A Hybrid Software and Hardware System for Collision Detection

Muiris Woulfe

24 September 2010

Submitted for examination: 22 December 2009

A thesis submitted to the Office of Graduate Studies of The University of Dublin,
Trinity College in candidacy for the degree of Doctor of Philosophy in Computer Science.

Declaration

I, the undersigned, declare that this thesis has not been submitted as an exercise for a degree at this or any other university and that it is entirely my own work. I agree that Trinity College Library may lend or copy this thesis upon request. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Muiris Woulfe
24 September 2010

© 2010 Muiris Woulfe

Abstract

Collision detection is a fundamental problem in myriad applications, spanning a wide variety of fields. Although the core concept is relatively straightforward, significant complexities result from the demand for interactive or real-time performance for ever-increasing workloads.

In this thesis, a different approach is applied to the problem of collision detection. This approach consists of a hybrid software and hardware system that utilises the properties of both software and hardware to achieve an acceleration over the current state of the art. At its core, the system consists of a custom microarchitecture performing broad phase collision detection, with the narrow phase executing in software. The microarchitecture consists of a plurality of memories internal to the microarchitecture, which parallelise the input data. These data are then formed into pairs using a custom data access algorithm, before being processed using a parallel version of the all-pairs algorithm. Two equivalent microarchitectures are presented: the first is designed for microarchitecture platforms with highly constrained resources, while the second supports many more objects. Both designs are prototyped using a reconfigurable hardware platform known as a field-programmable gate array (FPGA).

Despite the major advantages that can be achieved using this system, a question remains over how to deal with limitations on the size of internal memories in microarchitecture platforms. These limitations would significantly curtail the number of objects that could be processed by the microarchitectures. To answer this question, spatial partitioning is proposed. This partitions the environment into cells of appropriate size for processing by the microarchitecture. After the software reads a potentially colliding set (PCS) from the microarchitecture, narrow phase processing of the objects begins and this processing is overlapped with the microarchitecture's processing of the subsequent cell.

A framework for benchmarking collision detection algorithms is also presented. This framework is based on a physics engine, in order to accurately reflect scenarios found in many interactive applications, and includes a set of parameters that allow the creation of myriad practical benchmarks from a single generic benchmark. Collision detection algorithm developers can use the benchmark to rapidly determine which scenarios perform inefficiently with their algorithm. In addition, developers of applications utilising collision detection can recreate their scenarios using the benchmark to rapidly find the most efficient algorithm for those scenarios.

Acknowledgements

It's not the destination that's important; it's the journey that takes you there.

I wish to express my sincerest thanks and gratitude to my thesis advisor, Dr Michael Manzke, who provided constant encouragement, insight and guidance throughout my studies. His direction during the development of ideas and methodologies in the fields of collision detection and graphics hardware was invaluable. *Ich danke Ihnen vielmals.*

Thanks are also due to Dr John Dingliana, whose expertise in the field of collision detection helped me determine the optimal direction in which to proceed with my research. I also wish to acknowledge the contribution of Dr David Gregg, who provided me with access to a field-programmable gate array (FPGA) and its associated system, which were vital to the successful completion of this research.

I must extend my thanks to all of my colleagues in the Graphics, Vision and Visualisation Group (GV2) for their helpful advice. In particular, I wish to offer my earnest thanks to (in alphabetical order) Dr Ross Brennan, Dr Owen Callanan, Dr Eoin Creedon, Michael Doyle, Bartosz Fabianowski, Colin Fowler and Jiang Zhou. Their numerous suggestions and clarifications in a variety of fields helped shape this thesis.

I would like to especially thank my parents, to whom I am forever indebted for their support, encouragement and patience throughout this research.

Finally, I acknowledge the contribution of the Irish Research Council for Science, Engineering and Technology (IRCSET), funded by the National Development Plan (NDP), who supported the initial stages of this research.

Thesis Advisor and Examiners

Thesis Advisor

Dr Michael Manzke
Graphics, Vision and Visualisation Group (GV2)
School of Computer Science and Statistics
Trinity College Dublin
Ireland

External Examiner

Prof Gabriel Zachmann
Arbeitsgruppe Computer Graphik
Institut für Informatik
Technische Universität Clausthal
Germany

Computer Graphics Clausthal (CGC) Group
Department of Informatics
Technical University of Clausthal

Internal Examiner

Dr Steven Collins
Graphics, Vision and Visualisation Group (GV2)
School of Computer Science and Statistics
Trinity College Dublin
Ireland

Contents

Abbreviations xv

Mathematical Variables xxi

Units xxiii

1 Introduction 1
- 1.1 Benchmarking Framework . . . 5
- 1.2 Contributions . . . 7
  - 1.2.1 Benchmarking Framework . . . 8
- 1.3 Thesis Statement . . . 8
- 1.4 Publications . . . 9
  - 1.4.1 Directly Relevant . . . 9
  - 1.4.2 Indirectly Relevant . . . 9
- 1.5 Summary of Chapters . . . 10

2 Background and Related Work 13
- 2.1 Collision Detection . . . 13
  - 2.1.1 Broad Phase . . . 13
  - 2.1.2 Narrow Phase . . . 20
  - 2.1.3 Graphics Processing Units (GPUs) . . . 23
- 2.2 Hybrid Systems . . . 26
  - 2.2.1 Microarchitectures . . . 26
  - 2.2.2 Collision Detection . . . 27
  - 2.2.3 Physics Simulation . . . 29
  - 2.2.4 *N*-Body Simulations . . . 31
- 2.3 Benchmarking Frameworks . . . 33
- 2.4 Summary . . . 36

3 Design 37
- 3.1 Software and Hardware Divide . . . 37

3.2 Area-Efficient Microarchitecture . . . 39
3.3 Many-Object Microarchitecture . . . 47
3.4 Spatial Partitioning . . . 54
3.5 Other Collision Detection Types . . . 58
3.6 Benchmarking Framework . . . 58
3.6.1 Generic Benchmark . . . 59
3.6.2 Parameters . . . 59
3.6.3 Repeatability . . . 66
3.6.4 Measurement . . . 68
3.7 Summary . . . 68

4 Implementation **71**
4.1 Target Platform Evaluation . . . 71
4.2 Area-Efficient Microarchitecture . . . 78
4.2.1 Xilinx Virtex-II . . . 78
4.2.2 Xilinx Virtex-5 . . . 83
4.3 Many-Object Microarchitecture . . . 86
4.4 Spatial Partitioning . . . 87
4.5 Benchmarking Framework . . . 91
4.5.1 Collision Detection . . . 95
4.5.2 Measurement . . . 95
4.5.3 Application Programming Interface (API) . . . 95
4.5.4 Open Source . . . 96
4.6 Summary . . . 96

5 Results and Evaluation **97**
5.1 Benchmarking Framework . . . 97
5.2 Area-Efficient Microarchitecture . . . 113
5.2.1 Xilinx Virtex-II . . . 113
5.2.2 Xilinx Virtex-5 . . . 124
5.3 Many-Object Microarchitecture . . . 132
5.4 Spatial Partitioning . . . 140
5.4.1 Area-Efficient Microarchitecture, Xilinx Virtex-II . . . 140
5.4.2 Area-Efficient Microarchitecture, Xilinx Virtex-5 . . . 141
5.4.3 Many-Object Microarchitecture . . . 154
5.5 Evaluation . . . 159
5.6 Summary . . . 159

6 Conclusion and Future Work **161**
6.1 Future Work . . . 162
6.1.1 Benchmarking Framework . . . 165
6.2 Conclusion . . . 168

Appendices **171**

A Generalised Microarchitectures **171**
A.1 Design . . . 171
A.1.1 One-Source . . . 171
A.1.2 Two-Source . . . 174
A.2 Implementation . . . 179
A.3 Advantages . . . 179
A.4 Summary . . . 180

B Settings and Specifications **181**
B.1 Hardware . . . 181
B.2 Software . . . 183
B.3 Computers . . . 184

C Source Code **185**
C.1 Coding Style . . . 185
C.2 Area-Efficient Microarchitecture . . . 186
C.3 Many-Object Microarchitecture . . . 194

D Supplemental Results **205**
D.1 Standard Benchmark . . . 206
D.2 Area-Efficient Microarchitecture . . . 207
D.2.1 Xilinx Virtex-II . . . 207
D.2.2 Xilinx Virtex-5 . . . 211
D.3 Many-Object Microarchitecture . . . 215
D.4 Spatial Partitioning . . . 220
D.4.1 Area-Efficient Microarchitecture, Xilinx Virtex-5 . . . 220
D.4.2 Many-Object Microarchitecture . . . 224
D.5 Evaluation . . . 251

References **257**

Abbreviations

3D-DDA	3D digital differential analyser
AABB	axis-aligned bounding box
ADF	adaptively sampled distance field
ADT	abstract data type
AI	artificial intelligence
ALU	arithmetic logic unit
API	application programming interface
ASIC	application-specific integrated circuit
BART	A Benchmark for Animated Ray Tracing
BLAS	basic linear algebra subprograms
BRAM	block RAM
BSD	Berkeley Software Distribution
BSP	binary space partitioning
BV	bounding volume
BVH	bounding volume hierarchy
CA	conservative advancement
CAD	computer-aided design
CAE	computer-aided engineering
CCD	continuous collision detection
CD	collision detection
CFD	computational fluid dynamics
CG	coarse granularity
CPU	central processing unit
CSG	constructive solid geometry

CUDA	Compute Unified Device Architecture
DBVT	dynamic bounding volume tree
DDR SDRAM	double data rate SDRAM
DDR2 SDRAM	double data rate 2 SDRAM
DK	Dobkin-Kirkpatrick
DMA	direct memory access
DME	data movement engine
EPA	expanding polytope algorithm
FDH	fixed-direction hull
FG	fine granularity
FIFO	first-in first-out
FPE	floating-point engine
FPGA	field-programmable gate array
FSM	finite-state machine
GJK	Gilbert-Johnson-Keerthi
GPU	graphics processing unit
GRAPE	Gravity Pipe
GUI	graphical user interface
HAQMC	Hardware-Accelerated Quantum Monte Carlo
HDL	hardware description language
hgrid	hierarchical grid
HPCCD	hybrid parallel CCD
HUG	hierarchy of uniform grids
I/O	input/output
IC	integrated circuit
IEEE	Institute of Electrical and Electronics Engineers
ISA	instruction set architecture
ISA-GJK	incremental separating axis GJK
ISP	instruction set processor

***k*-DOP**	discrete oriented polytope
KDS	kinetic data structure
L1	level 1
L2	level 2
LC	Lin-Canny
LCP	linear complementarity problem
LDI	layered depth image
LNS	logarithmic number system
LOD	level of detail
LP	linear programming
LSS	line-swept sphere
LTS	long-term support
MD	molecular dynamics
MPI	Message Passing Interface
NaN	not a number
NAS	NASA Advanced Supercomputing
NASA	National Aeronautics and Space Administration
NPB	NAS Parallel Benchmarks
NRE	non-recurring engineering
NURBS	non-uniform rational B-spline
OBB	oriented bounding box
ODE	Open Dynamics Engine
ODE	ordinary differential equation
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
OS	operating system
PAL	Physics Abstraction Layer
PCB	printed circuit board
PCE	PPU control engine

PCI peripheral component interconnect
PCS potentially colliding set
PE processing element
PIO programmed input/output
PMC PCI mezzanine card
PPE power processing element
PPU physics processing unit
PSS point-swept sphere

QMC Quantum Monte Carlo

RAM random access memory
RD retroactive detection
RISC reduced instruction set computer
RPU Ray Processing Unit
RSS rectangle-swept sphere

s-tope spherically-extended polytope
SAP sweep and prune
SAT separating axis test
SDK software development kit
SDRAM synchronous dynamic RAM
SFU scalar floating-point unit
SI *le système international d'unités*
SIU scalar integer unit
SoC system on chip
SOLID Software Library for Interference Detection
SPARTA Simulation of Physics on a Real-Time Architecture
SPD Standard Procedural Databases
SPE synergistic processing element
SPH smoothed-particle hydrodynamics
SRAM static RAM
SSL sphere-swept line
SSP sphere-swept point
SSR sphere-swept rectangle

SSV sphere-swept volume
STL Standard Template Library

TANOR Tool for Accelerating N-Body Simulations on Reconfigurable Platforms
TAT turn around time
TR1 Technical Report 1

V-Clip Voronoi Clip
VFU vector floating-point unit
VHDL VHSIC HDL
VHSIC very high speed integrated circuit
VLIW very long instruction word

ZBT SSRAM zero bus turnaround synchronous static RAM

Mathematical Variables

a	internal memory address
c	limit on number of objects in spatial-partitioning cell
C_O	cost of one-time processing
C_P	cost of testing a primitive pair
C_U	cost of updating a node
C_V	cost of testing a bounding volume (BV) pair
d	depth of internal memory
f	clock frequency [hertz]
j	index of internal memory being accessed
m	parallelism factor of the microarchitectures
n	number of objects
N_P	number of primitive pairs to test
N_U	number of nodes to update
N_V	number of bounding volume (BV) pairs to test
p	number of ports on an internal memory
T	total cost of testing a bounding volume hierarchy (BVH) pair
w	width of internal memory [bit]

x spatial-partitioning cell size in x direction

y spatial-partitioning cell size in y direction

z spatial-partitioning cell size in z direction

Units

µs	microsecond, 0.000001 seconds
fps	frames per second
g	gram
GB	gigabyte, 1 000 000 000 bytes
GHz	gigahertz, 1 000 000 000 Hz
GiB	gibibyte, 1 073 741 824 bytes
Hz	hertz
kg	kilogram, 1000 g
KiB	kibibyte, 1024 bytes
m	metre
MB	megabyte, 1 000 000 bytes
MHz	megahertz, 1 000 000 Hz
MiB	mebibyte, 1 048 576 bytes
mm	millimetre, 0.001 m
rad	radian

Chapter 1

Introduction

Every new opinion, at its starting, is precisely in a minority of one.

—Thomas Carlyle, essayist and historian (1795–1881)

Collision detection is the process of determining if objects in a computer-simulated environment are intersecting. In cooperation with collision response, which deals with the effect of collisions on objects' behaviour, it ensures that objects do not travel through one another. Therefore, collision detection, which is also referred to as *intersection detection* or *interference detection*, is vital for maintaining the illusion of an environment comprising solid objects.

Collision detection is a fundamental ingredient of a multitude of varied applications, including computer games [Liu et al, 2008], virtual environments [Su, 2007], computer animation [Palmer and Grimsdale, 1995], robotics [Steinbach et al, 2006], computer-aided design (CAD) and computer-aided engineering (CAE) [Tang et al, 2007]. For example, computer games use collision detection to prevent characters from walking through walls or falling through floors. It can also be used to perform line-of-sight queries, enabling simulated characters to determine when a player character should be within view. Similarly, virtual environments, such as architectural simulations, use collision detection to ensure realistic character motion. In computer animation, collision detection is used for automating laborious animation processes such as ensuring that physically simulated cloth intersects neither itself nor adjacent objects during motion. The primary use of collision detection in robotics is motion planning, which is the process of determining a collision-free path for a robot in a restricted environment containing a known set of obstacles. Finally, in CAD and CAE, collision detection aids the designer or engineer in preventing the occurrence of overlapping objects. For the detailed models used in such applications, it can often be challenging to visually determine if such overlap is occurring. Therefore, collision detection is vital in preventing malformed models entering production.

The first challenge in collision detection is to execute the collision detection cycles rapidly enough to enable the application to meet the necessary frame rate constraints. Some applications, such as

computer animation, do not make significant frame rate demands due to the non-interactive nature of such applications. Others, including virtual environments, CAD and CAE, demand interactive frame rates of around 1 fps or greater. However, the most significant demands are made by applications such as computer games, which require real-time frame rates. These usually attempt to achieve the highest possible frame rate within the constraints imposed by the refresh rate of the monitor, although a frame rate of 30 fps is often sufficient. Visual and physical disparities in the simulation will only be noticed once the frame rate drops significantly below the refresh rate. It is also vital to maintain a constant frame rate for all of these applications, as researchers have demonstrated that when frame rates drop below a certain threshold, the user may experience motion sickness [Uliano et al, 1986; Hettinger and Riccio, 1992]. The situation is most challenging for applications utilising haptics, as 1000 collision detection cycles per second [Vahora et al, 1999] are required to achieve realistic tactile feedback.

The relatively short duration between frames is further reduced when, as in the vast majority of applications, collision detection comprises only a single stage of an application's program loop. As an example, the program loop of a computer game may comprise input processing, collision detection, collision response, physics, artificial intelligence (AI), audio and display, as illustrated in Figure 1.1. Other applications comprise similarly challenging program loops. In these cases, the duration between frames must be sufficient to execute all stages. It could be argued that the program loop can be accelerated by improving the performance of any stage. However, Lin and Gottschalk [1998] found that collision detection was a bottleneck for many applications. More recent studies demonstrated that collision detection, as part of a computer-game–style application, accounts for between 1% and 38.3% of the program loop, with an average application consuming 14%. These proportions were measured using the sweep and prune (SAP) algorithm and the 'Parameters (Ratio) Experiment' and 'Collisions (Ratio) Experiment' outlined in Section 5.1. They are significant proportions, which imply that major advantages could be gained through ameliorating the performance of collision detection.

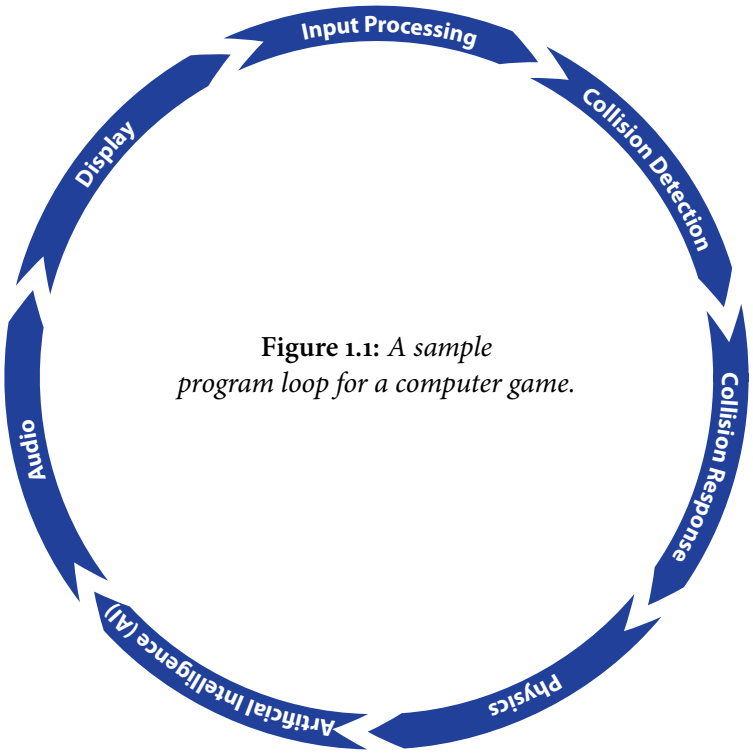


Figure 1.1: *A sample program loop for a computer game.*

The second challenge is to achieve an appropriate degree of accuracy within the relevant time constraints. Some applications, including robotics, CAD and CAE, demand a high degree of accu-

racy if the application is to function correctly. For other applications, such as computer games, virtual environments and computer animation, accuracy is less important and can be traded for speed. Some researchers have attempted to leverage this truism using perception techniques [O'Sullivan et al, 1999]. However, these applications still demand a certain level of accuracy or the application will appear to behave unrealistically.

To help achieve an appropriate frame rate and accuracy, a scheme referred to as hybrid collision detection is typically employed. For clarity, this thesis will refer to the scheme by its alternative name of two-phase collision detection. The scheme involves the division of the collision detection system into two distinct phases, referred to as the *broad phase* and the *narrow phase* in the taxonomy proposed by Hubbard [1993a]. In the literature, there exist a plurality of definitions for these terms but the spirit of the original definitions formulated by Hubbard is maintained here:

Broad Phase The first collision detection phase, which uses an approximate and fast test to enumerate appropriate pairs of objects into a potentially colliding set (PCS).

Narrow Phase The second collision detection phase, which checks the PCS using an accurate or exact algorithm. It is occasionally divided into two distinct phases, which are referred to as the mid and narrow phases.

This division is illustrated in Figure 1.2. Since the broad phase can rapidly eliminate pairs of objects that cannot possibly collide, two-phase collision detection can lead to a significant reduction in the computational time required to process the collision detection routines. For this reason, two-phase collision detection is currently used in almost all applications utilising collision detection.

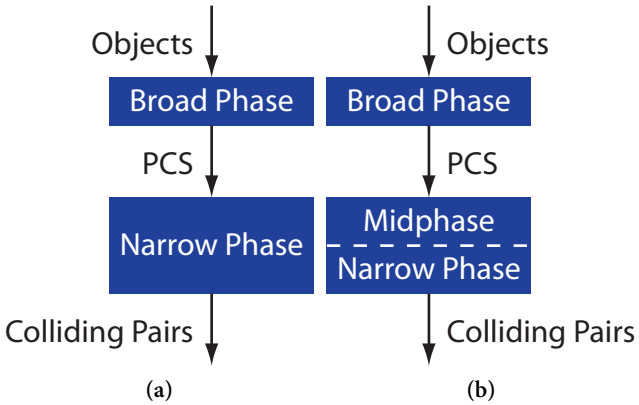


Figure 1.2: *Collision detection phases.* **(a)** *Standard, without midphase.* **(b)** *With midphase.*

Despite offering considerable benefits, two-phase collision detection inadequately solves the two challenges. In particular, the worst-case computational complexity of all broad phase algorithms is $O(n^2)$. This complexity also applies to all sophisticated algorithms that exploit temporal and spatial coherence to reduce the number of updates to the algorithm's data structures. This inherent problem is illustrated by studies showing that broad phase collision detection can account for between 2.5% and

98.7% of the overall collision detection execution time, with the broad phase of an average application accounting for 40.4%. These proportions were measured using the SAP algorithm and the 'Parameters (Ratio) Experiment' and 'Collisions (Ratio) Experiment' outlined in Section 5.1.

Even when none of the challenges prove problematic, it is desirable to lessen the impact of collision detection on the application. This would make more computational time available for the various stages of the program loop, allowing for the creation of more realistic and detailed environments. Adding realism and detail is vital for modern applications, particularly in computer games and computer animation, due to the insatiable appetite of users for increasingly believable simulated environments.

For the aforementioned reasons, it is imperative to improve the performance of collision detection. However, it is a relatively mature research area and few major developments have occurred over the past few years. Instead, the focus has been on optimisation of existing algorithms, providing incremental performance improvements. Although such incremental developments are beneficial, they fail to produce the advances necessary to truly surmount the aforementioned challenges. Therefore, it is vital to examine potential solutions that offer a radical departure from the current state of the art.

Focusing momentarily on existing algorithms, it is evident that the majority are targeted towards software implementation, although there has recently been significant interest in graphics processing unit (GPU) implementations. Both implementations are applicable to modern commodity computers and can achieve significant accelerations, although the general-purpose design of the CPU and GPU means they are unlikely to achieve the maximum attainable acceleration.

To bypass the limitations of these general-purpose platforms, a custom microarchitecture is proposed. A microarchitecture specifies the layout of the basic hardware elements required to execute a given algorithm. This could be implemented on a wide variety of hardware platforms including a field-programmable gate array (FPGA), an application-specific integrated circuit (ASIC) or as part of the fixed-function logic found in a CPU or GPU. Other algorithms for graphics applications, including ray-tracing [Woop et al, 2006], volume rendering [Knittel, 2007], 3D compositing [Moll et al, 1999], image processing [Knittel, 1996; Singh and Slous, 1998], culling for objects undergoing rasterisation [Hasselgren and Akenine-Möller, 2007] and point-based rending [Heinzle et al, 2008], have already been successfully implemented as microarchitectures. Therefore, it is evident that a custom microarchitecture could offer a significant acceleration when executing collision detection.

Microarchitectures gain some acceleration through the removal of superfluous logic found in general-purpose platforms. However, their primary source of acceleration results from the utilisation of parallelism, primarily of the kind where a large multitude of threads can execute simultaneously without interdependencies. These kinds of parallelism are particularly evident in algorithms that are compute-bound. In contrast, algorithms with less scope for parallelism or those that are memory-bound are best left to execute in software. The microarchitecture would provide a lower acceleration factor for such algorithms. Collision detection systems typically consist of both types of algorithms.

Therefore, a combination of software and hardware is most appropriate for accelerating collision detection.

In addition to providing the potential to accelerate many algorithms, an ancillary benefit is that microarchitectures provide the potential to reduce power consumption. Power consumption is increasingly an issue in computing, since greater quantities of energy are necessary to provide ever-faster performance. This problem is highlighted by the trend towards demanding computations, including the execution of interactive graphics applications, on power-sensitive mobile computing platforms. Microarchitectures provide a means to reduce this power consumption, as they typically achieve better performance per watt. This increased energy efficiency is a result of microarchitectures' use of parallelism and the removal of unnecessary logic. Such efficiencies can be realised even if the CPU remains idle during parts of the microarchitecture execution, as a result of technologies such as Intel SpeedStep [Gochman et al, 2003]. Alternatively, if the microarchitecture were hosted on the CPU, efficiencies could still be achieved as parts of the CPU could be disabled while not in use.

Based on this analysis, this thesis proposes the creation of a hybrid software and hardware system for collision detection. The system comprises a custom microarchitecture suitable for implementing the highly parallel, compute-bound parts of the system, coupled with software that implements the remainder. The custom microarchitecture implements a parallel version of the all-pairs broad phase algorithm, which uses a plurality of internal memories that allow the input data to be shared among many parallel threads. These memories are accessed using a custom data access algorithm that ensures every pair of objects is considered. The microarchitecture also demonstrates how an algorithm with an undesirable computational complexity can be made practical using parallelisation techniques. FPGA implementations of the microarchitecture are discussed and analysed, and theoretical CPU implementations are quantified. To overcome some of the disadvantages associated with the microarchitecture, software-based spatial partitioning is used to divide the environment into cells. These cells differ from those used in typical applications, as they are large enough to facilitate the efficient execution of the microarchitecture. Execution delays can be hidden as each PCS could be processed by the narrow phase while the microarchitecture is processing the subsequent cell. The narrow phase is implemented in software. The entire system focuses primarily on rigid bodies processed using static collision detection, but some straightforward adaptations that would allow the system to support deformable objects processed using dynamic collision detection are investigated. The system leverages the unique attributes of both software and hardware to achieve highly accelerated collision detection.

1.1 Benchmarking Framework

There exist standardised and comprehensive benchmarks in many areas. Standardised benchmarks facilitate the rapid comparison of different algorithms, so that it can be easily determined whether a new algorithm offers any appreciable benefit over an older algorithm. Likewise, comprehensive benchmarks allow for the thorough analysis of an algorithm across every potential scenario, so that those scenarios which cause an algorithm to perform inefficiently can be easily identified. Despite their ad-

vantages, there are essentially no standardised and comprehensive benchmarks in collision detection. This problem was discovered during work on the hybrid system and creates complexities for accurately determining the system's performance. The lack of such benchmarks has also resulted in a variety of other problems.

When new algorithms and adaptations of existing algorithms are formulated, the algorithms are often tested using a small selection of benchmarks designed by the algorithm developer. These benchmarks require a significant time investment and this time could be better spent on optimisation of the algorithm. Moreover, these benchmarks are unlikely to recreate the full range of scenarios that the algorithm will encounter. This is a significant problem, as Caselli et al [2002] determined that the behaviour of collision detection algorithms is often very sensitive to the design of the scenario. It would be advantageous if developers could rapidly test their algorithm using a spectrum of benchmarks to gain a more complete understanding of how their algorithm performs, while reducing time spent developing appropriate benchmarks. This would assist them in focusing on optimising those cases in which their algorithm fails to perform efficiently.

Benchmarks designed by algorithm developers also create challenges for developers of applications utilising collision detection. Execution times measured using these benchmarks are often published as results in collision detection research papers, making it difficult to determine the algorithm most amenable to a given application's scenarios. A developer could select an algorithm that appears to be superior to existing algorithms, but this algorithm might not perform efficiently with the scenarios used in the application. Alternatively, they could select the algorithm whose published benchmarks most closely match the application scenarios, but this removes the possibility of selecting a better algorithm whose developer did not publish certain benchmarks. In both cases, there is a significant risk that a developer will select a non-optimal algorithm. Therefore, it would be beneficial if an application developer could quickly determine the behaviour of a wide variety of collision detection algorithms using benchmarks approximating application scenarios.

To mitigate these issues, this thesis proposes a framework for benchmarking collision detection. This framework is a comprehensive tool for the generation of a wide variety of benchmarks, from which standardised benchmarks could be defined. It consists of a single generic benchmark that may be modified using a number of parameters to create a large range of practical benchmarks. These parameters reflect the geometrical and physical properties of simulated objects and provide enormous flexibility in the creation of benchmarks. In addition, the framework is based on a physics engine, in order to accurately reflect the scenarios most commonly found in interactive applications. A variety of statistics is measured during algorithm execution, to enable developers to understand algorithm performance. The framework was also used as a tool to quantify the performance of the hybrid system.

Using this framework, algorithm developers can define benchmarks spanning a very large test space, against which they can test their algorithm to determine which areas are problematic. After rectifying any issues highlighted by the framework, they can conclusively prove that their algorithm works efficiently across a broad range of potential scenarios. Alternatively, application developers can use the framework to implement benchmarks approximating scenarios found in their applications, to

quickly check a wide range of potential algorithms in order to determine the most appropriate one for their application. Therefore, such a framework should prove invaluable to both algorithm and application developers.

1.2 Contributions

The research documented in this thesis contributes to the field of collision detection in general and, specifically, to the area of broad phase collision detection. It is also of relevance to researchers whose interests lie in the area of hybrid systems, custom microarchitectures and the splitting of applications across software and hardware, as well as to those researching benchmarking.

The hybrid system attempts to optimally distribute a collision detection system across software and a custom hardware microarchitecture. It is recognised that custom microarchitectures provide the greatest acceleration to compute-bound algorithms exhibiting high degrees of parallelism. Based on this truism, **a number of collision detection algorithms are analysed to determine the optimal split between software and hardware (Section 3.1).** It is determined that the broad phase is most amenable to microarchitecture acceleration, while the narrow phase can achieve excellent performance by remaining in software.

Based on this split, **two microarchitectures (Sections 3.2 and 3.3)** for broad phase collision detection are designed. The first is area-efficient and suitable for platforms with highly constrained resources, while the second supports many objects on platforms with fewer limitations. Through exploitation of a plurality of internal memories, these microarchitectures demonstrate how data can be buffered to increase an algorithm's parallelism. For both of these microarchitectures, **a variety of parameters allows many of their aspects to be adjusted (Sections 3.2 and 3.3).** This permits the two microarchitectures to be implemented on a wide variety of target hardware platforms.

However, a disadvantage of the proposed microarchitectures is that they are somewhat constrained in terms of the number of objects supported. To overcome this limitation, a **software-based spatial partitioning scheme, customised to the requirements of the microarchitectures (Section 3.4),** is discussed. This scheme divides the environment into cells and places each object into the appropriate cell. These cells are subsequently transferred to the microarchitecture. The scheme differs from existing spatial partitioning schemes in that the number of objects per cell are maximised within certain limits, unlike traditional schemes for software collision detection, which tend to minimise the number of objects per cell to reduce the number of pairwise collision detection tests. Spatial partitioning has an auxiliary advantage in that it reduces the computational complexity of the system.

FPGA-based prototypes of the aforementioned microarchitectures (Sections 4.2 and 4.3) are subsequently discussed. These implementations demonstrate how the microarchitectures can be adapted for a specific target platform. Moreover, two different FPGAs are used, which further **highlights the adaptability of the microarchitectures (Sections 4.2 and 4.3).**

A **set of analyses (Sections 5.2 and 5.3)** are performed on the hybrid systems. These analyses find that the FPGA implementation can achieve an acceleration of 3.54× over a state-of-the-art software

algorithm, even when the communication overhead is taken into account. Removing the communication overhead increases the acceleration to 80.893×, while retargeting the microarchitecture for placement on a CPU would result in accelerations of up to 1565.546×.

Finally, the **spatial partitioning algorithms are investigated (Section 5.4).** These analyses show an acceleration of up to 2.387× including the communication overhead and of up to 10.443× excluding this overhead. An implementation retargeted for CPU placement provides an acceleration of up to 26.6×. In addition, it is determined that further accelerations could be achieved using more sophisticated spatial partitioning implementations. A variety of **spatial partitioning algorithms is also investigated (Section 5.4)** and the **optimal cell size is considered (Section 5.4).**

1.2.1 Benchmarking Framework

The benchmarking framework is a tool to generate benchmarks in order to facilitate the acceleration of collision detection algorithms. Unlike previous collision detection benchmarks, this framework utilises **a physics engine (Section 3.6)** and **a set of parameters (Section 3.6.2).** The use of a physics engine is significant as such an engine drives the collision detection present in the majority of interactive applications. Meanwhile, the set of parameters allows the user to create myriad practical benchmarks from a single generic benchmark. During execution of the benchmarks, **a variety of statistics (Section 3.6.4)** is recorded.

The framework is beneficial to collision detection algorithm developers who can modify the parameters to determine the performance of their algorithm across a wide range of scenarios, and to compare their algorithm against their competitors'. Using the provided data, they can adapt their algorithm to ameliorate its performance before retesting. To this end, the framework provides **an application programming interface (API) for the integration of additional collision detection algorithms (Section 4.5.3).** The framework is also beneficial for application developers, who can use the parameters to recreate application scenarios. Using the resultant benchmarks, they can rapidly analyse a wide range of collision detection algorithms with a view to determining the optimal.

Following implementation of the framework, **a variety of analyses is performed on a selection of broad phase collision detection algorithms (Section 5.1).** These analyses highlight the behaviour of the framework and demonstrate its utility, while revealing information about the performance of the three algorithms.

1.3 Thesis Statement

> Collision detection can be significantly accelerated using a hybrid software and hardware system. By appropriately leveraging the properties of both parts of the system, it is possible to make performance gains that would be unattainable using software alone.

1.4 Publications

1.4.1 Directly Relevant

Manzke, M and **Woulfe, M.** 2009. Micro-architecture system and method for data processing. Irish Patent s2009/0940.

Woulfe, M and Manzke, M. 2009. A framework for benchmarking interactive collision detection. In *Proceedings of the 25th Spring Conference on Computer Graphics (SCCG 2009)*. Comenius University, Bratislava, Slovakia, 221–228.

Woulfe, M, Dingliana, J and Manzke, M. 2008. Hardware accelerated broad phase collision detection for interactive entertainment applications. In *Proceedings of the 2008 Irish Graduate Student Symposium on Vision, Graphics and Visualisation (VGV 2008)*. Trinity College, Dublin, Ireland, 20. Poster.

Woulfe, M, Dingliana, J and Manzke, M. 2007. Hardware accelerated broad phase collision detection for realtime simulations. In *Proceedings of the 4th Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS 2007)*. Eurographics Association, Aire-la-Ville, Switzerland, 79–88.

Woulfe, M, Dingliana, J and Manzke, M. 2007. Hardware accelerated broad phase collision detection. In *Proceedings of the 34th International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2007)*. ACM, New York, USA. Poster.

1.4.2 Indirectly Relevant

Woulfe, M, Doyle, M and Manzke, M. 2010. Collision detection hardware optimised for raytracers. In *Proceedings of the 2010 Conference on High-Performance Graphics (HPG 2010)*. ACM, New York, USA. Poster.

Woulfe, M, Creedon, E, Brennan, R, Doyle, M and Manzke, M. 2009. Programming models for FPGA application accelerators. In *Proceedings of the 1st Workshop on Programming Models for Emerging Architectures (PMEA 2009)*. Barcelona Supercomputing Center, Barcelona, Spain, 77–83.

Woulfe, M and Manzke, M. 2006. Towards a Field-Programmable Physics Processor (FP^3). In *Proceedings of the 7th Irish Workshop on Computer Graphics (Eurographics Ireland Chapter 2006)*. IADT, Dún Laoghaire, Ireland, 44–50.

Woulfe, M and Manzke, M. 2006. Field-Programmable Physics Processor (FP^3). In *Proceedings of the 2006 Xilinx Academic Forum Ireland (XAFI 2006)*. Xilinx, Dublin, Ireland. Poster.

1.5 Summary of Chapters

Chapter 2: Background and Related Work This chapter provides a background to the contributions of this thesis by providing a detailed overview of prior research conducted in the areas of collision detection, hybrid systems and benchmarking frameworks.

Chapter 3: Design The design of all components comprising this thesis are outlined in Chapter 3. These components include a variety of hybrid systems for accelerating collision detection along with a benchmarking framework for analysing their performance. For the hybrid systems, two variations of the microarchitecture component are presented. The first is area-efficient and designed for hardware platforms with significantly constrained resources, while the second supports many objects but is restricted to larger platforms. A spatial partitioning scheme is presented to overcome object limitations in various microarchitecture platforms and to avoid undesirable computational complexities.

Chapter 4: Implementation This chapter considers the implementation of the aforementioned designs, including software and hardware implementations of the hybrid systems and a software implementation of the benchmarking framework. For the hybrid systems, the area-efficient design was implemented on both a Xilinx Virtex-II FPGA and a Xilinx Virtex-5 FPGA, while the many-object design was implemented on only a Xilinx Virtex-5 FPGA. Consequently, this chapter addresses the differences that arise when implementing the same design on differing platforms.

Chapter 5: Results and Evaluation The performance of the benchmarking framework and the hybrid systems are considered in this chapter. For the benchmarking framework, a variety of benchmarks is created and analysed to demonstrate the framework's utility. In the case of the hybrid systems, the benchmarking framework is used to thoroughly analyse their performance and to determine which scenarios perform most efficiently, along with any scenarios that inhibit performance.

Chapter 6: Conclusion and Future Work This chapter discusses the research presented throughout this thesis. Suggestions for future improvements and possible directions for future research are also outlined.

Appendix A: Generalised Microarchitectures This appendix considers how the microarchitectures can be generalised to support algorithms other than collision detection. This facilitates the distribution of the innovative microarchitecture concepts to other research areas.

Appendix B: Settings and Specifications A complete list of the compiler settings, software library versions and hardware platform specifications used for the different implementations are included in this appendix for reference purposes. These complete settings and specifications permit the different platforms to be thoroughly compared.

Appendix C: Source Code Source code for selected microarchitecture operations are included in this appendix. This code complements Chapter 4 in providing an understanding of how the designs are implemented.

Appendix D: Supplemental Results This appendix provides additional results for the hybrid systems. These facilitate an evaluation and understanding of the different systems and their implementations, but are of lesser importance than the results provided in Chapter 5.

Chapter 2

Background and Related Work

前事不忘，后事之师。
Qián shì bù wàng, hòu shì zhī shī.
Consider the past and you shall know the future.

—Chinese proverb

Background comprises the prior knowledge on which a given subject relies. It is associated with related work, which is the published research that builds on this knowledge. This chapter analyses this prior knowledge and research, commencing with a thorough review of collision detection. Broad phase algorithms are the focus, but important narrow phase and GPU-based algorithms are also presented. Following this review, hybrid systems are investigated in the context of general microarchitecture research, collision detection, physics simulation and N-body simulations. The chapter concludes with an examination of some benchmarking frameworks.

2.1 Collision Detection

As stated in Chapter 1, collision detection is typically divided into a broad phase and a narrow phase, in accordance with the two-phase collision detection scheme proposed by Hubbard [1993a]. In this section, the cited collision detection research is divided into these two categories using Hubbard's original definitions, although not all of the cited literature makes this distinction and some of the literature uses alternative definitions. This section concludes with a review of GPU-based collision detection.

2.1.1 Broad Phase

Broad phase collision detection typically exploits bounding volumes (BVs), which are simple geometric structures that enclose the complex geometry of the objects in order to reduce the execution time of

the collision tests. They can be considered a simplification of the true display geometry. The advantage of using BVs is that testing BVs for collision is much quicker and simpler than testing the individual polygons comprising complex, arbitrarily shaped objects. This is especially true for concave objects and, consequently, BVs tend to be convex.

Simple BV include spheres [Quinlan, 1994; Hubbard, 1996], whose primary advantage is that they are invariant under rotation. Axis-aligned bounding boxes (AABBs) [Baraff, 1992; Cohen et al, 1995] are cuboids whose edges are aligned with the world axes. Triboxes [Crosnier and Rossignac, 1999] adapt AABBs to include chamfered edges. Oriented bounding boxes (OBBs) [Gottschalk et al, 1996] generalise AABBs by removing the restriction that the edges of the cuboids must be aligned with the world axes. Discrete oriented polytopes (k-DOPs) or fixed-direction hulls (FDHs) [Held et al, 1996; Zikan and Konečný, 1997] generalise OBBs to k-sided parallelepipeds, where the surfaces of the k-DOPs consist of hyperplanes with normals from a fixed set of vectors.

More complex BVs include sphere-swept volumes (SSVs) [Larsen et al, 1999]. SSVs comprise: sphere-swept points (SSPs), which are also known as point-swept spheres (PSSes) or spheres; sphere-swept lines (SSLs), which are also known as line-swept spheres (LSSes), capsules, capped cylinders or spherocylinders; and sphere-swept rectangles (SSRs) [Larsen et al, 2000], which are also known as rectangle-swept spheres (RSSes) or lozenges. These BVs are constructed as the Minkowski sum of a sphere and the given primitive. The concept is generalised by the spherically-extended polytope (s-tope) [Hamlin et al, 1992], where the radius of the sphere can vary for each vertex. Other possible BVs include cylinders [Held, 1997], cones [Held, 1997], toroids [Held, 1997] and ellipsoids [Rimon and Boyd, 1992; Wang et al, 2001]. Spherical shells [Krishnan et al, 1998], which are formed by the intersection of the volume between two concentric spheres and a cone with its apex at the spheres' centres, and zonotopes [Guibas et al, 2003], which are centrally symmetric polyhedra exhibiting certain characteristics, have also been proposed. Finally, the most generic BV is the convex hull, which is constructed from the smallest convex-point set fully enclosing a given object. BVs are illustrated in Figure 2.1.

There is a significant challenge in determining the most appropriate BV for a given application, as there exists a tradeoff between execution speed and complexity. Simple BVs have short execution times for both BV construction and collision testing. However, they tend to enclose complex objects loosely, resulting in numerous potential collisions. In contrast, complex BVs tend to have much higher execution times, but their tight fit of the objects can significantly reduce the number of potential collisions detected. The optimal BV is the one that achieves the best execution times for the collision detection system as a whole. The intricacy is that the broad phase will execute more rapidly with simple BVs, but the resultant increased potential collisions will decelerate the narrow phase. Therefore, the challenge lies in finding the BV which leads to the optimal split between the two phases. However, the majority of applications tend not to address this challenge and instead use relatively simple BVs, with AABBs being the most widely used [van den Bergen, 2004, p 72]. Complex BVs have failed to achieve widespread use for broad phase collision detection, although some, such as the convex hull, are regularly used in the narrow phase where the goal is to determine collisions more accurately.

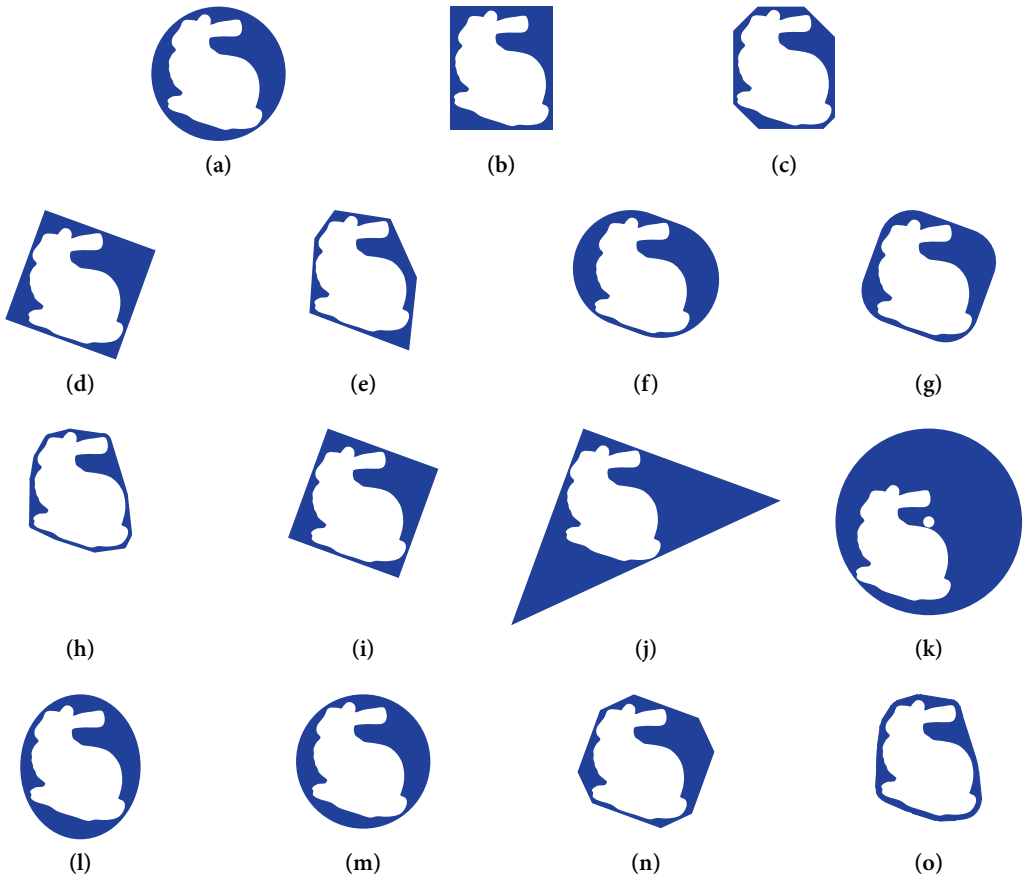


Figure 2.1: *2D representations of BVs.* **(a)** *Sphere.* **(b)** *AABB.* **(c)** *Tribox.* **(d)** *OBB.* **(e)** *8-DOP.* **(f)** *Capsule.* **(g)** *Lozenge.* **(h)** *S-tope.* **(i)** *Cylinder.* **(j)** *Cone.* **(k)** *Toroid.* **(l)** *Ellipsoid.* **(m)** *Spherical shell.* **(n)** *Zonotope.* **(o)** *Convex hull.*

Once the objects have been enclosed within appropriate BVs, it is necessary to process these BVs for collision using a broad phase collision detection algorithm. The most elementary of these is the brute force all-pairs algorithm. It checks each pair of BVs for collision, resulting in

$$\frac{n\,(n-1)}{2}$$

collision detection tests and a computational complexity of $O\,(n^2)$. All BV types are supported by all-pairs, making it one of the most flexible algorithms.

SAP [Cohen et al, 1995] or sort and sweep [Baraff, 1992, pp 53–57] improves all-pairs by exploiting temporal and geometrical coherence. Coherence relies on the observation that objects in most interactive applications tend to move relatively small distances between cycles. The algorithm uses three sorted sequences corresponding to the three coordinate axes to store the endpoints of each object, thereby implicitly creating AABBs. Each pair of objects has three flags indicating whether that pair is currently colliding on each coordinate axis. After each update iteration, the sequences are sorted. In the full sort variant [Baraff, 1992, pp 53–57], the sequences are sorted using any appropriate algorithm and are subsequently traversed to determine the colliding pairs. In the incremental sort variant [Cohen et al, 1995], the sequences are instead sorted with insertion sort, which only switches adjacent elements. When the right endpoint of one object is switched with the left endpoint of another, the collision flag for that pair is toggled. In both variants, a pair of objects is colliding if a collision is flagged on all three axes. An exemplar is presented in Figure 2.2. SAP has an average computational complexity of $O\,(n \log n)$ but a worst-case complexity of $O\,(n^2)$. It is the most widely used broad phase algorithm and is utilised by the Software Library for Interference Detection (SOLID) [van den Bergen, 2007], I-COLLIDE [Cohen et al, 1995] and V-Collide [Hudson et al, 1997] collision detection libraries.

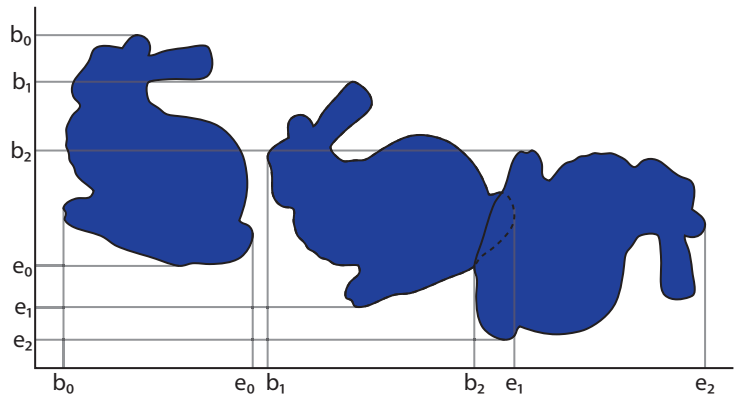


Figure 2.2: *2D exemplar of SAP. On the x-axis, object 0 collides with no object as no 'b_i' occurs between 'b_0' and 'e_0'. Objects 1 and 2 collide since 'b_2' occurs before 'e_1'. On the y-axis, all three objects collide since all three 'b' points occur before the three 'e' points. The actual colliding pairs are determined by the intersection of the colliding pairs for each axis, so the result is that objects 1 and 2 are colliding.*

Objects can also be processed using spatial partitioning algorithms, which attempt to minimise expensive pairwise tests by dividing the environment into cells and adding each object to the appropriate cell or cells. There are a number of variants.

Uniform Grid This grid is the simplest spatial partitioning algorithm due to its use of uniformly sized cells. Numerous adaptations have been proposed to deal with objects overlapping multiple cells and to move objects between cells at cycle intervals. The uniform grid has been used by Turk [1989, pp 12–17] for the acceleration of molecular simulations and by García-Alonso et al [1994] to simulate the kinematic analysis of complex articulated objects. Jin et al [2009] use a uniform grid to accelerate simulations involving uniformly distributed large objects undergoing deformation.

Hierarchical Grid (Hgrid) Simulations with variable-sized objects behave poorly with the uniform grid, as some objects will overlap many cells while others occupy only a small portion of each cell. The hierarchical grid (hgrid) attempts to ameliorate this situation by creating a hierarchy of grids with cells of different sizes, so that each object can be added to an appropriately sized cell. Implementations vary in the number of levels used, in whether an object is added to single or multiple levels and in the algorithm used to determine the cell size of each level. As for the uniform grid, numerous adaptations have been proposed to deal with objects overlapping multiple cells and to move objects between cells at cycle intervals.

Spatial Hashing Uniform and hierarchical grids do not perform efficiently in environments of changing size, as the grids must typically be rebuilt to accommodate new environment dimensions. Spatial hashing rectifies this problem by turning the cells into hash buckets, populated by an appropriate hash function. Since there are essentially no limits to the dimensions that can be hashed, this structure accommodates changing environment sizes. Moreover, spatial hashing is efficient for sparsely populated environments as only occupied cells must be stored. Despite spatial hashing's advantages, it is difficult to determine adjacent cells and multiple expensive hash table queries may be necessary, which can result in significant delays when objects move from one set of cells to another. Lefebvre and Hoppe [2006] outline a perfect and minimal hashing scheme and Teschner et al [2003] use spatial hashing for deformable bodies that can self-collide.

Hierarchical Spatial Hashing This scheme [Mirtich, 1996, 1998a; Eitz and Lixu, 2007] combines hierarchical grids and spatial hashing, to accommodate objects of varying size in an environment whose size can change. Like spatial hashing, there can be significant delays when objects move from one set of cells to another. Moreover, it is difficult to determine how cells at different levels correspond spatially without performing multiple expensive hash table queries.

Hierarchy of Uniform Grids (HUG) The four previous structures work best when objects are uniformly distributed throughout the environment and when one can afford to subdivide the environment finely enough to have an optimal number of objects per cell, even in very densely populated areas. The recursive grid or adaptive voxel subdivision, proposed by Jevans and Wyvill [1989], attempts to overcome this restriction. In this scheme, spatial hashing is initially used.

Afterwards, each cell with more than a specific quantity of objects is hashed and the process continues to create a per-cell hierarchical spatial hashing structure. This is often more efficient than traditional hierarchical spatial hashing, as the bottom-level grid does not span the entire environment, requiring less memory for a dense grid representation. This scheme was subsequently adapted to create the hierarchy of uniform grids (HUG) [Cazals et al, 1995; Cazals and Puech, 1997] or the adaptive grid [Klimaszewski and Sederburg, 1997]. Both structures are built by filtering the objects according to size, applying a clustering algorithm to objects of the same size and finally building the HUG. HUGs are essentially recursive grids where superfluous intermediate levels are removed, along with the need to reference low-level objects at higher levels of the hierarchy. Moreover, the lower-level grids do not necessarily align with those at higher levels, allowing each level to be smaller than in the recursive grid.

Octree This is a hierarchical structure that, starting with a top-level cell enclosing the entire environment, halves each cell along its x, y and z axes to create up to eight child cells. The major advantage of octrees is that they naturally adapt to the complexity of a scene, by deepening the appropriate branches of the tree when necessary, while simultaneously providing a uniform spatial representation. However, hierarchies of large depth and numerous nodes are often created, objects frequently appear in multiple adjacent nodes and hierarchy traversal is often time-consuming. Optimisations to the basic structure include loose octrees [Ulrich, 2000], where cells are expanded to minimise the problem of objects straddling cells, and linear octrees [Frisken and Perry, 2002], where nodes are stored in a 1D array using an octree-specific hash function. In 2D, an octree is known as a quadtree.

***k*-d Tree** A generalisation of the octree is the k-d tree [Bentley, 1975; Friedman et al, 1977]. Instead of simultaneously dividing along three axes, the k-d tree divides along any internal axis-aligned plane at each iteration, where k denotes the number of axes that can be divided. By considering only a single axis at each iteration, the algorithm for processing the tree is simpler than for the octree, leading to faster traversal. Despite this, k-d trees are rarely used for collision detection and are more commonly used in ray-tracing [Wald and Havran, 2006; Shevtsov et al, 2007].

Binary Space Partitioning (BSP) Tree This tree generalises the k-d tree by removing the requirement that only axis-aligned planes are used for division, enabling the tree to enclose objects more tightly. Binary space partitioning (BSP) trees can be used for both the broad and narrow phases, as they can enclose the objects for the broad phase and can then be divided more finely to enclose the polygons for the narrow phase. Tree construction and modification are expensive so the trees are typically used to hold only static background geometry. However, Luque et al [2005] have successfully managed to create a semi-adjusting variant to handle moving objects.

Hybrid It is possible to combine some of the aforementioned schemes to create new hybrid schemes. For example, an environment could first be partitioned using a uniform grid and each cell could then be partitioned using octrees. Little research has been conducted into such hybrid schemes.

A survey of a multitude of spatial partitioning schemes is provided by Samet [1989]. In general, single-level grids have computational complexities of $O(n)$ for their construction and updating phases, while trees and hierarchical grids have complexities of $O(n \log n)$ for construction and of $O(n)$ for updating. In practice, the execution time of the different algorithms vary greatly and tend to depend on the behaviour of the simulation, although all are typically less efficient than SAP.

Finally, there exists a variety of rarely used algorithms. Six and Wood [1982] describe an output-sensitive algorithm that uses a space-sweeping technique with AABBs, where a plane orthogonal to a coordinate axis is swept from minus infinity to plus infinity, iterating over the coordinates corresponding to the extremes of the AABBs. Intersections of AABBs cut by the plane are stored in a segment tree and a range tree [Preparata and Shamos, 1985, pp 13–15, 81–85], allowing for fast insertions, deletions and queries. Consequently, the algorithm achieves a subquadratic time bound. Hopcroft et al [1983] describe a similar technique for spheres. An alternative approach, proposed by van den Bergen [2004, pp 215–217], applies the 3D digital differential analyser (3D-DDA) ray-casting algorithm [Amanatides and Woo, 1987] to a nonuniform grid whose subdivision is determined by AABB edges. The algorithm outlined by Vaněček Jr [1994] culls back-faces, since objects moving apart cannot collide, thereby effectively halving the number of collisions tests performed.

All of these algorithms only perform static collision detection by default. Static collision detection checks for collisions between objects at discrete intervals, essentially reducing their motion to a set of discrete positions and orientations. This can lead to tunnelling, where objects appear to pass through one another when a collision is missed due to its occurrence within an interval. To avoid this problem, dynamic collision detection or continuous collision detection (CCD) can be used. This considers the full motion of an object between the beginning and end of an interval. It is typically performed by constructing a swept volume that extends over the entire motion of an object during an interval. The collision test is performed on the swept volumes instead of the traditional BVs. Such tests are complex, so the object motion during an interval is typically assumed to be linear, thereby reducing these swept volumes to elongated boxes. These elongated boxes are referred to as either speedboxes or space-time bounds and are used by almost all commercial collision detection systems. Speedboxes can be constructed on a per-object or per-pair basis. In the case of a per-pair speedbox, only the relative motion of the pair of objects is considered, so that one object is essentially static. Despite offering improved accuracy in the majority of cases, they fail to accurately enclose objects undergoing non-linear motion.

An alternative to speedboxes are Hubbard's space-time bounds [Hubbard, 1993a,b]. These are 4D structures that provide a conservative estimate of where an object may be located in the future. To construct the structures, an object's position and velocity are used to create a 3D shape called a parabolic horn. To simplify the collision detection test, this parabolic horn is enclosed within a simpler structure called a hypertrapezoid. The hypertrapezoid is then transformed into a Hubbard space-time bound by combining it with a cutting plane formed by the object's acceleration. Similar but less sophisticated 4D structures are also used by Foisy et al [1990]. The complexity of these schemes means they are rarely used in practice.

In all broad phase algorithms, colliding objects are often penetrating each other by a non-trivial distance when a collision is detected. Applying a collision response to penetrating objects can result in irregular behaviour, so it is beneficial to correct the positions of the objects using one of a variety of algorithms. The first option, retroactive detection (RD) [Baraff, 1995], advances the simulation by the standard timestep but rolls back the simulation using a root finding method once a collision is detected. All of the objects are then rolled back to their locations at the time of the collision and the simulation is advanced as before. An alternative is conservative advancement (CA) [Snyder et al, 1993], which maintains a sorted queue of lower bounds on the times of predicted collisions. The simulation moves forward by the time at the top of the queue until the entire simulation timestep has been performed. A dynamic version of CA has been proposed by Tang, Kim and Manocha [2009]. Timewarp [Mirtich, 2000] attempts to combine the benefits of both RD and CA. This algorithm advances all of the objects independently using small timesteps until a collision occurs. At this point, all objects are rolled back to the point of collision and the simulation proceeds as before. Since each object advances independently and does not wait on the progress of other objects, performance improvements can be realised.

2.1.2 Narrow Phase

Most narrow phase collision detection algorithms use bounding volume hierarchies (BVHs), which are hierarchical data structures whose nodes consist of BVs. The root BV corresponds to all the primitives describing an object's geometry, while each child corresponds to some subset of these primitives. The leaf node BVs typically store a single polyhedron, although some variants store multiple polyhedra. Examples of BVHs include sphere trees [Quinlan, 1994; Hubbard, 1996], AABB trees [van den Bergen, 1997; Zachmann, 2000], OBB trees [Gottschalk et al, 1996] and k-DOP trees [Held et al, 1996; Zikan and Konečný, 1997]. One noteworthy BVH type is the SSV tree [Larsen et al, 1999, 2000], which combines three types of SSV in a single hierarchy. This is permissible as the collision test for the three objects can be described as a single algorithm with early exits for SSPs and SSLs. For all BVH types, collision queries are executed by recursively traversing two BVHs in tandem. Each step tests whether the current two BVs collide. If they do not, the recursion branch is terminated; if they do, the BVs' children are tested for collision. This process is continued until the possibility of collision is either proved or disproved.

Despite the advantages of BVHs, some problems remain. For example, Dingliana and O'Sullivan [2000] find that it may not be possible to traverse BVHs to their lowest levels within the frame rate constraints imposed by interactive applications. Their solution is to apply a priority to outstanding BVs and only check as many BVs as the time available permits. After this time has elapsed, any pairs that have not been pruned are assumed to be colliding. This scheme works based on the findings of Barzel et al [1996], which indicate that it is not possible to distinguish between physically correct and physically plausible behaviour, and of Uno and Slater [1997], which indicate that the perceived quality of most interactive applications is dependent not on exact simulation but on real-time response. Gottschalk [2000] finds that recursively traversing BVHs is often suboptimal since the number of nodes is often very large and memory stack overflows can result. Iterative traversal coupled with a FIFO queue is proposed as a solution.

As previously stated, polyhedra describing object geometry are typically stored in the leaf nodes of BVHs and are processed as a final collision test. The algorithms for this final collision test are often restricted to convex polyhedra to minimise runtime. Collisions between convex polyhedra can be detected using the separating axis test (SAT), which looks for an axis separating two polyhedra, as in Chung and Wang [1996]. Alternatively, collision detection algorithms can rely on the theorem stating that minimum separation distance between two objects is the same as the minimum distance from the origin of the Minkowski sums of a and $-b$ to the surface of the sums. One such algorithm that exploits this theorem is Gilbert-Johnson-Keerthi (GJK) [Gilbert et al, 1988; Gilbert and Foo, 1990], which has an average computational complexity of $O(n)$. It uses randomly selected pairs of vertices from each object to define simplices within each polyhedron, along with a corresponding simplex within the Minkowski sum. Local optimisation is used to refine the simplices until they find the closest point on the Minkowski sum from its origin. Optimisations include those by Cameron [1997], who uses hill climbing from the set of witness points found during the previous iteration, and incremental separating axis GJK (ISA-GJK) [van den Bergen, 1999], which exploits caching and coherence. Another alternative is to track features using geometric locality. Lin-Canny (LC) [Lin and Canny, 1991] uses this technique by calculating the polyhedra's Voronoi regions and updating each pairs' closest features at every cycle, for a near-constant execution time. Voronoi Clip (V-Clip) [Mirtich, 1998b] improves LC's robustness by avoiding geometric degeneracies. Guibas et al [1999] combine LC and Dobkin-Kirkpatrick (DK) hierarchies [Dobkin and Kirkpatrick, 1990], while Ehmann and Lin [2000] design multi-level Voronoi Marching using an error-bounded level of detail (LOD) hierarchy for progressive refinement. In addition, there has recently been interest in kinetic data structures (KDSes), which keep track of closest features for moving polyhedra, while exploiting coherence and geometric locality [Basch et al, 2004; Agarwal et al, 2008]. Performance is separation sensitive and object movement is restricted to linear motion, translation along an algebraic trajectory or algebraic rigid motion.

Examining the complete narrow phase, the cost formula [He, 1999] for determining expected performance is

$$T = N_V C_V + N_P C_P + N_U C_U + C_O,$$

where T is the total cost of testing a BVH pair, N_V is the number of BV pairs to test, C_V is the cost of testing a BV pair, N_P is the number of primitive pairs to test, C_P is the cost of testing a primitive pair, N_U is the number of nodes to update, C_U is the cost of updating a node and C_O is the cost of one-time processing. Primitive pairs refer to the polyhedra stored in leaf nodes. The cost formula highlights that the tradeoff between execution speed and complexity that affects the broad phase also affects the narrow phase, making it challenging to design an efficient BVH. In particular, N_V and N_P are minimised by using a complex BV, while C_V and C_P tend to increase with these complex BVs.

Distance fields provide an alternative to BVHs and have been successfully used in commercial computer games. These are scalar fields specifying the minimum distance to an object, where the distance may be signed to distinguish between the inside and outside of the object. The basic concept can be significantly optimised by storing the distances within octrees and only creating octree cells for those distances than cannot be sufficiently approximated through bilinear interpolation, thereby creating a

structure known as an adaptively sampled distance field (ADF) [Frisken et al, 2000]. Distance-field–based collision detection typically stores a set of sample points on the objects' surfaces in addition to their distance fields [Guendelman et al, 2003; Erleben, 2005]. When a pair of objects collides, the points of one object are tested against the distance field of the other. A ray can be traced in the direction of the gradient to find the exact contact point and normal [Guendelman et al, 2003]. Alternatively, if a point is detected to be inside the other object according to the distance field sign, it can become the contact point, while the gradient of the distance field can become the contact normal [Erleben, 2005]. A thorough review of the topic can be found in Jones et al [2006].

Recently, there has been significant research interest in the area of deformable bodies. Deformable bodies are vital for the accurate simulation of cloth [Govindaraju, Knott et al, 2005; Selle et al, 2009] or surgery [Guy and Debunne, 2004; Raghupathi et al, 2004] but this deformation introduces additional problems for narrow phase collision detection. In particular, the possibility of self-collision now arises as different parts of an object can collide with each other. Rigid body algorithms ignore this possibility and a straightforward adaptation of these could result in undetected collisions. Deformable bodies also mean that less of the collision detection algorithm can be performed as a pre-processing step.

As for rigid body collision detection, the majority of deformable algorithms exploit BVHs. These BVHs must be rebuilt or updated between cycles to ensure they remain accurate. To increase the performance to interactive rates, the BVH is typically constructed during a pre-processing step as before and only the BV extents are updated between cycles. This typically leads to a less tightly fitting BVH, which results in more false collisions. Van den Bergen [1997] determined that refitting a BVH in this way is approximately ten times faster than rebuilding, even when considering the resultant increased collisions, providing object topology is preserved.

Efficient BVH refitting algorithms exist for sphere trees [Brown et al, 2001] and AABB trees [van den Bergen, 2001; Larsson and Akenine-Möller, 2001]. Of these, AABB trees are widely considered the optimal BVH type, due to the rapid execution times of their refitting operations. Another issue for deformable collision detection is the arity of the BVHs. Rigid bodies typically use binary trees, but Larsson and Akenine-Möller [2001] and Mezger et al [2003] find that quaternary and octary trees offer better performance for deformable bodies. A final important factor concerns the construction and update of the hierarchy. Larsson and Akenine-Möller [2001] determined that a bottom-up strategy works better when the algorithm reaches many deep nodes, but that a top-down strategy works better otherwise. To utilise these findings, the authors propose a hybrid strategy that constructs the top half of the tree using a bottom-up strategy and constructs the bottom half using a top-down strategy when required. This hybrid strategy is more efficient as many nodes are only updated when accessed. This comes at the expense of higher memory requirements, as the BVHs must record whether a node has been updated and the leaf primitives must be replicated within the hierarchy. An alternative strategy is proposed by Mezger et al [2003], where the BVs are inflated instead of refitted. This inflation continues for several cycles until the objects move a distance greater than that of the inflation.

Stochastic or inexact methods adapt BVHs, with algorithms similar to those of Dingliana and O'Sullivan [2000]. Mendoza and O'Sullivan [2006] outline a direct adaptation of the earlier algorithm,

while Klein and Zachmann [2003] design a variant that determines the likelihood of collisions using probabilities. By using these probabilities, no polyhedra need to be stored in the BVH leaves as none will be tested for collision. This algorithm's major advantages are that the runtime can be controlled by specifying the desired accuracy and BVH traversal is guided by the probabilities, resulting in faster convergence.

Distance fields provide an alternative to BVH-based dynamic collision detection. For example, Fuhrmann et al [2005] use distance fields to detect collisions between rigid objects and highly deformable objects such as cloth. A technique radically different to both BVHs and distance fields is presented by Teschner et al [2003], where the environment is checked for collisions using spatial hashing. Similarly, Zhang and Yuen [2000] use a uniform grid for clothed human animation, while Jin et al [2009] also use a uniform grid to accelerate simulations involving uniformly distributed large objects undergoing deformation.

Dynamic collision detection techniques are considered challenging to design for deformable bodies, due to the possibility of self-collision. Successful designs have been proposed by Redon et al [2005], Curtis et al [2008] and Tang, Manocha and Tong [2009]. Tang, Manocha and Tong [2009] also leverage the multiple cores present in modern CPUs to reduce the impact of having to perform a multitude of complex collision tests.

Comprehensive surveys of collision detection algorithms can be found in Lin and Gottschalk [1998], Jiménez et al [2001], Lin and Manocha [2004], van den Bergen [2004], Fares and Hamam [2005], Ericson [2005], Kockara et al [2007] and Avril et al [2009]. Teschner et al [2005] survey deformable collision detection.

2.1.3 Graphics Processing Units (GPUs)

Since their introduction, GPUs have evolved from their fixed-function rasterisation origins into programmable graphics pipelines. This has been reflected in the development of high-level programming languages such as Compute Unified Device Architecture (CUDA) [NVIDIA, 2009c] and Open Computing Language (OpenCL) [Munshi, 2008]. In addition, their computational power has increased at a rate superseding Moore's law [Owens et al, 2007]. Consequently, they have been targeted for the implementation of a wide variety of computationally demanding tasks including collision detection.

When combined with CPU software, a GPU forms a hybrid system similar to the software and hardware microarchitecture system outlined in this thesis. In addition, one of the defining characteristics of GPUs is their high degree of parallelism, which is a characteristic they share with microarchitectures. Although GPUs and microarchitectures are significantly different due to the general-purpose nature of GPUs, these similarities mean it is beneficial to review some GPU-based collision detection systems before considering a system utilising a microarchitecture.

In terms of broad phase algorithms, the most prominent design is that of the CULLIDE family of systems. For a simulation with n objects, the original design [Govindaraju et al, 2003] writes every object to the GPU depth buffer n times but switches the last object to be written on each iteration. If this last object is fully visible, it is non-colliding, but if the last object is partially occluded, it must be

tested by the CPU-based narrow phase for collision. The algorithm's execution time varies with the depth-complexity of the scene. Quick-CULLIDE [Govindaraju, Knott et al, 2005; Govindaraju, Lin and Manocha, 2005] builds on this using chromatic decomposition to detect self-collisions between non-adjacent primitives. Govindaraju et al [2007] add support for dynamic collision detection with deformable bodies using swept BVs. An alternative design is proposed by Le Grand [2007], who uses a broad phase based on a uniform grid. The design is very similar to those traditionally implemented on CPUs, highlighting the similarities between current CPUs and GPUs. Despite this, the author achieves an acceleration of approximately 26× over the equivalent algorithm executing on a CPU. GPU-based spatial subdivision algorithms have also been designed for uniform grids [Purcell et al, 2002; Purcell, 2004], quadtrees and octrees [Lefohn et al, 2006] and k-d trees [Foley and Sugerman, 2005; Horn et al, 2007], although none of this research focuses specifically on collision detection. Coombe et al [2004] and Lefebvre et al [2005] consider the dynamic update of quadtrees and octrees respectively.

In terms of GPU-based narrow phase algorithms, the earliest approaches tended to focus on the object-object test. The algorithms proposed by Myszkowski et al [1995] and Baciu et al [1998] write the two objects to be tested to the stencil buffer. The pixels are considered equivalent to a ray-cast towards the objects, so that by noting where the rays enter and exit the objects, it is possible to determine whether they are colliding. For convex objects, the algorithms are relatively accurate, but for concave objects, the determination of a collision does not necessarily mean a collision has occurred. Knott and Pai [2003] solve this problem by adapting the algorithm to determine whether a ray passes through the objects an odd or even number of times. Objects are colliding when an odd number of passes is detected. An alternative algorithm, capable of processing deformable bodies, is proposed by Heidelberger et al [2003]. This algorithm first calculates the intersection of the AABBs enclosing the objects, before computing layered depth image (LDI) representations of each object within the intersections. The algorithm finally deduces the colliding objects using volumetric collision queries. Heidelberger et al [2004] expand the algorithm to support self-collision. A unified GPU framework for proximity queries including dynamic collision detection between deformable bodies is proposed by Sud et al [2006]. Finally, Rueda and Ortega [2008] describe the design of GPU-based geometric algorithms, which are directly relevant to collision detection.

BVHs can also be implemented on a GPU to create an alternative CPU-style narrow phase. One such example, by Greß and Zachmann [2003], uses an AABB tree that leverages GPU parallelism using breadth-first traversal. However, texture size restrictions limit model complexity and the traversal algorithm has an overhead of $\log n$ above that of a traditional software implementation. Horn [2005] eases the restrictions on model complexity and Greß et al [2006] further adapt the design to support deformable models represented as non-uniform rational B-splines (NURBS). The hybrid parallel CCD (HPCCD) system [Kim, Heo et al, 2009] focuses explicitly on the concept of a hybrid system comprising a multi-core CPU and multiple GPUs. The CPU software constructs AABB trees, while the GPUs perform the object-object tests. GPU-based AABB trees have also been explored in the context of ray-tracing [Thrane and Simonsen, 2005; Günther et al, 2007]. A novel BVH construction algorithm for both collision detection and ray-tracing is outlined by Lauterbach et al [2009].

Collision detection also features in GPU-based physics engines. Kipfer et al [2004] simulate particle systems and consider inter-particle collisions, with the GPU sorting particles into a PCS. Kolb et al [2004] present a related particle system that supports collisions between particles and scene geometry based on depth comparisons. Zeller [2005] models mass-spring dynamics to simulate cloth. The system includes collision detection supporting deformable bodies and self-collisions. Havok FX was a commercially available physics engine that implemented GPU-based physics and collision detection for both rigid bodies and particles. The engine supported single and multiple GPUs. A survey of GPU research, encompassing architectures, programming techniques and algorithms, is presented in Owens et al [2007].

Overall, GPU techniques have traditionally suffered from geometric errors introduced by the discretisation to image-space resolution [Jang et al, 2007]. However, this is often unproblematic for many applications that do not require a high degree of realism. Nevertheless, it is often possible to avoid these geometric errors, although this avoidance can result in the screen resolution having a significant effect on algorithm performance [Greß et al, 2006]. Another problem encountered by GPU-based collision detection algorithms is the delay incurred during buffer readbacks. These readback delays are a result of pipeline stalls and GPU designs that favour writing over reading. In addition, some algorithms require multiple readbacks during each iteration, resulting in additional delays, but it is often possible to minimise or eliminate these readbacks to improve performance [Zhang et al, 2008]. Due to a variety of architectural developments, both the resolution and readback problems are of lesser consequence in modern GPUs and can be effectively disregarded. However, some challenges remain. Firstly, it may not be possible to use the GPU-computed data directly in the CPU software without expensive conversions. In addition, the algorithm designer is limited to the GPU vendor's design and must work within these constraints. A custom microarchitecture, in contrast, allows an algorithm designer to layout all necessary components in the configuration best suited to the algorithm under consideration. Therefore, a custom microarchitecture unlocks the potential of achieving a greater acceleration than a GPU for a given algorithm.

All of this research naturally targets current GPU designs and, despite their move towards greater flexibility, their designs have remained relatively unchanged over the years. Some researchers have determined that these designs limit performance and that it would be desirable to create a radically different design. One example is Larrabee [Seiler et al, 2008], proposed by Intel. This uses a multitude of Pentium cores adapted to include a wide vector-processing unit and some fixed-function logic that performs rasterisation operations. In addition, out-of-order execution has been disabled to achieve higher performance per watt and per unit of area when executing highly parallel algorithms. The design greatly improves GPU flexibility and programmability, although it comes at the expense of overall performance. Despite this, the increased flexibility may enable some algorithms to execute more quickly than on an equivalent GPU. The authors also briefly discuss the implementation of the SAP and GJK collision detection algorithms on Larrabee. A more radical alternative is the Ray Processing Unit (RPU) [Woop et al, 2005]. This generates graphics through ray-tracing instead of rasterisation and contains programmable shaders similar to those found in GPUs. Another GPU-style design is the

Cell Broadband Engine [Chen, Raghavan et al, 2007], developed by Sony Computer Entertainment, Toshiba and IBM. Although not a GPU, it shares many characteristics with these platforms as it is targeted towards the implementation of graphics algorithms. It contains a single power processing element (PPE), similar to existing PowerPC processors, which manages the execution of eight synergistic processing elements (SPEs). These SPEs contain vector-processing units that can operate on both integers and floating-point numbers. Very high floating-point performance is achievable but this comes at the expense of an extremely complex programming model where caches, memory transfers and the flow of data through the processor must be explicitly managed. The Cell Broadband Engine is used as the PlayStation 3 processor, although its variant disables one SPE to increase yield, and another SPE is dedicated to the operating system (OS) and is inaccessible to developers. Although these designs provide certain advantages over current GPUs, they all lack the key feature of custom microarchitectures, which is that an algorithm designer completely determines the layout of the components implementing that algorithm. For this reason, it should still be possible to gain a greater acceleration with a custom microarchitecture than with any of these designs.

2.2 Hybrid Systems

Hybrid software and hardware systems have been successfully used to accelerate many algorithm types. This section focuses on the microarchitecture component of such systems, as a microarchitecture forms the major contribution of the system described in this thesis. After outlining research into the acceleration of algorithms using microarchitectures, a variety of relevant systems, including collision detection, physics simulation and N-body simulations, are discussed.

2.2.1 Microarchitectures

A number of researchers have investigated the reasons microarchitectures can achieve greater accelerations than software. Benitez [2002] analyses microarchitecture-based coprocessors and determines that performance is primarily influenced by three metrics: hardware capacity, memory and system bus bandwidth. In addition, Benitez suggests that coprocessors are most efficient when placed as close to the CPU as possible. Guo et al [2004] perform a similar analysis but concentrate on the performance of the algorithm executing on the microarchitecture, instead of its interaction with the other elements of the system. They determine that microarchitectures are best suited to computationally intensive algorithms and that control-flow intensive algorithms should remain in software. They also determine that four metrics primarily influence algorithm execution speed:

Parallelism This should be maximised within the constraints of available logic resources and obtainable memory bandwidth.

Instruction Efficiency Many operations, which would comprise multiple CPU instructions, can be executed simultaneously in a microarchitecture. For example, an operation inside a loop and

the update of a loop counter could be performed concurrently. Efficiency is further maximised by separating the control from the datapath and executing the two simultaneously.

Latency Hiding Unavoidable latencies, such as those associated with I/O requests, can often be hidden by effectively exploiting features such as pipelining.

Memory Layout Microarchitecture memory designs are effectively pliable and can be reshaped to accommodate the needs of an algorithm.

After implementing the microarchitecture on a suitable hardware platform, it is necessary to analyse its performance. Such analyses are routinely performed as part of almost all microarchitecture research. The vast majority of these quantify the execution time of the microarchitecture alone. While this is an interesting and useful metric, it is also beneficial to examine the acceleration achieved when the data transfer costs present in most hybrid systems are considered. Tripp et al [2005] perform an analysis of these data transfer costs and claim that the overall acceleration of a hybrid system is determined by Amdahl's law [Amdahl, 1967]. A similar approach is taken by Rousseaux et al [2007], who implement the basic linear algebra subprograms (BLAS) matrix multiplication 'dgemm' operation on a microarchitecture. Their timing analyses consider the global runtime of the algorithm at the software level, including data preprocessing, transferral and postprocessing. They additionally compute a theoretical performance and demonstrate how this theoretical figure can never be achieved due to issues surrounding microarchitecture design complexity, data transfer latencies and background software processes. They ultimately conclude that the primary limiting factors are sources external to the microarchitecture.

Che et al [2008] analyse the applicability of CPUs, GPUs and microarchitectures to the acceleration of computationally intensive algorithms. Their results indicate that microarchitectures provide the best overall performance in their test cases, although they criticise the difficulty of microarchitecture programming models. They also determine that microarchitectures work best in applications demanding many detailed low-level hardware control operations that cannot be efficiently implemented in a high-level language. In contrast, GPUs are found to work best for applications with no interdependencies in the dataflow.

2.2.2 Collision Detection

The foremost collision detection microarchitecture is the CollisionChip, which focuses exclusively on narrow phase collision detection. The initial design [Zachmann and Knittel, 2003] targets an ASIC and uses a microarchitecture based on 24-DOP binary trees storing triangles in the leaf nodes. A novel algorithm, designed to reduce memory accesses and node transformations, traverses a single tree that combines the trees of the two objects being tested for collision. The k-DOPs and triangles are tested for collision using a specialised SAT. Knittel and Zachmann [2003] adapt the microarchitecture to use 24-bit floating-point instead of single precision floating-point. Raabe et al [2005] adjust the configuration of the external memory banks and target an alternative ASIC architecture. However, this design

requires a 756-bit connection between the ASIC and the memories, which would necessitate the design of a custom printed circuit board (PCB) and would be prohibitively expensive in practice. Hochgürtel et al [2005] adapt the microarchitecture to use fixed-point arithmetic. The authors design a custom rounding scheme that results in a larger k-DOP as naïve rounding of floating-point to fixed-point would result in false negatives. The smaller scale of fixed-point numbers is overcome by normalising the numbers relative to the size of the largest object. Using this normalisation scheme, 16-bits permit k-DOPs the size of a skyscraper and a 6 mm screw and 36-bits permit k-DOPs the size of the sun and a 6 mm screw.

After porting the CollisionChip microarchitecture to a Xilinx Virtex-II XC2V6000 FPGA of speed grade four [Xilinx, 2007b], hosted on an Alpha Data ADM-XRC-II [Alpha Data, 2002b], an acceleration of 4× was achieved [Raabe et al, 2006a,b]. The authors hypothesise that this figure could be increased by implementing multiple CollisionChip pipelines in parallel, but doing so leads to memory bandwidth becoming a limiting factor. To compensate for this, Raabe and Zavelberg [2008] implement a custom caching scheme to achieve an acceleration of 10×. Additionally, Raabe et al [2008] analyse the use of FPGAs' dynamic reconfigurability. Using this reconfigurability, the intersection test can be switched between the standard triangle-triangle test and a new quadrangle-quadrangle test. No performance results are published but the authors have determined that the reconfiguration incurs a penalty of only 0.1%. Although this adaptation adds flexibility to the microarchitecture, it is unlikely to be of much practical utility as the meshes comprising objects in the vast majority of applications are described solely as triangles. Finally, Raabe et al [2009] work around some of the problems imposed by fixed-point, by returning a flag denoting whether a collision is guaranteed or whether the result is potentially incorrect. The quantity of potentially incorrect results is reduced by converting each triangle to a fixed-point representation on the FPGA and normalising against the largest triangle from each pair tested. Therefore, if two small triangles are tested, the discretisation grid imposed by the fixed-point representation should not result in numerical problems. Depending on the application, an exact collision test could be performed in software for results denoted as being potentially incorrect. Additional details on all of these FPGA-based collision detection microarchitectures are outlined in Raabe [2008].

Overall, the CollisionChip microarchitectures use a novel algorithm, adapted to microarchitecture implementation, to achieve good performance. However, all designs are limited to detecting collisions between two instances of the same high-polygon rigid body model located in static positions, which is extremely limiting and unsuitable for the majority of applications. This is unlikely to be a major problem as it is conceivable that this limitation could be easily overcome. Another disadvantage of the system is that although 24-DOP trees have been successfully used in commercial collision detection systems to model interactive applications' static objects, such as environment geometry, it is doubtful that this BVH would achieve sufficient performance for the dynamic objects of typical interactive applications, which tend to have relatively low polygon counts. Finally, all provided results fail to quantify the execution time of tree construction in software along with the communication delay between the software and the microarchitecture.

An alternative FPGA-based collision detection microarchitecture is outlined by Atay et al [2005a,b]. This is based on the fast triangle-triangle intersection test designed by Möller [1997] and uses a 32-bit fixed-point number representation. Since the geometry comprising the majority of interactive applications are described solely as triangles, this microarchitecture can achieve results accurate to the image rendered by the GPU. When simulated for a Xilinx Virtex-4 XC4VLX200 FPGA [Xilinx, 2007a], the microarchitecture achieves an acceleration of approximately 36×. Despite this acceleration, the complexity of the triangle-triangle intersection test means that it fails to achieve interactive performance for many applications. However, it is sufficiently fast for the types of robot motion planning targeted by this microarchitecture [Atay and Bayazit, 2006]. As for the CollisionChip microarchitectures, the results fail to quantify the communication delay between the software and microarchitecture. Moreover, the research uses an RS-232 communication link for transferring data, which is likely to adversely affect the performance of any practical application utilising the microarchitecture.

Finally, Wu et al [2009] describe a departure from the standard models of collision detection and focus on a microarchitecture for detecting collisions using the affine interior-point algorithm [Akgunduz et al, 2005]. This algorithm is used to solve linear programming (LP) formulations and its primary advantage is that it can detect inter-frame collisions without multiple timesteps. The microarchitecture is implemented on a Cyclone II EP2C35F672C6 FPGA [Altera, 2008], hosted on an DE2 board [Altera, 2006]. Real-time performance is achieved, although the tests were limited to detecting collisions between two pentagonal prisms and did not consider communication.

All of these microarchitectures only perform narrow phase collision detection and do not consider the broad phase. Although a narrow phase microarchitecture could offer significant accelerations, the preceding broad phase could become a considerable limiting factor. Therefore, the broad phase microarchitecture outlined in this thesis fills a niche as it could either synergistically complement these microarchitectures or operate independently.

2.2.3 Physics Simulation

Physics simulation is directly related to collision detection, as every physics engine must comprise collision detection to realistically simulate the behaviour of objects. For this reason, it is beneficial to examine some physics simulation microarchitectures and hardware designs, with a focus on their collision detection.

The AGEIA PhysX was a commercially available ASIC physics processing unit (PPU), which was essentially a general-purpose processor and not a microarchitecture. Details have not been published in academic fora, although nine patents shed some light on possible internal details. The main patent [Davis et al, 2005b] describes a system on chip (SoC), that comprises a multitude of processing cells containing a PPU control engine (PCE), data movement engine (DME) and floating-point engine (FPE). The PCEs are reduced instruction set computer (RISC) microprocessors, which primarily control the operation of the DMEs and FPEs but can also perform some general-purpose operations. The DMEs are responsible for moving data between the CPU and PPU in an efficient manner and they use ultra-threading to allow near 100% utilisation through the prevention of stalls when loading data.

The single-precision FPEs are hybrid vector/very long instruction word (VLIW) processors containing scalar integer units (SIUs), scalar floating-point units (SFUs) and vector floating-point units (VFUs) to perform operations on a variety of operand types. The patent also describes a simpler design where the connections between components in the system are less complex and there is less internal communication. In this design, the FPEs become more like general-purpose processors by including registers and arithmetic logic units (ALUs) in addition to the previously described elements. In both designs, myriad processing cells are combined to form the overall design. Each cell is assigned a specific function, such as rigid body collision detection, particle collision detection, explicit ordinary differential equation (ODE) solving or implicit ODE solving. Other patents provide information on a method for communicating data between the CPU and PPU [Davis et al, 2005a], on integrating the PPU into a conventional computer [Davis et al, 2005c], on the design of a parallel linear complementarity problem (LCP) solver [Zhang et al, 2009a,b; Tonge et al, 2005], on designing a PPU instruction set architecture (ISA) [Maher et al, 2005], on allowing asynchronous and parallel execution of a PPU [Bordes and Sequeira, 2006] and on directly connecting a PPU and GPU for optimal execution [Bordes et al, 2006]. Despite these patents, it remains unclear which parts of the collision detection system are executed on the PPU and which are executed in the associated software driver, but it is likely that both phases could be implemented on the PPU due to the multitude of vector floating-point operations in the FPEs. The patents may have covered AGEIA's plans, with a less ambitious version being manufactured. Moreover, no verifiable performance metrics have been published so it is difficult to determine the performance advantage offered by the AGEIA PhysX. Following an acquisition by NVIDIA, the ASIC is no longer available and the physics algorithms are accelerated using an equivalent GPU implementation.

Alternatively, Yeh et al [2007] thoroughly analyses a set of physics benchmarks modelled with a modified version of Open Dynamics Engine (ODE) [Smith, 2009]. These analyses demonstrate that current architectures cannot maximally accelerate physics simulation, as a mix of coarse granularity (CG) and fine granularity (FG) cores would be required. To rectify this, Yeh et al design a processor consisting of general-purpose CG and FG cores that can efficiently execute physics simulation along with many other applications. Simulations demonstrate that the design could exhibit a significant acceleration over software, although current technology does not allow such a design to be fabricated due to its considerable size. The design fails to exploit the full potential of the hardware for the broad phase as the authors merely attempt to execute algorithms optimised for software on a small number of CPU-like CG cores.

A less general approach is taken by the Simulation of Physics on a Real-Time Architecture (SPARTA) [Bishop et al, 2000b], which is a simple physics microarchitecture that focuses on deformations modelled using the mass-spring system. Simple collision detection is included. Due to size constraints, only the spring force computation pipeline is implemented on a FLEX 10K EPF10K250 FPGA [Altera, 2003], hosted on a custom PCB, with collision detection and resolution remaining in software. In a final SPARTA system, the authors claim that it would be beneficial if the software were responsible for broad phase collision detection [Bishop et al, 2000a]. Some optimisations to SPARTA are described by Bishop and Kelliher [2001]. The design is modified to use forward Euler integration

with a 16-bit floating-point number representation. It is not pipelined and performance is limited.

A second generation of SPARTA, called HELLAS, is outlined by Bishop and Kelliher [2003] and Yardi et al [2006]. This adapted microarchitecture uses an implicit solver and is sufficiently general to be used for other floating-point intensive applications. It consists of four processing elements (PEs), each comprising a functional unit, substantial static RAM (SRAM) resources, control and control memory. The microarchitecture is implemented on an ASIC and achieves a clock frequency of 227 MHz, although the authors' host breadboard limits the clock frequency to 16.1 kHz. Consequently, no performance results are provided.

The particle simulator proposed by Beeckler and Gross [2005, 2008] is a microarchitecture for physics simulation that focuses solely on particle physics. It consists of a pipeline with four primary stages: force calculation and summation; integration and updates; collision detection and response; and 3D rendering. Either an 18-bit fixed-point or floating-point representation is used. The collision detection stage can detect collisions between particles and other objects within the environment, although these are restricted to simple objects such as planes. In addition to reporting whether a collision has occurred, the collision detection stage computes an estimated point of intersection and a surface normal vector. If the collision is between a particle and a plane, the plane is also provided along with the side of the plane on which the collision occurred. However, despite the apparent completeness of the microarchitecture, collisions involving particles are much easier to detect that those involving general objects and, consequently, the microarchitecture is not of general utility.

In summation, a few of these microarchitectures and designs consider the hardware implementation of broad phase collision detection, unlike their collision detection counterparts. These tend to focus on using general-purpose cores for the broad phase. Although such designs facilitate the direct porting of software algorithms to the microarchitectures, they fail to unlock the full potential of the hardware since they do not exploit all of the hardware's benefits, such as increased parallelism or the removal of superfluous general-purpose logic. Therefore, it is beneficial to create an appropriate broad phase microarchitecture specialised for this part of the collision detection system.

2.2.4 *N*-Body Simulations

Outside collision detection and physics simulation, there exists a variety of microarchitectures with features similar to the microarchitecture outlined in this thesis. In particular, microachitectures tasked with executing the category of algorithms known as N-body simulations are extremely similar. These simulations consider the behaviour of a system comprising N objects, where all of the objects can potentially interact with one another. Applications of N-body simulations include astrophysics, molecular dynamics (MD), plasma physics, fluid mechanics and semiconductor device characterisation. Of particular interest are the ways in which these microarchitectures manage the data arriving from the CPU, as this thesis proposes a novel approach for buffering such data to maximise parallelism.

The Gravity Pipe (GRAPE) project [Makino, 2006] uses an ASIC-based microarchitecture to accelerate the force computation from N-body astrophysical simulations. The microarchitecture contains a pipeline that traverses the list of objects sequentially, storing the current object in an internal register.

After updating this current object, the list of objects is traversed a second time to apply the effect of the current object to the other objects. Six of these force calculation pipelines are placed on each ASIC and the complete system comprises 2048 ASICs. An alternative is proposed by Spurzem et al [2009], who investigate the performance of FPGAs and GPUs in the context of the smoothed-particle hydrodynamics (SPH) calculations used in astrophysical simulations. Both the FPGA and GPU designs read an object and a list of its neighbours from external memory, limiting the overall performance to the communication speed. The authors determine that GPUs provide a moderately better execution time, although they foresee FPGAs as a better solution in the longer term since a greater proportion of their resources can be effectively utilised. They also propose a hybrid system combining FPGAs and GPUs.

Azizi et al [2004] focus on MD instead of astrophysics. Their FPGA-implemented design uses fixed-point number representations. Each pair is read sequentially from an external SRAM but this results in poor performance, since other microarchitecture components write results to the same SRAM and the read and write requests must be interleaved. The researchers exploit the reconfigurability of the FPGA by allowing the user to specify precision, scaling factors and the number of particles supported. These factors generate an appropriate VHDL description, which is subsequently synthesised, translated, mapped, placed and routed for implementation on the FPGA. Another MD approach is outlined by Patel et al [2006] in a paper detailing the design of a Message Passing Interface (MPI) programming model for clusters of FPGAs.

The Hardware-Accelerated Quantum Monte Carlo (HAQMC) [Gothandaraman et al, 2009] uses the FPGAs present in Cray XD1 supercomputers to implement a microarchitecture that uses Quantum Monte Carlo (QMC) to solve the Schrödinger equation for obtaining the ground-state properties of systems comprising many objects. On each iteration, the list of objects is transferred to the microarchitecture's internal memories and processed sequentially.

All of these N-body simulations can contain an extremely large number of computations. To optimise their performance, processing compression [Fujita and Jinya, 2009] can be used to remove redundant calculations that have no effect on the results. This compression uses asynchronous data processing based on redundant variables and a prioritisation scheme. Another scheme for the acceleration of N-body simulations is the Tool for Accelerating N-Body Simulations on Reconfigurable Platforms (TANOR) [Kim, Deng et al, 2009]. Using a single MATLAB description and a variety of parameters specifying features such as the use of fixed- or floating-point representations, TANOR can create an N-body microarchitecture for implementation on an FPGA. The resultant microarchitectures read their data from the CPU sequentially. The authors demonstrate the validity of TANOR using a Gaußian kernel, astrophysics gravitation computations and force calculation in MD.

In summation, there exist a wide range of microarchitectures for performing N-body simulations, with each optimised for particular scenarios. Despite the variety of implementations, none of the designs proposes a radical method for storing or parallelising data. However, this lack of novelty may result from the size of the N-body simulations restricting the availability of microarchitecture resources.

2.3 Benchmarking Frameworks

Before reviewing benchmarking frameworks, it is useful to consider the characteristics of a sophisticated benchmark. The first characteristic is repeatability, which means that the same results should be attained whenever the benchmark is executed with a given algorithm. Repeatability allows researchers and developers to obtain results that can be independently verified by others. The second characteristic is comprehensiveness, which means that the benchmark should be sufficiently comprehensive to exercise all relevant aspects of the algorithm under consideration. Such comprehensiveness allows researchers and developers to find the conditions under which an algorithm performs poorly, facilitating the improvement of that algorithm. In addition, comprehensiveness means that the benchmark should not focus exclusively on a single collision detection or application type, and should not be restricted to scenarios that are implausible or unrepresentative of those found in practical applications. The final characteristic is that the benchmark should have achieved widespread use, so that the performance of a variety of algorithms, recorded using the benchmark, would have been published. This facilitates the comparison of additional, newer algorithms to those published and lends credibility to the benchmark. This characteristic is probably the most difficult to achieve since a benchmark that has met the first two characteristics may not meet this characteristic, as it is subject to whims of researchers and developers.

Researchers in many areas of computer graphics, and in computer science in general, use benchmarks. For example, there exist two main benchmarks in the field of ray-tracing. The first, Standard Procedural Databases (SPD) [Haines, 1987], includes models for a recursive tetrahedral pyramid, a fractal mountain with spheres, dodecahedral rings, a tree, meshed gears, a 'sphereflake' and a Utah teapot on a reflective surface. The key feature is that these are procedurally generated from a LOD parameter, thereby enabling models of differing complexities to be generated from a single basic model. The second, A Benchmark for Animated Ray Tracing (BART) [Lext et al, 2001], consists of detailed animated environments. These environments are a kitchen with a small car moving around the floor, moving robots surrounded by skyscrapers and a museum with animated abstract art. Each environment attempts to test different features of current ray-tracing methodologies. Both SPD and BART are repeatable, relatively comprehensive and have achieved widespread use. In terms of comprehensiveness, SPD provide more scope for modification but BART is focused on creating realistic scenarios, making the two benchmarks relatively equal in terms of this characteristic. Outside of ray-tracing, GPUs can be benchmarked using 3DMark Vantage [Futuremark, 2009]. This application consists of two graphics tests that create detailed environments similar to those found in computer games, two CPU tests that attempt to quantify the impact of the CPU on the overall performance of a typical computer game and six feature tests that exercise specific GPU features. 3DMark Vantage is repeatable. It is relatively comprehensive in that it tests all relevant GPU features, yet it fails to provide the ability to modify or adapt the tests. It has also achieved widespread use.

Outside the field of computer graphics, the NAS Parallel Benchmarks (NPB) [Bailey et al, 1994] quantify the performance of parallel compute clusters using eleven representative algorithm types from the field of computational fluid dynamics (CFD). These measure overall system performance as well

as the execution of specific operations such as I/O. The NPB are repeatable, relatively comprehensive and have achieved widespread use. Similarly, NetSpec [Jonkman, 1994; Lee, 1995] quantifies the performance of networks. This is extremely flexible, as instead of using a fixed set of benchmarks, a basic framework is provided. This framework can create a range of practical benchmarks using a simple programming language that describes the behaviour of the network traffic. NetSpec fulfils all three characteristics, being repeatable, comprehensive and having achieved widespread use.

The benchmarks in these fields allow developers to accurately test the performance of their designs in plausible scenarios, while facilitating comparisons between different algorithms. However, there are few benchmarks in the area of collision detection and all current attempts have significant limitations. One such attempt [Held et al, 1995] creates three benchmark scenes to test the performance of collision detection algorithms in simulations involving many small stationary objects and one large moving object. The first scene uses test objects such as chess pieces, mechanical parts, aircraft and animals, while the second scene uses tetrahedra. These objects are randomly placed by using a BSP tree to divide the environment. In the third scene, terrain data are used instead of objects. Following construction, all three scenes add one of six objects at a random feasible location with a random flight path. The object is advanced along the flight path until a collision occurs, at which point the object's path is changed using simplified physics. The benchmark is not repeatable due to the use of a random number generator. It is relatively comprehensive as it contains many options, but it is limited by its specialisation to a very small subset of collision detection applications that involve many small static objects and one large moving object. Moreover, it has failed to achieve widespread use. An alternative benchmark is outlined by Zachmann [1998] in a paper proposing dynamically aligned k-DOP trees. The benchmark provides car doors, car door locks, car exteriors and sets of pipes as models. Trenkel et al [2007] outline a development of this benchmark, which uses the Apollo 13 capsule, the Eagle Transporter from the television series *Space: 1999*, a helicopter, a chandelier, a chair, a *Star Wars* AT-ST Walker, a set of pipes, a laurel wreath, the Buddha, the *Star Trek: Deep Space Nine* space station, the Stanford Bunny, a castle, a Ferrari, a cobra, a door lock, a sphere, a grid, a sponge and a torus as models. Its flexibility is enhanced through the addition of three parameters – object type, the rotation of one object and the number of object placements to test. Both benchmarks are repeatable but not very comprehensive, since they focus on offline algorithms for static objects with very large numbers of vertices. Moreover, providing only three parameters is limiting. Finally, like the previous benchmark, they have failed to achieve widespread use. Caselli et al [2002] take an alternative approach and test a range of collision detection algorithms for their suitability in probabilistic motion planners, by attempting to plan the motion of an object through a variety of grid-like structures. The described benchmark is repeatable but not comprehensive as it is not of general utility and is restricted to a fixed set of scenarios. As with the previous algorithms, it has failed to achieve widespread use. An application for emulating a haptic device, to which benchmarks can be attached, is outlined by Cao [2006] but this is unsuitable for benchmarking non-haptic algorithm behaviour. Since this is only an emulator and not a benchmark, it is not possible to analyse it under the outlined characteristics. Due to the aforementioned problems with each of these benchmarks, they have failed to gain traction within the collision detection community.

Other researchers have focused on the benchmarking of physics engines and, as part of this research, have benchmarked the collision detection systems of these engines. The Physics Abstraction Layer (PAL) [Boeing and Bräunl, 2007] provides a unified interface to a variety of physics engines, on which a set of benchmarks has been constructed. The collision detection benchmark consists of sixty-four spheres falling under the influence of gravity into an inverted square pyramid. This benchmark is repeatable but not comprehensive as it lacks flexibility and does not work, for instance, with variable quantities of objects. It has also failed to achieve widespread use. Seugling and Rölin [2006] also benchmark physics engines but are more focused on the overall behaviour of the system than on the performance of any particular aspect. Their benchmark is repeatable but not comprehensive and it has not achieved widespread use. Banta et al [2006] take a different approach and benchmark physics from a hardware perspective. Using a gravity-driven simulation that can be adapted using an LOD parameter and which contains a cloth and a cube, the authors focus on branch prediction, instruction profiling, cache miss rates and memory footprints. Future iterations will include collision detection and an analysis of the algorithms' maximum available parallelism. As for the previous benchmark, it is repeatable but not comprehensive and it has not achieved widespread use.

Instead of using one of the previously outlined benchmarks, most collision detection developers design their own benchmarks. For example, I-COLLIDE [Cohen et al, 1995] was tested using a static scene comprising a single object type parameterised by the quantity of objects, their complexity, their linear velocity, their angular velocity and the density of objects in the scene. Van den Bergen [1997] uses three static benchmarks consisting of a single object type – a torus, teapot or *Star Wars* X-wing – initialised at random locations within an enclosing cube. Govindaraju, Knott et al [2005] take a different approach as their algorithm is designed to work with deformable bodies. They created five benchmarks. The first three model cloth of different complexities, the fourth models catheter in liver chemoembolisation and the fifth models folding curtains. All of these benchmarks created by collision detection developers are repeatable but not very comprehensive, as they focus on the types of problems that their associated algorithms are designed to solve, making them insufficient for testing the behaviour of algorithms across a wide spectrum of scenarios. In addition, although each paper has been widely cited, these benchmarks have failed to achieve widespread use. This failure can be attributed to their lack of comprehensiveness.

In summation, almost all collision detection benchmarks achieve the first characteristic of repeatability. Despite this, there exists a variety of problems with current benchmarks. Most are not sufficiently comprehensive in that they consist of a small, fixed set of benchmarks that only measure the performance of a limited selection of simulations. These cannot be adapted as technology progresses and applications become more complex. Moreover, many of these benchmarks are specialised for a small subset of possible collision detection simulations, such as those including detailed meshes or haptics. Consequently, all collision detection benchmarks have failed to achieve widespread use. To overcome these problems, this thesis proposes combining the underlying concepts of benchmarks such as NetSpec, which provides a framework that can create a wide variety of practical benchmarks, with collision detection benchmarking concepts to form a complete collision detection benchmarking framework.

2.4 Summary

In this chapter, a thorough review of the area of collision detection has been presented. Broad phase collision detection has been discussed in detail, as this phase is the most germane to the hybrid system considered in this thesis. In addition, narrow phase and GPU-based algorithms were reviewed as it is useful to be aware of their impact, if any, within such a hybrid system. As part of these reviews, deformable bodies and dynamic collision detection were briefly considered.

Following on from this collision detection review, hybrid systems were analysed in the context of general microarchitectures, collision detection, physics simulation and N-body simulations. It was determined that there exists no true broad phase collision detection microarchitecture. Some general-purpose hardware designs for physics simulation support broad phase collision detection, although these would encounter many of the disadvantages inherent in CPUs and GPUs. Moreover, it was discovered that no design offered an ambitious solution to the parallelisation of input data. Based on this, it was deduced that there was significant scope for the acceleration of broad phase collision detection using a highly parallel microarchitecture, as outlined in the following chapters of this thesis.

The chapter concluded with a review of benchmarking frameworks. This review first investigated frameworks throughout different fields of computer science, which uncovered that some frameworks were based on a single generic benchmark that could be adapted into many practical benchmarks. In contrast, a subsequent review of benchmarks in collision detection found all such benchmarks inflexible due to their reliance on scenarios that were either fixed or barely modifiable. Consequently, it was deduced that there was scope for a benchmarking framework that could be modified by means of parameters, as will also be outlined in this thesis.

Chapter 3

Design

To design is to communicate clearly by whatever means you can control or master.

—Milton Glaser, graphic designer (1929–)

Design is the planning that lays the basis for the creation of an object or system. This chapter outlines the designs for all of the thesis' systems. It commences with an evaluation of the attributes of the different algorithms involved in a collision detection system, with a view to determining the optimal split of these algorithms across software and hardware. Afterwards, two variants of the microarchitecture are described: the first is area-efficient while the second supports many more objects. Spatial partitioning is subsequently analysed as a means to overcome the microarchitectures' object limitations and to work around their undesirable computational complexities. The application of the microarchitectures to other types of collision detection, such as deformable bodies processed using dynamic collision detection, is briefly outlined before the chapter concludes with the design of the benchmarking framework.

3.1 Software and Hardware Divide

The initial stage in the design of a hybrid software and hardware system is the determination of the most appropriate platform for the different portions of the system. An investigation highlighted that broad phase algorithms generally exhibit a high degree of parallelism, while narrow phase algorithms generally exhibit a lower degree of parallelism. The structure of microarchitectures makes them amenable to the efficient execution of parallel algorithms, but they are unlikely to offer much performance improvement for algorithms with a low degree of parallelism. In addition, broad phase algorithms tend to be more compute-bound than narrow phase algorithms, which have more intensive memory requirements. Microarchitectures exploit parallelism to provide acceleration to compute-bound algorithms but parallelism tends to offer fewer advantages to memory-bound algorithms. Finally, broad

phase algorithms tend to use standard, recurring collision tests performed in a well-defined sequence. Such well-defined sequences lend themselves to microarchitecture implementation, due to the simplicity of their control logic and the facilitation of efficient pipelining and replication through the reuse of computation engines. In contrast, the narrow phase is dominated by large abstract data types (ADTs) representing the hierarchies. The complexity of traversing these would consume large quantities of hardware resources, reducing the possibility of a high degree of acceleration. These findings suggest that the best performance could be achieved by offloading the broad phase to a hardware platform implementing a custom microarchitecture, while the narrow phase could remain in software and achieve good performance.

However, this division of labour between the microarchitecture and the software would make little sense if only a small proportion of the execution time were spent in the broad phase. Therefore, it is necessary to examine the performance of the broad phase in detail. Studies conducted as part of this research show that broad phase collision detection can account for between 2.5% and 98.7% of the overall collision detection execution time, with the broad phase of an average application accounting for 40.4%. These proportions were measured using the SAP algorithm and the 'Parameters (Ratio) Experiment' and 'Collisions (Ratio) Experiment' outlined in Section 5.1. Based on the size of these percentages, it is evident that it would be beneficial to accelerate the broad phase in most cases. In addition, the worst-case scenario would be detrimental to application performance and accelerating the broad phase would be of major benefit in this case.

From these percentages, it is evident that the broad phase can have a highly variable execution time. This is undesirable as developers of interactive applications typically look for predictable behaviour, which enables them to assign a proportion of execution time to each stage of the program loop. Therefore, it is beneficial to examine some scenarios that result in lengthy broad phase execution times and, consequently, in this variability. Firstly, large quantities of objects are the most common cause of lengthy broad phase execution times. These many-object scenarios are challenging because the broad phase must consider all object pairs for potential collision. Secondly, scenarios containing objects moving rapidly and in essentially random motions are problematic, as these scenarios typically eliminate the gains made by those algorithms utilising coherence. Thirdly, a small number of very large objects that span the environment can be problematic when combined with a large number of smaller objects. Due to the size of the large objects, most of the smaller objects will need to be checked against the larger ones. Although each of these three scenarios can significantly affect the broad phase, they typically have less effect on the narrow phase, as most of the problematic pairs are eliminated from consideration before narrow phase processing. Therefore, these scenarios tend to stress the broad phase. In contrast, a scenario that results in many potential collisions being returned from the broad phase would burden the narrow phase by increasing the number of objects that must be checked for collision using lengthy narrow phase testing. In this way, the broad phase affects the narrow phase performance. The presence of such scenarios further highlights the need to accelerate the broad phase and to gain performance accelerations in those scenarios that typically perform poorly.

Based on the preceding analysis, it was determined that a broad phase microarchitecture combined

with narrow phase software was the optimal solution. In particular, the object processing part of the broad phase was selected for microarchitecture implementation, while it was considered best to keep the BV updating in software as this often uses the narrow phase BVHs, which would not be transferred to the microarchitecture. Moreover, this scheme lets the software control the overall behaviour of the system, while the microarchitecture functions as a coprocessor. This reflects the labour division typically used for such hybrid systems.

3.2 Area-Efficient Microarchitecture

There are many broad phase collision detection algorithms, of which the most widely used is incremental SAP [Cohen et al, 1995]. By exploiting coherence, this algorithm achieves excellent performance across a wide range of simulations including those commonly found in interactive applications. However, it is widely acknowledged that it is difficult to effectively parallelise the algorithm across more threads of execution than the number of coordinate axes being processed. One solution is to switch from incremental SAP to full-sort SAP [Baraff, 1992] but this reduces coherence and the execution time of the algorithm tends to increase substantially. Moreover, microarchitectures for sorting tend to perform poorly, as they either inadequately exploit parallelism [Beechick et al, 1996] or have significant performance-to-area tradeoffs [Wong et al, 2005]. For these reasons, it is evident that SAP would be a poor choice for the microarchitecture. This finding corresponds to those of Chen, Chhugani et al [2007] that the best serial algorithms can have poor parallel scalability.

In contrast, the all-pairs algorithm is embarrassingly parallel, making it an ideal candidate for microarchitecture implementation. AABBs are used as the BV, since their simplicity means storage and computational demands are minimised, enabling more collisions to be detected in parallel. Moreover, AABBs have been successfully used by the I-COLLIDE [Cohen et al, 1995] and SOLID [van den Bergen, 2007] collision detection libraries. A sequential version of the AABB all-pairs algorithm is presented in Algorithm 3.1.

The microarchitecture based on this algorithm consists of a pipeline implementing two primary operations – buffer and compare – coupled with a communication interface as illustrated in Figure 3.1. A detailed schematic is provided in Figure 3.2. As the size of the underlying hardware platform used for the implementation can vary, the microarchitecture is extensible by a factor m. Therefore, on larger hardware platforms, the microarchitecture can take full advantage of available resources and gain the maximum achievable acceleration, while it will still fit and execute efficiently on smaller platforms.

Buffer The buffer stores each AABB's data in an efficient manner for processing by the compare operation. Initially, the buffer uses the communication interface to read every AABB from external memory. These are stored in $6m$ memories corresponding to m memories for each of the minimum x, maximum x, minimum y, maximum y, minimum z and maximum z values of the AABBs. The relevant data are replicated across each set of m memories so that they are duplicated, as illustrated in Table 3.1. To achieve this data layout, the same address is specified when writing to each of the m me-

function AllPairsAABB(n, x, y, z)
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    for $j \leftarrow i + 1$ **to** n **do**
      collision $\leftarrow \left(x^i_{\max} \geq x^j_{\min}\right) \wedge \left(x^i_{\min} \leq x^j_{\max}\right)$
        $\wedge \left(y^i_{\max} \geq y^j_{\min}\right) \wedge \left(y^i_{\min} \leq y^j_{\max}\right)$
        $\wedge \left(z^i_{\max} \geq z^j_{\min}\right) \wedge \left(z^i_{\min} \leq z^j_{\max}\right)$
      if *collision* **then**
        result $\leftarrow$ *result* $\cup \langle i, j \rangle$
      end if
    end for
  end for
  return *result*
end function

Algorithm 3.1: *Sequential AABB all-pairs algorithm.*

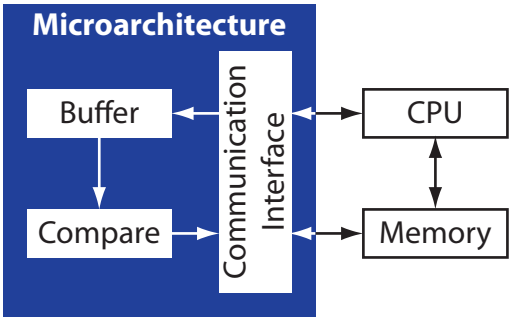


Figure 3.1: *Overview of the area-efficient microarchitecture showing the connection of the two primary operations and the communication interface.*

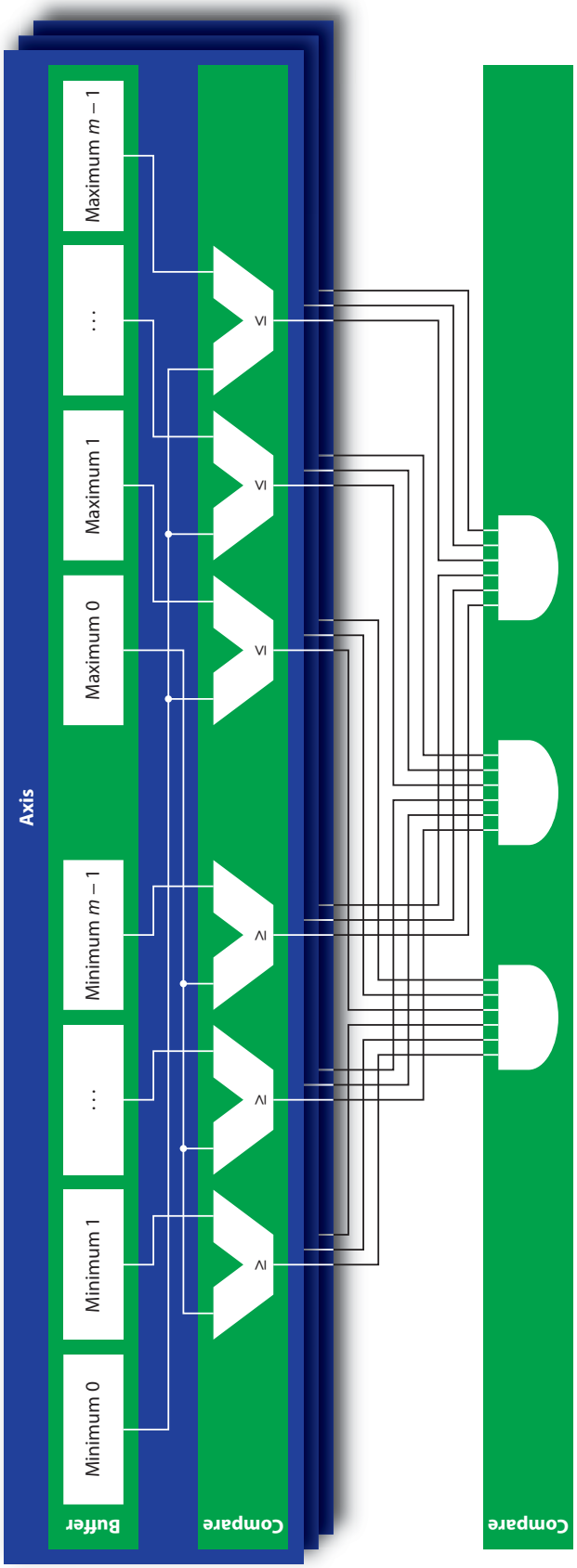


Figure 3.2: *Detailed schematic of the area-efficient microarchitecture.*

	Memory			
Address	0	1	2	3
0	0	0	0	0
1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
4	4	4	4	4
5	5	5	5	5
6	6	6	6	6
7	7	7	7	7
8	8	8	8	8
9	9	9	9	9

Table 3.1: *Buffer layout of the area-efficient microarchitecture with $m = 4$ and $n = 10$. This layout is replicated across the minimum x, maximum x, minimum y, maximum y, minimum z and maximum z memories.*

mories. This structure and data layout essentially create six m-port memories, thereby allowing $6m$ data to be outputted in a single clock cycle and permitting a high degree of parallelism. Previous attempts at creating m-port memories [So et al, 2004; Wang and Gu, 2007] avoid data replication but have variable access times. This variability would decrease the performance of the microarchitecture, necessitating the data replication.

The AABBs are outputted from the buffer in a specific sequence to perform the necessary comparisons. Originally, the address input of the first memory is set to 0, while the address inputs of the remaining $m - 1$ memories are set to the $m - 1$ successive addresses. On the subsequent cycle, the address input of the first memory retains its value and the address inputs of the remaining $m - 1$ memories are each incremented by $m - 1$. This sequence continues until one of the address inputs selects address $n - 1$. At this stage, the address input of the first memory is set to 1, while the address inputs of the remaining $m - 1$ memories are set to the $m - 1$ subsequent addresses. The sequence continues until the address input of the first memory selects address $n - 2$. This permits addresses after $n - 1$ to be accessed; these must be subsequently removed. The comparisons to be performed are exemplified in Table 3.2, while the entire sequence is exemplified in Table 3.3 and Algorithm 3.2.

Although the design uses single-port memories, it is equally amenable to memories with p ports, for $p > 1$. In these cases, the sequence can be trivially extended across the extra ports by considering each port as a separate memory. Extra ports should result in increased performance with lower resource utilisation.

The maximum value of n is d, the depth of a single memory. If d is insufficient, multiple memories may be combined to create a deeper memory. However, this may limit the value of m, which may decrease overall performance. Therefore, it is necessary to investigate the tradeoffs for each platform

	Memory					
Cycle	0	1	0	2	0	3
0	0	1	0	2	0	3
1	0	4	0	5	0	6
2	0	7	0	8	0	9
3	1	2	1	3	1	4
4	1	5	1	6	1	7
5	1	8	1	9	1	–
6	2	3	2	4	2	5
7	2	6	2	7	2	8
8	2	9	2	–	2	–
9	3	4	3	5	3	6
10	3	7	3	8	3	9
11	4	5	4	6	4	7
12	4	8	4	9	4	–
13	5	6	5	7	5	8
14	5	9	5	–	5	–
15	6	7	6	8	6	9
16	7	8	7	9	7	–
17	8	9	8	–	8	–

Table 3.2: *Sequence of comparisons for the area-efficient microarchitecture with m = 4 and n = 10. Invalid indices are marked '–'.*

procedure AreaEfficientControl(n, m)
  for $i \leftarrow 0$ **to** $n - 2$ **do**
    $address^0 \leftarrow i$
    for $j \leftarrow 0$ **to** $\left\lceil \frac{n-i-1}{m-1} \right\rceil - 1$ **do**
      $address^1 \leftarrow j(m-1) + i + 1$
      $\vdots$
      $address^{m-1} \leftarrow j(m-1) + i + m - 1$
      wait
    end for
  end for
end procedure

Algorithm 3.2: *Algorithm to generate control signals for the buffer of the area-efficient microarchitecture. '***wait***' indicates that the current control signals should be maintained for a single clock cycle.*

	Memory			
Cycle	0	1	2	3
0	0	1	2	3
1	0	4	5	6
2	0	7	8	9
3	1	2	3	4
4	1	5	6	7
5	1	8	9	–
6	2	3	4	5
7	2	6	7	8
8	2	9	–	–
9	3	4	5	6
10	3	7	8	9
11	4	5	6	7
12	4	8	9	–
13	5	6	7	8
14	5	9	–	–
15	6	7	8	9
16	7	8	9	–
17	8	9	–	–

Table 3.3: *Sequence of addresses for the buffer of the area-efficient microarchitecture with $m = 4$ and $n = 10$. This sequence is replicated across the minimum x, maximum x, minimum y, maximum y, minimum z and maximum z memories. Invalid addresses are marked '–'.*

before deciding on memory depth. The memory bandwidth is

$$f m p w \text{ bit/s}$$

where f is the clock frequency of the microarchitecture in hertz and w is the bit-width of a single memory location. The number of cycles required to generate the sequence is

$$\sum_{i=0}^{n-2} \left\lceil \frac{n-i-1}{mp-1} \right\rceil.$$

Compare The compare operation performs the comparison from the all-pairs algorithm using the data supplied by the buffer. It comprises $3(mp-1)$ greater-than-or-equal-to comparators and $3(mp-1)$ less-than-or-equal-to comparators. The outputs are connected to $mp-1$ logical AND gates. Each gate takes six inputs corresponding to the six comparator results forming a single AABB pair.

Communication Interface The communication interface manages the transfer of data between the microarchitecture and the software. Many details are determined by the underlying hardware platform used for the implementation, including the communication protocol and the layout of the external memories. However, certain functionality must be present in all implementations.

At the beginning of each collision detection cycle, the AABBs are transferred to the microarchitecture's external memories. These AABBs are represented as six data corresponding to the minimum x, maximum x, minimum y, maximum y, minimum z and maximum z. A register is used to communicate the number of AABBs written to memory. After processing the AABBs, the indices of the colliding AABBs are written to different external memories, from which the software can read them. A register records the number of collisions detected. An interrupt could be used to signal completion.

In many platforms, the external memories are unlikely to be sufficiently wide to permit $mp-1$ collisions to be written in parallel. Even if all collisions could be written simultaneously, these would be distributed across a number of discrete memories and reading each memory from the software would be likely to incur a substantial overhead. To surmount these challenges, the collisions could be serialised before being written to external memory. Although serialisation can create a bottleneck, most simulations will result in less than one collision per clock cycle as there tend to be far fewer collisions than objects. The serialisation could be performed using $2(mp-1)$ FIFOs, which would store the indices of the colliding AABBs as they become available. Each collision would require two FIFOs – one for the index of the first AABB involved in a collision and another for the index of the second. On some platforms, fixed-width FIFOs are provided and it may be beneficial to combine the two FIFOs corresponding to a single collision into a single FIFO, as both FIFOs would use the same control signals. The FIFOs would be serviced by a round robin scheduler, which would emit one collision per cycle as long as unserviced collisions are present. The outlined serialisation module is illustrated in Figure 3.3.

On some platforms, the microarchitecture may be able to directly access the CPU's memories. In

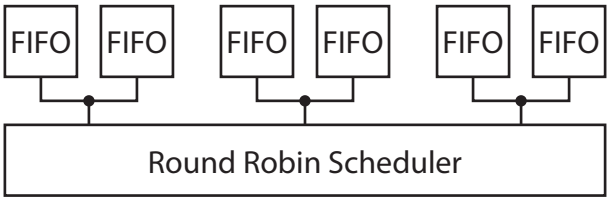


Figure 3.3: *Module for serialising collisions before they are written to the external memories.*

such cases, the software only needs to set a register indicating that data are available at a particular memory address and that execution can proceed. This eliminates the time spent by the CPU in the transferral of data, but there may be significant latencies involved in reading the CPU's memories from the microarchitecture. Results would still be written to one of the microarchitecture's external memories, so the software would have to access these memories as before.

The optimal data layout for transferring the AABBs to the microarchitecture is one which consists of the AABBs listed sequentially, with no extraneous data between them. However, this layout may be time-consuming to construct in software and it may be beneficial to transfer the ADTs containing the AABBs to the microarchitecture directly. If the microarchitecture understands the format of the ADTs a priori, there should be little difficulty in their interpretation. This optimisation would be most useful if the microarchitecture were reading directly from the CPU's memories. However, more data will need to be transferred to and processed by the microarchitecture and this may result in inefficiencies.

The aforementioned schemes are all optional and dependent on the nature of the target hardware platform. Concrete details are provided for the implementations in Chapter 4.

Number Representation One of the major advantages of microarchitectures is that there is complete flexibility with regard to the representation of real numbers. This differs from software, where the representation is constrained to one of those supported by the CPU. By appropriately adjusting the representation in a microarchitecture, significant accelerations can often be achieved. Therefore, it is necessary to consider the most appropriate representation for the collision detection microarchitecture.

Firstly, the microarchitecture could use the IEEE 754 floating-point representation [IEEE, 2008], which is the representation typically used in software. There is a wealth of research highlighting the potential of floating-point computations on microarchitectures [Underwood, 2004; Govindu et al, 2004; Scrofano et al, 2008]. Moreover, this representation can be used by both the software and the microarchitecture so that no translation between representations is required when communicating data. Floating-point comes in both single- and double-precision variants. However, single-precision is likely to be sufficient as it is the default precision of many collision detection libraries, including SOLID [van den Bergen, 2007] and the Bullet Physics SDK [Coumans, 2009]. This indicates that it offers sufficient precision for the majority of interactive applications. Double- or custom-precision could also be appropriate, depending on the nature of the application.

Alternatively, fixed-point representations could be used. These store each number in an integer-like representation for increased performance and decreased resource utilisation, at the cost of decreased

precision. These are unlikely to offer significant benefit for the collision detection microarchitecture, as the only real number units are comparators and these tend not to exhibit poor performance or high resource utilisation with any representation, especially when pipelined. Moreover, these representations require the addition of a float-to-fixed converter to translate each floating-point number before it is written to the buffer, resulting in significant delays and increased resource utilisation. Alternatively, the translation could be performed in software but this is likely to incur similar delays. The low accuracy of fixed-point also means that numbers would need to be adjusted to ensure that each AABB fully encloses its object. This adjustment involves choosing the fixed-point numbers at or before the AABB minima and the fixed-point numbers at or after the AABB maxima. This could potentially result in increased collisions, which could lead to longer narrow phase execution times, although any impact is likely to be minimal. Overall, this analysis indicates that floating-point representations are better suited to the microarchitecture.

If the concept of fixed-point representations is taken further, the result is integer representations. These reduce the real numbers describing the simulated environment into integers, thereby eliminating all position information after the decimal point. Their primary advantage is that they consume fewer resources than fixed-point representations. In addition, they can execute rapidly as most arithmetic operations can be performed in a single clock cycle. There would still be a need to translate the data transferred to the microarchitecture, although this translation would be significantly less complex than with a fixed-point representation. In addition, there would be a loss of precision greater than that experienced by the fixed-point representation, although this is unlikely to be problematic as integer representations have been successfully used in commercial collision detection systems. Despite their various advantages, these representations are unlikely to offer significant benefit, as the comparators are the only real number components in the microarchitecture. All of this shows that floating-point representations are better suited to the microarchitecture, although integer representations may be beneficial for very tightly constrained platforms.

Another alternative is the logarithmic number system (LNS) [Swartzlander Jr and Alexopoulos, 1975; Matoušek et al, 2002]. This stores numbers in a logarithmic representation, resulting in rapid multiplication and division at the expense of slower addition and subtraction. The comparators are likely to exhibit similar performance with the LNS as for floating-point, so the representation offers little benefit to the collision detection microarchitecture. Moreover, the use of the LNS would require the addition of a float-to-LNS converter to translate each floating-point number before it is written to the buffer, resulting in significant delays and increased resource utilisation. As before, the translation could be performed in software but this is likely to incur similar delays. Therefore, the LNS is likely to be a suboptimal representation for the microarchitecture.

3.3 Many-Object Microarchitecture

The area-efficient microarchitecture achieves good performance using minimal resources. However, it makes inefficient use of these resources since each AABB is replicated across m memories, limiting

the number of AABBs that can be processed to d. For larger hardware platforms, it is desirable to remove this data replication so that more AABBs can be processed, even if this comes at the expense of increased resource utilisation.

The many-object microarchitecture, which removes the data replication, consists of a pipeline implementing three primary operations – buffer, reorder and compare – coupled with a communication interface as illustrated in Figure 3.4. A detailed schematic is provided in Figure 3.5. As for the area-efficient microarchitecture, the many-object microarchitecture's size can be adapted using a factor m.

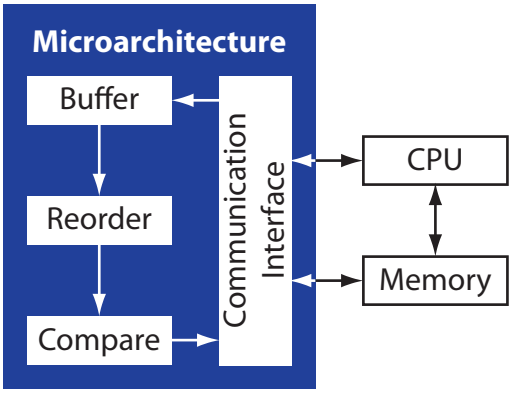


Figure 3.4: *Overview of the many-object microarchitecture showing the connection of the three primary operations and the communication interface.*

Buffer Like the area-efficient microarchitecture, the buffer uses the communication interface to initially read every AABB from external memory. These are stored in $6m$ dual-port memories corresponding to m memories for each of the minimum x, maximum x, minimum y, maximum y, minimum z and maximum z values of the AABBs. The relevant data are stored across the m memories so that none are duplicated, as illustrated in Table 3.4. To achieve this data layout, a write enable consisting of a single active bit is used when writing to the m memories. This write enable starts at the first memory and is rotated by one bit on each cycle. The corresponding write address, which is supplied to each of the m memories, begins at 0 and is incremented after m cycles. This structure and data layout creates a division between the memory address and the index of the AABB being retrieved but the index can be computed using $am + j$ where a is the address and j is the memory being accessed.

The AABBs are outputted from the buffer in a specific sequence to perform the necessary comparisons. The use of dual-port memories enables data from two addresses (A and B) to be outputted on each clock cycle. Originally, every A is set to 0, while the first B is set to 1 and the remaining Bs are set to 0. On the subsequent cycle, the second B is incremented to 1 and the other Bs retain their previous values. This sequence continues until the first B selects

$$\left\lceil \frac{n}{m} \right\rceil,$$

which ensures all comparisons to AABB $n - 1$ have taken place. At this stage, every A is set to 1, the

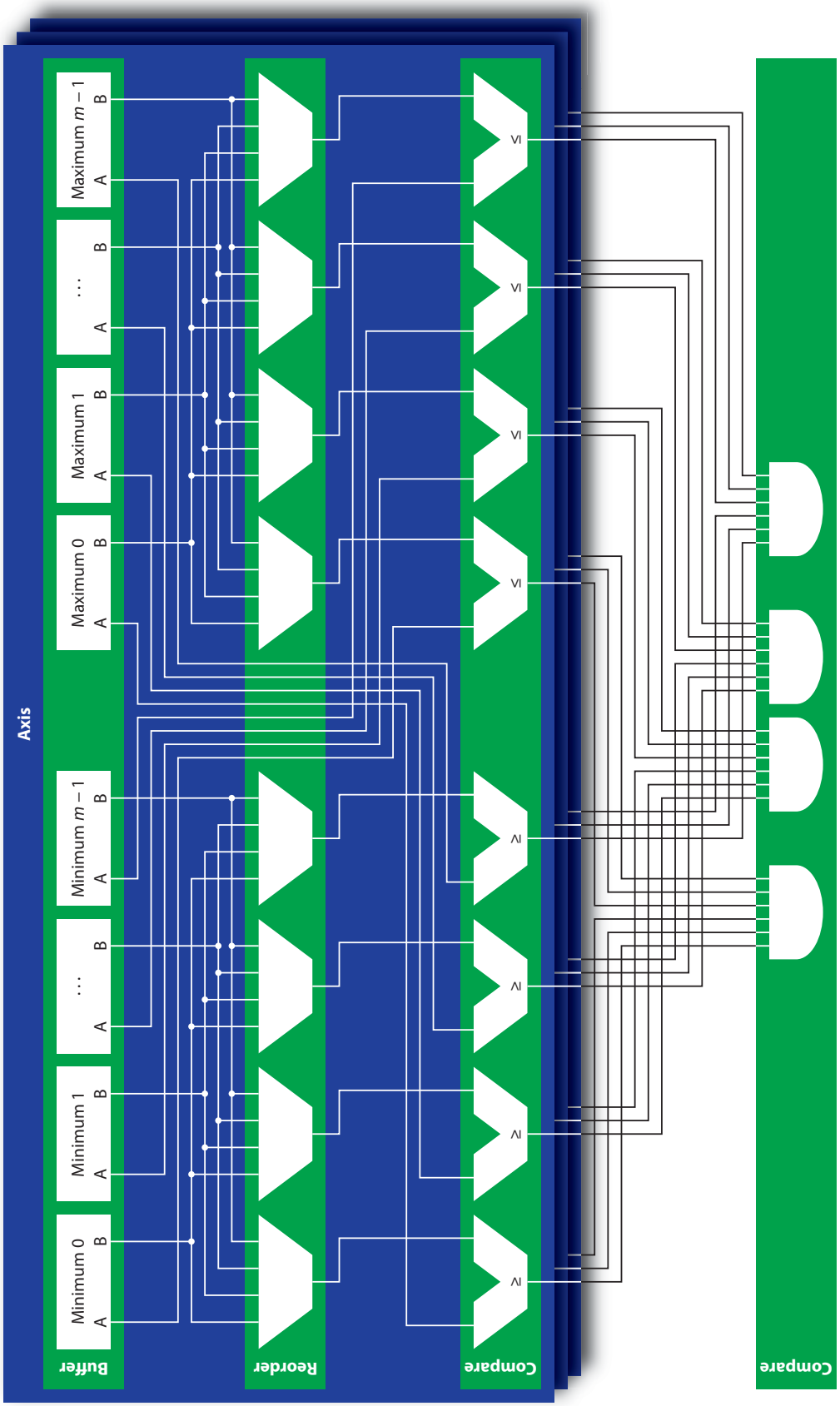


Figure 3.5: *Detailed schematic of the many-object microarchitecture.*

	Memory			
Address	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	–	–

Table 3.4: *Buffer layout of the many-object microarchitecture with $m = 4$ and $n = 10$. This layout is replicated across the minimum x, maximum x, minimum y, maximum y, minimum z and maximum z memories. Locations containing no data are marked '–'.*

first B is set to 2 and the remaining Bs are set to 1. This sequence continues until some A selects

$$\left\lceil \frac{n}{m} \right\rceil - 1,$$

which is the address corresponding to AABB $n - 2$, and the first B selects

$$\left\lceil \frac{n}{m} \right\rceil.$$

This permits data after AABB $n - 1$ to be accessed; these must be subsequently removed. The comparisons to be performed are exemplified in Table 3.5, while the entire sequence is exemplified in Table 3.6 and Algorithm 3.3.

	Memory Port							
Cycle	A	B	A	B	A	B	A	B
0	0	1	1	2	2	3	3	4
1	0	2	1	3	2	4	3	5
2	0	3	1	4	2	5	3	6
3	0	4	1	5	2	6	3	7
4	0	5	1	6	2	7	3	8
5	0	6	1	7	2	8	3	9
6	0	7	1	8	2	9	3	–
7	0	8	1	9	2	–	3	–
8	0	9	1	–	2	–	3	–
9	4	5	5	6	6	7	7	8
10	4	6	5	7	6	8	7	9
11	4	7	5	8	6	9	7	–
12	4	8	5	9	6	–	7	–
13	4	9	5	–	6	–	7	–
14	8	9	–	–	–	–	–	–

Table 3.5: *Sequence of comparisons for the many-object microarchitecture with $m = 4$ and $n = 10$. Invalid indices are marked '–'. The colours correspond to Table 3.6.*

	A								B											
	Address				AABB Index				Address				AABB Index				Multiplexer			
Cycle	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	0	0	0	0	1	2	3	1	0	0	0	4	1	2	3	1	2	3	0
1	0	0	0	0	0	1	2	3	1	1	0	0	4	5	2	3	2	3	0	1
2	0	0	0	0	0	1	2	3	1	1	1	0	4	5	6	3	3	0	1	2
3	0	0	0	0	0	1	2	3	1	1	1	1	4	5	6	7	0	1	2	3
4	0	0	0	0	0	1	2	3	2	1	1	1	8	5	6	7	1	2	3	0
5	0	0	0	0	0	1	2	3	2	2	1	1	8	9	6	7	2	3	0	1
6	0	0	0	0	0	1	2	3	2	2	2	1	8	9	–	7	3	0	1	2
7	0	0	0	0	0	1	2	3	2	2	2	2	8	9	–	–	0	1	2	3
8	0	0	0	0	0	1	2	3	3	2	2	2	–	9	–	–	1	2	3	0
9	1	1	1	1	4	5	6	7	2	1	1	1	8	5	6	7	1	2	3	0
10	1	1	1	1	4	5	6	7	2	2	1	1	8	9	6	7	2	3	0	1
11	1	1	1	1	4	5	6	7	2	2	2	1	8	9	–	7	3	0	1	2
12	1	1	1	1	4	5	6	7	2	2	2	2	8	9	–	–	0	1	2	3
13	1	1	1	1	4	5	6	7	3	2	2	2	–	9	–	–	1	2	3	0
14	2	2	2	2	8	–	–	–	3	2	2	2	–	9	–	–	1	2	3	0

Table 3.6: *Sequence of addresses and AABB indices for the buffer, along with the corresponding multiplexer selectors for the reorder operation, of the many-object microarchitecture with $m = 4$ and $n = 10$. This sequence is replicated across the minimum x, maximum x, minimum y, maximum y, minimum z and maximum z memories. Invalid indices are marked '–'. The colours, which correspond to Table 3.5, highlight the indices used in each comparison.*

```
procedure ManyObjectControl(n, m)
    for i ← 0 to ⌈n/m⌉ − 1 do
        address_A ← i
        index_A^0 ← im
        index_A^1 ← im + 1
                  ⋮
        index_A^{m−1} ← im + m − 1

        address_B^0 ← i + 1
        address_B^1 ← i
                  ⋮
        address_B^{m−1} ← i
        index_B^0 ← m(i + 1)
        index_B^1 ← im + 1
                  ⋮
        index_B^{m−1} ← im + m − 1

        multiplexer^0 ← 1
        multiplexer^1 ← 2
                  ⋮
        multiplexer^{m−1} ← 0
        next ← 1
        wait

        for j ← 0 to n − index_A^0 − 1 do
            if next = 0 then
                address_B^0 ← address_B^0 + 1
                index_B^0 ← address_B^0 × m
            else if next = 1 then
                address_B^1 ← address_B^1 + 1
                index_B^1 ← (address_B^1 × m) + 1
            else if ··· then
                ⋮
            else
                address_B^{m−1} ← address_B^{m−1} + 1
                index_B^{m−1} ← (address_B^{m−1} × m) + m − 1
            end if

            multiplexer^0 ← (multiplexer^0 + 1) mod m
            multiplexer^1 ← (multiplexer^1 + 1) mod m
                  ⋮
            multiplexer^{m−1} ← (multiplexer^{m−1} + 1) mod m
            next ← (next + 1) mod m
            wait
        end for
    end for
end procedure
```

Algorithm 3.3: *Algorithm to generate control signals for the buffer and reorder operations of the many-object microarchitecture. '***wait***' indicates that the current control signals should be maintained for a single clock cycle.*

Although the design uses dual-port memories, it is equally amenable to either single- ($p = 1$) or multi-port ($p > 2$) memories. If $p = 1$, the buffer layout must be modified so that each value is replicated across the memories once, as exemplified in Table 3.7. The formula for converting an address into an index becomes

$$\frac{am}{2} + j \mod 2.$$

In the sequence of memory requests, A corresponds to the first set of $m/2$ memories and B corresponds to the second set of $m/2$ memories. If $p > 2$ and p is even, the buffer layout is left unchanged. During the sequence of memory requests, the system is expanded by essentially executing multiple cycles simultaneously. For example, in the case of $p = 4$, A of the first cycle would correspond to the quad-port memory's A, B of the first cycle would correspond to the quad-port memory's B, A of the second cycle would correspond to the quad-port memory's C and B of the second cycle would correspond to the quad-port memory's D. If $p > 2$ and p is odd, one port can be ignored so that p essentially becomes even. The aforementioned sequence can then be applied.

	Memory			
Address	0	1	2	3
0	0	1	0	1
1	2	3	2	3
2	4	5	4	5
3	6	7	6	7
4	8	9	8	9

Table 3.7: *Buffer layout for the many-object microarchitecture with $m = 4$, $n = 10$ and $p = 1$. This layout is replicated across the minimum x, maximum x, minimum y, maximum y, minimum z and maximum z memories.*

The maximum value of n is $(dm)/2$ when $p = 1$, or dm when $p > 1$. As for the area-efficient microarchitecture, multiple memories may be combined to create a deeper memory, but this may limit the value of m, which may decrease overall performance. The memory bandwidth is

$$fmpw \text{ bit/s}.$$

The number of cycles required to generate the sequence is

$$\frac{2\sum_{i=0}^{\lceil n/m \rceil - 1}(n - im - 1)}{p}.$$

Reorder If the data emitted from the buffer were immediately sent to the compare operation, only a fraction of the required comparisons would take place and some of these comparisons would be repeated multiple times. The reorder operation is designed to rectify this using $3mp$ multiplexers to reorder the data to create the appropriate sequence of comparisons. These multiplexers can consume signi-

ficant hardware resources, which is why this microarchitecture is less area efficient than the previous microarchitecture.

Originally, multiplexer 0's selector is initialised to 1, multiplexer 1's is initialised to 2, multiplexer $m-2$'s is initialised to $m-1$ and multiplexer $m-1$'s is initialised to 0. On each cycle, each selector is incremented by 1 mod m. The sequence restarts each time the buffer's A inputs are modified. This is exemplified in Table 3.6 and Algorithm 3.3.

Compare The compare operation performs the comparison from the all-pairs algorithm using the data supplied by the reorder operation. It comprises $1.5mp$ greater-than-or-equal-to comparators and $1.5mp$ less-than-or-equal-to comparators. The outputs are connected to $0.5mp$ logical AND gates. Each gate takes six inputs corresponding to the six comparator results forming a single AABB pair.

Communication Interface, Number Representation The communication interface can use the same design as for the area-efficient microarchitecture. Additionally, the same decisions apply when selecting an appropriate number representation.

3.4 Spatial Partitioning

Although the many-object microarchitecture can store many more AABBs than the area-efficient one, it is still limited by the depth of its internal memories. To overcome these limitations, software-based spatial partitioning is added to the system as its first stage. This divides the list of AABBs into appropriately sized cells for processing by the microarchitectures. Another advantage of software-based spatial partitioning is that it should reduce the average computational complexity of all-pairs. In this way, the microarchitecture has been leveraged for those algorithms ideally suited to microarchitecture implementation, while the software eliminates many of the microarchitecture's disadvantages.

The addition of spatial partitioning effectively results in a system comprising three main stages.

Spatial Partitioning The first stage executes in software and divides the simulated environment into cells that contain an appropriate number of AABBs for processing by the microarchitecture. It forms the first stage of the broad phase.

Object-Object Tests The second stage executes on the microarchitecture and performs object-object collision detection tests on the contents of each cell. It forms the second stage of the broad phase.

Narrow Phase The third stage executes in software and performs conventional narrow phase collision detection.

The three stages are summarised in Table 3.8.

There exist a multitude of spatial partitioning algorithms and each makes a different tradeoff between the execution times of the construction and traversal phases. Two such algorithms are investigated in the context of the collision detection system. These algorithms represent two general families of spatial partitioning algorithms – grid-based and hierarchical.

Stage	Platform	Phase
Spatial partitioning	Software	Broad phase
Object-object tests	Hardware	Broad phase
Narrow phase	Software	Narrow phase

Table 3.8: *The hybrid system's three stages.*

Uniform Grid This grid creates cells whose size in each dimension is based on the AABB whose size in that dimension is least, so that each cell can hold a maximum of six AABBs as per Bentley et al [1977]. Since this could result in the transfer of a multitude of small cells to the microarchitecture, the cells are expanded to contain at most the maximum number of AABBs supported by the microarchitecture. This expansion can be expressed as a linear programming task, where c represents the maximum cell size supported by the microarchitecture:

$$\begin{aligned}&\text{maximise } x, y, z\\ &\text{subject to } 2x + 2y + 2z + (x-1)(y-1)(z-1) \leq c.\end{aligned}$$

The cell sizing factors x, y and z for different values of c are listed in Table 3.9. At the beginning of the simulation, the grid cells are constructed and are not changed for the remainder of the simulation, since the cells are only based on the sizes of the environment and the smallest object, which are assumed to remain unchanged for the duration of the simulation. If these were anticipated to change, this should be taken into account prior to constructing the grid. After constructing the cells, each AABB is placed in every cell overlapped by that AABB and, when updating the grid, coherence is exploited by only updating AABBs that have moved between collision detection cycles. However, this scheme still results in a multitude of small cells as their size is designed to never exceed c objects and many cells contain far fewer than c objects. Therefore, on each cycle, the list of cells is traversed and combined into a larger temporary cell, while the size of the larger cell remains less than c.

c	x	y	z	Maximum Objects
1024	10	11	11	964
2048	13	13	14	1952
4096	16	17	17	3940
8192	21	21	21	8126

Table 3.9: *Cell sizing factors for different values of c. 'Maximum Objects' indicates the maximum number of objects that can fit in each cell based on the cell sizing factors. In each case, this is less than c.*

AABB Tree Narrow phase collision detection often utilises hierarchies of AABBs known as AABB trees. However, it is hypothesised that AABB trees could also be used as an efficient broad phase spatial partitioning algorithm. The system tests this hypothesis using a scheme similar to that

described by van den Bergen [1997]. To construct the tree, top-down recursive subdivision is used. At each step, the objects are tightly bounded with an AABB. This AABB is partitioned by splitting it at its spatial median along its longest side. Occasionally, it may not be possible to create child cells smaller than their parents when splitting along the longest side. In these cases, each side is tested for its suitability as a splitting plane until an appropriate side is located. The objects are added to the appropriate child node and the process continues until each AABB contains a quantity of objects less than or equal to c. During construction, the leaf nodes are recorded and these are used for the collision check. Coherence is not exploited and the tree is reconstructed on each collision detection cycle.

Both algorithms have been designed to preclude the possibility of having to process cells larger than the maximum size supported by the microarchitecture. In the case of the uniform grid, this is achieved by creating cells that can contain, at most, the maximum number of objects supported. For the AABB tree, it is achieved by repeatedly splitting the tree's nodes until the number of objects in each node is less than or equal to the maximum supported. It was necessary to design the algorithms in this way so that the microarchitecture could be used in all cases, without requiring a fallback strategy.

In certain cases, objects may lie on the splitting planes of grid cells. When this occurs, each split object will be added to the cells on both sides of the plane, resulting in duplicate objects. These duplicate objects may, in turn, result in duplicate pairs. Consequently, the same collision could be detected in multiple cells. This multiple detection is ignored in the broad phase, as it would be more time consuming to check for duplicate pairs than to perform the superfluous collision checks. Instead, the narrow phase eliminates multiple occurrences of a potential collision before it begins processing.

After executing the spatial partitioning algorithm, the contents of each cell are added to a data structure whose design is influenced by the microarchitecture's implementation details. This data structure is subsequently transferred to the microarchitecture. Since the microarchitecture refers to the AABBs by their index within the cell and not by the 'global' indices used by the software, it is necessary to create a second data structure to retain the global indices while the collision tests are performed by the microarchitecture. This data structure is essentially a 2D array, where the grid cells are stored vertically and the per-cell indices are stored horizontally. After the spatial partitioning algorithm computes the contents of each cell, the objects are added to the array corresponding to the appropriate grid cell in the order in which they are to be transferred to the microarchitecture. The position of the objects within this array implicitly creates a per-cell index for each object. After each grid cell is processed by the microarchitecture, the global index can be computed by selecting the array entry corresponding to the cell number and the per-cell index returned by the microarchitecture. Although simple, this scheme is very effective as the straightforward integer arithmetic used to access the array is highly efficient. The scheme is illustrated in Figure 3.6.

The addition of spatial partitioning also necessitates modification of the microarchitectures. These modified microarchitectures process the cells sequentially so that, for example, processing of the second cell only begins once processing of the first has completed. An appropriately modified microarchitecture will be able to process simulations involving single or multiple cells. Consequently, all simu-

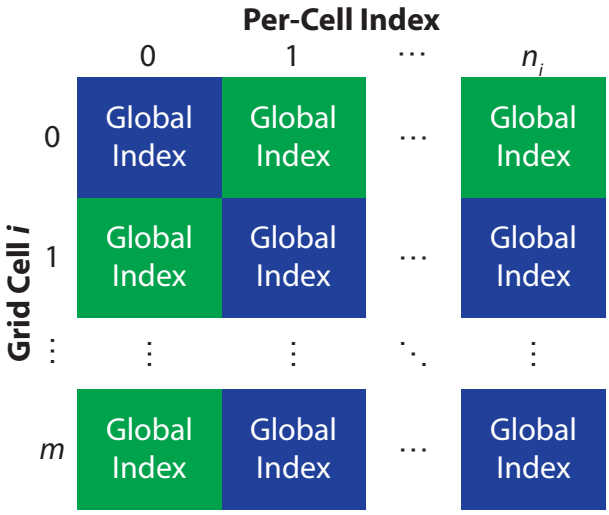


Figure 3.6: *Data structure used to convert per-cell indices to global indices after processing by the microarchitectures. n_i may differ with i, where i refers to the grid cell.*

lations that can execute on the original microarchitectures will also execute on the adapted microarchitectures, although the differences in the interface would necessitate some minor modifications to the software.

The aforementioned microarchitecture modification only affects the communication interfaces. At the beginning of each collision detection cycle, all of the cells are transferred to the microarchitecture's external memories, with a marker delimiting the end of each cell. A register indicates the number of cells written to memory, instead of the number of AABBs. After processing each cell, the indices of the colliding AABBs are written to external memories from which the software can read them. Alternate cells are written to alternate memories so that one cell can be read while another is being written, thereby preventing the communication interface from becoming a bottleneck. Moreover, narrow phase processing can begin on the contents of each cell once it is read by the software, so that the software does not have to idle while the microarchitecture is executing. Each memory has three associated registers recording the cell currently stored in that memory, the number of collisions detected for that cell and the last cell read by the software from that memory. This last register allows the microarchitecture to determine when the PCS can be overwritten.

No other microarchitecture modifications are required. However, the buffer could be modified so that two sets of $6m$ memories are used. With this design, the first cell is read into the first set of memories and processing commences. During processing, the second cell is read into the second set of memories. After processing of the first cell completes, the memories are switched and processing commences using the second set of memories. During this processing, the third cell is read into the first set of memories. This sequence continues until all cells have been read. The advantage of this design is that it reduces the time spent reading data from external memory, since the data reads are overlapped with processing for all cells after the first. However, this design may limit the value of m, as significant memory resources will be consumed by the second set of memories.

3.5 Other Collision Detection Types

All of these microarchitectures and, consequently, all of the hybrid systems, focus on the acceleration of static collision detection for rigid bodies. Static collision detection only checks for collisions between objects at discrete intervals, essentially reducing the objects' motion to a set of discrete positions and orientations. Rigid bodies are objects that cannot undergo deformation. This narrow focus reflects the collision detection used in the majority of interactive applications. However, there has recently been increased interest in both dynamic collision detection and deformable bodies.

Dynamic collision detection or CCD considers the entire motion of an object between the beginning and end of an interval. As stated in Section 2.1.1, speedboxes are typically used to perform the dynamic collision detection tests. These speedboxes could be directly processed by the microarchitectures if sufficient adaptations were made. Alternatively, the speedboxes could be enclosed within AABBs and processed by the microarchitectures without modification, although this is likely to result in poor performance. This would be especially problematic if the objects moved a substantial non–axis-aligned distance, as a very large AABB would be required to fully enclose the objects' motion.

Deformable bodies are objects, such as cloth, whose geometry can change or scale over time. They are typically modelled using the same broad phase BVs as their rigid equivalents. The only difference is in the narrow phase, where the BVHs are refitted during deformation. Therefore, the microarchitectures can detect collisions between deformable bodies without modification.

Although the focus of the collision detection microarchitectures and hybrid systems is on static collision detection for rigid bodies, it is clear that the microarchitectures offer sufficient flexibility to support other forms of collision detection. In particular, both dynamic collision detection and deformable bodies are supported directly. Moreover, dynamic collision detection could be further accelerated through minor modifications to the design.

3.6 Benchmarking Framework

Turning to the benchmarking framework, the first stage in its design is to consider the challenges involved. Firstly, the framework must be repeatable so that different researchers and developers can independently achieve the same results. Secondly, it must be sufficiently realistic to recreate scenarios found in representative applications. Thirdly, it must be adequately flexible to emulate all common scenarios but this flexibility must not result in an overly complex framework. Fourthly, the framework must be extensible to allow for the integration of new collision detection algorithms. Finally, it must record accurate and relevant performance data. The design attempts to address these five challenges.

The range of applications utilising collision detection is extremely large, so it would be challenging to create a framework that comprehensively emulates simulations found in every potential application. For this reason, the framework focuses only on simulations encountered in interactive applications. By analysing recent interactive applications, it is evident that the majority use some form of physics engine to control the motion of the objects. It is also evident that the majority of objects are simulated

using rigid bodies modelled as polygonal meshes, due to the simplicity and rapid execution times of such objects. Finally, it is evident that the majority of applications contain some form of ground object. As a result of these findings, it was decided to use a physics engine in the framework to permit the accurate recreation of appropriate simulations. Similarly, it was decided to focus on rigid bodies modelled as polygonal objects. It was also decided to include a ground plane, unlike the majority of existing benchmarks. This is a simplification of the polygonal ground object found in most interactive applications. In typical ground objects, each polygon is enclosed within its own BV. These BVs are not checked against each other but may be checked against the other BVs in the environment. Ground objects tend to result in objects clustering and overlapping along the y axis. Consequently, if the collision detection algorithm has been poorly designed, objects on the ground may necessitate significant computation before they can be eliminated from consideration. Sophisticated algorithms avoid this problem either by processing axes in a specific order or by deactivating stationary objects. It is acceptable to simplify the ground object to a ground plane as the important clustering and overlapping behaviour is preserved, but the ground is simplified so as not to overcomplicate or overly influence the benchmark.

3.6.1 Generic Benchmark

In all interactive applications, the quantities used to describe the simulations are essentially unitless. The user can interpret these unitless quantities as belonging to any system of units, providing that the system remains consistent. For clarity, the following sections utilise *le système international d'unités* (SI) [BIPM and *Organisation intergouvernementale de la Convention du Mètre*, 2006].

The core of the framework is a generic benchmark with a cube containing a variable quantity of objects. The cube's dimensions are

$$\sqrt[3]{n \times 50^3 \times 5}\,\mathrm{m} = 85.5\sqrt[3]{n}\,\mathrm{m},$$

where n is the quantity of objects. The dimensions are based on those of a default object, which is 50 m along each axis, resulting in a volume of 50^3 m^3. This volume is expanded by $\sqrt[3]{5}$ m to create unoccupied space between each object. The formula ensures that the density of objects remains constant unless object sizes are altered.

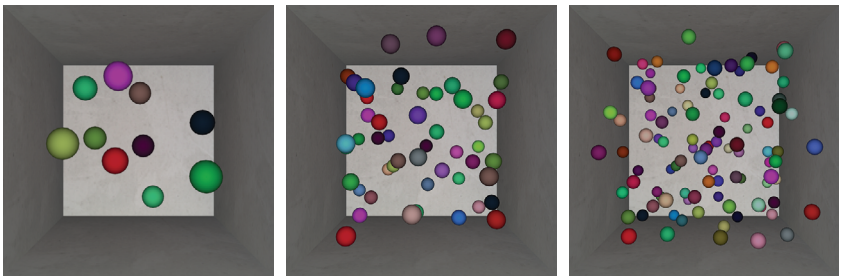
3.6.2 Parameters

The framework includes ten parameters to allow the generic benchmark to be translated into an almost infinite variety of practical benchmarks. These parameters specify the initial state of each benchmark and describe the geometrical and physical properties of rigid bodies. They should be sufficient to enable representative application scenarios to be recreated as benchmarks.

Cycles All collision detection algorithms execute as a series of discrete cycles or time steps. This parameter controls the number of cycles that are executed, with the default being 1000 cycles.

Different benchmarks require different numbers of cycles before achieving a steady state. For example, a benchmark consisting of an object falling under the influence of gravity will achieve a steady state after a certain period of time, once the object meets the ground. However, if gravity were doubled, the object would reach the ground faster and it would take fewer cycles for the benchmark to reach a steady state. This parameter provides an opportunity to adapt the number of cycles to the behaviour of the benchmark under consideration.

Objects – Quantity This parameter modifies the quantity of objects, with the default being ten objects. The effect of modifying this parameter can be observed in Figure 3.7.



(a) (b) (c)

Figure 3.7: *Object quantity parameter.* **(a)** *Ten objects.* **(b)** *Fifty objects.* **(c)** *100 objects.*

Object quantity influences algorithm behaviour, since greater quantities tend to take longer to process than smaller quantities. In addition, poorly designed algorithms may not be able to cope with larger quantities and may make inefficient memory requests.

Objects – Type This parameter selects the type of the objects, with an option of spheres, cuboids, cylinders, cones, Stanford Bunnies [Turk and Levoy, 1994], Stanford Armadillos [Krishnamurthy and Levoy, 1996] or a combination of objects determined by the uniform probability distribution. The default is spheres. As the Stanford Bunny and Armadillo are concave objects, they are constructed as a compound set of convex hulls with twenty-two convex hulls for the Bunny and twenty-five for the Armadillo. The effect of modifying this parameter can be observed in Figure 3.8.

Object type influences algorithm execution speed, as some algorithms optimise for specific primitive objects while others consider every object using a generic BV. Additionally, complex, concave objects tend to be processed differently than simpler convex primitives and can execute at very different speeds with different algorithms.

Objects – Location This parameter determines the initial location of the objects, using a constant pattern or a choice of the uniform or normal probability distributions. The default is the constant pattern, where objects are placed in rows to create a grid formation. The distance between adjacent objects is determined by adding the two size parameters. If every object is a constant size, adjacent objects would not be separated by empty space so 5 m is added to the distance. When using a probability

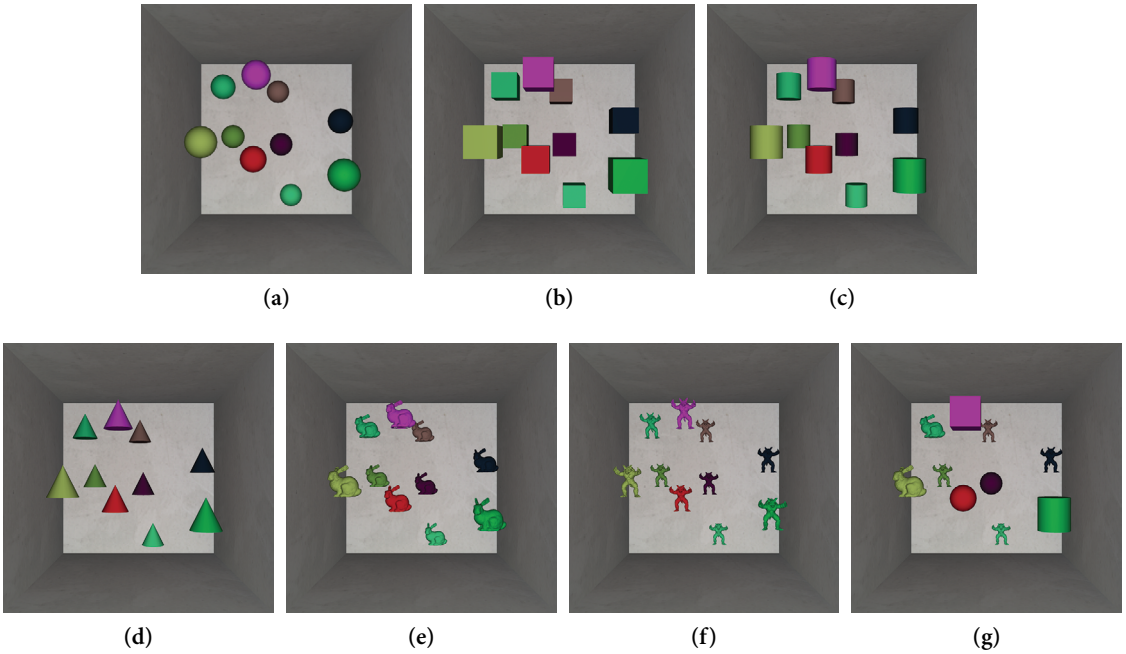


Figure 3.8: *Object type parameter.* **(a)** *Spheres.* **(b)** *Cuboids.* **(c)** *Cylinders.* **(d)** *Cones.* **(e)** *Stanford Bunnies.* **(f)** *Stanford Armadillos.* **(g)** *Uniform.*

distribution to determine initial locations, there is potential for objects to overlap each other or the cube walls. Therefore, once an object location is generated, it is checked to ensure there is no overlap. If it overlaps, a new location is generated according to the distribution. The distribution parameters are determined by the size of the cube. For the uniform distribution, the minimum and maximum are the coordinates of the cube walls. For the normal distribution, the mean is zero and the variance is the coordinates of the cube walls in the positive direction divided by three. This choice of variance ensures 99.7% of objects are within the walls. The 0.3% of objects outside the walls are detected and new normal locations within the walls are generated. The effect of modifying this parameter can be observed in Figure 3.9.

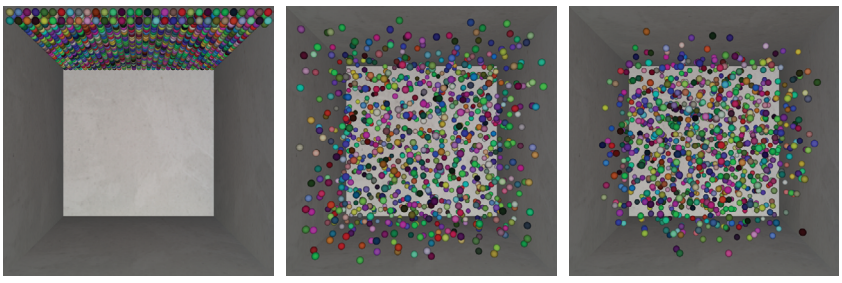
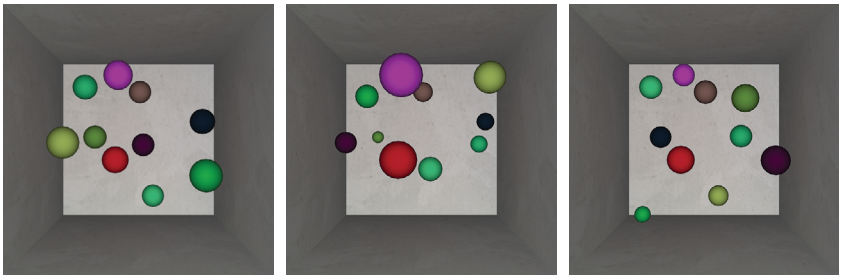


Figure 3.9: *Object location parameter. Benchmarks contain 1000 objects.* **(a)** *Constant.* **(b)** *Uniform.* **(c)** *Normal.*

Object location influences algorithm behaviour as the constant pattern distributes objects equally within one region of the cube, which highlights inefficient broad phase algorithms that make assumptions regarding object placement. The uniform distribution tends to place objects throughout the cube, while the normal distribution tends to cluster objects in the centre of the cube. A cluster of objects is likely to result in increased collisions, as the objects within the cluster interact with each other.

Size This parameter determines the size of the objects, using a constant value or a choice of the uniform or normal probability distributions. The default is a constant 50 m. When using the probability distributions the size of every object may differ in each dimension. Any sizes less than or equal to zero will be discarded and a new size will be generated. The effect of modifying this parameter can be observed in Figure 3.10. This parameter also allows the size of the cube surrounding the objects to essentially be made smaller or larger. For example, the cube can become relatively larger by making the objects smaller.



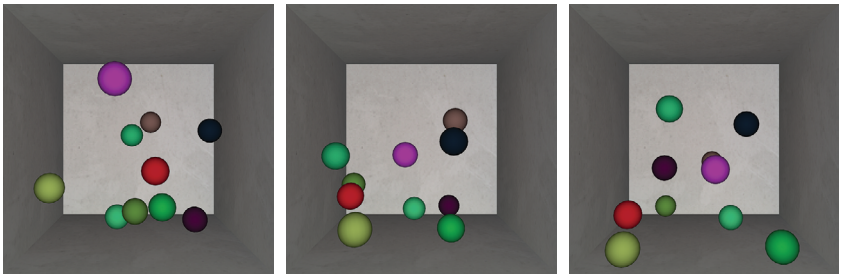
(a) (b) (c)

Figure 3.10: *Size parameter.* **(a)** *Constant 50 m.* **(b)** *Uniform with a minimum of 25 m and a maximum of 75 m.* **(c)** *Normal with a mean of 50 m and a variance of 10 m.*

Size influences algorithm execution speed as larger objects tend to result in more collisions. The uniform distribution tends to create objects across a full range of sizes, while the normal distribution tends to create objects whose sizes are centred on the mean.

Mass This parameter controls the mass of the objects, using a constant value or a choice of the uniform or normal probability distributions. The default is a constant 1 kg. The effect of modifying this parameter can be observed in Figure 3.11.

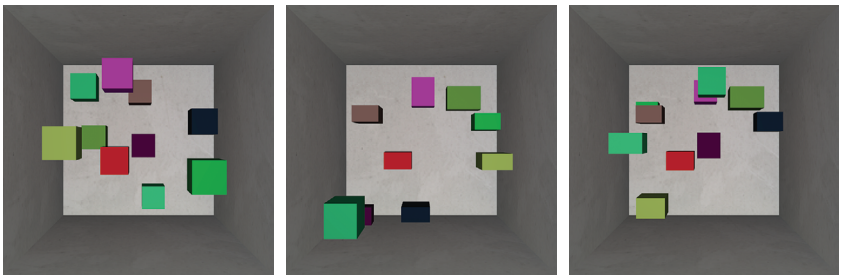
Mass influences algorithm behaviour as the mass of the objects affects their linear momenta, calculated as the product of their mass and velocity, which in turn affects their behaviour when undergoing collisions. The linear momentum of an object after a collision is determined by its mass and velocity before that collision [Fowles, 1985, pp 169, 177]. Angular momentum behaves similarly [Fowles, 1985, p 172]. Behaviour during a collision affects subsequent object behaviour and may result in either increased or decreased collisions.



(a) (b) (c)

Figure 3.11: *Mass parameter. Benchmarks executed for 250 cycles with linear velocities determined by the uniform probably distribution with a minimum of* $(-25, -25, -25)$ *m/s and a maximum of* $(25, 25, 25)$ *m/s.* **(a)** *Constant 1 kg.* **(b)** *Uniform with a minimum of 10 kg and a maximum of 100 kg.* **(c)** *Normal with a mean of 50 kg and a variance of 20 kg.*

Aspect Ratio This parameter determines the aspect ratio of the objects, using a constant value or a choice of the uniform or normal probability distributions. This ratio can be considered to be of the form $1 : y : z$ where y and z are the user-modifiable parameters. The default is $1 : 1 : 1$. Spheres are unaffected by this parameter and cones are affected only in the y dimension. The aspect ratio of all other objects can be controlled in both the y and z dimensions. The effect of modifying this parameter can be observed in Figure 3.12.

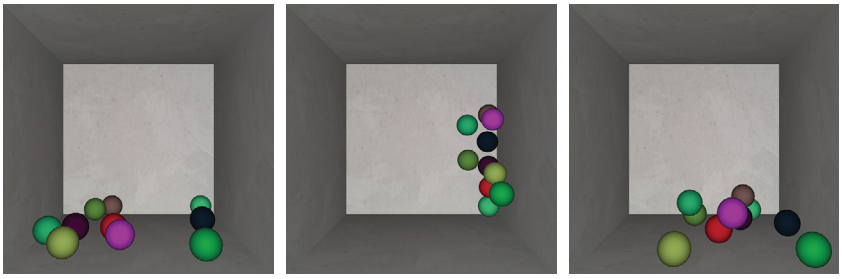


(a) (b) (c)

Figure 3.12: *Aspect ratio parameter.* **(a)** *Constant* $1 : 1 : 1$. **(b)** *Uniform with a minimum of* $1 : 0.5 : 0.5$ *and a maximum of* $1 : 1.5 : 1.5$. **(c)** *Normal with a mean of* $1 : 1 : 1$ *and a variance of* $1 : 0.2 : 0.2$.

Aspect ratio influences algorithm execution speed by adjusting the shape of the objects. Most collision detection algorithms utilise BVs and some types of BV may be difficult to fit around long, thin objects. This parameter can be used to reveal a suboptimal BV selection, as such a suboptimal selection would result in many collisions in the PCS and the collision detection algorithm would execute slowly.

Acceleration This parameter controls the acceleration that acts on every object, using a constant value or a choice of the uniform or normal probability distributions. The default is a constant $(0, -9.8, 0)$ m/s^2, corresponding to acceleration due to gravity. The acceleration is applied as a single value to the entire environment. The effect of modifying this parameter can be observed in Figure 3.13.

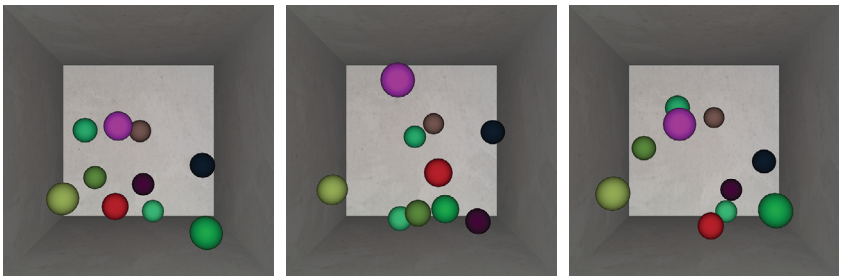


(a) (b) (c)

Figure 3.13: *Acceleration parameter. Benchmarks executed for 1000 cycles.* **(a)** *Constant* $(0, -9.8, 0)$ *m/s²*. **(b)** *Uniform with a minimum of* $(-2.5, -2.5, -2.5)$ *m/s² and a maximum of* $(2.5, 2.5, 2.5)$ *m/s²*. **(c)** *Normal with a mean of* $(0, 0, 0)$ *m/s² and a variance of* $(1, 1, 1)$ *m/s²*.

Acceleration influences algorithm behaviour by determining the motion of the objects within the environment, which influences the number of collisions. For example, the acceleration could be used to move all of the objects towards one wall of the cube. Alternatively, if zero acceleration were applied, objects would float, resulting in zero collisions unless linear or angular velocities were employed.

Linear Velocity This parameter controls the linear velocity of the objects, using a constant value or a choice of the uniform or normal probability distributions. The default is a constant $(0, 0, 0)$ m/s. The effect of modifying this parameter can be observed in Figure 3.14.



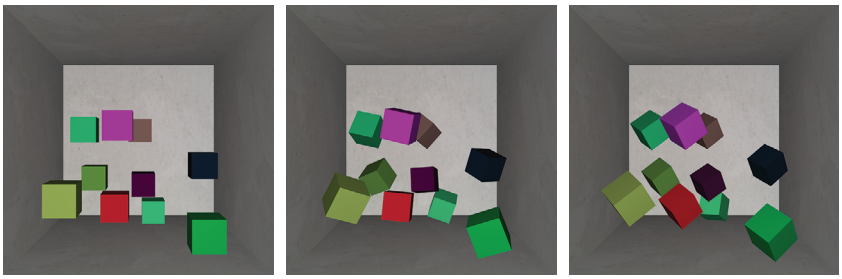
(a) (b) (c)

Figure 3.14: *Linear velocity parameter. Benchmarks executed for 250 cycles.* **(a)** *Constant* $(0, 0, 0)$ *m/s.* **(b)** *Uniform with a minimum of* $(-25, -25, -25)$ *m/s and a maximum of* $(25, 25, 25)$ *m/s.* **(c)** *Normal with a mean of* $(0, 0, 0)$ *m/s and a variance of* $(10, 10, 10)$ *m/s.*

Linear velocity influences algorithm execution speed by determining the linear motion of objects. This parameter has an effect similar to that of the acceleration but different values can be applied to different objects using a probability distribution. In particular, if objects move towards each other, the number of collisions will be maximised.

Angular Velocity This parameter controls the angular velocity of the objects, using a constant value or a choice of the uniform or normal probability distributions. The default is a constant $(0, 0, 0)$ rad/s.

The effect of modifying this parameter can be observed in Figure 3.15.



(a) (b) (c)

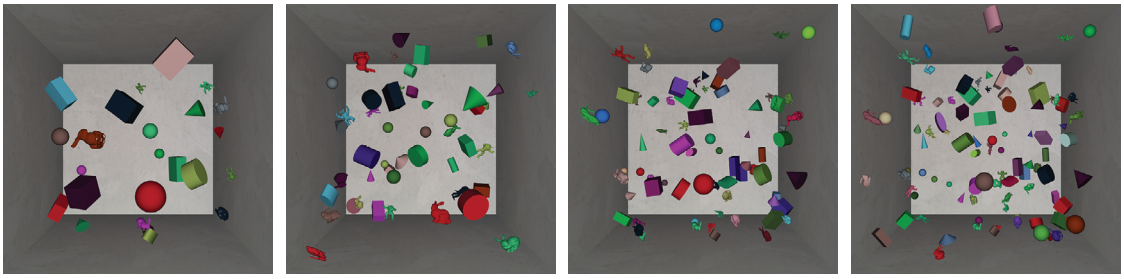
Figure 3.15: *Angular velocity parameter. Benchmarks executed for 250 cycles.* **(a)** *Constant* $(0, 0, 0)$ *rad/s.* **(b)** *Uniform with a minimum of* $(-2.5, -2.5, -2.5)$ *rad/s and a maximum of* $(2.5, 2.5, 2.5)$ *rad/s.* **(c)** *Normal with a mean of* $(0, 0, 0)$ *rad/s and a variance of* $(1, 1, 1)$ *rad/s.*

Angular velocity influences algorithm behaviour by determining the rotational motion of objects. This adds a degree of complexity to the simulation due to the difficulty of fitting BVs to rotated objects. Moreover, some BVs fit rotated objects better than others. Some algorithms also use conservative or loose-fitting BVs, which can accommodate rotations, thereby precluding the need to refit the BVs at each cycle. These larger BVs may result in more potential collisions but they may reduce execution time due to the reduced need to refit. The optimality of the collision detection algorithm's choice of BV and its behaviour in relation to these BVs can be tested using this parameter.

Remarks Comprehensive benchmarks can be created by combining the aforementioned parameters. For example, the representative benchmarks illustrated in Figure 3.16 use the following parameters:

- **Objects – Quantity:** 25, 50, 75, 100
- **Objects – Type:** Uniform probability distribution
- **Objects – Location:** Uniform probability distribution
- **Size:** Uniform probability distribution
  - **Minimum:** 25 m
  - **Maximum:** 50 m
- **Acceleration:** Constant $(0, 0, 0)$ m/s^2
- **Linear Velocity:** Uniform probability distribution
  - **Minimum:** $(-25, -25, -25)$ m/s
  - **Maximum:** $(25, 25, 25)$ m/s
- **Angular Velocity:** Uniform probability distribution
  - **Minimum:** $(-2.5, -2.5, -2.5)$ rad/s
  - **Maximum:** $(2.5, 2.5, 2.5)$ rad/s

When creating a benchmark, each uniform distribution takes minimum and maximum values as parameters, while each normal distribution takes mean and variance values as parameters.



(a) (b) (c) (d)

Figure 3.16: *Comprehensive benchmarks created by combining parameters. Benchmarks executed for 250 cycles.* **(a)** *Twenty-five objects.* **(b)** *Fifty objects.* **(c)** *Seventy-five objects.* **(d)** *100 objects.*

3.6.3 Repeatability

One of the most important points to address in a benchmarking framework is repeatability, since this allows different researchers and developers to independently achieve the same results, as per the scientific method. However, the primary difficulty in achieving repeatability is the diversity of computer components that are available. Consequently, each researcher or developer using the framework is likely to be using different CPUs and memories. One possible solution would be to restrict those using the framework to a particular system but this would place a significant burden on the users. Moreover, the results from the selected system would become meaningless as technology progressed and the system became unrepresentative of those in common use. Alternatively, users could be requested to publish their system specifications alongside their results so that others could gain an understanding of the system used. The latter approach is the one taken by the majority of benchmarks and is recommended for the benchmarking framework outlined in this thesis.

Within the framework, the topic of repeatability was addressed by seeding the random number generators with a constant value instead of the timestamp value used in typical applications. This value, hexadecimal AAAAAAAA, was used as the seed for the two random number generators that correspond to the uniform and normal probability distributions. The majority of existing benchmarks use similar solutions to ensure repeatability.

Most benchmarks go no further in addressing the topic of repeatability. However, it is beneficial to examine the topic in more detail due to its centrality to achieving reliable results. In particular, it is important to examine all other sources that may cause results to differ. Such examination suggests that these sources include the physics engine, the benchmarking framework, the implementation of the random number generators and the CPU floating-point behaviour.

Firstly, if the version of the physics engine were to differ, it is plausible that the results may also differ. This is because the algorithms used in different physics engine revisions often vary, as developers attempt to fix bugs while improving performance and accuracy. Such algorithmic modifications could result in objects moving along differing trajectories, resulting in variability in terms of the measured results. The two solutions proposed to address the diversity of technology could also be applied here.

However, a better solution would be to use only major revisions of the physics engine and to publish the physics engine version along with any results. This still creates problems for the comparison of results over a long timeframe, but provides a compromise between the progression of technology and the repeatability of results.

The same problem could also occur with different versions of the benchmarking framework, as it may not be possible to advance the framework while keeping its behaviour identical to previous versions. Consequently, it is suggested that any published results should also include the version of the benchmarking framework used.

Variations could also result from differences in the implementation of the random number generators. Although the benchmarking framework uses only the well-defined uniform and normal probability distributions seeded with a constant value, minor variations could occur within the implementations. This problem could be ameliorated by specifying which library should be used to supply the random number generators. Alternatively, as for the physics engine and benchmarking framework, the libraries could be specified along with any published results. However, any differences occurring within the random number generator implementations are likely to be extremely small due to the well-defined nature of the two probability distributions.

Finally, CPU floating-point behaviour could be problematic for repeatability. Over time, as CPU technology develops, floating-point behaviour tends to become more sophisticated through the implementation of the more esoteric features of the IEEE floating-point standard [IEEE, 2008]. In addition, the number of internal bits used for intermediate results computed during floating-point operations tends to increase. The increased accuracy resulting from these features may lead to differing results between benchmarks executed on different CPUs. As before, this problem could be resolved by specifying the CPU along with any published results.

In addition to the previously outlined solutions, there exists a further solution to the physics engine, benchmarking framework, random number generator implementation and CPU floating-point behaviour issues. Since the problems basically revolve around differing versions of the various entities, the solution is to attempt to eliminate or reduce the influence of these entities on the results. To do this, the object paths of a particular benchmark could be exported from the framework. These paths would be directly applied to the objects, instead of simulating the motion of the objects using the physics engine. This would eliminate the problems caused by differing physics engine, benchmarking framework and random number generator versions, but would not entirely eliminate the problem of differing CPU floating-point behaviour. However, the latter problem would be minimised as many of the floating-point computations would be eliminated. The issue would remain in the application of the floating-point coordinates to the simulated objects and in the benchmarking algorithm under consideration.

This solution provides a sophisticated means of overcoming the various difficulties, but it is ultimately problematic as it essentially removes the entire raison d'être of the framework by eliminating the comprehensiveness offered by the variety of parameters. This could be partially overcome by defining a set of standardised benchmarks that exercise a collision detection system in typical but varied ways.

These could then be exported as object paths, as described above, and used for analysing collision detection performance. The definition of these standardised benchmarks remains outside the scope of this thesis.

This does not solve all problems, however, as algorithm developers would be unable to sweep through a wide test space to find those scenarios that do not execute efficiently with their algorithm, which was one of the original benefits of the framework. In addition, application developers could not recreate application scenarios to determine which algorithm performs most efficiently in those scenarios. Consequently, if the proposed solution were to be implemented, it is recommended that the benchmarking framework should remain a viable option for developers, as it offers a degree of flexibility that would be lost with the proposal. The combination of fixed benchmarks with the benchmarking framework would ultimately provide a solution that encompasses the advantages of both.

3.6.4 Measurement

For each cycle, the execution times of the broad and narrow phases are recorded separately. The overall cycle execution time is determined by summing the times of the two phases. Based on these measurements, the mean, standard deviation, minimum, maximum and overall cycle execution times are computed. The mean permits a rapid insight into the behaviour of the algorithm under consideration, while the standard deviation, minimum and maximum highlight the fluctuation between cycles. When the mean value is examined, the disparity between different benchmarks' cycle execution times can seem minor. This can belie the true difference between the benchmarks' overall execution times; the overall overcomes this problem as it permits visualisation of these differences.

3.7 Summary

In this chapter, the split between software and hardware for the hybrid collision detection system was first analysed. This analysis determined that the broad phase was more amenable to microarchitecture implementation, due to the significant scope for parallelisation in broad phase algorithms coupled with their compute-bound nature and their use of standard recurring collision tests performed in a well-defined sequence. It was also established that the narrow phase could be left in software and still achieve good performance. Based on these findings, two variants of the microarchitecture were designed. Both were based on a parallel version of the all-pairs algorithm and used a plurality of internal memories to parallelise the input data. The first variant was area-efficient, making it suitable for hardware platforms with highly constrained resources. However, this replicated data throughout the internal memories, resulting in inefficient use of resources. The second variant rectified this inefficiency by storing each datum in only a single memory, allowing many more objects to be processed. However, this came at the expense of decreased parallelism and increased consumption of hardware resources.

Both microarchitectures suffer from restrictions on the number of objects they can support, based on the limitations of the hardware platform on which they will be implemented. These restrictions would reduce the practicality of the microarchitectures; spatial partitioning was proposed as a solution.

This divides the environment into cells using either the uniform grid or AABB tree algorithm. Each cell can be sent to the microarchitecture and processed sequentially, and the PCS of each cell can be processed by the narrow phase while another cell is being processed by the microarchitecture. Another advantage of spatial partitioning is its ability to reduce the undesirable computational complexities that may result from the use of the all-pairs algorithm.

After outlining spatial partitioning, the application of the microarchitectures to dynamic collision detection with deformable bodies was analysed. It was established that the current designs are directly applicable to these collision detection types, although some adaptations could be made to improve their performance.

The chapter concluded with a discussion of the design of the benchmarking framework. This framework uses a physics engine to accurately recreate scenarios from interactive applications, while a set of ten parameters allows the geometrical and physical properties of the objects to be adjusted. These parameters allow a single generic benchmark to become myriad practical benchmarks, creating a comprehensive benchmarking framework.

Chapter 4

Implementation

> *It is not always what we know or analyze* before *we make a decision that makes it a great decision. It is what we do* after *we make the decision to implement and execute it that makes it a good decision.*
>
> —C William Pollard (*The Soul of the Firm*), Fortune 500 CEO (1938–)

Implementation is the realisation of a design. It makes concrete the abstract details of a design and reduces the list of alternatives provided by a design into a single solution. This chapter details the realisations of all of the designs outlined in the previous chapter. It begins with an appraisal of the different platforms that could be used to implement the hybrid system's microarchitecture. Subsequently, implementations of the two microarchitectures are outlined, followed by an account of the spatial partitioning implementation. The chapter concludes with a discussion of the implementation of the benchmarking framework in software.

4.1 Target Platform Evaluation

As a first step towards implementing the hybrid system, it was necessary to consider potential platforms for the custom microarchitectures. The two primary options were FPGAs and ASICs.

Field-Programmable Gate Arrays (FPGAs) FPGAs are integrated circuits (ICs) that can be programmed 'in the field' after manufacture. They comprise programmable logic blocks linked via programmable interconnect. FPGAs are often used for prototyping as their programmability results in a low turn around time (TAT) of between one and four weeks [Zuchowski et al, 2002], permitting a multitude of designs to be rapidly tested. This programmability incurs significant performance, area and power consumption penalties, although these penalties are lower than for general-purpose hardware [Kumar et al, 2009].

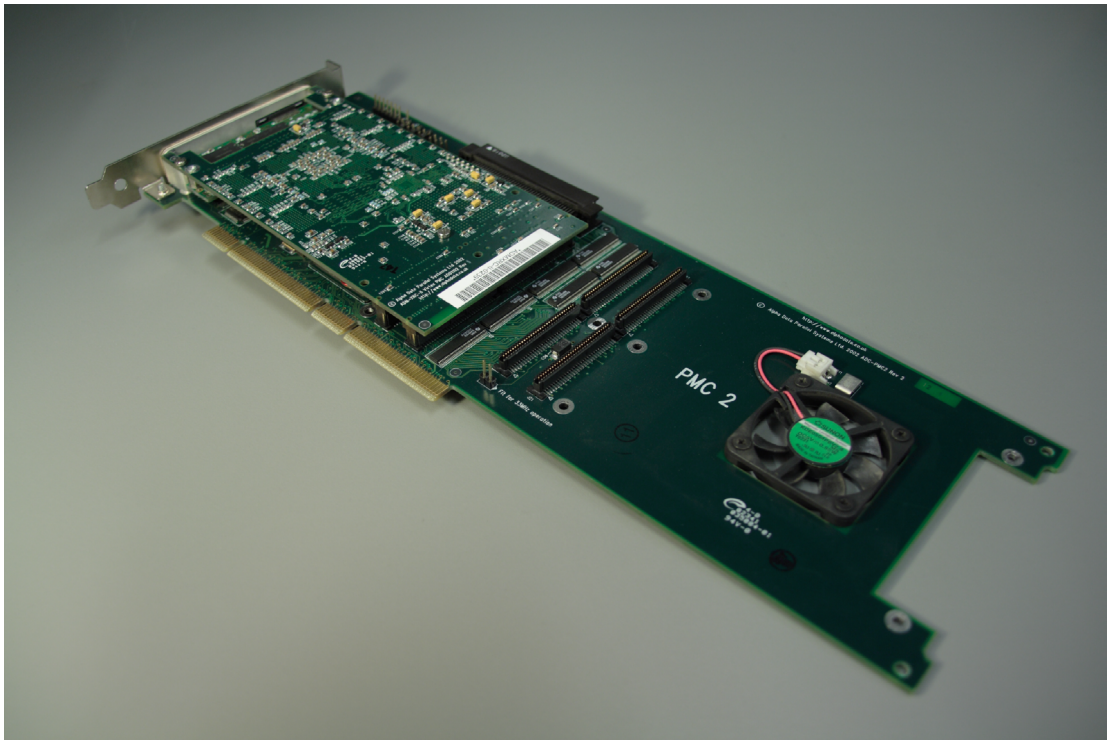
Application-Specific Integrated Circuits (ASICs) ASICs are ICs designed for specific applications, which cannot be programmed after manufacture. The TAT is significantly higher, since a set of masks must be created from the final design and sent to a manufacturer. This process can take between two and five months [Zuchowski et al, 2002]. In return for these considerable delays, performance is typically improved by a factor of three or four, area utilisation is decreased by a factor between twenty and thirty-five and dynamic power consumption is reduced by a factor of ten [Kuon and Rose, 2007]. The cost per unit is lower than for FPGAs, once non-recurring engineering (NRE) costs have been amortised. For these reasons, ASICs are often used for finalised designs that may have been prototyped on FPGAs.

Based on this analysis, FPGAs were selected as a target platform for the prototype implementation. Although their programmability incurs significant penalties, these were considered acceptable for the process of determining whether it is possible to accelerate collision detection utilising a custom microarchitecture. Moreover, an FPGA's programmability allows all of the microarchitectures to be implemented on a single IC at different times. This obviates the need to purchase or fabricate multiple ICs for the different designs.

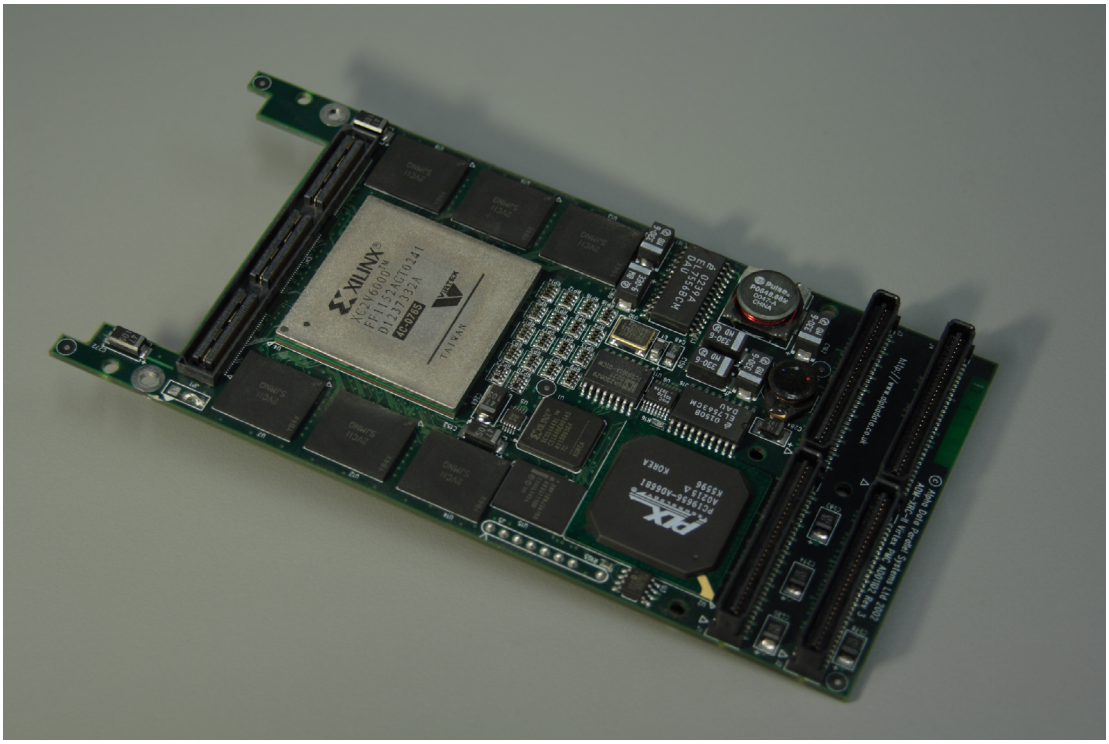
Although FPGAs are a good choice for prototyping, their significant disadvantages make them less beneficial in the longer term. For such long-term use, an ASIC would offer better performance, smaller die size and lower power consumption. Another alternative would be to implement the microarchitectures as fixed-function logic on commodity CPUs. CPUs offer advantages such as effectively having no communication delay between the software and the microarchitecture, coupled with the highest clock frequencies of all IC types. These advantages would provide the microarchitecture with significant gains. One more promising alternative would be to implement the microarchitectures as part of the fixed-function logic of commodity GPUs, due to the significant overlap between applications that use the GPU and those that use collision detection. A GPU implementation could also attempt to reuse some of the graphics data for performing collision detection, so that communication between the software and the GPU would be minimised in order to increase overall performance. However, there would still be significant communication delays and clock frequencies would be lower than those of a CPU implementation. Placing the microarchitecture on the CPU or GPU would be primarily interesting in the context of video game consoles, as the majority of applications that execute on these devices would use collision detection and the microarchitecture would rarely go unutilised. In addition, a single console design is typically maintained for many years, making it imperative for their designers to include any technologies that assist the consoles in exhibiting good performance throughout their lifespan.

Having decided on an FPGA as the target platform for the microarchitectures, it was necessary to select a particular FPGA model hosted on an appropriate FPGA coprocessor module that provided an interface between the software and the FPGA. Two different FPGAs and FPGA coprocessor modules were selected.

The first FPGA selected was the Xilinx Virtex-II XC2V6000 of speed grade four [Xilinx, 2007b], hosted on an Alpha Data ADM-XRC-II [Alpha Data, 2002b], as shown in Figure 4.1. This module was



(a)



(b)

Figure 4.1: **(a)** *Alpha Data ADC-PMC-2 hosting the Alpha Data ADM-XRC-II module.* **(b)** *Alpha Data ADM-XRC-II module.*

originally manufactured in 2001. The FPGA comprises 33 792 logic slices and 144 2.25 KiB memories for a total of 324 KiB of internal memory. The module is conntected to the CPU via the peripheral component interconnect (PCI) bus. It is implemented as a PCI mezzanine card (PMC) module, which fits into an Alpha Data ADC-PMC2 [Alpha Data, 2002a] PCI carrier, which translates the PCI bus signals on the PMC card for use on the PCI bus of the host computer.

The module comprises six independent 36-bit wide 1024 KiB banks of zero bus turnaround synchronous static RAM (ZBT SSRAM). Each bank has a clock frequency of 133.$\bar{3}$ MHz, resulting in a theoretical maximum bandwidth of 599.$\bar{9}$ MB/s, although this is dependent on the clock frequency attained by the microarchitecture. Data transfers between the software and the FPGA must use these memories to store the data being transferred and all reads and writes must pass through the FPGA. Consequently, requests from the software and the FPGA must be interleaved, which may result in some delay between the initiation and completion of a transfer. The PCI bus between the PCI carrier card and the host system is 64-bits wide with a clock frequency of 66.$\bar{6}$ MHz, allowing for a theoretical maximum bandwidth of 533.$\bar{3}$ MB/s. However, the PCI bus between the module and PCI carrier card is only 32-bits wide with a clock frequency of 66.$\bar{6}$ MHz, allowing for a theoretical maximum bandwidth of 266.$\bar{6}$ MB/s. Therefore, the theoretical maximum bandwidth over the PCI bus as a whole is 266.$\bar{6}$ MB/s. The connections between the module, its external memories and the host system are illustrated in Figure 4.2.

The second FPGA selected was the Xilinx Virtex-5 XC5VLX330 of speed grade two [Xilinx, 2009a], hosted on a DRC Accelium AC2030 [DRC Computer, 2008b], as shown in Figure 4.3. This module was originally manufactured in 2008. The FPGA comprises 51 840 logic slices and 288 4.5 KiB memories for a total of 1296 KiB of internal memory. The module connects to the CPU via the high-speed HyperTransport bus [HyperTransport Consortium, 2009]. HyperTransport has substantially lower latencies than PCI due to the direct connection of the CPU to the FPGA with no intermediate bridging.

The module comprises three types of memory. First, there are two 256 MiB banks of low-latency memory that are each 64-bits wide with a clock frequency of 200 MHz, allowing for a bandwidth of 2.8 GB/s. Second, there are two 4 GiB banks of module memory that are each 128-bits wide with a clock frequency of 200 MHz, allowing for a bandwidth of 6.4 GB/s. Finally, there are two 8 GiB banks of mainboard memory that are each 128-bits wide with a clock frequency of 200 MHz, allowing for a bandwidth of 6.4 GB/s. Data transfers between the software and the FPGA must use these memories to store the data being transferred and all reads and writes must pass through the FPGA. Consequently, requests from the software and the FPGA must be interleaved, which may result in some delay between the initiation and completion of a transfer. The HyperTransport bus between the CPU and the FPGA module is 64-bits wide with a clock frequency of 200 MHz, allowing for a theoretical maximum bandwidth of 9.6 GB/s. The connections between the module, its external memories and the host system are illustrated in Figure 4.4.

It was beneficial to use two different FPGAs and modules for a variety of reasons. Firstly, their differences are essential for demonstrating that the microarchitectures are sufficiently flexible to adapt to differing platforms. In addition, the considerable difference in the manufacture date of the two FPGA

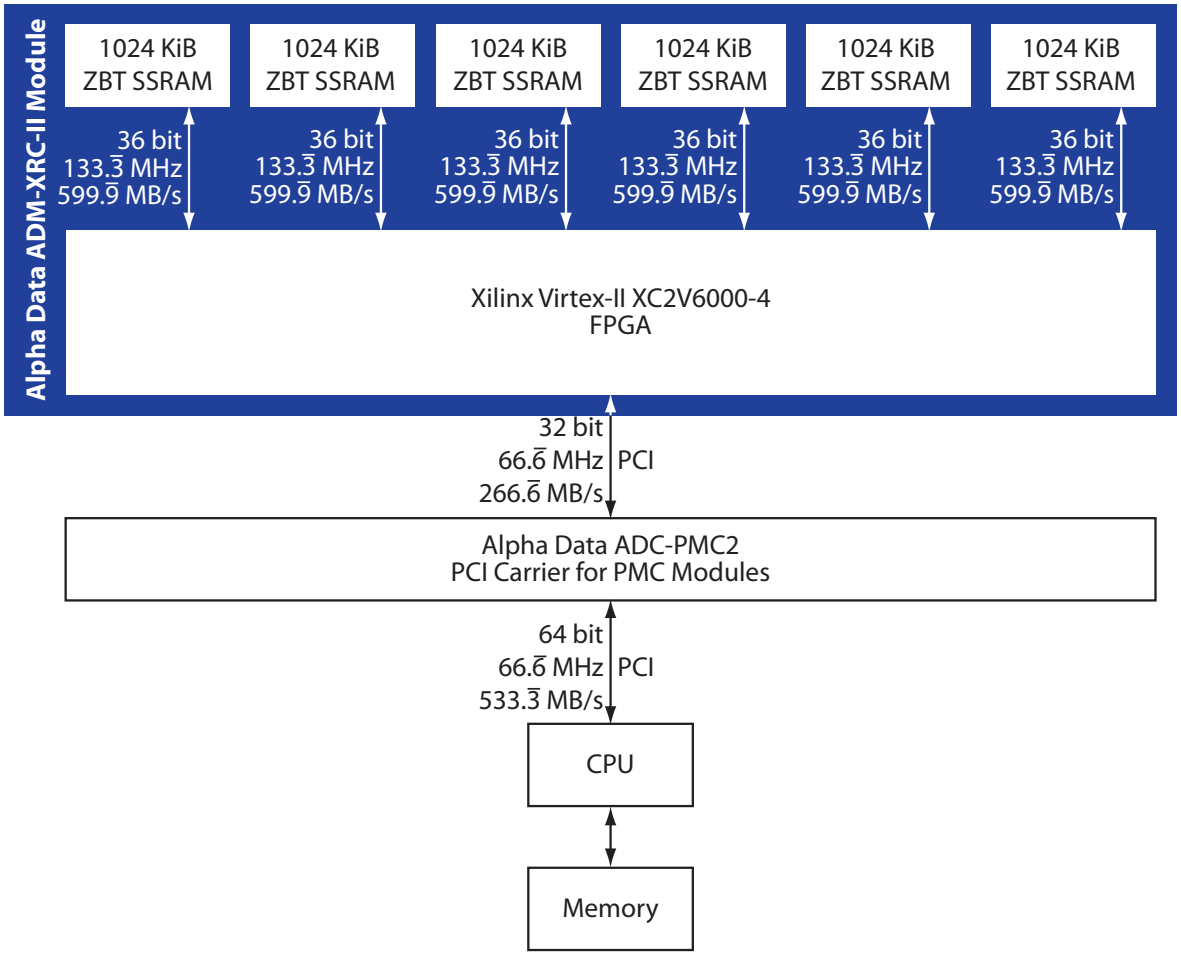


Figure 4.2: *Connections between the Alpha Data ADM-XRC-II, its external memories and the host system.*

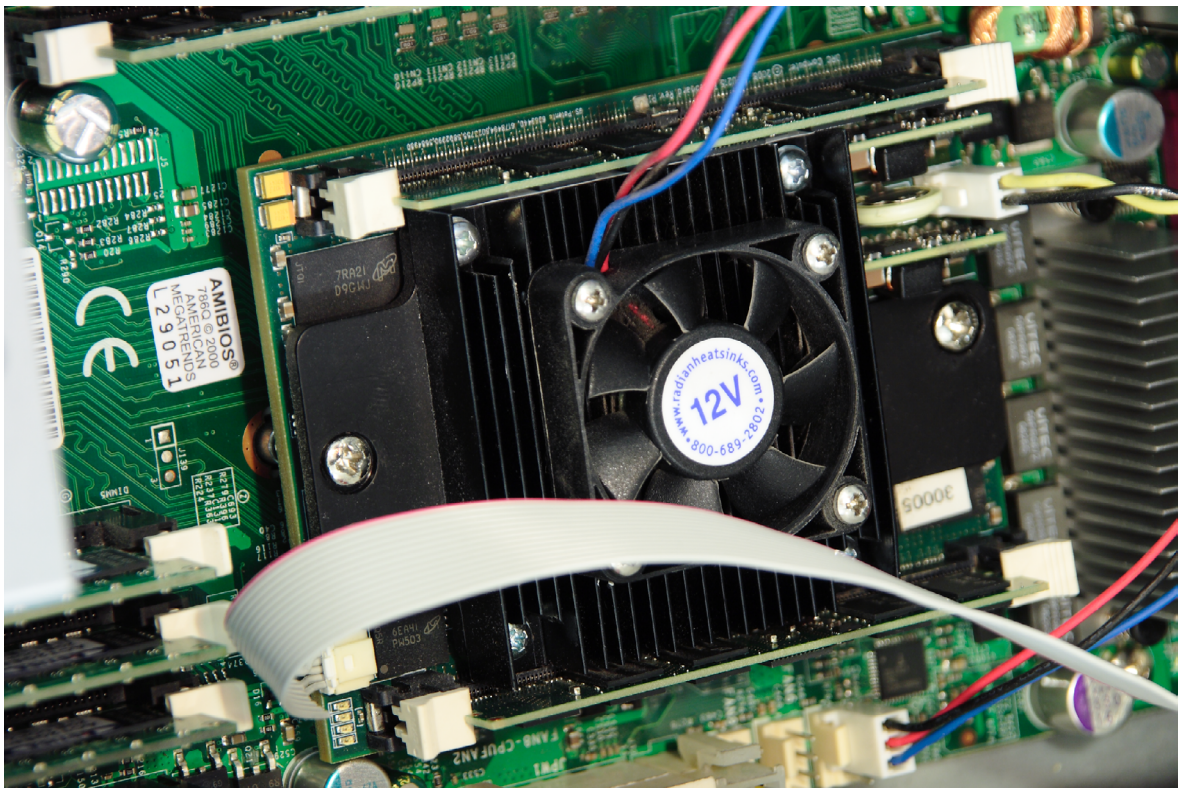


Figure 4.3: *DRC Accelium AC2030 module hosted in a DRC Accelium AD2311 workstation.*

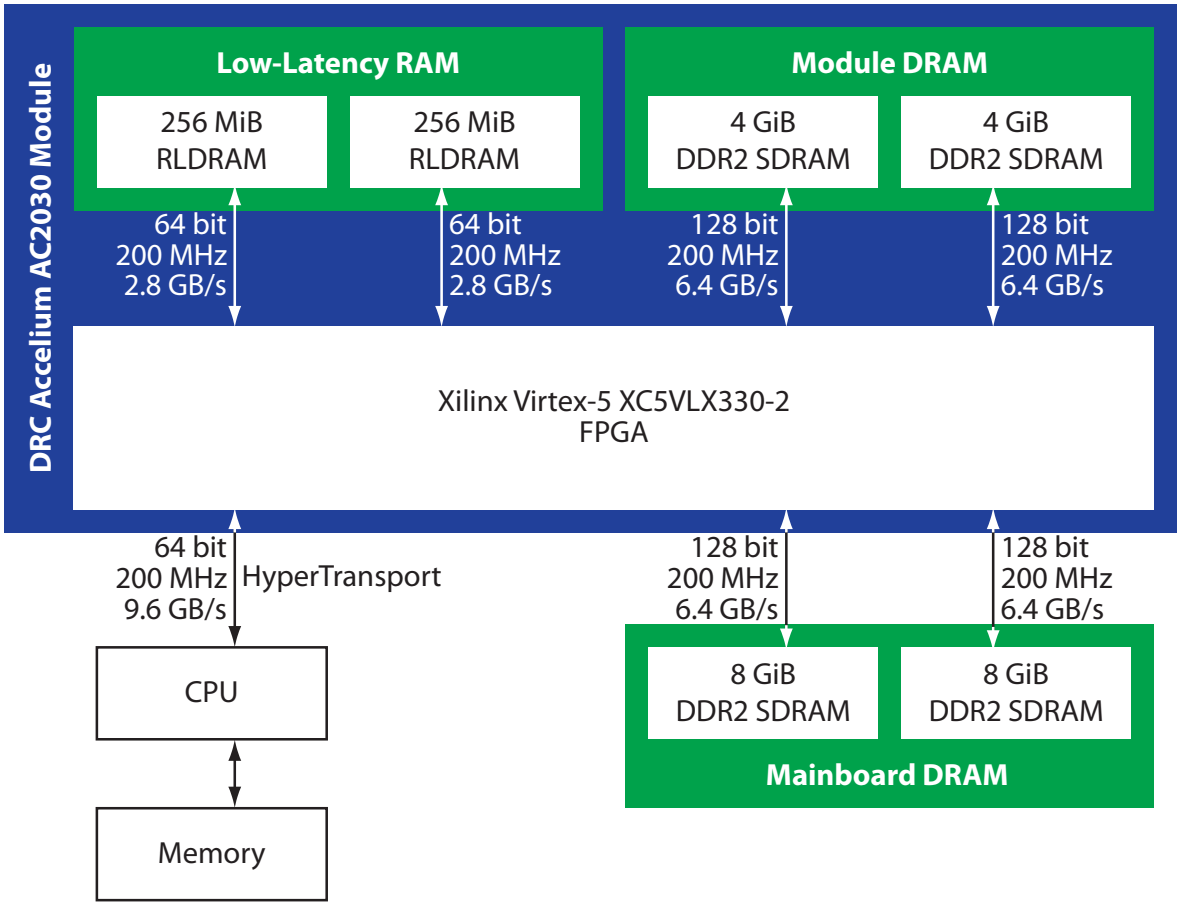


Figure 4.4: *Connections between the DRC Accelium AC2030, its external memories and the host system.*

modules can be used to demonstrate how the progression of technology influences the performance of the microarchitectures. Finally, the different modules allow for an examination of the impact of module features, such as differing communication buses and memory layouts. For these three reasons, it is evident that the use of two different FPGAs and modules is of significant benefit.

4.2 Area-Efficient Microarchitecture

Two implementations of the area-efficient microarchitecture were created. The first was implemented on the Xilinx Virtex-II FPGA, while the second was implemented on the Xilinx Virtex-5 FPGA. The two implementations share many details although there is considerable difference in key aspects of their implementation.

4.2.1 Xilinx Virtex-II

Microarchitecture As a first step towards implementation, it was necessary to examine the hardware API supplied by the vendor of the module as a rigid API could limit the functionality of the microarchitecture. However, Alpha Data supplied no API in order to allow all of the FPGA's logic to be used by the microarchitecture. The lack of an API meant that the interface code between the FPGA module and the FPGA, and between the CPU and the FPGA, would need to be developed for each microarchitecture. Such development is extremely time consuming and is of little significance to the microarchitecture. Consequently, the communication interface was constructed using the example 'memory' microarchitecture supplied by Alpha Data [2009]. This example was designed to provide the necessary interface logic to developers, while providing a means to override its inbuilt functionality with that of a custom microarchitecture. The example also provided access to all of the module's memories and allowed for the implementation of user-defined registers. It may have been possible to reduce resource utilisation by reimplementing the interface logic to access only the resources required by the microarchitecture, but tests determined that the example consumed few resources, making reimplementation of little benefit.

After examining the API, it was necessary to determine the optimal values of the variables m, p and d. Having determined that the limiting resource of the implementation was the availability of internal memories or block RAMs (BRAMs) [Wilton et al, 1995], m was set to sixteen, which was the highest possible value for the FPGA when hosting the microarchitecture. As all BRAMs were dual-ported, p was set to two, to allow for maximum resource utilisation. Finally, d was set to 512, as the depth of a single BRAM is 512 when each line is 36-bits wide. In addition, one set of buffer memories was implemented as two buffers would have effectively halved the value of m, resulting in significantly poorer performance.

Next, the communication interface had to be adapted for the FPGA module. This primarily consisted of selecting which external memories would store each data set. For the AABBs, the first three ZBT SSRAMs were used. Each AABB was stored as three minimum data at one address followed by three maximum data at the following address, allowing an AABB to be read in two clock cycles. By splitting

the AABBs across the ZBT SSRAMs, dataflow through the FPGA was maximised. The PCSes emitted by the microarchitecture were stored in the fourth ZBT SSRAM. The final two ZBT SSRAMs were left unutilised. Based on this memory scheme, the input and output data layouts shown in Figure 4.5 were created.

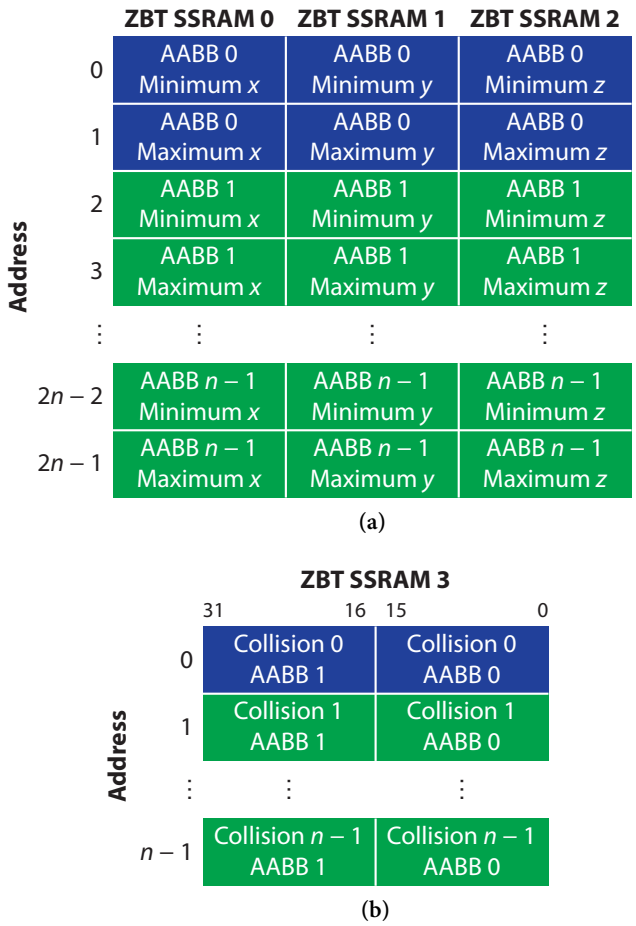


Figure 4.5: *Data layouts for the Xilinx Virtex-II implementation of the area-efficient microarchitecture.* **(a)** *Input.* **(b)** *Output.*

Some other adaptations and choices had to be made when implementing the communication interface. Firstly, due to the limitations of the PCI bus, the design alternative whereby the CPU memory would be accessed directly from the FPGA was not explored. It was also decided not to transfer the ADTs to the FPGA directly, as these contained large quantities of physics and collision detection data and the transfer of these large quantities would consume significant time. In addition, the communication interface included serialisation, so that the collisions were written to the single ZBT SSRAM and did not require $mp - 1$ banks of external memory. Each set of two FIFOs in this design was reduced to a single FIFO to minimise the number of BRAMs consumed. Interrupts were not used as Moll and Shand [1997] determined that PCI interrupts suffer from high and variable latencies.

For the number representation, single-precision floating-point was selected due to the advantages

outlined in Section 3.2. In addition, the 32-bit width of single-precision worked well with the 36-bit width of the BRAMs, although the width of the BRAMs could have been reduced if desired. The comparators had a latency of two cycles when implemented using Xilinx CORE Generator. In contrast, the fixed-point equivalents would have had latencies of one cycle and the float-to-fixed converter would have had a latency of six cycles. Although all elements were pipelined so that the effective latency was reduced, it was evident that the floating-point implementation would be more efficient. Similarly, the integer comparators would have had latencies of one cycle and the float-to-integer converter would also have had a latency of one cycle. This would result in better performance than the fixed-point representation, but there would have been no real benefit over the floating-point representation. Furthermore, use of either fixed-point or integer representations would have resulted in difficulties associated with reduced precision. Finally, the BRAMs were the limiting factor in terms of the microarchitecture implementation, so it was not advantageous to use the smaller fixed-point or integer comparators.

Software After finalising the microarchitecture for implementation on the Xilinx Virtex-II, it was necessary to consider the software. This primarily concerned the API [Alpha Data, 2009] supplied by the module vendor, Alpha Data, as this controlled all interactions between the software and the microarchitecture. It was necessary to examine this API in detail as its poor usage could result in poor overall performance. A study of the API revealed functionality for performing memory, register and interrupt operations.

Write Memory Writing data from the CPU to the FPGA's memories can be performed using either programmed input/output (PIO) or direct memory access (DMA). With PIO, all transactions are performed immediately, while DMA provides the option of performing transactions asynchronously or immediately. In the case of an immediate transaction, the end of the transaction essentially functions as a barrier, so that all functions subsequent to the transaction occur after its completion. The data for the transaction are specified as an array of 32-bit elements. The desired memory bank, initial address and number of data must also be specified. Multiple banks cannot be written simultaneously.

Read Memory Reading data from the FPGA's memories works analogously to the write memory functionality and provides the same options.

Write Register Writing data to an FPGA register is performed using PIO. The datum and register address must be specified.

Read Register Reading data from an FPGA register works analogously to the write register functionality and provides the same options.

Register Interrupt Handler An interrupt handler, which will execute if an interrupt is generated by the FPGA, can also be registered using the API.

In addition to this functionality, the API provides significant functionality for configuring and managing the FPGA and module. However, most of this functionality must be used in a defined pattern

during initialisation and termination of the system and it is, therefore, not germane to the adaptation of the communication interface.

It is clear from the API that the previously described memory layout may not be efficient in terms of software, as the data would need to be arranged into three separate arrays corresponding to the three ZBT SSRAMs and these arrays would need to be transferred in three separate transactions. This implementation decision was designed to maximise dataflow through the FPGA, but its effect on the overall system was not studied in detail and it may merit future investigation. However, any effect is likely to be of negligible impact.

Combining the Microarchitecture and the Software Based on the aforementioned implementation decisions, a nine-step procedure for using the FPGA to perform broad phase collision detection was designed:

1. Software: Update the AABBs based on the objects' current locations and rotations.
2. Software: Write all of the AABB x data into an array.
3. Software to FPGA: Write the AABB x data array to ZBT SSRAM bank 0 using an immediate-mode DMA transaction.
4. Software: Write all of the AABB y data into an array.
5. Software to FPGA: Write the AABB y data array to ZBT SSRAM bank 1 using an immediate-mode DMA transaction.
6. Software: Write all of the AABB z data into an array.
7. Software to FPGA: Write the AABB z data array to ZBT SSRAM bank 2 using an immediate-mode DMA transaction.
8. Software to FPGA: Write the number of AABBs transferred to the 'AABB Register', indicating that processing of the AABBs should commence.
9. FPGA: Process the AABBs.
10. Software to FPGA: Read the 'Collisions Register' to determine the size of the PCS. This step is repeated until the 'Collisions Register' contains a value other than −1.
11. Software to FPGA: Read the PCS into an array from ZBT SSRAM bank 3 using an immediate-mode DMA transaction.
12. Software: Transfer the PCS to the narrow phase.

This sequence is illustrated in Figure 4.6, which highlights that the software must initiate all communication with the FPGA.

Compilation After finalising all of these details, the entire microarchitecture was implemented in VHDL [IEEE, 2009]. It was synthesised, translated, mapped, placed and routed using Xilinx ISE 10.1.03 [Xilinx, 2008] so that it was possible to execute the entire microarchitecture on the FPGA. The design was optimised for speed. A complete list of the settings used can be found in Section B.1.

The complementary software implementation consisted of an adapted version of the benchmarking framework. This version transferred the list of AABBs from the Bullet Physics SDK to the FPGA at

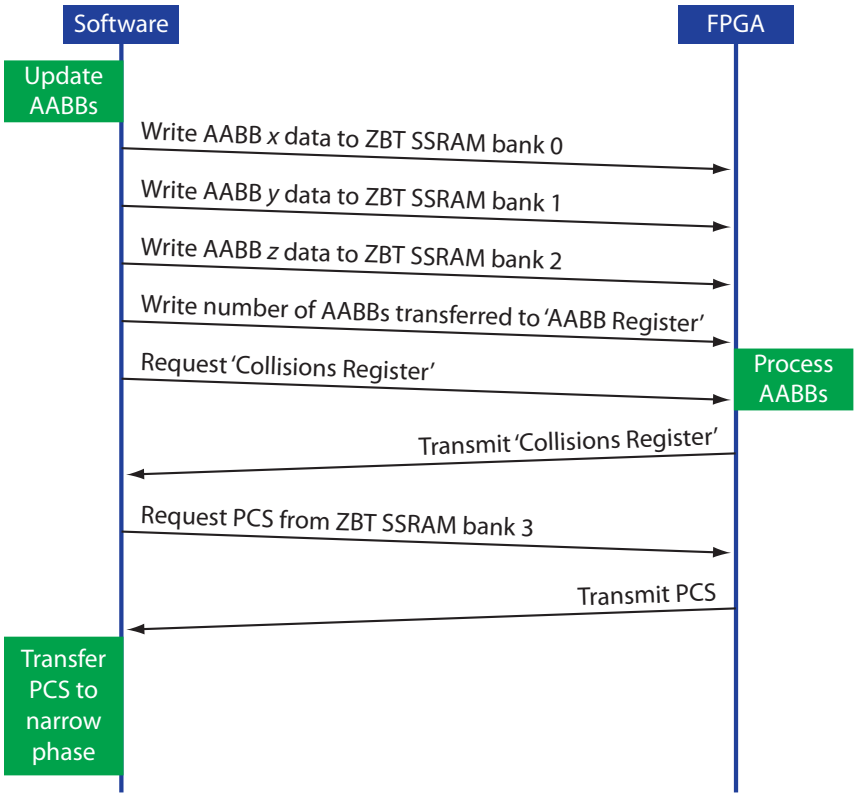


Figure 4.6: *Sequence diagram illustrating the steps involved in using the Xilinx Virtex-II implementation of the area-efficient microarchitecture.*

the start of each cycle, before subsequently reading the PCS from the FPGA. The PCS would then be supplied to the Bullet Physics SDK for narrow phase processing, in accordance with the previously outlined sequence. The design was compiled using Microsoft Visual C++ 2008 Service Pack 1 with Feature Pack [Microsoft, 2009a] and optimised for speed. A complete list of the settings used can be found in Section B.2.

4.2.2 Xilinx Virtex-5

Microarchitecture Unlike Alpha Data, DRC provides an unchangeable hardware core that must be included in all designs. This controls access to the memories and HyperTransport bus and allows interleaving of CPU and FPGA requests for these resources. To access the functionality of this core, DRC provides a hardware API [DRC Computer, 2009]. This API provides signals that can be manipulated in order to access all of the module's memory banks. These signals are manipulated in a predefined sequence:

1. Request access to the desired memory banks. Multiple memory banks can be requested and accessed simultaneously.
2. Wait until access to the memory banks is granted.
3. Specify the address, data, data length and whether the transaction is a read or write.

Transactions consuming multiple clock cycles can be controlled by the microarchitecture using a write enable signal, which is used to indicate whether the data is valid. The core can control the transaction using write ready and wait signals.

The majority of the implementation decisions made for the Xilinx Virtex-II implementation were maintained for the Xilinx Virtex-5 implementation. Despite increased BRAM resources, BRAMs remained the limiting factor and m had to be retained at sixteen, as the HyperTransport functionality consumed significant quantities of BRAMs. A higher value of m could almost certainly be achieved using a device from the SXT range of Xilinx Virtex FPGAs, which provide increased BRAM resources at the cost of decreased logic slices. If a Xilinx Virtex-5 XC5VSX240T were used, it is likely that $m = 64$ could be achieved, although this is dependent on the availability and arrangement of logic slices that would contain the non-BRAM resources. Similarly, a Xilinx Virtex-6 XC6VSX475T [Xilinx, 2009b] should permit an increase of m to 128. In the present Xilinx Virtex-5 implementation, d was increased to 1024 due to the Xilinx Virtex-5's larger BRAMs.

The communication interface was also adapted to support the different memories provided by the FPGA module. The AABBs were stored in the mainboard memories as these are of sufficient width to read an entire AABB in a single clock cycle. The three minimum data and one word of padding were stored in the first bank and the three maximum data and one word of padding were stored in the second bank. As for the Xilinx Virtex-II implementation, this memory layout maximised FPGA dataflow. Each PCS emitted by the microarchitecture was stored in the first module memory. The second module memory and the two low-latency memories were left unutilised. Based on this scheme, the input and output data layouts shown in Figure 4.7 were created.

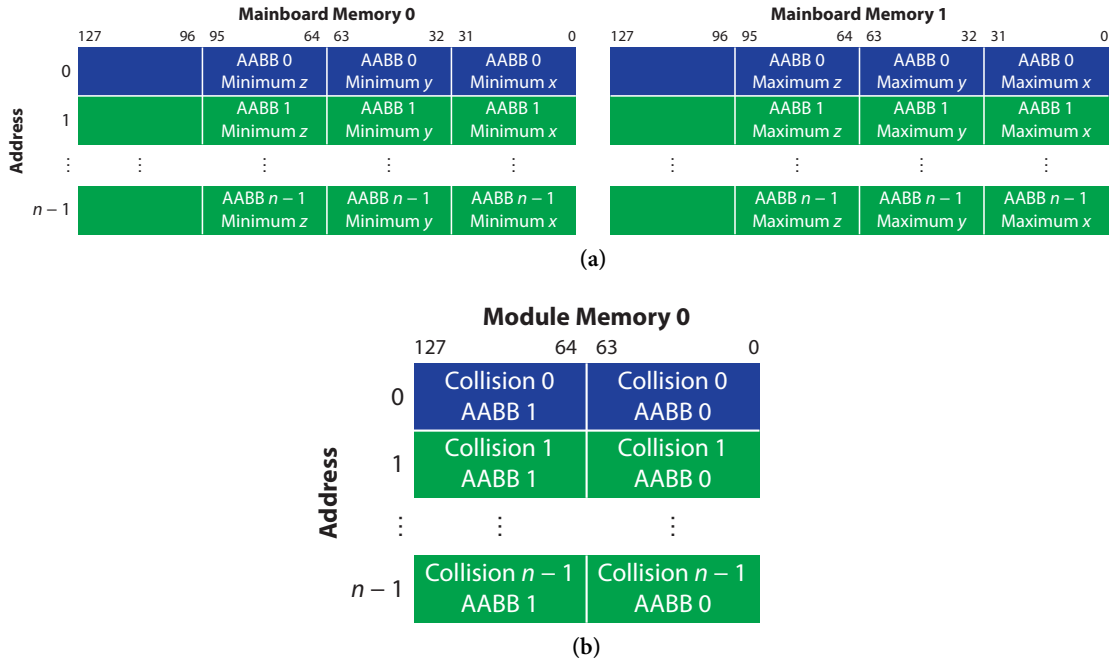


Figure 4.7: *Data layouts for the Xilinx Virtex-5 implementation of the area-efficient microarchitecture.* **(a)** *Input.* **(b)** *Output.*

The module memories are 128-bits wide and writing a single collision to a 128-bit word is inefficient. Therefore, to improve the performance, the serialiser was adapted to output four collisions on each clock cycle. This was done by dividing the round-robin scheduler into four segments, with each managing a quarter of the input data sources. One datum was taken from each of the four schedulers on each clock cycle to create one word of the output memory. If less than four collisions were available in a single clock cycle, the available collisions were combined until a set of four was created. After processing had completed, any remaining collisions were written to memory, even if they could not be combined to create a complete set.

Software Examining the DRC software API [DRC Computer, 2009], it was evident that this was very similar to Alpha Data's API. It primarily focused on providing functionality for memory and register operations.

Write Memory Writing data from the CPU to the FPGA's memories is always performed using asynchronous DMA. The data for the transaction are specified as an array of 8-bit, 32-bit or 64-bit elements. The destination memory bank, initial address and number of data must also be specified. Multiple banks cannot be written simultaneously.

Read Memory Reading data from the FPGA's memories works analogously to the write memory functionality and provides the same options.

Write Register Writing data to an FPGA register is performed by specifying the datum and register address.

Read Register Reading data from an FPGA register works analogously to the write register functionality and provides the same options.

Write Barrier As all writes to the FPGA's memories are performed using asynchronous DMA, there is the possibility that operations will be performed out of order. For some applications, this can cause the system to exhibit race conditions. The barrier solves this problem by ensuring that all previous write operations from a given CPU have completed before the execution of subsequent operations commences.

In addition to the aforementioned functionality, the API allows for configuration and management of the FPGA and module in a similar manner to the equivalent Alpha Data functionality. However, a notable difference between the Alpha Data and DRC APIs is that the DRC API provides no interrupt functionality, as interrupts are not supported by the module.

Combining the Microarchitecture and the Software Based on the aforementioned implementation decisions, the procedure for using the Xilinx Virtex-5 was adapted from that of the Xilinx Virtex-II:

1. Software: Update the AABBs based on the objects' current locations and rotations.
2. Software: Write all of the AABB minimum data into an array.
3. Software to FPGA: Write the AABB minimum data array to mainboard memory bank 0.
4. Software: Write all of the AABB maximum data into an array.
5. Software to FPGA: Write the AABB maximum data array to mainboard memory bank 1.
6. Software: Execute a barrier to ensure all prior data has been written to the FPGA memory banks.
7. Software to FPGA: Write the number of AABBs transferred to the 'AABB Register', indicating that processing of the AABBs should commence.
8. FPGA: Process the AABBs.
9. Software to FPGA: Read the 'Collisions Register' to determine the size of the PCS. This step is repeated until the 'Collisions Register' contains a value other than −1.
10. Software to FPGA: Read the PCS into an array from module memory bank 0.
11. Software: Transfer the PCS to the narrow phase.

This sequence is illustrated in Figure 4.8. As for the Xilinx Virtex-II implementation, the software must initiate all communication with the FPGA.

Compilation The entire microarchitecture was implemented in VHDL [IEEE, 2009] with some Verilog [IEEE, 2006] used for interfacing with the DRC Accelium AC2030. It was synthesised, translated, mapped, placed and routed using Xilinx ISE 10.1.03 [Xilinx, 2008] so that it was possible to execute the entire microarchitecture on the FPGA. The design was optimised for speed. A complete list of the settings used can be found in Section B.1.

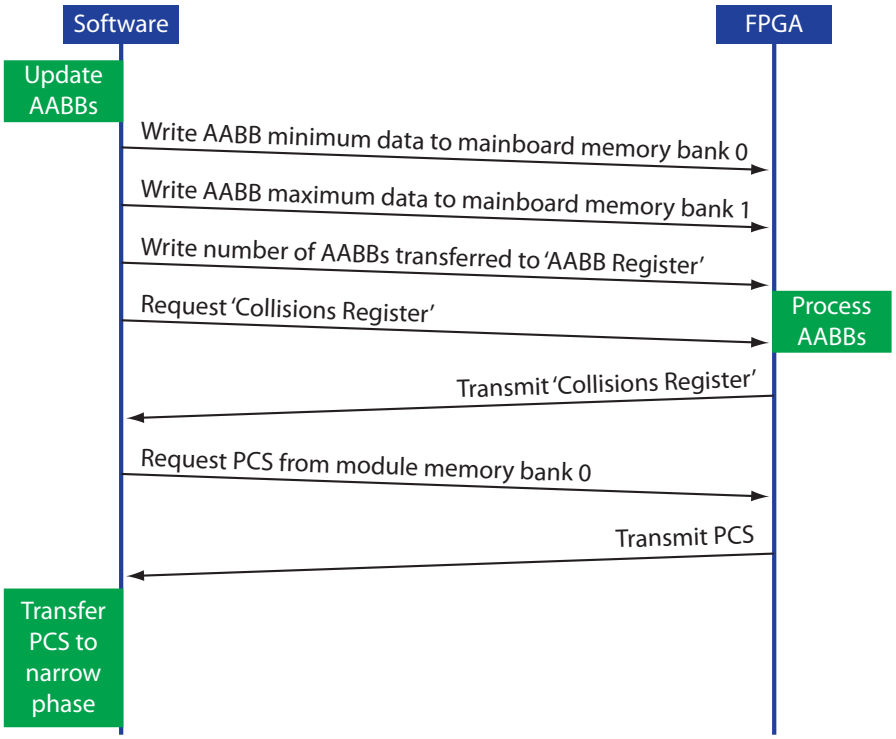


Figure 4.8: *Sequence diagram illustrating the steps involved in using the Xilinx Virtex-5 implementation of the area-efficient microarchitecture.*

The complementary software implementation remained the same as for the Xilinx Virtex-II implementation, but the code for accessing the Alpha Data ADM-XRC-II's memories was adapted to access the DRC Accelium AC2030's memories instead.

4.3 Many-Object Microarchitecture

The many-object microarchitecture was only implemented on the Xilinx Virtex-5 as its considerable size prevented it from fitting on the smaller Xilinx Virtex-II. The same implementation decisions were made as for the Xilinx Virtex-5 implementation of the area-efficient microarchitecture, and the existing complementary software implementation was used with no adaptations. Consequently, the procedure for using the FPGA to perform broad phase collision detection remained the same.

Despite not being able to increase m above sixteen, there were sufficient BRAMs to duplicate the buffer using the design alternative outlined in Section 3.4. With this alternative, the AABBs of one cell would be read while another was being processed. This optimisation was not implemented as it would have increased resource utilisation significantly, which would be liable to reduce the FPGA's clock frequency and, consequently, overall performance. It should, however, be beneficial for an ASIC, CPU or GPU implementation, where resource utilisation tends to have a lesser impact on clock frequency.

4.4 Spatial Partitioning

Microarchitecture When adapting the microarchitectures to support spatial partitioning, the same decisions were made as before. The only changes required were in the communication interfaces. The AABBs were stored in the same memories as for the previous microarchitectures but with a 'not a number' (NaN) separating each cell. The storage scheme for the PCSes required additional, FPGA-module–specific adaptation.

In the Xilinx Virtex-II implementation, this adaptation involved writing the PCS corresponding to the first cell to the third ZBT SSRAM, the PCS corresponding to the second cell to the fourth ZBT SSRAM and the PCS corresponding to the third cell to the fifth ZBT SSRAM. Whenever the software read the PCS corresponding to the first cell and the PCS corresponding to the fourth cell became available, the new PCS would be written to the third ZBT SSRAM, overwriting its earlier contents. This pattern continued until all PCSes were written to the ZBT SSRAMs. With this updated implementation, all of the Alpha Data ADM-XRC-II's ZBT SSRAMs were utilised. Based on this memory scheme, the updated input and output data layouts shown in Figure 4.9 were created.

In the Xilinx Virtex-5 implementations, this adaptation involved writing the PCS corresponding to the first cell to the first module memory and the PCS corresponding to the second cell to the second module memory. Whenever the software read the PCS corresponding to the first cell and the PCS corresponding to the third cell became available, the new PCS would be written to the first module memory, overwriting its earlier contents. This pattern continued until all PCSes were written to the module memories. With this updated implementation, all of the DRC Accelium AC2030's mainboard and module memories were utilised, but its two low-latency memories remained unutilised. This updated scheme resulted in the input and output data layouts shown in Figure 4.10.

The VHDL buffer control logic for the area-efficient and many-object microarchitectures, which roughly corresponds to Algorithms 3.2 and 3.3, is listed in Appendix C. This code forms the primary contribution of the microarchitectures in terms of algorithm development.

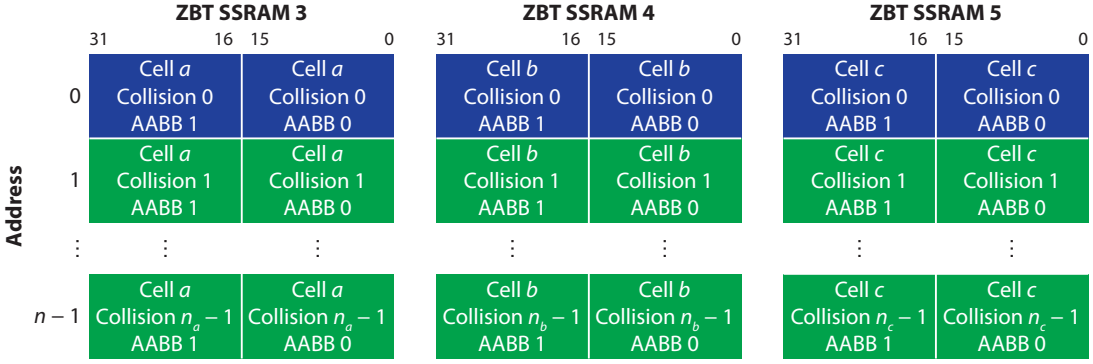
Software The complementary software implementation was adapted to support the spatial partitioning algorithms. These algorithms were not parallelised for multi-core processors. Further, it was not possible to perform narrow phase collision detection on individual PCSes as they arrived from the FPGA, as the design of the Bullet Physics SDK precluded the overlapping of its broad and narrow phases. If the Bullet Physics SDK were substantially overhauled to support this overlapping, it is likely that the overall execution time would reduce, since the software would spend less time waiting for the FPGA to compute the PCSes.

Combining the Microarchitecture and the Software Based on the aforementioned modifications, the prior procedure for using the Xilinx Virtex-II FPGA was adapted to support spatial partitioning:

1. Software: Update the AABBs based on the objects' current locations and rotations.
2. Software: Write all of the AABB x data into an array.

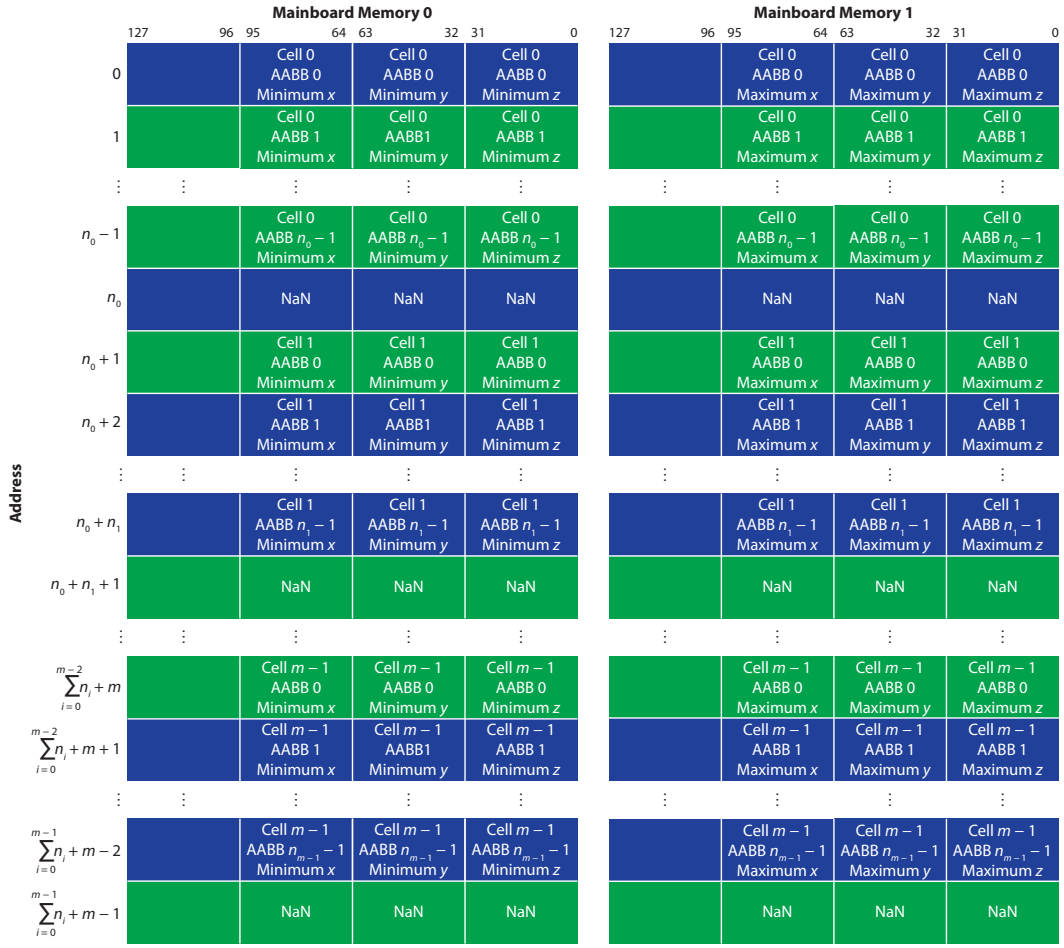
Address	ZBT SSRAM 0	ZBT SSRAM 1	ZBT SSRAM 2
0	Cell 0 AABB 0 Minimum x	Cell 0 AABB 0 Minimum y	Cell 0 AABB 0 Minimum z
1	Cell 0 AABB 0 Maximum x	Cell 0 AABB 0 Maximum y	Cell 0 AABB 0 Maximum z
2	Cell 0 AABB 1 Minimum x	Cell 0 AABB1 Minimum y	Cell 0 AABB 1 Minimum z
3	Cell 0 AABB 1 Maximum x	Cell 0 AABB 1 Maximum y	Cell 0 AABB 1 Maximum z
⋮	⋮	⋮	⋮
$2n_0 - 2$	Cell 0 AABB $n_0 - 1$ Minimum x	Cell 0 AABB $n_0 - 1$ Minimum y	Cell 0 AABB $n_0 - 1$ Minimum z
$2n_0 - 1$	Cell 0 AABB $n_0 - 1$ Maximum x	Cell 0 AABB $n_0 - 1$ Maximum y	Cell 0 AABB $n_0 - 1$ Maximum z
$2n_0$	NaN	NaN	NaN
$2n_0 + 1$	Cell 1 AABB 0 Minimum x	Cell 1 AABB 0 Minimum y	Cell 1 AABB 0 Minimum z
$2n_0 + 2$	Cell 1 AABB 0 Maximum x	Cell 1 AABB 0 Maximum y	Cell 1 AABB 0 Maximum z
$2n_0 + 3$	Cell 1 AABB 1 Minimum x	Cell 1 AABB1 Minimum y	Cell 1 AABB 1 Minimum z
$2n_0 + 4$	Cell 1 AABB 1 Maximum x	Cell 1 AABB 1 Maximum y	Cell 1 AABB 1 Maximum z
⋮	⋮	⋮	⋮
$2n_0 + 2n_1 - 1$	Cell 1 AABB $n_1 - 1$ Minimum x	Cell 1 AABB $n_1 - 1$ Minimum y	Cell 1 AABB $n_1 - 1$ Minimum z
$2n_0 + 2n_1$	Cell 1 AABB $n_1 - 1$ Maximum x	Cell 1 AABB $n_1 - 1$ Maximum y	Cell 1 AABB $n_1 - 1$ Maximum z
$2n_0 + 2n_1 + 1$	NaN	NaN	NaN
⋮	⋮	⋮	⋮
$\sum_{i=0}^{m-2} 2n_i + m$	Cell $m-1$ AABB 0 Minimum x	Cell $m-1$ AABB 0 Minimum y	Cell $m-1$ AABB 0 Minimum z
$\sum_{i=0}^{m-2} 2n_i + m + 1$	Cell $m-1$ AABB 0 Maximum x	Cell $m-1$ AABB 0 Maximum y	Cell $m-1$ AABB 0 Maximum z
$\sum_{i=0}^{m-2} 2n_i + m + 2$	Cell $m-1$ AABB 1 Minimum x	Cell $m-1$ AABB1 Minimum y	Cell $m-1$ AABB 1 Minimum z
$\sum_{i=0}^{m-2} 2n_i + m + 3$	Cell $m-1$ AABB 1 Maximum x	Cell $m-1$ AABB 1 Maximum y	Cell $m-1$ AABB 1 Maximum z
⋮	⋮	⋮	⋮
$\sum_{i=0}^{m-1} 2n_i + m - 3$	Cell $m-1$ AABB $n_{m-1} - 1$ Minimum x	Cell $m-1$ AABB $n_{m-1} - 1$ Minimum y	Cell $m-1$ AABB $n_{m-1} - 1$ Minimum z
$\sum_{i=0}^{m-1} 2n_i + m - 2$	Cell $m-1$ AABB $n_{m-1} - 1$ Maximum x	Cell $m-1$ AABB $n_{m-1} - 1$ Maximum y	Cell $m-1$ AABB $n_{m-1} - 1$ Maximum z
$\sum_{i=0}^{m-1} 2n_i + m - 1$	NaN	NaN	NaN

(a)

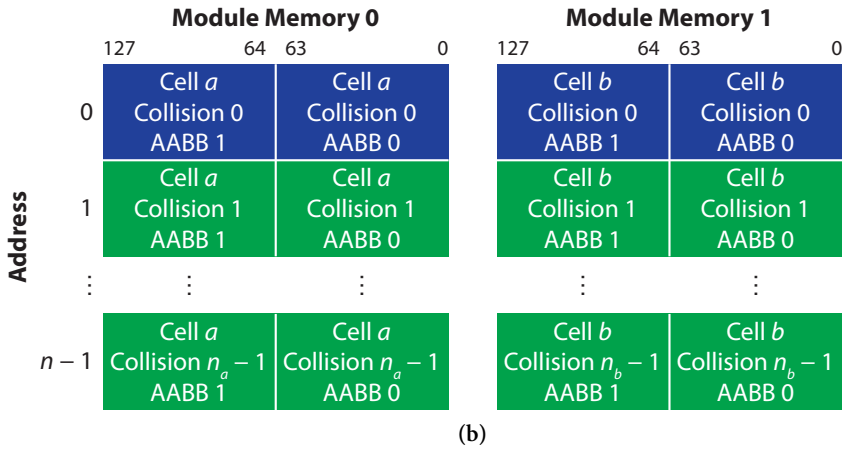


(b)

Figure 4.9: *Data layouts for the Xilinx Virtex-II implementation of the area-efficient microarchitecture supporting spatial partitioning.* **(a)** *Input.* **(b)** *Output.*



(a)



(b)

Figure 4.10: *Data layouts for the Xilinx Virtex-5 implementations of the microarchitectures supporting spatial partitioning.* **(a)** *Input.* **(b)** *Output.*

3. Software to FPGA: Write the AABB x data array to ZBT SSRAM bank 0 using an immediate-mode DMA transaction.
4. Software: Write all of the AABB y data into an array.
5. Software to FPGA: Write the AABB y data array to ZBT SSRAM bank 1 using an immediate-mode DMA transaction.
6. Software: Write all of the AABB z data into an array.
7. Software to FPGA: Write the AABB z data array to ZBT SSRAM bank 2 using an immediate-mode DMA transaction.
8. Software to FPGA: Write the number of cells transferred to the 'Cell Register', indicating that processing of the AABBs should commence.
9. FPGA: Process cell 0's AABBs.
10. Software to FPGA: Read the 'Collisions 0 Register' to determine the size of the PCS corresponding to cell 0. This step is repeated until the 'Collisions 0 Register' contains a value other than –1.
11. Software to FPGA: Read the PCS into an array from ZBT SSRAM bank 3 using an immediate-mode DMA transaction.
12. FPGA: Process cell 1's AABBs.
13. Software to FPGA: Read the 'Collisions 1 Register' to determine the size of the PCS corresponding to cell 1. This step is repeated until the 'Collisions 1 Register' contains a value other than –1.
14. Software to FPGA: Read the PCS into an array from ZBT SSRAM bank 4 using an immediate-mode DMA transaction.
15. FPGA: Process cell 2's AABBs.
16. Software to FPGA: Read the 'Collisions 2 Register' to determine the size of the PCS corresponding to cell 2. This step is repeated until the 'Collisions 2 Register' contains a value other than –1.
17. Software to FPGA: Read the PCS into an array from ZBT SSRAM bank 5 using an immediate-mode DMA transaction.

18. Repeat steps nine to seventeen until all PCSes are read.
19. Software: Transfer the PCSes to the narrow phase.

In this sequence, the FPGA processing of the AABBs can be overlapped with the reading of a collisions register and the corresponding PCS by the software. The sequence is illustrated in Figure 4.11.

Similar adaptations were made to the Xilinx Virtex-5 sequence to support spatial partitioning:

1. Software: Update the AABBs based on the objects' current locations and rotations.
2. Software: Write all of the AABB minimum data into an array.
3. Software to FPGA: Write theAABB minimum data array to mainboard memory bank 0.
4. Software: Write all of the AABB maximum data into an array.
5. Software to FPGA: Write theAABB maximum data array to mainboard memory bank 1.
6. Software: Execute a barrier to ensure all prior data has been written to the FPGA memory banks.
7. Software to FPGA: Write the number of cells transferred to the 'Cell Register', indicating that processing of the AABBs should commence.
8. FPGA: Process cell 0's AABBs.
9. Software to FPGA: Read the 'Collisions 0 Register' to determine the size of the PCS corresponding to cell 0. This step is repeated until the 'Collisions 0 Register' contains a value other than −1.
10. Software to FPGA: Read the PCS into an array from module memory bank 0.
11. FPGA: Process cell 1's AABBs.
12. Software to FPGA: Read the 'Collisions 1 Register' to determine the size of the PCS corresponding to cell 1. This step is repeated until the 'Collisions 1 Register' contains a value other than −1.
13. Software to FPGA: Read the PCS into an array from module memory bank 1.
14. Repeat steps eight to thirteen until all PCSes are read.
15. Software: Transfer the PCSes to the narrow phase.

As for the equivalent Xilinx Virtex-II sequence, the FPGA processing of the AABBs can be overlapped with the reading of a collisions register and the corresponding PCS by the software. This sequence is illustrated in Figure 4.12.

4.5 Benchmarking Framework

The benchmarking framework was implemented in C++ [ISO and IEC, 2003], with the uniform and normal probability distributions using the relevant classes from the Technical Report 1 (TR1) [ISO and IEC, 2005] additions to the C++ Standard Template Library (STL). Additional data structures were provided by Boost 1.36.0 [Boost, 2008] and Boost 1.40.0 [Boost, 2009]. All floating-point numbers were implemented as single precision, but these could be changed to double precision using a preprocessor directive. The Bullet Physics SDK 2.75 [Coumans, 2009] was selected to provide physics simulation, due to its status as a mature, open source, cross-platform engine featuring collision detection and physics, optional multithreaded execution and a PlayStation 3 port. These features were desirable as they expanded the flexibility of the framework. The code was compiled using G++ 4.2.4

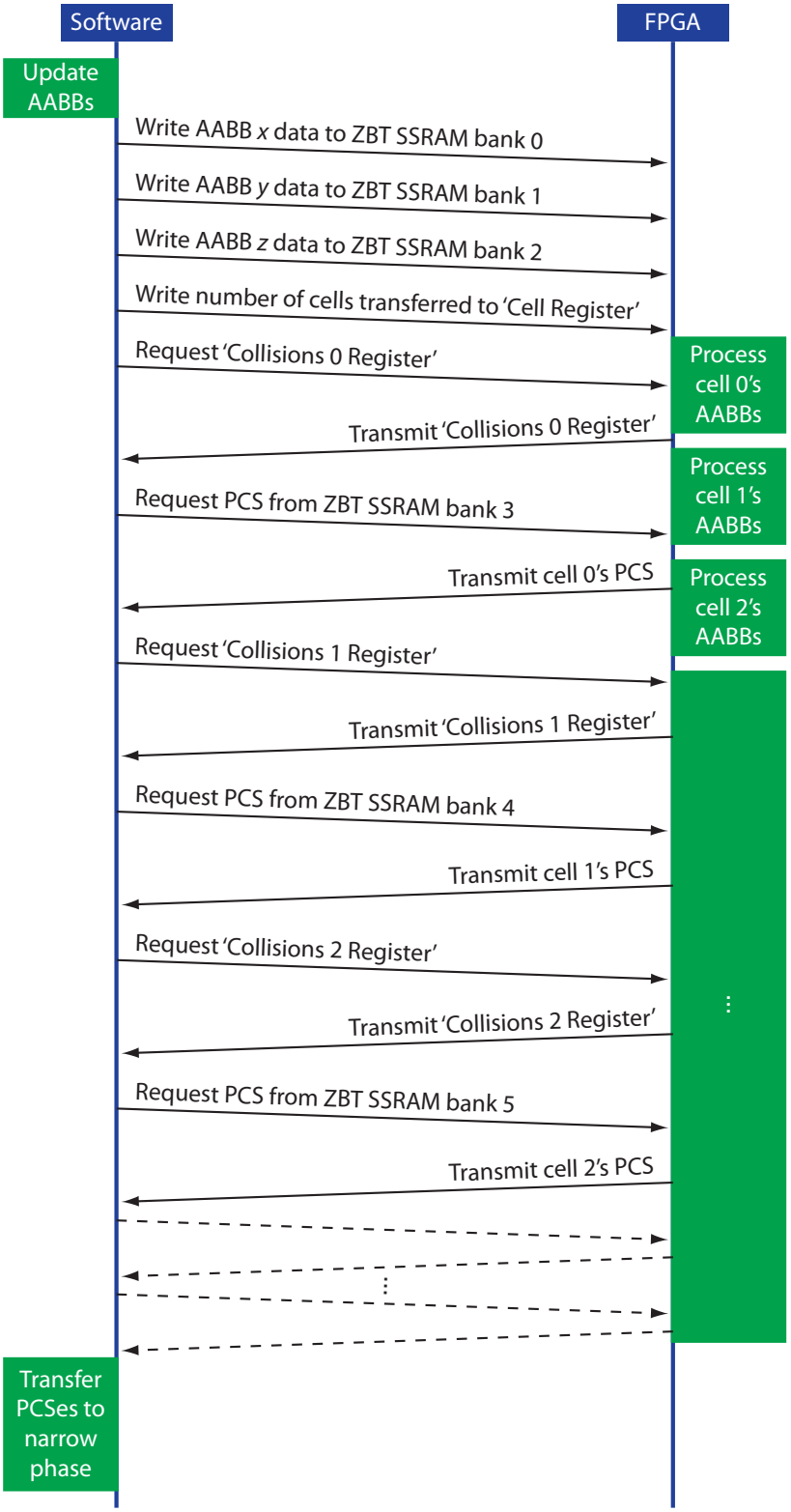


Figure 4.11: *Sequence diagram illustrating the steps involved in using the Xilinx Virtex-II implementation of the area-efficient microarchitecture supporting spatial partitioning.*

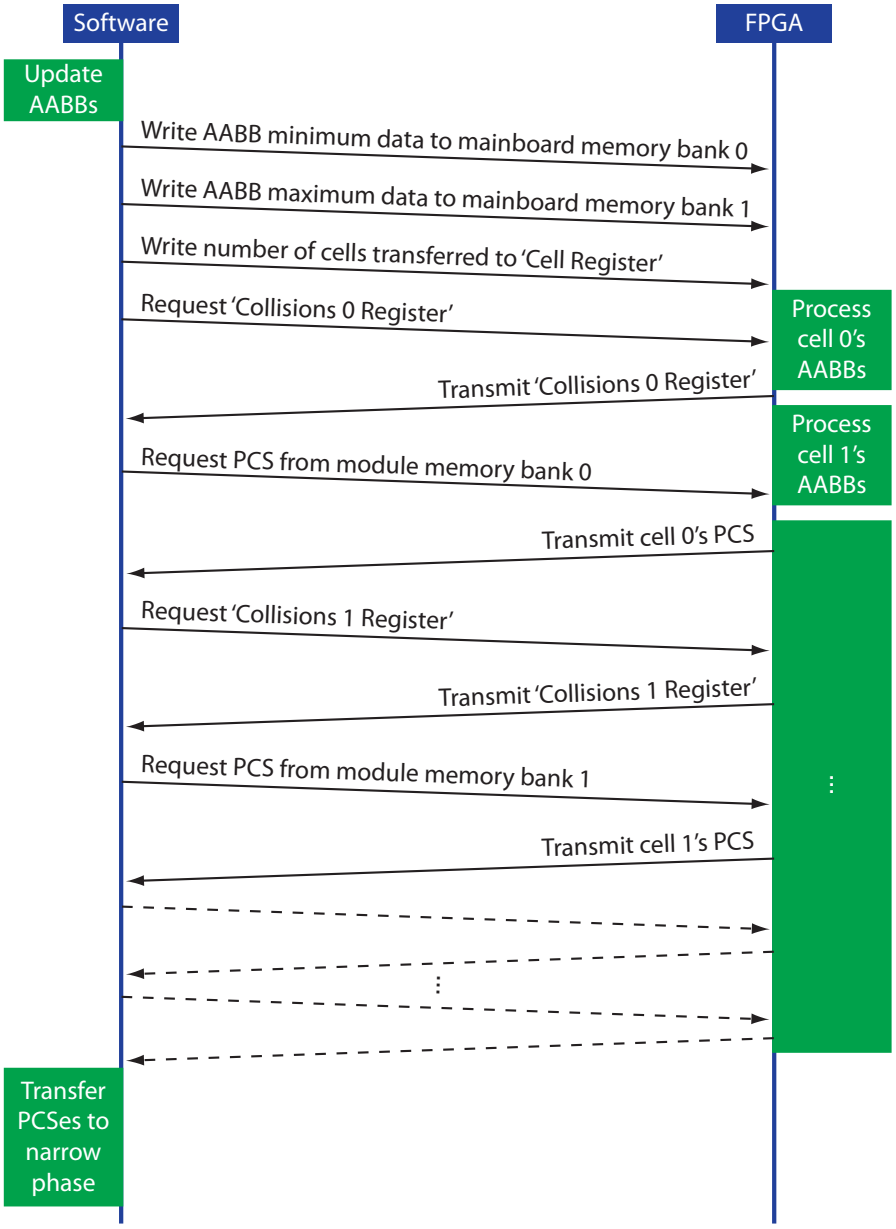


Figure 4.12: *Sequence diagram illustrating the steps involved in using the Xilinx Virtex-5 implementations of the microarchitectures supporting spatial partitioning.*

[Free Software Foundation, 2008] and Microsoft Visual C++ 2008 Service Pack 1 with Feature Pack [Microsoft, 2009a] and optimised for speed. A complete list of settings are provided in Section B.2.

To assist users in modifying the parameters, it was decided to create a graphical user interface (GUI) comprising dialogue boxes with appropriate parameter controls. These were created using the open source, cross-platform wxWidgets 2.8.7.1 [Smart et al, 2007] and wxWidgets 2.8.10 [Smart et al, 2009]. The dialogue boxes are illustrated in Figure 4.13. A graphical display, enabling users to visualise the effect of each parameter, was created using OpenGL [Segal and Akeley, 2009]. Figures 3.7 to 3.16 contain screenshots of this OpenGL visualisation.

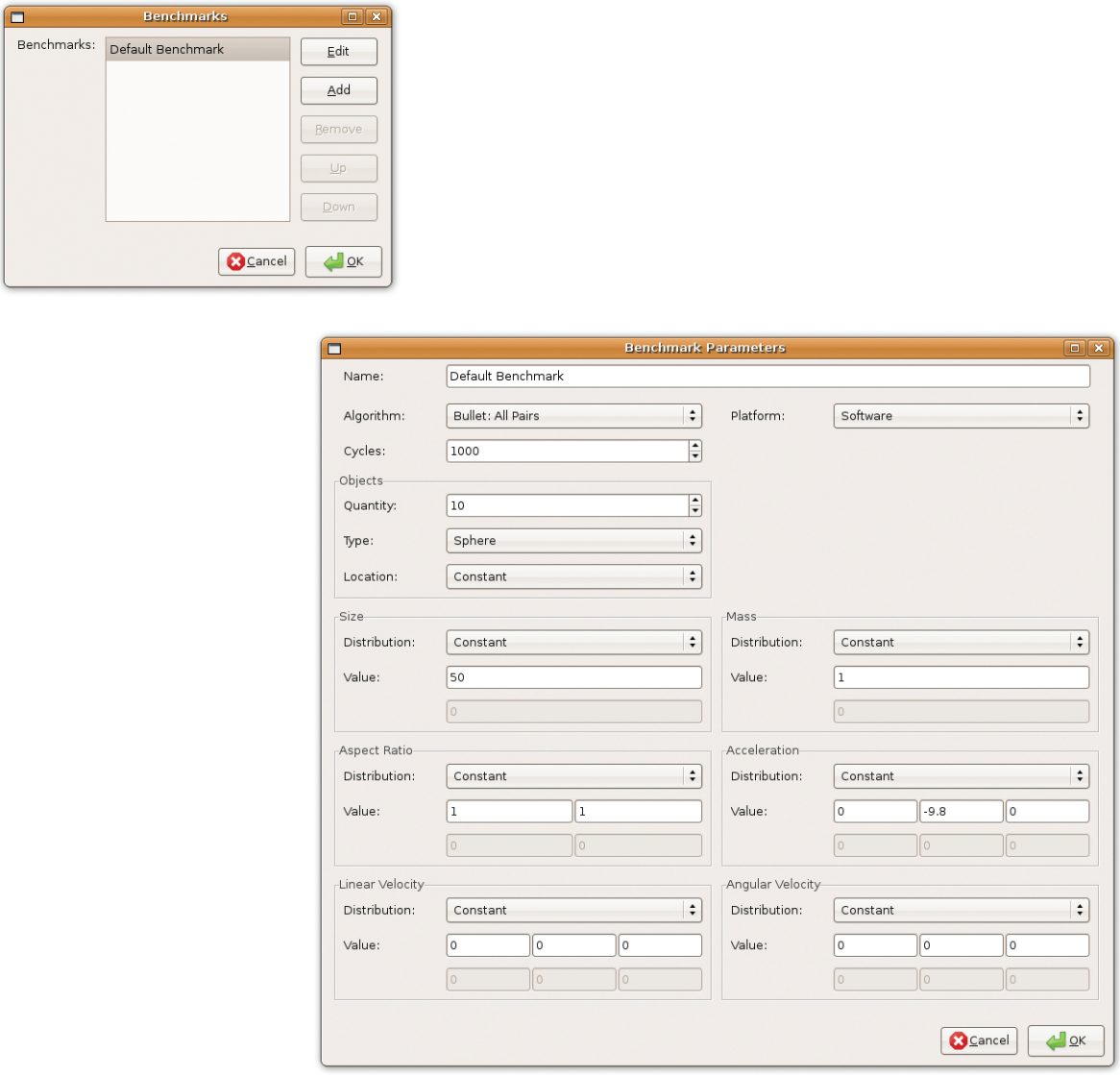


Figure 4.13: *Dialogue boxes for controlling the parameters of the benchmarking framework.*

4.5.1 Collision Detection

The implemented collision detection system was based on the one provided with the Bullet Physics SDK. For comparison purposes, it was possible to change the broad phase algorithm, although the narrow phase algorithm remained constant. The three broad phase algorithm options were all-pairs (referred to as 'simple' by the Bullet Physics SDK), incremental SAP [Cohen et al, 1995] and the custom dynamic bounding volume tree (DBVT), which used two AABB trees with one storing the static objects and the other storing the dynamic objects, with objects migrating between trees throughout the simulation as appropriate. For SAP, a fixed-point implementation was used for less than 16 384 objects, including those objects comprising the enclosing cube, while its floating-point equivalent was used for more than 16 384 objects, as per the best practices outlined in the Bullet Physics SDK documentation.

The Bullet Physics SDK has three phases – a broad phase, a midphase and a narrow phase. The midphase and narrow phase can be considered to comprise Hubbard's narrow phase. The midphase performs culling on complex collision shapes, such as compound shapes and concave triangle meshes. It employs an AABB tree accessed using stackless tree traversal. The narrow phase performs the final collision check using data from the broad phase and midphase. It performs distance, penetration or time-of-impact queries. Primitive objects such as spheres, cuboids, cylinders and cones are tested using the appropriate primitive collision test. For more complex objects, the GJK [Gilbert et al, 1988; Gilbert and Foo, 1990] convex distance calculation algorithm is used. The expanding polytope algorithm (EPA) [van den Bergen, 2001] provides the penetration depth.

4.5.2 Measurement

It was fundamental that the execution times of the different collision detection phases were recorded accurately, so high-resolution hardware performance counters were employed to achieve maximum precision. These timers are located in the CPU and are accessed using instructions such as the `rdtsc` [Intel, 2009] found in the x86 architecture. The timer code was integrated into the framework with sufficient flexibility to enable the benchmarking of algorithms executing in software or on an external coprocessor.

4.5.3 Application Programming Interface (API)

The framework was implemented with extensibility in mind, to facilitate algorithm and application developers to analyse a variety of collision detection algorithms. To this end, an API was created to enable the retrieval of geometry data from the framework and for supplying the framework with a list of collisions.

To add an algorithm to the framework, the developer first inherits from a collision detection class. The inherited class' initialisation method can then be overridden to initialise the appropriate algorithm data structures. Inside this class, a variety of method calls can be used to retrieve object properties and geometry. To perform the collision detection routines, the class' collision detection method can be overridden with the appropriate method from the algorithm under consideration. Inside this method,

the algorithm can call a selection of framework methods to retrieve object status data. This method will be called during each time step. For each collision, a point on both colliding objects, a contact normal pointing from the second to the first object and the distance between the two contact points is required. Using this data, the Bullet Physics SDK can perform the appropriate physics calculations.

4.5.4 Open Source

To further assist developers, a version of the framework corresponding to Woulfe and Manzke [2009] is available for download under the Berkeley Software Distribution (BSD) licence from http://gv2.cs.tcd.ie/benchmarkcd/. It compiles and executes under both GNU/Linux and Microsoft Windows. A set of experiments are included, which can be used directly or as a starting point for more targeted evaluations. By providing the framework for download, algorithm and application developers do not have to reprogram the framework to test their algorithms and application scenarios. Therefore, developers can publish results from one algorithm and it should be relatively straightforward to compare these results against those published by other developers for different algorithms. Open sourcing the framework substantially reduces development time, since developers can immediately download, compile and execute the framework instead of reimplementing it. This should make the framework more useful, potentially resulting in greater utilisation within the collision detection community.

4.6 Summary

This chapter began with an evaluation of a variety of target hardware platforms suitable for implementing the microarchitectures. This evaluation resulted in the selection of two FPGAs as prototyping platforms. For longer-term use, it was determined that implementation on more sophisticated platforms, such as ASICs, or as part of the fixed-function logic in CPUs and GPUs would offer better results. After this selection, the implementation of the two microarchitectures on these platforms was outlined, focusing on the adaptations of the generic microarchitectures outlined in Chapter 3. In particular, it was determined that the optimal value of m for each implementation was sixteen. The optimal storage of AABB data using each of the FPGA modules' external memories was also investigated in detail.

Subsequent to the discussion of the area-efficient and many-object microarchitectures, the implementation of the spatial partitioning concept was discussed. This discussion concentrated on the adaptations required for the FPGA implementations of the microarchitectures. The communication interface changes were highlighted. Modifications to the software and the sequence of interactions between the CPU and FPGA were also considered.

The software implementation of the benchmarking framework was also outlined in this chapter. Three broad phase algorithms from the Bullet Physics SDK were implemented, providing the user with three reference algorithms that should prove useful during collision detection evaluations. Additional algorithms can be easily added to the framework using the provided API, making it useful for algorithm developers, who can add their own algorithms for testing purposes. The framework was also open sourced so that developers will not need to reimplement the framework for their own algorithms.

Chapter 5

Results and Evaluation

However beautiful the strategy, you should occasionally look at the results.

—Sir Winston Churchill, British Prime Minister (1874–1965)

Results are performance metrics and other quantitative data characterising the behaviour of a system. This chapter thoroughly characterises the behaviour of all of the systems outlined previously. It commences with an appraisal of the benchmarking framework, based on a set of five experiments. These consider the three algorithms from the Bullet Physics SDK and evaluate fluctuations in cycle execution times, the impact of the different parameters and the effect of collisions. Subsequently, the hybrid software and hardware systems with the area-efficient and many-object microarchitectures are similarly appraised using a range of experiments. These experiments consider the systems' performance from both a software and a microarchitecture viewpoint. The experiments also evaluate the expected performance of a theoretical implementation of the microarchitecture on the fixed-function logic of a CPU. A thorough investigation of spatial partitioning with both microarchitectures and a variety of cell sizes is also conducted. The chapter concludes with a complete evaluation of the results, providing guidelines on how to select the most appropriate hybrid system for a particular scenario.

5.1 Benchmarking Framework

The benchmarking framework was evaluated prior to the hybrid software and hardware systems, as the framework would be used for evaluating these systems. Therefore, it was vital to thoroughly understand the framework's behaviour and to ensure that it functioned correctly. Consequently, this evaluation is both a complete appraisal of the benchmarking framework and an initial stage in the appraisal of the hybrid systems. It was also essential to determine the relative performance of the different Bullet Physics SDK collision detection algorithms, so that the most efficient software algorithm could be

found. This algorithm could then be used to determine the relative performance of the hybrid system when compared against a state-of-the-art software algorithm.

The benchmarking framework was executed on 'Computer 1', comprising a Quad-Core AMD Opteron 2350 (2 GHz) CPU [AMD, 2007] with 8 GiB RAM running Ubuntu GNU/Linux 8.04 LTS Desktop (64-bit) [Canonical, 2008]. All benchmarking framework experiments were performed entirely in software. A complete list of the computer's specifications can be located in Section B.3.

Cycle Execution Time Experiment Figure 5.1

The purpose of this experiment was to measure the fluctuation of broad and narrow phase execution times across an application's cycles. This would allow the behaviour of the three algorithms to be compared in detail, demonstrating how developers could use the framework to explore detailed algorithm behaviour.

The experiment utilised a benchmark with the following parameters:

- **Objects – Quantity:** 500
- **Objects – Type:** Uniform probability distribution
- **Objects – Location:** Uniform probability distribution
- **Size:** Uniform probability distribution
  - **Minimum:** 25 m
  - **Maximum:** 50 m
- **Acceleration:** Constant $(0, 0, 0)$ m/s^2
- **Linear Velocity:** Uniform probability distribution
  - **Minimum:** $(-25, -25, -25)$ m/s
  - **Maximum:** $(25, 25, 25)$ m/s
- **Angular Velocity:** Uniform probability distribution
  - **Minimum:** $(-2.5, -2.5, -2.5)$ rad/s
  - **Maximum:** $(2.5, 2.5, 2.5)$ rad/s

All unspecified parameters were left at the defaults outlined in Section 3.6.2. The parameters were selected to be representative of scenarios found in interactive applications. For example, the quantity of objects was sufficiently high to exercise the collision detection algorithms but within the object limits of typical applications. Moreover, object distribution, types, sizes, linear velocities and angular velocities varied as few applications contain multiple instances of a single object type moving in lockstep. Varying the angular velocity was particularly important since it was vital to test the behaviour of the collision detection algorithms in the presence of rotating objects, which occur in almost all interactive applications.

The benchmark was executed for 1000 cycles using the all-pairs, SAP and DBVT broad phase algorithms. Each cycle's execution time was recorded separately for the broad and narrow phases.

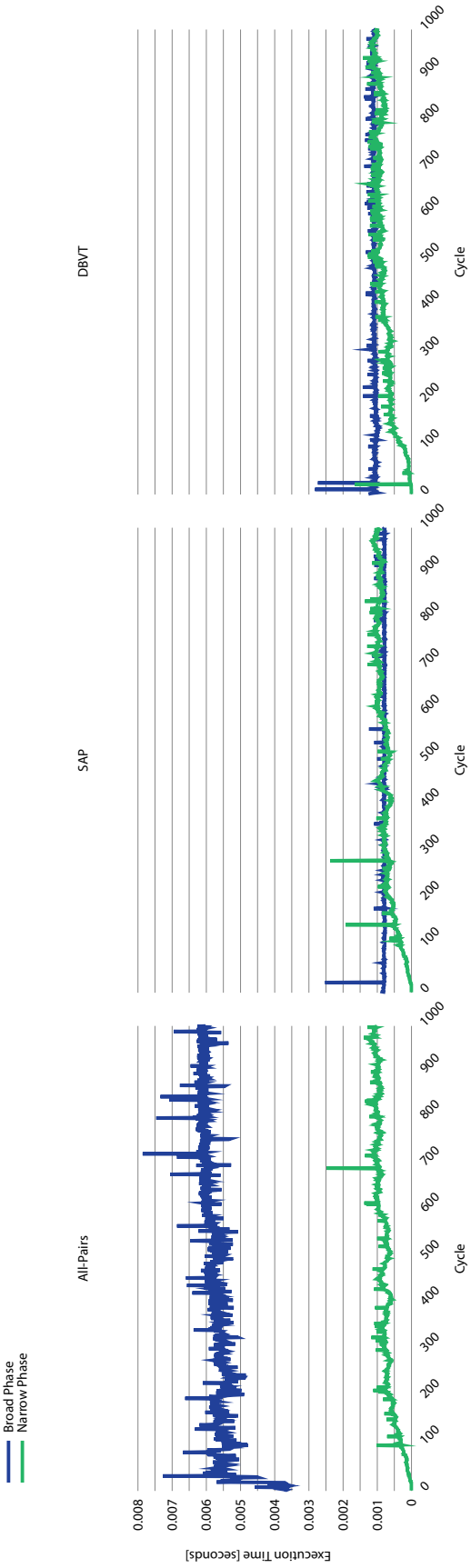


Figure 5.1: *Execution times for the 'Cycle Execution Time Experiment' of Section 5.1.*

Examining the results, it is evident that the all-pairs broad phase exhibits significant variability in cycle execution time throughout the benchmarks. Initially, this may appear unusual as all-pairs does not exploit coherence and the same number of collision tests must be performed on every cycle. However, this variability exists because the collision test involves the checking of six logic statements and short-circuited evaluation discontinues this checking once the possibility of collision has been eliminated. Therefore, many collisions or near collisions will result in longer execution times as more axes will need to be checked. The results also reveal that the all-pairs broad phase starts with a relatively low execution time before it increases significantly. This increase occurs because the initial positions of the objects result in no collisions, but as the objects begin to move, collisions and near-collisions occur. After the increase, the execution time exhibits a gradual upward trend and there is significant variability in individual execution times, with peaks occurring throughout the simulation. The all-pairs narrow phase takes considerably less time to execute than the broad phase, although it also exhibits an increase at the beginning of the simulation. However, this increase is more gradual and extends over more cycles than that of the broad phase. The increase is followed by a gradual upward trend, as for the broad phase.

Significantly better performance is attained using the SAP broad phase. This improvement is evident in every cycle of the simulation. Moreover, the algorithm is almost constant in terms of execution time as, despite some variability, the execution times are all centred on a constant mean. These results demonstrate that SAP's utilisation of coherence is beneficial in achieving fast execution times. The SAP narrow phase behaves the same as for all-pairs'.

DBVT's broad phase has a similar, albeit slightly longer, execution time to SAP's. Despite the overall mean being higher, the longest cycle execution time is lower for DBVT than for SAP. In addition, a noticeable characteristic of the DBVT broad phase is the peaks that occur during the initialisation of the simulation. These are attributable to the startup costs involved in constructing the AABB trees. They are also attributable to the movement of objects between the static and dynamic trees, which occurs while the algorithm is attempting to determine the most appropriate tree for each object. The DBVT narrow phase is extremely similar to those of all-pairs and SAP, but there is an additional peak during initialisation. This peak is a result of DBVT returning a larger PCS from the broad phase. The large PCS is undesirable as it demonstrates that the broad phase is inefficient during initialisation, although this is unlikely to make a significant contribution to the overall execution time of any simulation.

To ameliorate the performance of the algorithms, it would be advisable to focus on their behaviour after initialisation. The execution time increase that occurs as more collisions arise leads to deterioration in the performance of the three algorithms, particularly in the narrow phase. It would also be advisable to attempt to minimise the variation of the execution times, particularly for all-pairs and DBVT. In the case of all-pairs, it may be possible to do this by altering the order in which axes are tested on each iteration, as collisions along certain axes may occur more frequently than those along others. For DBVT, it would be advantageous for the user to indicate to the Bullet Physics SDK which objects are static and which are dynamic at the start of the simulation. This would obviate the need to move objects between the static and dynamic trees during the initial collision detection cycles. These

suggestions, if implemented, should help to improve overall execution times.

Based on these results, application developers would be advised to select SAP as this algorithm exhibits the best overall performance. In addition, SAP exhibits the least variability, making it easier to achieve constant frame rates. Constant frame rates are beneficial for developers of real-time applications, as the majority look for predictable behaviour that can be allocated a constant fraction of CPU time. DBVT is almost as efficient as SAP in terms of execution times and variability, while all-pairs is poorest in terms of both metrics. However, it would be best for an application developer to recreate their application scenarios in the benchmarking framework and repeat the experiment to ensure SAP remains the most efficient algorithm for their application.

Parameters Experiment Figure 5.2

The purpose of this experiment was to determine the effect of each parameter on the three algorithms. This would demonstrate the validity of the range of parameters selected for inclusion in the benchmarking framework. It would also show how developers could use the framework to explore different aspects of an algorithm's behaviour.

The experiment utilised twenty-two benchmarks with the following parameters:

1.
2.
   - **Objects – Quantity:** 100
   - **Objects – Quantity:** 500
   - **Objects – Type:** Spheres
3.
   - **Objects – Quantity:** 500
   - **Objects – Type:** Cuboids
4.
   - **Objects – Quantity:** 500
   - **Objects – Type:** Cylinders
5.
   - **Objects – Quantity:** 500
   - **Objects – Type:** Cones
6.
   - **Objects – Quantity:** 500
   - **Objects – Type:** Stanford Bunnies
7.
   - **Objects – Quantity:** 500
   - **Objects – Type:** Stanford Armadillos
8.
   - **Objects – Quantity:** 500
   - **Objects – Type:** Uniform probability distribution
9.
   - **Objects – Quantity:** 500
   - **Objects – Location:** Uniform probability distribution
10.
    - **Objects – Quantity:** 500
    - **Objects – Location:** Normal probability distribution
11.
    - **Objects – Quantity:** 500
    - **Size:** Uniform probability distribution
      - **Minimum:** 25 m
      - **Maximum:** 75 m

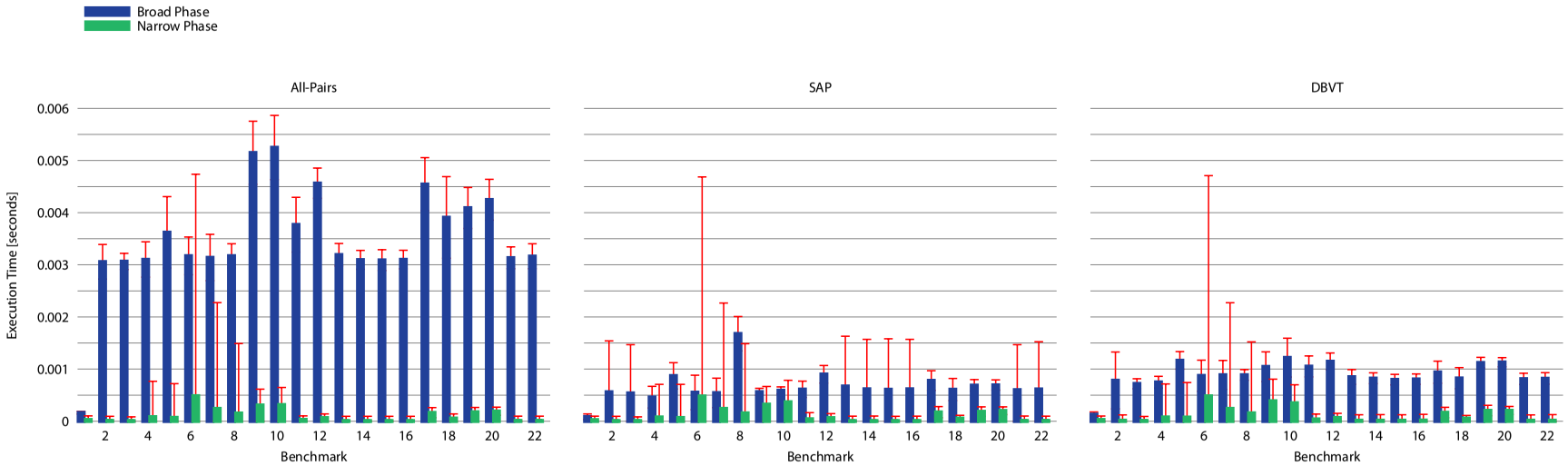


Figure 5.2: *Execution times for the 'Parameters Experiment' of Section 5.1.*

12. • **Objects – Quantity:** 500
    • **Size:** Normal probability distribution
        – **Mean:** 50 m
        – **Variance:** 10 m
13. • **Objects – Quantity:** 500
    • **Mass:** Uniform probability distribution
        – **Minimum:** 10 kg
        – **Maximum:** 100 kg
14. • **Objects – Quantity:** 500
    • **Mass:** Normal probability distribution
        – **Mean:** 50 kg
        – **Variance:** 20 kg
15. • **Objects – Quantity:** 500
    • **Aspect Ratio:** Uniform probability distribution
        – **Minimum:** 1 : 0.5 : 0.5
        – **Maximum:** 1 : 1.5 : 1.5
16. • **Objects – Quantity:** 500
    • **Aspect Ratio:** Normal probability distribution
        – **Mean:** 1 : 1 : 1
        – **Variance:** 1 : 0.2 : 0.2
17. • **Objects – Quantity:** 500
    • **Acceleration:** Uniform probability distribution
        – **Minimum:** $(-2.5, -2.5, -2.5)$ m/s^2
        – **Maximum:** $(2.5, 2.5, 2.5)$ m/s^2
18. • **Objects – Quantity:** 500
    • **Acceleration:** Normal probability distribution
        – **Mean:** $(0, 0, 0)$ m/s^2
        – **Variance:** $(1, 1, 1)$ m/s^2
19. • **Objects – Quantity:** 500
    • **Linear Velocity:** Uniform probability distribution
        – **Minimum:** $(-25, -25, -25)$ m/s
        – **Maximum:** $(25, 25, 25)$ m/s
20. • **Objects – Quantity:** 500
    • **Linear Velocity:** Normal probability distribution
        – **Mean:** $(0, 0, 0)$ m/s
        – **Variance:** $(10, 10, 10)$ m/s

21. • **Objects – Quantity:** 500
    • **Angular Velocity:** Uniform probability distribution
        – **Minimum:** $(-2.5, -2.5, -2.5)$ rad/s
        – **Maximum:** $(2.5, 2.5, 2.5)$ rad/s
22. • **Objects – Quantity:** 500
    • **Angular Velocity:** Normal probability distribution
        – **Mean:** $(0, 0, 0)$ rad/s
        – **Variance:** $(1, 1, 1)$ rad/s

All unspecified parameters were left at the defaults outlined in Section 3.6.2. The parameters were selected to be relatively representative of scenarios found in interactive applications.

The benchmarks were executed for 1000 cycles using the all-pairs, SAP and DBVT broad phase algorithms. Each cycle's execution time was recorded separately for the broad and narrow phases. The mean and standard deviation of the cycle execution times were subsequently computed.

The results illustrate how seemingly minor modification of the parameters can have a significant influence on algorithm execution times. For example, examining the all-pairs broad phase reveals that benchmark one has a significantly below-average execution time due to the lower object count. It is also evident that benchmarks nine and ten take longest to execute as these object distributions result in more collisions. Benchmarks eleven to twelve and seventeen to twenty have above-average execution times, as these parameters also result in increased collisions. Finally, benchmark five is relatively inefficient as AABBs enclose cones poorly, resulting in a disproportionate number of collision tests. With other object types, there is a greater possibility that a broad phase collision will result in a narrow phase collision, causing a collision response to move the colliding objects apart. Since this does not happen with the cones, many potential collisions will be detected as the objects gradually approach each other, before the collision response is finally applied. Turning attention to the all-pairs broad phase standard deviation, it is evident that cycle execution times undergo major fluctuations with certain benchmarks. For example, benchmarks five and eighteen have the largest standard deviations. For the former, this is due to the use of cones, and for the latter, it appears that the given acceleration maximises collisions during certain cycles. For the remaining benchmarks, the standard deviation varies significantly but none of the results are particularly noteworthy.

It is evident that the benchmarks that take longer to execute using all-pairs for the broad phase also take longer to execute in the narrow phase, since more potential collisions need to be checked with these benchmarks. However, benchmarks six and seven only take an average amount of time to execute in the broad phase yet take considerable time in the narrow phase. This is because the complex Stanford Bunnies and Armadillos are easily tested for collision as AABBs in the broad phase, but require complex testing of all their constituent polygons in the narrow phase. It is also apparent, from the all-pairs narrow phase standard deviation, that the variability in cycle execution times mimics that of the broad phase, except for benchmarks six to eight. These benchmarks, which contain Stanford Bunnies and Armadillos, often require the narrow phase algorithm to descend a variable number of tree levels before rejecting a collision. This variability results in higher standard deviations.

Focusing on the SAP broad phase, it is evident that the performance of this algorithm is superior to all-pairs across all of the benchmarks. The algorithm only exhibits a small acceleration for the first benchmark, but all subsequent benchmarks are greatly accelerated. Therefore, it is evident that object quantity has a more significant impact on all-pairs than SAP due to all-pairs' computational complexity of $O(n^2)$ compared to SAP's $O(n \log n)$. The SAP broad phase's longest execution time occurs in benchmark eight, as this combination of object types deteriorates the algorithm's performance. Examining the broad phase standard deviation, it is evident that there is significant degradation in benchmarks two to three, thirteen to sixteen and twenty-one to twenty-two. These benchmarks result in objects rapidly changing their placement within the environment, and this is inefficient for SAP as coherence cannot be effectively exploited. SAP's narrow phase behaves similarly to that of all-pairs'.

DBVT's broad phase takes slightly longer than SAP's to execute all benchmarks, except number eight. Its superior performance for the eighth benchmark suggests that DBVT works efficiently for scenarios containing a wide variety of objects. The algorithm is also significantly quicker than all-pairs for all of the benchmarks. Those benchmarks exhibiting high standard deviations in the broad phase are the same as those exhibiting high standard deviations with all-pairs. However, in many cases, the standard deviations are reduced, which may make it easier for a real-time application to achieve a consistent frame rate. The narrow phase mimics that of both all-pairs and SAP.

To accelerate the three algorithms, it would be advisable to focus primarily on the performance of cones as an object type, since their usage results in above-average execution times for all of the algorithms. To rectify this performance degradation, a BV that encloses cones more tightly could be used. This could be implemented for all-pairs and DBVT but it is not a solution for SAP, as the BV for this algorithm cannot be changed. Moreover, such a change is unlikely to be of much benefit as an alternate BV is liable to deteriorate performance in the majority of cases. In addition to cones, objects comprising convex hulls exhibit poor performance with all algorithms, making these of interest when attempting to improve performance. However, it is likely to be difficult to accelerate scenarios containing these object types, as the poor performance is caused by object complexity, which cannot be easily resolved. Furthermore, in the case of all-pairs, random object locations, object sizes, accelerations and linear velocities should be investigated, since these extend execution times. Altering the order in which axes are checked for collision or only checking certain axes in the broad phase may improve these scenarios.

Of greater benefit would be an attempt to reduce the execution time of benchmark eight with SAP, as this algorithm exhibits the best performance for every other benchmark. Moreover, few applications have only one object type and it is, therefore, important to achieve good execution times for scenarios involving a selection of objects. However, there are few optimisations that can be performed on SAP due to the rigid nature of the algorithm. Despite this, it would be worth investigating whether performance could be improved either by checking only certain axes for collision or by reordering the sequence of axes checked. Finally, the standard deviation measurements could be used to attempt to reduce the variability in the per-cycle execution times and improve the overall performance. This would be of particular benefit to the SAP broad phase, which exhibits high standard deviations, and to the

narrow phase of benchmarks six to eight, which exhibit significantly higher standard deviations than any of the other benchmarks. High standard deviations should be investigated in more detail, by recording the per-cycle execution times across the benchmark as in the 'Cycle Execution Time Experiment', before attempting to rectify them.

Application developers would be advised to select SAP as this exhibits the best overall performance. However, if application scenarios mimicked those of benchmark eight, in that they contained a wide distribution of object types including concave objects constructed from convex hulls, then it would be best to select DBVT. Similarly, if the overriding concern were to maintain a consistent frame rate, as in many real-time applications, then it would be best to also select DBVT for application scenarios mimicking benchmarks thirteen to seventeen and twenty-one to twenty-two. However, as for the 'Cycle Execution Time Experiment', it would be best for an application developer to recreate their application scenarios in the benchmarking framework and repeat the experiment to ensure SAP or DBVT remains the most efficient algorithm for their application.

Parameters (Ratio) Experiment Figure 5.3

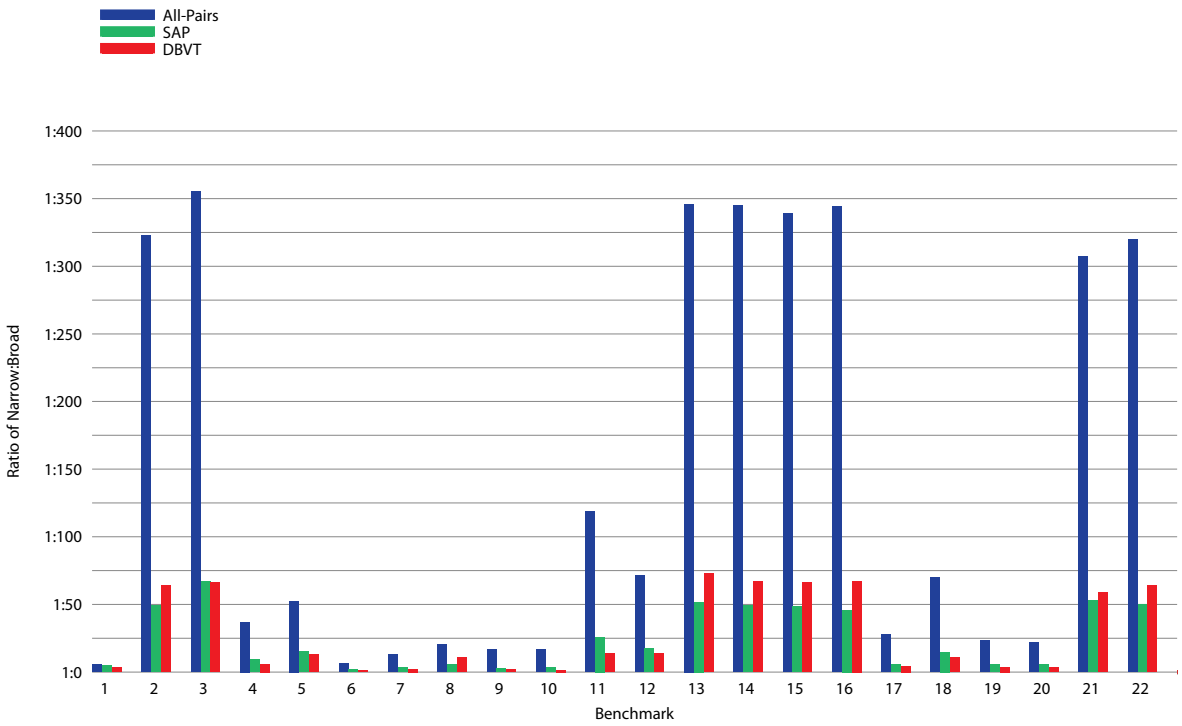


Figure 5.3: *Results for the 'Parameters (Ratio) Experiment' of Section 5.1.*

The purpose of this experiment was to determine the ratio of the narrow phase to the broad phase execution times for the three algorithms. This would facilitate a holistic understanding of the collision detection system. In addition, it would demonstrate whether the broad phase consumes a sufficient portion of the collision detection system to warrant microarchitecture implementation.

The broad and narrow phase execution times of the twenty-two benchmarks from the 'Parameters Experiment' were used to compute the ratio of the narrow phase to the broad phase execution times.

From the results, it can be observed that the broad phase execution times are substantially longer than the narrow phase execution times for every algorithm and benchmark. It is also evident that, for each benchmark, the proportion consumed by the all-pairs broad phase is always the largest, while the lowest varies between SAP and DBVT. Finally, it is clear that there exists significant variability across the benchmarks.

The ratios clearly prove the validity of attempting to accelerate the broad phase using the microarchitectures, as it is evident that accelerating the broad phase will have a greater overall effect on the system than accelerating the narrow phase. It is also evident that the results are of little benefit for optimising the various algorithms. However, in certain cases, it may be beneficial to attempt to balance the ratios of the broad and narrow phases. In these cases, it would be best to consult the results from the 'Parameters Experiment' as these provide more detail on the performance of the algorithms executing the benchmarks. Similarly, the results provide little information for application developers attempting to select an appropriate algorithm.

Collisions Experiment Figures 5.4 and 5.5

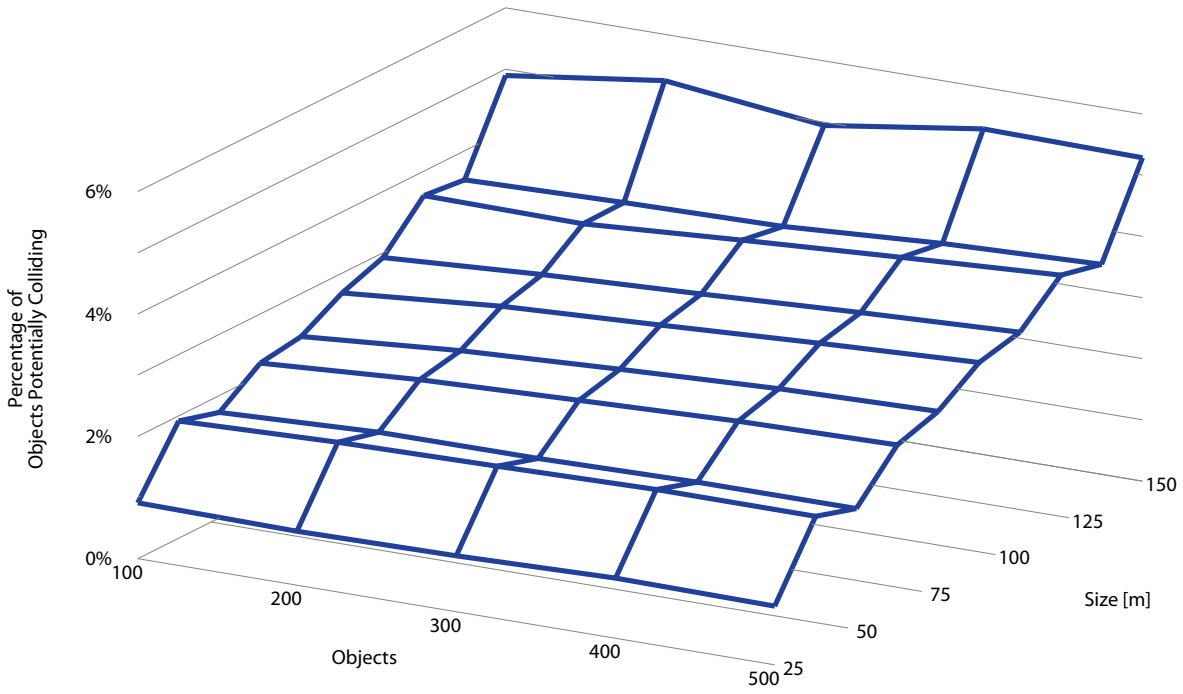


Figure 5.4: *Results for the 'Collisions Experiment' of Section 5.1.*

The purpose of this experiment was to determine how altering the number of collisions affected the performance of the different algorithms. This would also demonstrate how a developer could sweep through a range of parameter values using the benchmarking framework.

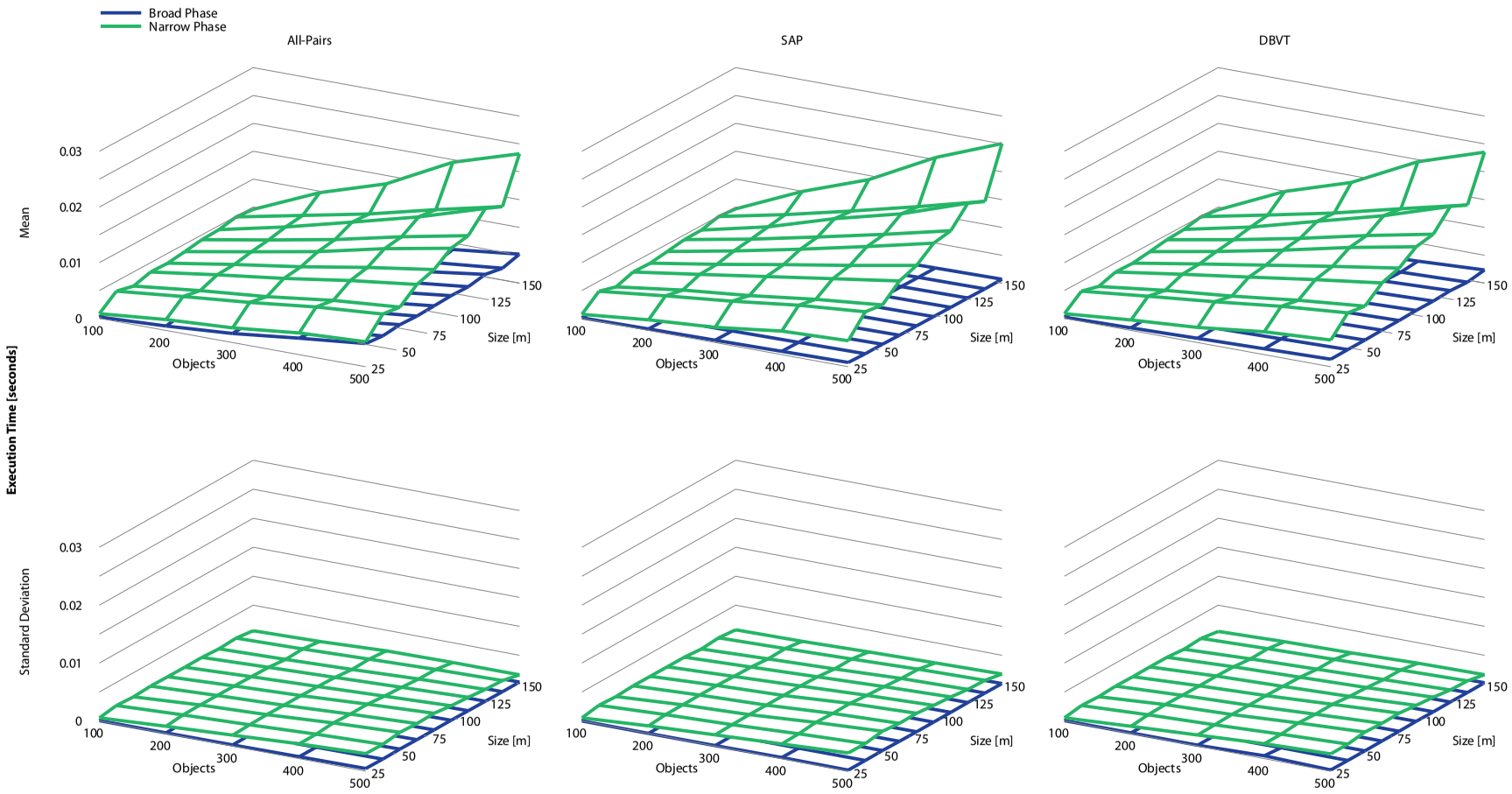


Figure 5.5: *Execution times for the 'Collisions Experiment' of Section 5.1.*

The experiment utilised thirty benchmarks based on the benchmark from the 'Cycle Execution Time Experiment':

- **Objects – Quantity:** 100, 200, 300, 400, 500
- **Objects – Type:** Uniform probability distribution
- **Size:** 25 m, 50 m, 75 m, 100 m, 125 m, 150 m
- **Acceleration:** Constant $(0, 0, 0)$ m/s^2
- **Linear Velocity:** Uniform probability distribution
  - **Minimum:** $(-25, -25, -25)$ m/s
  - **Maximum:** $(25, 25, 25)$ m/s
- **Angular Velocity:** Uniform probability distribution
  - **Minimum:** $(-2.5, -2.5, -2.5)$ rad/s
  - **Maximum:** $(2.5, 2.5, 2.5)$ rad/s

All unspecified parameters were left at the defaults outlined in Section 3.6.2. Representative benchmarks are illustrated in Figure 5.6.

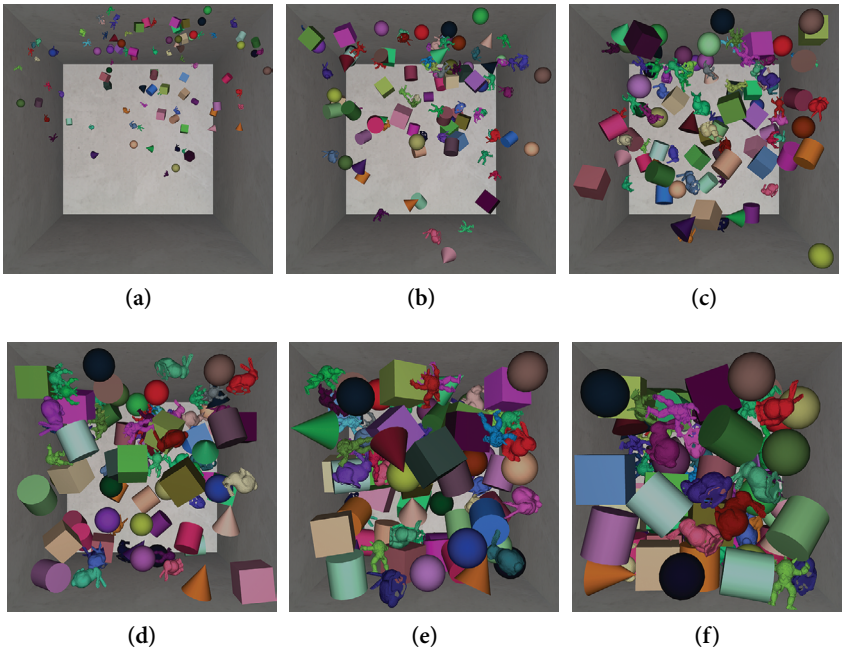


Figure 5.6: *100-object benchmarks for the 'Collisions Experiment' of Section 5.1.* **(a)** *25 m-objects.* **(b)** *50 m-objects.* **(c)** *75 m-objects.* **(d)** *100 m-objects.* **(e)** *125 m-objects.* **(f)** *150 m-objects.*

The benchmarks were executed for 1000 cycles using the all-pairs, SAP and DBVT broad phase algorithms. Each cycle's execution time was recorded separately for the broad and narrow phases. The mean and standard deviation of the cycle execution times were subsequently computed. In addition, the number of potentially colliding objects returned by the broad phase was measured and this was converted to a percentage based on the number of simulated objects.

The results demonstrate that the number of potentially colliding objects increases in accordance with both object count and size. However, the increase in relation to object count can be effectively disregarded since the percentage of potentially colliding objects remains relatively constant. This consistency occurs because the benchmarking framework is designed to keep object density constant, making the likelihood of collision relatively constant. Therefore, the size of the objects is the most significant factor affecting the number of potential collisions.

Further examination of the results shows that all-pairs exhibits the worst performance in the broad phase. DBVT achieves slightly better broad phase results while SAP displays the best performance. Of greater note is how the object counts and sizes affect execution time. For all broad phase algorithms, it is clear that object count has the most significant impact on execution time. For example, between the 100-object 25 m-benchmark and the 500-object 25 m-benchmark, there is an execution time increase of 1602% for all-pairs, of 567% for SAP and of 519% for DBVT. In contrast, object size has a significantly lower impact, particularly in the case of SAP. Quantifying the effect using the 500-object 25 m and 500-object 150 m benchmarks shows an execution time increase of 25% for all-pairs and of 76% for DBVT, but a decrease of 8% for SAP. Object count has the more significant effect as additional objects result in additional broad phase processing. For all of the algorithms, the standard deviation remains low and relatively constant.

Focusing on the narrow phase, it is evident that object size has a much larger impact on execution time, while the number of objects has a lesser impact. For example, a 637% increase is observed for the SAP narrow phase between the 100-object 25 m and 500-object 25 m benchmarks, while a 538% increase is observed between the 500-object 25 m and 500-object 150 m benchmarks. This reflects the pattern created by the number of potential collisions. This correlation between potential collisions and narrow phase execution times occurs due to increased potential collisions in the broad phase resulting in additional expensive BVH traversals in the narrow phase. The standard deviations show similar, albeit slightly more erratic, behaviour.

To accelerate the algorithms, it would be advisable to improve the broad phase performance of scenarios with large numbers of objects. However, there are unlikely to be any algorithmic developments that can significantly improve performance in these cases, as it is to be expected that additional objects take longer to process. It would also be beneficial to focus on scenarios where increased potential collisions are detected in the broad phase, as these deteriorate the performance of the narrow phase. Accurate algorithms with tightly fitting BVs could result in an improvement but it is likely that these would reduce performance in cases where fewer collisions occur. Overall, it would be challenging to find algorithmic improvements to rectify the issues detected using this experiment.

Application developers would be advised to select SAP as this exhibits the best overall performance. However, as for the previous experiments, it would be best for an application developer to recreate their application scenarios in the benchmarking framework and repeat the experiment to ensure SAP remains the most efficient algorithm for their application.

Collisions (Ratio) Experiment Figure 5.7

As for the 'Parameters (Ratio) Experiment', the purpose of this experiment was to determine the ratio of the narrow phase to the broad phase execution times for the three algorithms. This would facilitate an understanding of the collision detection system as a whole. In addition, it would demonstrate whether the broad phase consumes a sufficient portion of the collision detection system to warrant microarchitecture implementation.

The broad and narrow phase execution times of the thirty benchmarks from the 'Collisions Experiment' were used to compute the ratio of the narrow phase to the broad phase execution times.

The results show very different behaviour than the 'Parameters (Ratio) Experiment', with the broad phase now consuming significantly less time than the narrow phase for every algorithm and benchmark. This change occurred when the narrow phase execution time increased in accordance with the number of potential collisions, and it demonstrates how the ratios are extremely sensitive to the scenarios under consideration. Despite the change, the broad phase still takes significant time to execute. The results also show that, for each benchmark, the proportion consumed by the all-pairs broad phase is always the largest. DBVT lies in the middle while SAP consumes the smallest proportion.

Despite the differing results achieved using this experiment, it still appears to be beneficial to attempt to accelerate the broad phase using the microarchitectures. This is because these benchmarks focus on densely packing objects to maximise collisions, which are scenarios that are less likely to occur in practice than those in the 'Parameters (Ratio) Experiment' benchmarks. Moreover, the broad phase still accounts for significant execution time. Therefore, it can be inferred that the broad phase accounts for much, if not most, of the collision detection execution time in typical scenarios. Consequently, the broad phase microarchitectures are a valid approach. As for the 'Parameters (Ratio) Experiment', the results are of little benefit for optimising the various algorithms. In addition, they provide little information for application developers attempting to select an appropriate algorithm. In these cases, it would be more beneficial to consult the results from the 'Collisions Experiment', which provide additional detail.

Evaluation From the benchmarking framework experiments, it is clear that SAP offers the best overall performance for the majority of applications. This result is in accordance with current practice in the field of collision detection. DBVT also offers excellent performance for certain scenarios but it typically fails to perform as efficiently as SAP. Based on these findings, SAP was selected as the primary algorithm against which to compare the performance of the hybrid systems. If an acceleration could be achieved against this algorithm, it would be clear that the hybrid system offered an appreciable benefit over what could be achieved in software. In addition, all-pairs was selected as a secondary algorithm against which to compare the performance of the hybrid systems. This would enable a direct comparison of the performance of the software and hardware implementations of this algorithm.

The benchmarking framework experiments can also be used to determine the proportion of the collision detection cycle consumed by the broad phase. Using the 'Parameters (Ratio) Experiment' and the 'Collisions (Ratio) Experiment', it was determined that broad phase collision detection accounts for

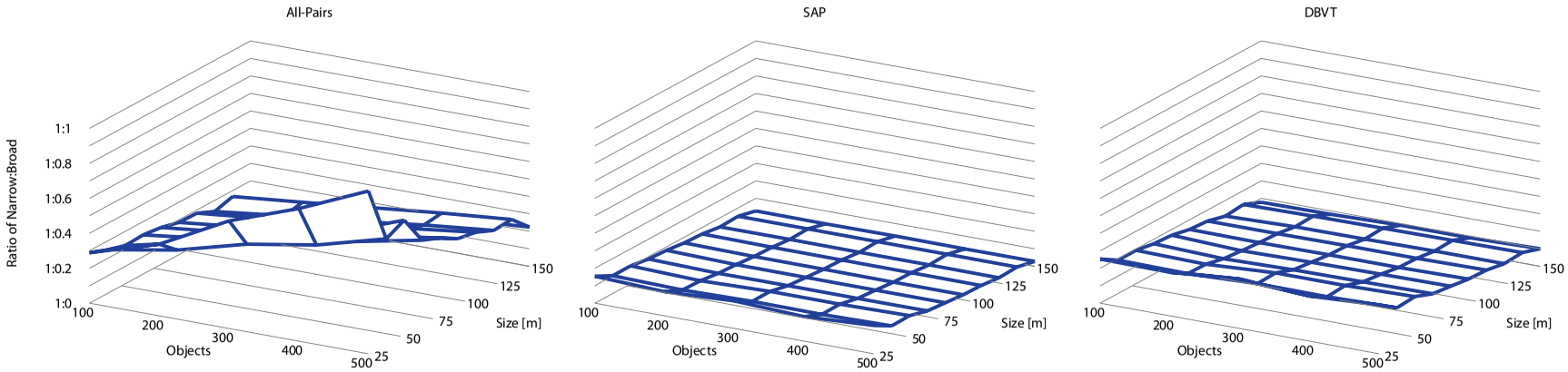


Figure 5.7: *Results for the 'Collisions (Ratio) Experiment' of Section 5.1.*

between 2.5% and 98.7% of the overall collision detection execution time, with the average accounting for 40.4%. In addition, the entire collision detection system using SAP consumes between 1% and 38.3% of the program loop, with an average application consuming 14%. Therefore, an average application typically spends 5.7% of its program loop computation time performing broad phase collision detection.

Throughout the following discussion, it will be necessary to determine whether the hybrid systems can execute a program loop at either 30 fps or 60 fps, which are equivalent to either one frame per $0.0\bar{3}$ seconds or one frame per $0.01\bar{6}$ seconds. Based on the metric of 5.7%, it would be necessary to perform the broad phase collision detection in 0.0019 seconds to achieve 30 fps, or in 0.00095 seconds to achieve 60 fps. These execution times will be used throughout the proceeding examination.

5.2 Area-Efficient Microarchitecture

After analysing the behaviour of the collision detection benchmarking framework and having used it to thoroughly analyse the performance of the three broad phase algorithms supplied with the Bullet Physics SDK, it was necessary to examine the hybrid systems that used the area-efficient microarchitecture. Systems including Xilinx Virtex-II and Xilinx Virtex-5 implementations of the microarchitecture were analysed.

5.2.1 Xilinx Virtex-II

All of the benchmarks for the hybrid system with the Xilinx Virtex-II implementation of the area-efficient microarchitecture were executed on 'Computer 2', comprising a 1.8 GHz Intel Pentium 4 [Intel, 2002] with 2 GiB RAM running Microsoft Windows XP Professional Service Pack 3 (32-bit) [Microsoft, 2009b]. A complete list of the computer's specifications can be located in Section B.3.

Metrics Table 5.1

	Total	Including Interface	Excluding Interface
Slices	33 792	24 607 (72%)	22 050 (65%)
BRAMs	144	128 (88%)	128 (88%)
Clock Frequency		76.6 MHz	81.3 MHz
Memory Throughput		29.41 GB/s	31.22 GB/s
Effective Bandwidth			240 MB/s
Effective Unidirectional Latency			1.68 µs

Table 5.1: *'Metrics' of Section 5.2.1.*

It was beneficial to examine the basic characteristics of the implemented microarchitecture prior to analysing the system's performance. The purpose of this was to provide an indication of the maximum

possible execution speed and to facilitate an understanding of any limitations of the implementation.

The resource utilisation metrics and clock frequencies from the Xilinx ISE place and route report were recorded for two variants of the microarchitecture. The first variant included the interface logic from the 'memory' example supplied by Alpha Data, while the second excluded this interface. Although the implementation with no interface cannot work on the Alpha Data ADM-XRC-II module, its metrics illustrate the portion of the design consumed by the collision detection microarchitecture alone. After recording these metrics, the clock frequencies were used to compute the sustained memory throughput that could be achieved while the system was outputting data. In addition, the effective bandwidth and effective unidirectional latency between the CPU and FPGA were measured. These last two metrics were determined by the CPU and the FPGA, and were not dependent on the microarchitecture.

Software-Based Execution Time Experiment Figure 5.8

The purpose of this experiment was to measure the performance of the hybrid system relative to a software system, using execution times measured from the software. This would provide an understanding of the acceleration offered by the microarchitecture when communication overheads and software processing of the data were considered.

The experiment utilised five benchmarks based on the benchmark from the 'Cycle Execution Time Experiment':

- **Objects – Quantity:** 100, 200, 300, 400, 500
- **Objects – Type:** Uniform probability distribution
- **Objects – Location:** Uniform probability distribution
- **Size:** Uniform probability distribution
  - **Minimum:** 25 m
  - **Maximum:** 50 m
- **Acceleration:** Constant $(0, 0, 0)$ m/s^2
- **Linear Velocity:** Uniform probability distribution
  - **Minimum:** $(-25, -25, -25)$ m/s
  - **Maximum:** $(25, 25, 25)$ m/s
- **Angular Velocity:** Uniform probability distribution
  - **Minimum:** $(-2.5, -2.5, -2.5)$ rad/s
  - **Maximum:** $(2.5, 2.5, 2.5)$ rad/s

Each of the five benchmarks had a different quantity of objects. All unspecified parameters were left at the defaults outlined in Section 3.6.2.

The benchmarks were executed for 1000 cycles using the hybrid system and the Bullet Physics SDK's all-pairs and SAP. Each cycle's broad phase execution time was recorded and the mean of the cycle execution times was subsequently computed. The execution time metrics, corresponding to Figure 5.8, were:

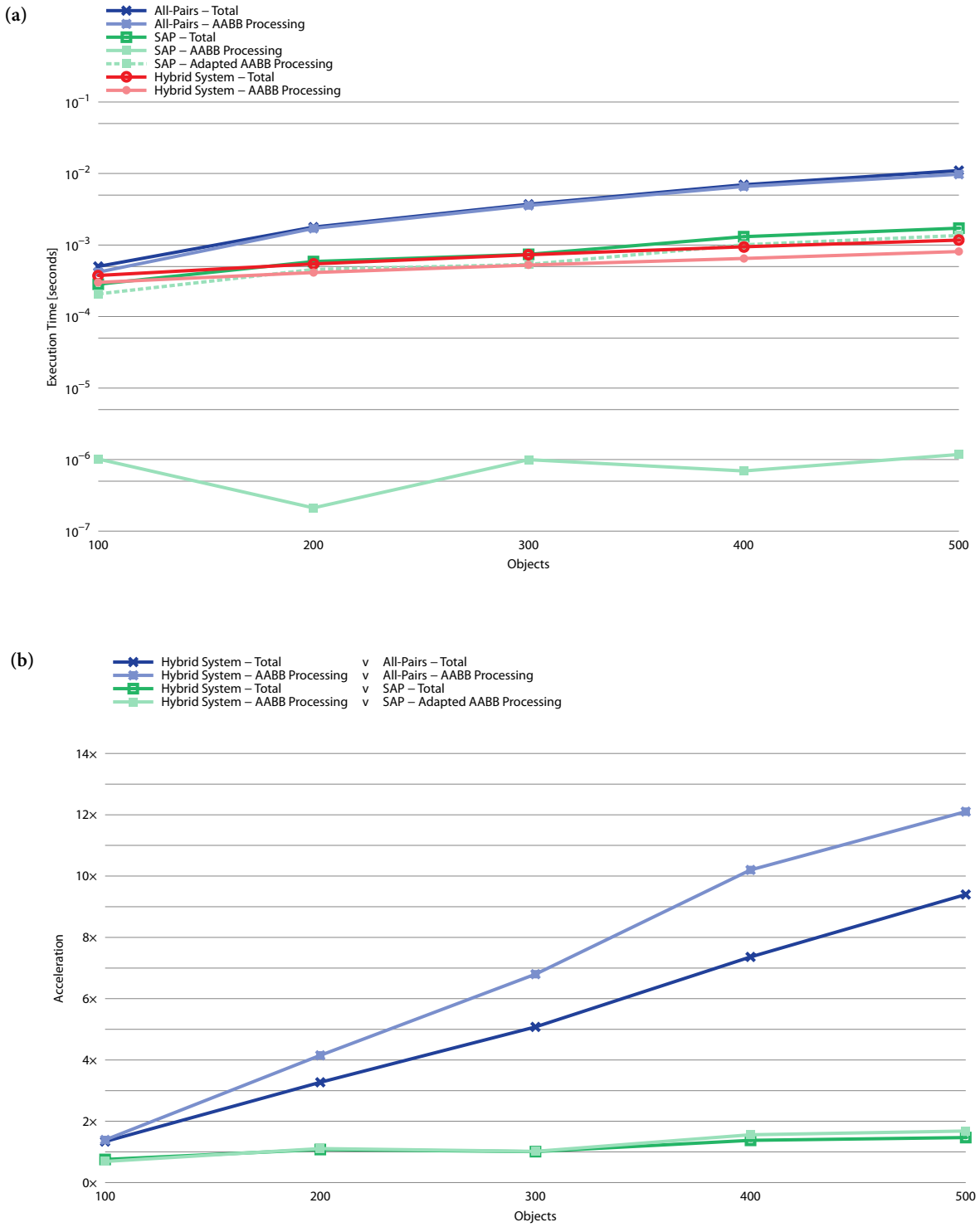


Figure 5.8: *Results for the 'Software-Based Execution Time Experiment' of Section 5.2.1.* **(a)** *Execution times. The equivalent linear-scale graph is Figure D.2.* **(b)** *Accelerations. The data are tabulated in Table D.1.*

All-Pairs – Total The total time consumed by the Bullet Physics SDK's all-pairs. This comprises AABB updating, which involves recomputing the AABBs to reflect updated object positions, and AABB processing, which involves the execution of the collision tests on the AABB pairs.

All-Pairs – AABB Processing The time consumed by the Bullet Physics SDK's all-pairs during AABB processing.

SAP – Total The total time consumed by the Bullet Physics SDK's SAP.

SAP – AABB Processing The time consumed by the Bullet Physics SDK's SAP during AABB processing.

SAP – Adapted AABB Processing The total time consumed by the Bullet Physics SDK's SAP less the time consumed by the hybrid system's AABB updating. This metric was computed because SAP's AABB processing does not reflect the true value of processing, since most of it is performed in tandem with updating due to the nature of the algorithm.

Hybrid System – Total The total time consumed by the hybrid system. This includes the software-based AABB updating and preparation of data for transfer, as well as the microarchitecture-based AABB processing. The two data transfers are also included.

Hybrid System – AABB Processing The time consumed by the hybrid system during AABB processing. This includes the software-based preparation of data for transfer, as well as the microarchitecture-based AABB processing. The two data transfers are also included.

Based on these recorded execution times, a variety of acceleration metrics was computed. These metrics directly compared pairs of the previously described total and AABB processing execution times. AABB processing times were compared as these exclude AABB updating, which was always performed entirely in software and created a relatively consistent overhead. Consequently, AABB processing times better reflect the performance of the microarchitecture.

Examining the results for the software algorithms, it is evident that all-pairs exhibits a fairly steep upward trend for both the total and AABB processing execution times. SAP exhibits a similar upward trend for the total execution time, although the AABB processing stays relatively constant. In addition, the aforementioned peculiarity in the division of SAP between AABB updating and AABB processing can be observed. Turning attention to the hybrid system, it is clear that the system's upward trend is significantly lower than for either all-pairs or SAP. This demonstrates that although the hybrid system suffers from a computational complexity of $O(n^2)$, the effective exploitation of parallelism significantly reduces the algorithm's growth rate. In particular, it is evident that the 500-object benchmark exhibits the longest execution time. This consumes 0.00117 seconds, which is sufficiently fast to allow applications to execute at 30 fps or greater, thereby highlighting the validity of the system.

The associated acceleration graph shows that the hybrid system achieves an acceleration for all of the benchmarks, except when compared against SAP with 100 objects, when the hybrid system achieves only 0.752× the performance of the SAP total or 0.69× of the SAP adapted AABB processing.

This deceleration can be attributed to the communication costs, which cannot be amortised by the microarchitecture for relatively small quantities of objects. Moreover, as the number of objects increases, the acceleration factor increases. This is beneficial as larger simulations tend to have a more significant negative impact on the performance of software algorithms, and the hybrid system now provides a means to overcome this impact. Comparing the execution time against all-pairs results in a greater acceleration than comparing against SAP due to all-pairs' poor performance. The acceleration factor is up to 12.1× compared to all-pairs and up to 1.7× compared to SAP. Finally, it is evident that a greater acceleration is typically achievable when comparing AABB processing instead of total execution time.

Overall, the microarchitecture achieves excellent performance for all benchmarks except for the 100-object benchmark, where a slight deceleration is encountered due to communication costs. This could be rectified by using a more sophisticated interconnect technology than PCI. However, the interconnect is constrained by the choice of FPGA module and cannot be altered. Therefore, there is no solution to this problem for this implementation. Despite this, the interconnect would clearly be worth considering if the microarchitecture were placed on a platform where the interconnect could be chosen.

Collisions Experiment Figure 5.9

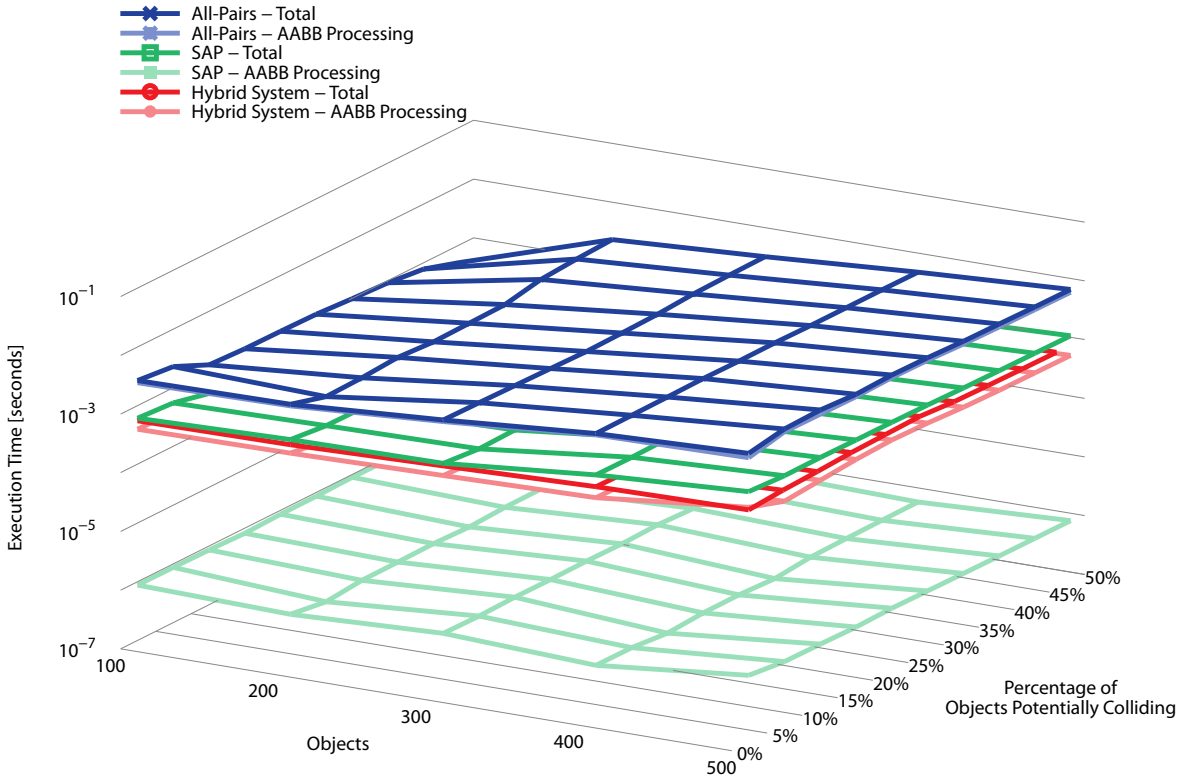


Figure 5.9: *Results for the 'Collisions Experiment' of Section 5.2.1. The equivalent linear-scale graph is Figure D.3.*

The purpose of this experiment was to determine the impact of the size of the PCS on the execution speed. This would provide information on whether an increase in the number of potential collisions detected would result in increased execution times.

The number of potential collisions detected by the broad phase was recorded for each cycle of the five benchmarks from the 'Software-Based Execution Time Experiment'. The 'SAP – Adapted AABB Processing' metric was not measured for this experiment as it was determined that it did not accurately reflect the cost of the collisions. From the measurements, the size of the PCS in relation to the number of objects was computed and this was evaluated against the cycle execution times.

From the results, it is evident that the number of potential collisions significantly degrades all-pairs' performance. For the benchmark with 500 objects, between the least and most potential collisions, all-pairs exhibits a 36% increase in the total broad phase execution time and a 48% increase in the AABB processing execution time. In contrast, SAP execution time is somewhat reduced when more potential collisions occur. It exhibits a decrease of 7% in the total and of 18% in AABB processing. The number of potential collisions also has an impact on the hybrid system, with an increase of 48% in the total but a decrease of 7% in AABB processing. The hybrid system's increase in the total execution time is due to unavoidable behaviour of the Bullet Physics SDK. The decrease in the hybrid system's AABB processing time must occur in the communication interface, as it is the only part of the microarchitecture without consistent execution times. Due to this inconsistency, the decrease cannot be relied upon in all scenarios, but is clear that increased collisions have little impact on the performance of the microarchitecture. In addition, variability between adjacent collision values is significantly lower than for either software algorithm, making it simpler to achieve a constant frame rate.

Hardware-Based Execution Time Experiment Figure 5.10

The purpose of this experiment was to measure the performance of the hybrid system relative to a software system, using the number of clock cycles consumed by the microarchitecture. This would provide an understanding of the acceleration offered by the microarchitecture without consideration of communication overheads and software processing of the data. Moreover, this would be used to effectively deduce the performance of the hybrid system without interference from the host system. In addition, the majority of existing microarchitecture performance data do not include communication overheads, so these results would facilitate a direct comparison with these data.

The number of clock cycles consumed by the microarchitecture was recorded for the five benchmarks from the 'Software-Based Execution Time Experiment' using clock cycle counters embedded within the microarchitecture. The benchmarks were executed for 1000 cycles and the mean of the clock cycle counts was subsequently computed. The clock cycle metrics, corresponding to Figure 5.10, were:

Excluding Communication Interface The clock cycles consumed by the microarchitecture excluding all communication interface logic.

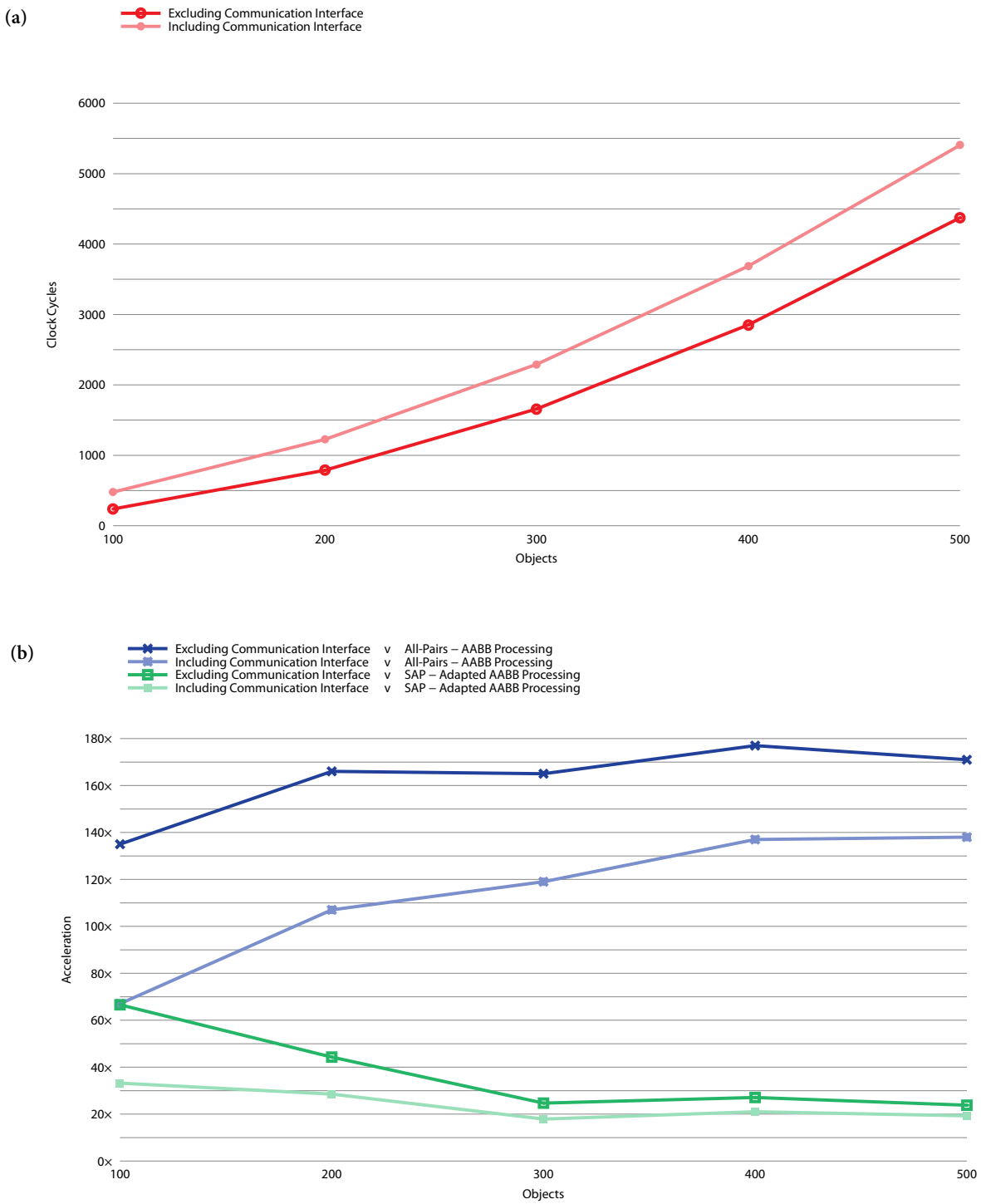


Figure 5.10: *Results for 'Hardware-Based Execution Time Experiment' of Section 5.2.1.* **(a)** *Clock cycles.* **(b)** *Accelerations. The data are tabulated in Table D.2.*

Including Communication Interface The clock cycles consumed by the microarchitecture including all communication interface logic. This includes the communication interface outlined in Sections 3.2 and 4.2.1, in addition to the communication interface logic from the 'memory' example supplied by Alpha Data.

The clock cycle counts were subsequently converted into execution times using the microarchitecture's clock frequency. Based on these execution times, a variety of acceleration metrics was computed. These metrics compared the metrics from this experiment with those from the Bullet Physics SDK algorithms from the 'Software-Based Execution Time Experiment'. All accelerations were based on AABB processing execution times, as the cycle counts cannot consider AABB updating costs.

From these results, it is evident that the communication interface accounts for a significant overhead and that performance would be improved if this could be reduced. It is also clear that these clock cycle counts result in significantly lower execution times than those measured using the CPU. Therefore, it can be inferred that the majority of the execution time is spent in the software-based preparation of data for transfer and in the two data transfers. The $O(n^2)$ computational complexity of the microarchitecture becomes apparent in the results, despite the microarchitecture implementation of all-pairs having been effectively divided by m.

The clock-cycle–based accelerations provided by the microarchitecture mimic those of the the execution-time–based accelerations, although all of the acceleration factors are now much improved. Comparing the hybrid system to all-pairs results in the greatest acceleration, with up to 177.269× achievable, although a very significant acceleration is evident when compared to SAP, with up to 66.647× achievable. However, the $O(n^2)$ computational complexity of the microarchitecture results in a slight drop-off in accelerations when compared to SAP at larger object counts. This is unproblematic as the acceleration factor remains high and does not descend below 17.893× for any object count.

As in the 'Software-Based Execution Time Experiment', the communication interface has been highlighted as a significant source of inefficiency. However, unlike the earlier experiment, the inefficiency present does not result from the interconnect used, since its effect is not considered in this experiment. Instead, the memory access requests are resulting in this performance degradation. This could be ameliorated by using more sophisticated memories, although any gains are likely to be minimal as it is doubtful that much efficiency could be gained using any memory type. Moreover, like the interconnect, the memories are constrained by the choice of FPGA module and cannot be altered. As before, they would be worth considering if the microarchitecture were placed on a platform where the memories could be chosen. This is not a major issue, however, as the execution times remain low despite the effect of the memory accesses.

A further issue highlighted by the microarchitecture is the undesirable $O(n^2)$ computational complexity. For up to 500 objects, this is unproblematic as the accelerations remain high. However, it is evident that accelerations may drop off more significantly if larger quantities of objects were simulated. Larger quantities are not supported by the Xilinx Virtex-II implementation of the area-efficient microarchitecture, but are supported by the Xilinx Virtex-5 implementations of the area-efficient and many-object microarchitectures. Further discussion of the issue is deferred to these implementations.

CPU-Based Execution Time Experiment Figure 5.11

The previously described experiments demonstrate the acceleration that was achieved using an FPGA implementation of the microarchitecture. However, these accelerations were curtailed by the clock frequency of the FPGA. In a practical system, an FPGA would not typically be used and, instead, the microarchitecture would be placed on a non-reconfigurable device such as an ASIC or in the fixed-function logic of a CPU or GPU. A CPU is a particularly interesting platform for the microarchitecture as there would be essentially no communication overhead between the software and the microarchitecture. Therefore, the purpose of this experiment was to determine the performance of the microarchitecture if it were implemented on a CPU.

The clock cycles counts from the 'Hardware-Based Execution Time Experiment' were used to compute two metrics, which correspond to Figure 5.11:

Excluding L2 Cache The 'Excluding Communication Interface' clock cycle counts from the 'Hardware-Based Execution Time Experiment' converted to execution times using the 1.8 GHz clock frequency of the host system's CPU.

Including L2 Cache The 'Excluding L2 Cache' metric with the estimated effect of the level 2 (L2) cache taken into account. To compute this, it was assumed that all of the data used by the microarchitecture were not available from the level 1 (L1) cache but were present in the L2 cache. This was a valid assumption as the data would have been used by the CPU during AABB updating, but the quantity of data processed during AABB updating means that it would be unlikely that the relevant data would be present in the small L1 cache. Based on Shanley [2005, pp 1009–1059], it was determined that the L1 data cache of the 1.8 GHz Intel Pentium 4 [Intel, 2002] was 8 KiB in size, was 4-way set associative and had a 64-byte line size. The L2 unified cache was 512 KiB in size, was 8-way set associative and had a 128-byte line size. Based on these details, data would be read from the L2 cache in 64-byte blocks. An L1 cache miss, resulting in an L2 cache access, costed approximately twenty-seven cycles according to Manegold and Boncz [2000]. Based on this data and the formula $24n$, which reflects the numbers of bytes required to store n objects, it was possible to compute the effect of the L2 cache assumption. The cache accesses required and the cycle penalties for each quantity of objects are tabulated in Table 5.2. The complete cycle values, converted into execution times, were added to the 'Excluding L2 Cache' execution times to compute the effect of the L2 cache.

Based on these execution times, a variety of acceleration metrics was computed. These metrics compared the metrics from this experiment with those from the Bullet Physics SDK algorithms from the 'Software-Based Execution Time Experiment'. All accelerations were based on AABB processing execution times, as the CPU-based execution times cannot consider AABB updating costs.

From these results, it can be seen that the overall pattern of execution times and accelerations remains similar to the 'Hardware-Based Execution Time Experiment'. Therefore, the same undesirable $O(n^2)$ computational complexity is apparent, although this is of less consequence as a result of the

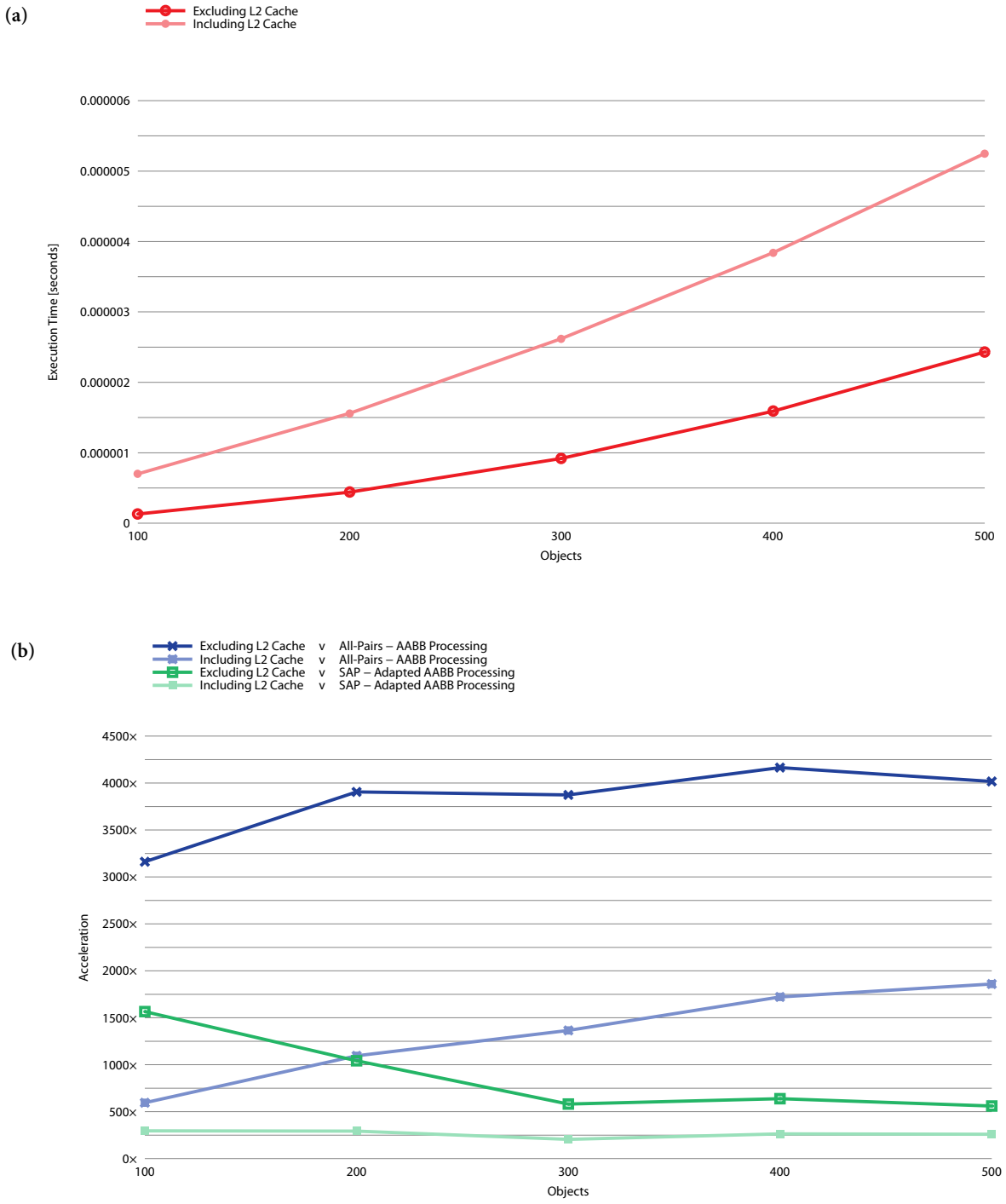


Figure 5.11: *Results for the 'CPU-Based Execution Time Experiment' of Section 5.2.1.* **(a)** *Execution times.* **(b)** *Accelerations. The data are tabulated in Table D.3.*

Objects	Cache Accesses	Cycle Penalty
100	38	1026
200	75	2025
300	113	3051
400	150	4050
500	188	5076

Table 5.2: *The cycle penalties resulting from modelling the L2 cache for the 'CPU-Based Execution Time Experiment' of Section 5.2.1.*

increased clock frequency. The results also show that the L2 cache has a significant effect on the execution times of the benchmarks, and that this effect increases with higher object counts, as additional cache accesses become necessary. Despite this, the high clock frequency means that execution times are still low, making this of little concern. Moreover, there is little scope for reducing this effect as the cache is designed to be as efficient as possible. The longest execution time is 0.0000024 seconds when the L2 cache is disregarded and 0.0000052 seconds when the L2 cache is considered. Both execution times occur for the 500-object benchmark. These extremely short execution times are sufficiently high to permit all applications to execute at 60 fps or greater.

Focusing on the accelerations, these extend to 4164.038× when compared to all-pairs and to 1565.546× when compared to SAP, when disregarding the effect of the L2 cache. Considering the L2 cache reduces the acceleration to 1859.244× when compared to all-pairs and to 294.779× when compared to SAP. Overall, each acceleration has increased by 23.5× when the effect of the L2 cache is disregarded.

It is likely that the value of m could be significantly increased due to the better resources available to CPUs. This would further increase the acceleration, although the value of m would be tempered by the demand for resources from other parts of the CPU. Alternatively, the microarchitecture could be implemented on the fixed-function logic of a GPU. One example, the NVIDIA GeForce GTX 285, is clocked at 1476 MHz [NVIDIA, 2009b]. Implementing the microarchitecture on this GPU would increase all of the FPGA-clock-cycle–based accelerations by 19.3× if the effect of the caches were disregarded. As for the CPU, it is likely that m could be increased, resulting in additional accelerations. One disadvantage of the GPU, however, is that the necessary communication overhead would reduce the acceleration. Despite this, a reasonable acceleration, exceeding that achievable on the FPGA, can be expected.

Evaluation Overall, the experiments have demonstrated that very significant accelerations are achievable using the hybrid system with the Xilinx Virtex-II implementation of the area-efficient microarchitecture. Compared to all-pairs, the accelerations extend to 12.1× when measured from a software perspective, to 177.269× when measured on the FPGA, to 1859.244× when measured on a theoretical CPU and to 4164.038× when measured on a theoretical CPU with the effect of the L2 cache disregar-

ded. In contrast, compared to SAP, the accelerations extend to 1.68× when measured from a software perspective, to 66.647× when measured on the FPGA, to 294.779× when measured on a theoretical CPU and to 1565.546× when measured on a theoretical CPU with the effect of the L2 cache disregarded. Every benchmark was significantly accelerated by the hybrid system, except for the 100-object benchmark when measured from a software perspective, where a slight deceleration was encountered.

Two issues were highlighted by the experiments. The first issue was the significant effect of the communication interface, which resulted in the deceleration of the 100-object benchmark. Within the communication interface, it was found that the PCI bus connecting the FPGA and CPU was inefficient and that the FPGA's memory accesses resulted in significant overhead. However, neither could be altered within the context of the Alpha Data ADM-XRC-II. The second issue was the microarchitecture's $O(n^2)$ computational complexity, which resulted in a slight reduction in acceleration at higher object counts. However, this was unproblematic for up to the 500 objects supported by the Xilinx Virtex-II implementation of the area-efficient microarchitecture. As both issues were fairly insignificant, it is clear that the hybrid system with the Xilinx Virtex-II implementation of the area-efficient microarchitecture is greatly beneficial for the acceleration of collision detection.

5.2.2 Xilinx Virtex-5

The same experiments were executed using the hybrid system with the Xilinx Virtex-5 implementation of the area-efficient microarchitecture as for the equivalent Xilinx Virtex-II system. Therefore, many experiment details, such as the purpose of the experiments and the metrics recorded, remain the same as in Section 5.2.1. These details are not repeated within this section.

All of the benchmarks for the hybrid system were executed on 'Computer 1', comprising a Quad-Core AMD Opteron 2350 (2 GHz) CPU [AMD, 2007] with 8 GiB RAM running Ubuntu GNU/Linux 8.04 LTS Desktop (64-bit) [Canonical, 2008]. A complete list of the computer's specifications can be located in Section B.3.

Metrics Table 5.3

	Total	Including Interface	Excluding Interface
Slices	51 840	27 595 (53%)	11 163 (21%)
BRAMs	288	202 (70%)	131 (45%)
Clock Frequency		166.$\bar{6}$ MHz	204 MHz
Memory Throughput		64.13 GB/s	78.336 GB/s
Effective Bandwidth			1340 MB/s
Effective Unidirectional Latency			0.32 µs

Table 5.3: *'Metrics' of Section 5.2.2.*

The resource utilisation metrics and clock frequencies from the Xilinx ISE place and route report were recorded for two variants of the microarchitecture. The first variant included the memory and HyperTransport interface logic supplied by DRC, while the second excluded this interface. Although the implementation with no interface cannot work on the DRC Accelium AC2030 module, its metrics illustrate the portion of the design consumed by the collision detection microarchitecture alone. After recording these metrics, the clock frequencies were used to compute the sustained memory throughput that could be achieved while the system was outputting data. It is evident that, without the interface, there would be sufficient slice and BRAM resources available to double m to thirty-two. In this case, the memory throughput would double to 156.672 GB/s, providing the clock frequency of 204 MHz were maintained. In addition, the effective bandwidth and effective unidirectional latency between the CPU and FPGA were measured. These metrics were determined by the CPU and the FPGA, and were not dependent on the microarchitecture.

Software-Based Execution Time Experiment Figure 5.12

The experiment utilised ten benchmarks based on the benchmarks from the Xilinx Virtex-II 'Software-Based Execution Time Experiment':

- **Objects – Quantity:** 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000
- **Objects – Type:** Uniform probability distribution
- **Objects – Location:** Uniform probability distribution
- **Size:** Uniform probability distribution
  - **Minimum:** 25 m
  - **Maximum:** 50 m
- **Acceleration:** Constant $(0, 0, 0)$ m/s^2
- **Linear Velocity:** Uniform probability distribution
  - **Minimum:** $(-25, -25, -25)$ m/s
  - **Maximum:** $(25, 25, 25)$ m/s
- **Angular Velocity:** Uniform probability distribution
  - **Minimum:** $(-2.5, -2.5, -2.5)$ rad/s
  - **Maximum:** $(2.5, 2.5, 2.5)$ rad/s

Each of the ten benchmarks had a different quantity of objects. All unspecified parameters were left at the defaults outlined in Section 3.6.2.

For the Bullet Physics SDK algorithms, the execution times follow the pattern of the equivalent Xilinx Virtex-II experiment, with both all-pairs and SAP exhibiting an upward trend over time. As before, all-pairs has a much more significant upward trend due to its $O(n^2)$ computational complexity. It is also clear that the hybrid system has significantly lower execution times than the other algorithms, although it too exhibits an upward trend. These execution times outperform those of the Xilinx Virtex-II implementation due to the combination of higher clock frequency, more sophisticated FPGA and

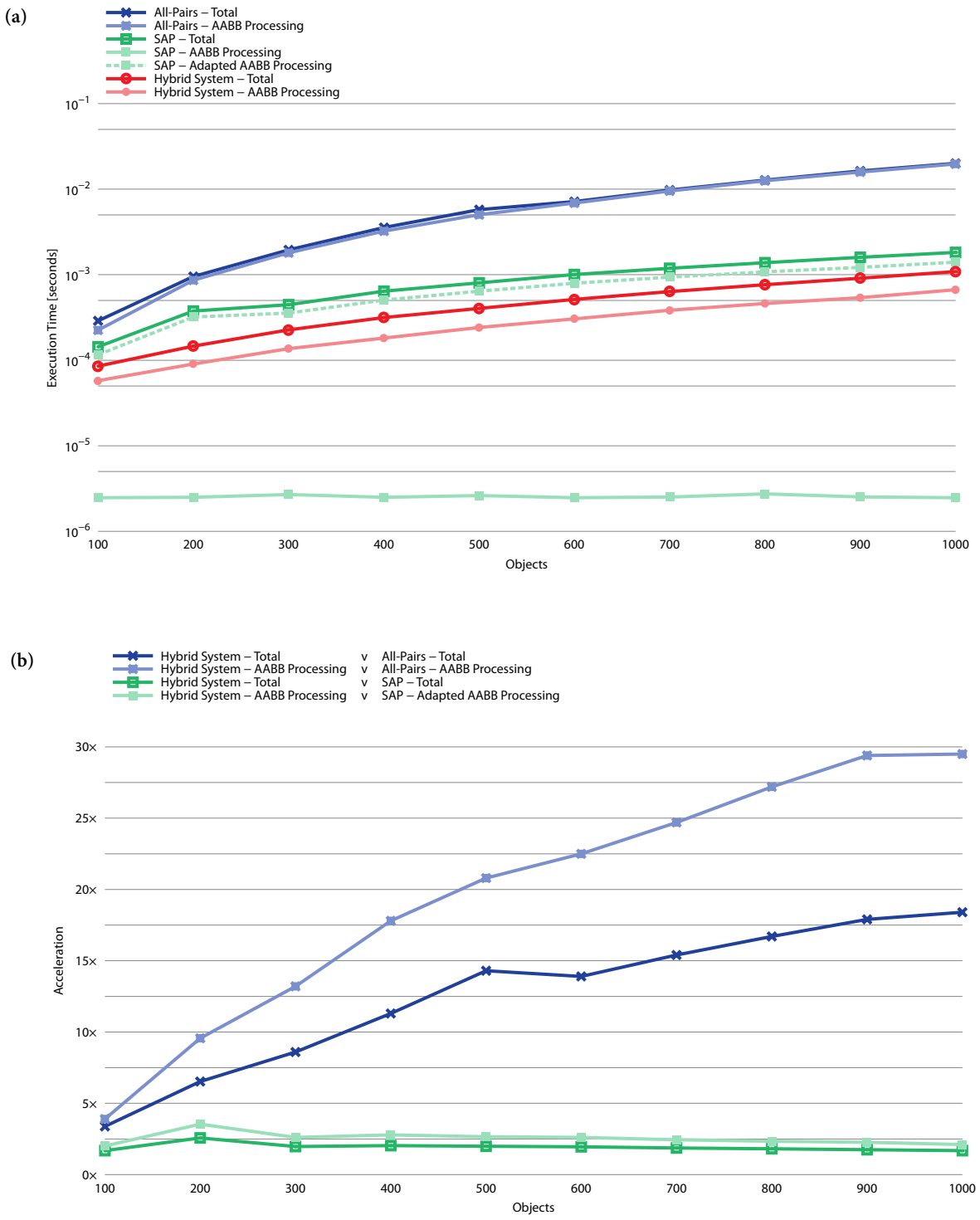


Figure 5.12: *Results for the 'Software-Based Execution Time Experiment' of Section 5.2.2.* **(a)** *Execution times. The equivalent linear-scale graph is Figure D.6.* **(b)** *Accelerations. The data are tabulated in Table D.4.*

faster FPGA-CPU interconnect. The system's longest execution time is 0.00109 seconds, which is well within the time boundaries necessary to achieve 30 fps.

Examining the accelerations reveals a pattern close to that of the Xilinx Virtex-II implementation. The highest acceleration is 29.5× when compared to all-pairs. A comparison with SAP yields an acceleration of up to 3.54×. In general, the accelerations are approximately two times greater than for the Xilinx Virtex-II implementation and all benchmarks achieve an acceleration. One noticeable characteristic of the acceleration offered the hybrid system compared to SAP is that it peaks at 200 to 400 objects before decreasing slightly. This decrease is the unfortunate consequence of the microarchitecture's O (n^2) computational complexity, which was also evident in the Xilinx Virtex-II implementation. However, as for the previous implementation, the acceleration factor remains high for all benchmarks and this slight deceleration can be effectively disregarded for all object quantities. Detailed discussion of the issue is, therefore, deferred until the many-object microarchitecture.

Collisions Experiment Figure 5.13

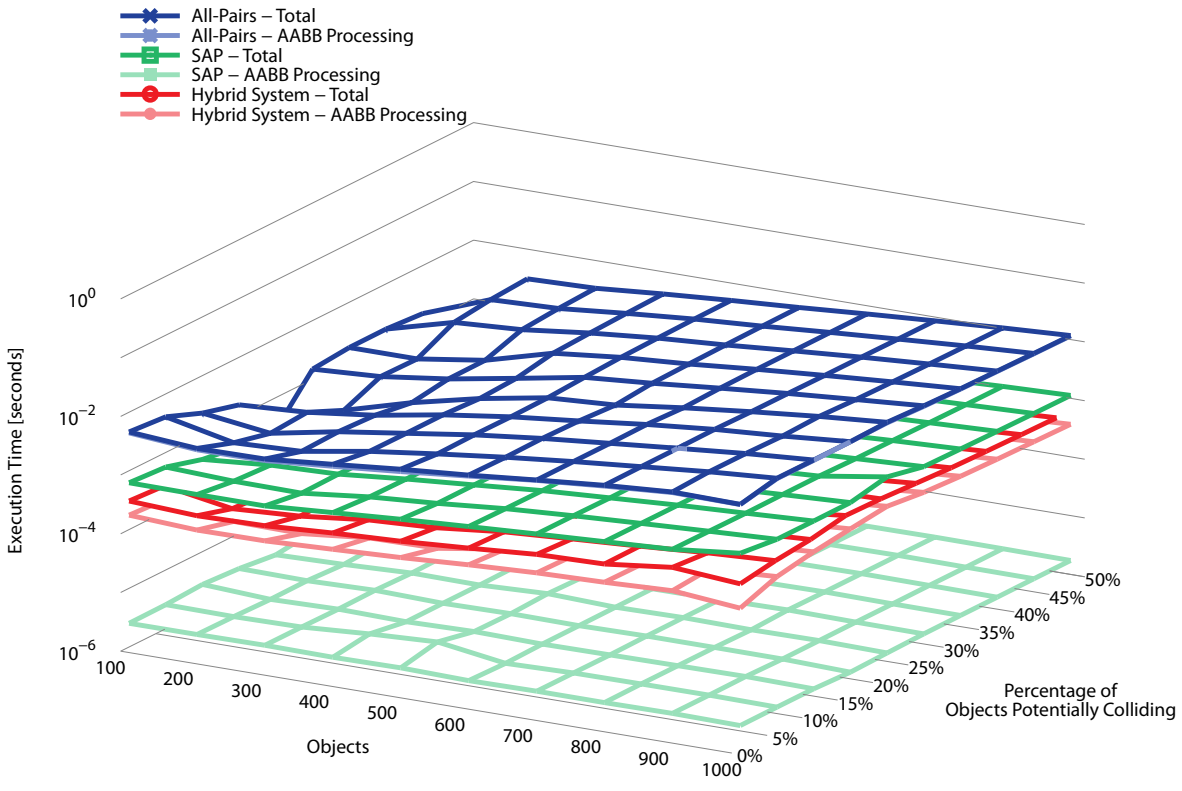


Figure 5.13: *Results for the 'Collisions Experiment' of Section 5.2.2. The equivalent linear-scale graph is Figure D.7.*

The results show that, for the benchmark with 1000 objects, between the least and most potential collisions, all-pairs exhibits a 45% increase in the total broad phase execution time a 42% increase in the AABB processing execution time. In contrast, SAP exhibits a decrease of 16% in the total but an increase

of 1% in AABB processing. The hybrid system exhibits a deterioration in performance, with an increase of 95% in the total and of 238% in AABB processing. Variability between adjacent values is significantly lower than for either software algorithm, making it simpler to achieve a constant frame rate.

Unlike the Xilinx Virtex-II implementation, the microarchitecture exhibits a very significant increase in AABB processing times. This increase must occur within in the communication interface, as it is the only part of the microarchitecture without consistent execution times. Within the interface, the execution time of the serialisation modules will increase as more data arrives on the parallel data streams. Similarly, more memory accesses will be required and these accesses may involve some latency, as memory requests may not be instantly accepted. Finally, it will take longer to read more potential collisions from the software as more requests to the FPGA module's memories will be required. In this case, it appears that memory access latencies are a significant source of the increased execution times. This is because the DRC Accelium AC2030's memories optimise for short write bursts, which is a pattern that cannot be effectively exploited by the potential collisions emitted by the microarchitecture. In contrast, the Alpha Data ADM-XRC-II attempts to achieve moderate performance for all access patterns.

The only way to reduce the effect of increased collisions would be to modify the FPGA module's memories to reduce access latencies for the write patterns used. However, this is not possible as the memory type is constrained by the FPGA module. Therefore, there is no way to rectify the timing increases that occur with increased collisions. Despite the performance degradation, significant accelerations are achieved even when the execution time increases are factored in, making the impact of these increases of little consequence. However, these results show that external memories would clearly be worth considering in detail if the microarchitecture were placed on a platform where the memories could be selected.

Hardware-Based Execution Time Experiment Figure 5.14

Both the number of clock cycles and the accelerations exhibit a similar trend to those of the Xilinx Virtex-II implementation, but the acceleration of the hybrid system now extends to 195.212× when compared to all-pairs and to 80.893× when compared to SAP. It is also clear that the maximum acceleration occurs around the 100- to 200-object mark, after which the acceleration decreases somewhat due to the $O(n^2)$ computational complexity. As before, this is unproblematic as the acceleration factors remain high up to 1000 objects and a significant acceleration is achieved despite this decrease. It is also clear that the communication interface creates a significant overhead. This results from the memories used, which are constrained by the choice of FPGA module. This is also unproblematic due to the high degree of acceleration achieved.

CPU-Based Execution Time Experiment Figure 5.15

All experiment details remained the same as for the Xilinx Virtex-II 'CPU-Based Execution Time Experiment' detailed in Section 5.2.1, except for the type of CPU used. Instead of the previous 1.8 GHz Intel Pentium 4 [Intel, 2002], the 2 GHz Quad-Core AMD Opteron 2350 [AMD, 2007] present in the

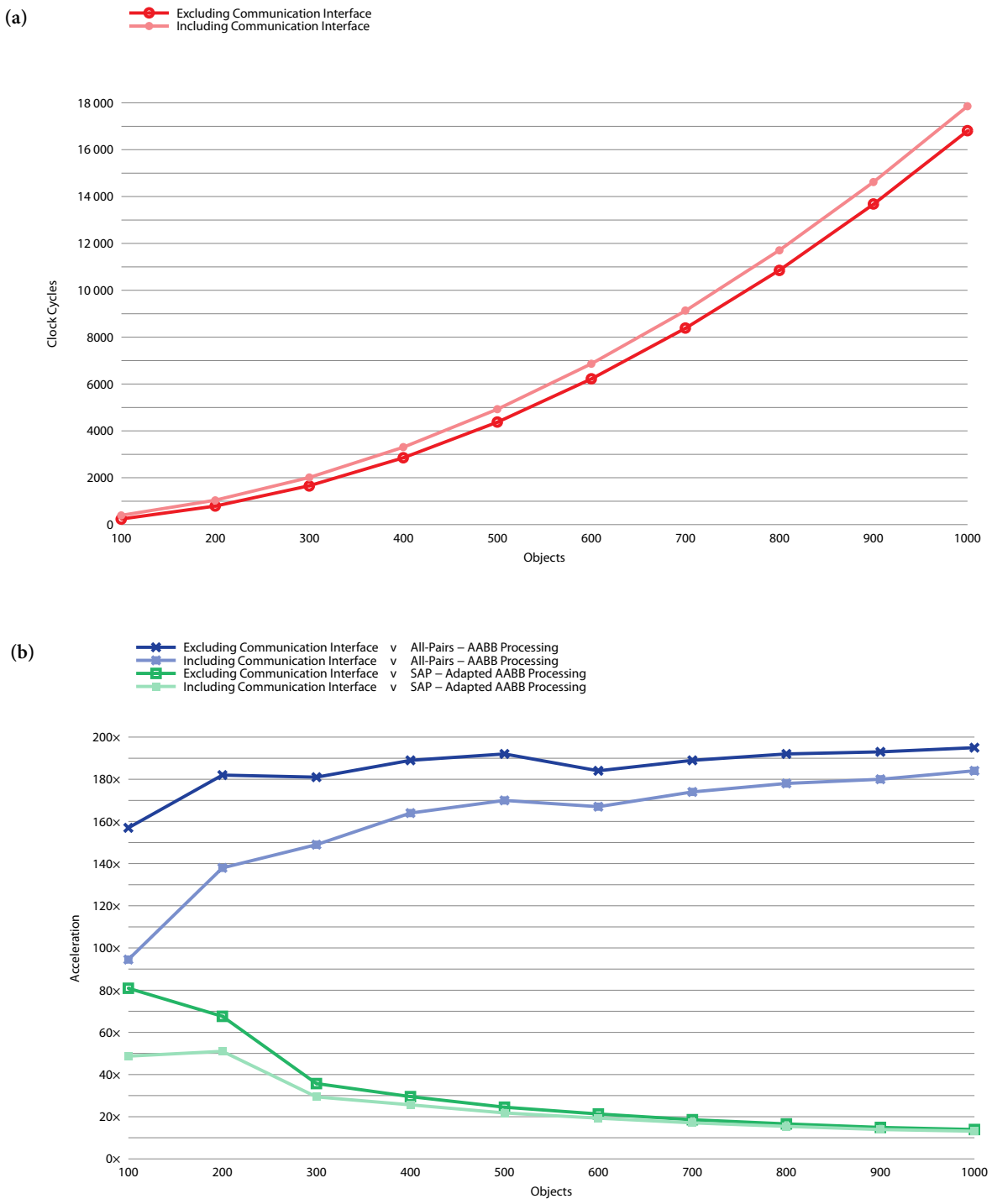


Figure 5.14: *Results for the 'Hardware-Based Execution Time Experiment' of Section 5.2.2.* **(a)** *Clock cycles.* **(b)** *Accelerations. The data are tabulated in Table D.5.*

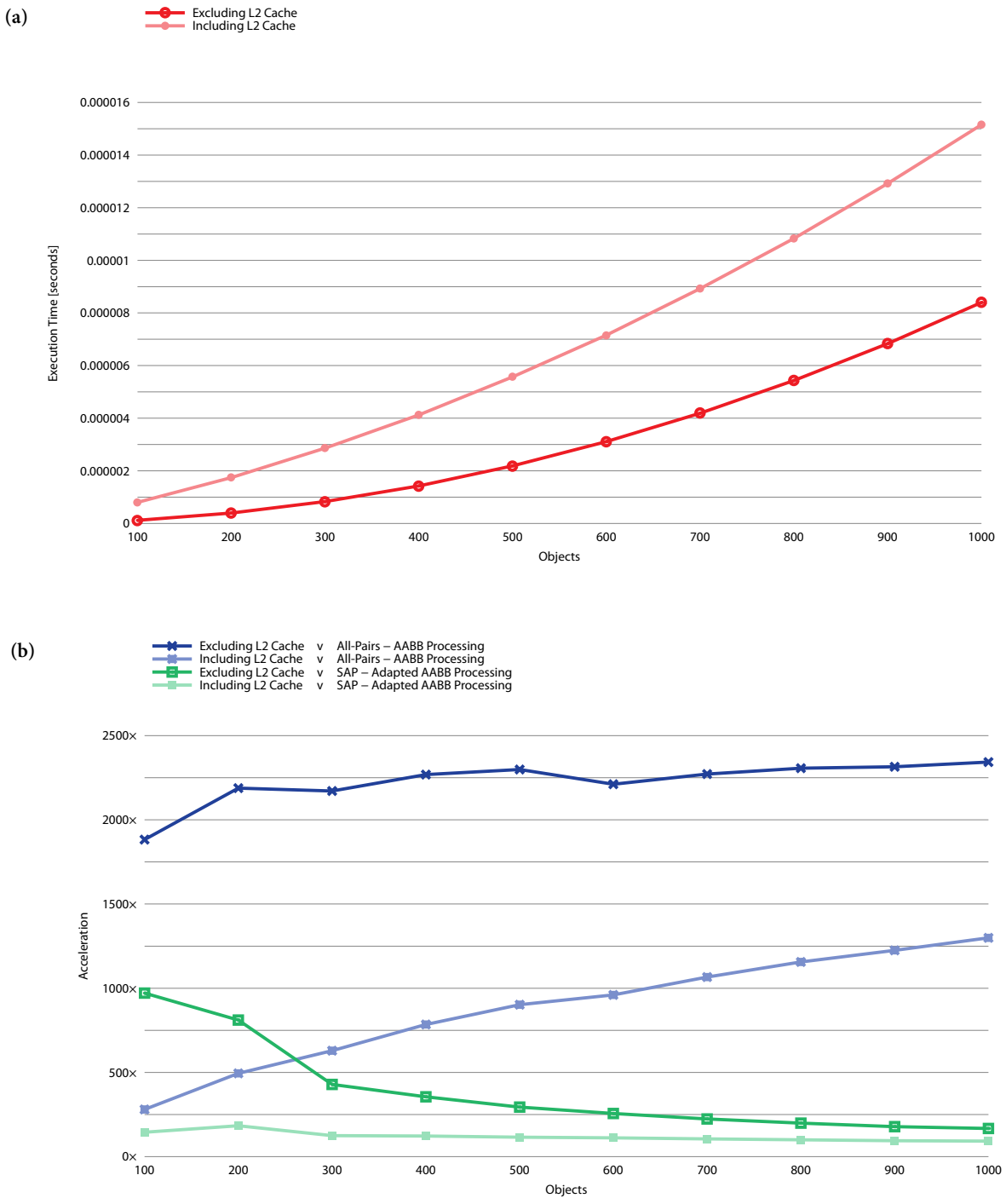


Figure 5.15: *Results for the 'CPU-Based Execution Time Experiment' of Section 5.2.2.* **(a)** *Execution times.* **(b)** *Accelerations. The data are tabulated in Table D.6.*

Xilinx Virtex-5 hybrid system was used. Using Manegold and Boncz [2000], it was determined that the L1 cache was 512 KiB in size, consisting of four 128 KiB caches per core. Each cache was 8-way set associative with a 64-byte line size. The L2 cache was 2 MiB in size, consisting of four 512 KiB caches per core. Each cache was 8-way set associative with a 64-byte line size. Based on this information, data would be read into the L2 cache in 64-byte blocks. It was determined that the penalty associated with accessing the L2 cache was thirty-six cycles. Using this information, the cache accesses required and the cycle penalties for each quantity of objects were computed. These are tabulated in Table 5.4.

Objects	Cache Accesses	Cycle Penalty
100	38	1368
200	75	2700
300	113	4068
400	150	5400
500	188	6768
600	225	8100
700	263	9468
800	300	10 800
900	338	12 168
1000	375	13 500

Table 5.4: *The cycle penalties resulting from modelling the L2 cache for the 'CPU-Based Execution Time Experiment' of Section 5.2.2.*

From the results, it is clear that the overall pattern of execution times and accelerations remains similar to the Xilinx Virtex-II 'CPU-Based Execution Time Experiment'. However, it is clear that the L2 cache has a greater effect on the execution times of the benchmarks due to the larger miss penalty. As before, the high clock frequency means that execution times are extremely low and this miss penalty is of little concern. The longest execution time is 0.0000084 seconds when the L2 cache is disregarded and 0.000015 seconds when the L2 cache is considered. Both execution times occur for the 1000-object benchmark. These extremely short execution times are sufficiently high to permit all applications to execute at 60 fps or greater. As for the previous experiments, the effect of the $O(n^2)$ computational complexity can be observed but the low execution times make this inconsequential.

Focusing on the accelerations, these extend to 2342.453× when compared to all-pairs and to 970.711× when compared to SAP, when disregarding the effect of the L2 cache. Considering the L2 cache reduces the acceleration to 1298.904× when compared to all-pairs and to 183.434× when compared to SAP. These accelerations are lower than for the Xilinx Virtex-II implementation, as the software is relatively faster on the Xilinx Virtex-5 system, which makes the hybrid system appear less efficient. The reduction in software execution times is primarily due to the 6 GiB increase in RAM, as the 0.2 GHz increase in clock frequency provides little benefit. The lower accelerations were not evident in either the 'Software-Based Execution Time Experiment', where the faster HyperTransport interconnect compensated for the increase in software execution speed, or the 'Hardware-Based Execution Time Ex-

periment', where the faster clock frequency of the FPGA compensated. The faster frequency does not compensate in this experiment as the increase is too low. Despite this, the accelerations provided by the theoretical CPU are still extremely large. Overall, each acceleration has increased by 12× when the effect of the L2 cache is disregarded. The NVIDIA GeForce GTX 285 would provide an acceleration of 8.9× over the FPGA-clock-cycle–based accelerations.

Evaluation All of these experiments have demonstrated that superior accelerations are achievable using the hybrid system with the Xilinx Virtex-5 implementation of the area-efficient microarchitecture, than with the Xilinx Virtex-II equivalent. Moreover, unlike the Xilinx Virtex-II implementation, every benchmark was accelerated by the hybrid system. Compared to all-pairs, the accelerations extended to 29.5× when measured from a software perspective, to 195.212× when measured on the FPGA, to 1298.904× when measured on a theoretical CPU and to 2342.543× when measured on a theoretical CPU with the effect of the L2 cache disregarded. In contrast, compared to SAP, the accelerations extended to 3.54× when measured from a software perspective, to 80.893× when measured on the FPGA, to 183.434× when measured on a theoretical CPU and to 970.711× when measured on a theoretical CPU with the effect of the L2 cache disregarded.

The two issues highlighted by the Xilinx Virtex-II implementation were also highlighted by the Xilinx Virtex-5 implementation. These were the effect of the communication interface, which could not be solved within the constraints of the DRC Accelium AC2030, and the $O(n^2)$ computational complexity. Neither issue was problematic due to the significant accelerations achieved.

5.3 Many-Object Microarchitecture

The same experiments were executed using the hybrid system with the Xilinx Virtex-5 implementation of the many-object microarchitecture as with the Xilinx Virtex-II and Xilinx Virtex-5 area-efficient systems. Therefore, many experiment details, such as the purpose of the experiments and the metrics recorded, remain the same as in Sections 5.2.1 and 5.2.2. These details are not repeated within this section.

All of the benchmarks for the hybrid system were executed on 'Computer 1', comprising a Quad-Core AMD Opteron 2350 (2 GHz) CPU [AMD, 2007] with 8 GiB RAM running Ubuntu GNU/Linux 8.04 LTS Desktop (64-bit) [Canonical, 2008]. A complete list of the computer's specifications can be located in Section B.3.

Metrics Table 5.5

The resource utilisation metrics and clock frequencies from the Xilinx ISE place and route report were recorded for the microarchitecture both including and excluding the interface logic. One noticeable characteristic of the resource utilisation metrics is that they show only a slight increase over the area-efficient microarchitecture's utilisation metrics. Despite this, examination has shown that the many-object microarchitecture consumes significantly more resources than the area-efficient microar-

	Total	Including Interface	Excluding Interface
Slices	51 840	29 053 (56%)	13 204 (25%)
BRAMs	288	187 (64%)	116 (40%)
Clock Frequency		133.$\bar{3}$ MHz	192.3 MHz
Memory Throughput		59.1$\bar{9}$ GB/s	73.843 GB/s
Effective Bandwidth			1340 MB/s
Effective Unidirectional Latency			0.32 µs

Table 5.5: *'Metrics' of Section 5.3.*

chitecture because of the multiplexers used in the reorder operation. The discrepancy between the reported metrics and these examinations results from the presence of sophisticated slices in the Xilinx Virtex-5 that support both multiplexing and typical logic. Therefore, the Xilinx ISE synthesiser was able to place the multiplexers within the same slices as existing logic, resulting in relatively low overall resource consumption. The Xilinx Virtex-II contained less sophisticated logic slices, which did not contain hardware that could be used for multiplexing. Instead, the multiplexers had to be implemented using generic reconfigurable logic, resulting in insufficient logic for the microarchitecture.

It is evident that, without the interface, there would be sufficient slice and BRAM resources available to double m to thirty-two. In this case, the memory throughput would double to 147.686 GB/s, providing the clock speed of 192.3 MHz were maintained. The memory throughput metrics are slightly lower than for the area-efficient microarchitecture as fewer collision tests were performed in parallel. However, by performing fewer collision tests in parallel, it is possible to eliminate the area-efficient microarchitecture's data duplication.

Software-Based Execution Time Experiment Figure 5.16

The experiment utilised sixteen benchmarks based on the benchmarks from the Xilinx Virtex-5 'Software-Based Execution Time Experiment':

- **Objects – Quantity:** 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 4000, 8000, 16 000
- **Objects – Type:** Uniform probability distribution
- **Objects – Location:** Uniform probability distribution
- **Size:** Uniform probability distribution
  - **Minimum:** 25 m
  - **Maximum:** 50 m
- **Acceleration:** Constant $(0, 0, 0)$ m/s^2
- **Linear Velocity:** Uniform probability distribution
  - **Minimum:** $(-25, -25, -25)$ m/s
  - **Maximum:** $(25, 25, 25)$ m/s

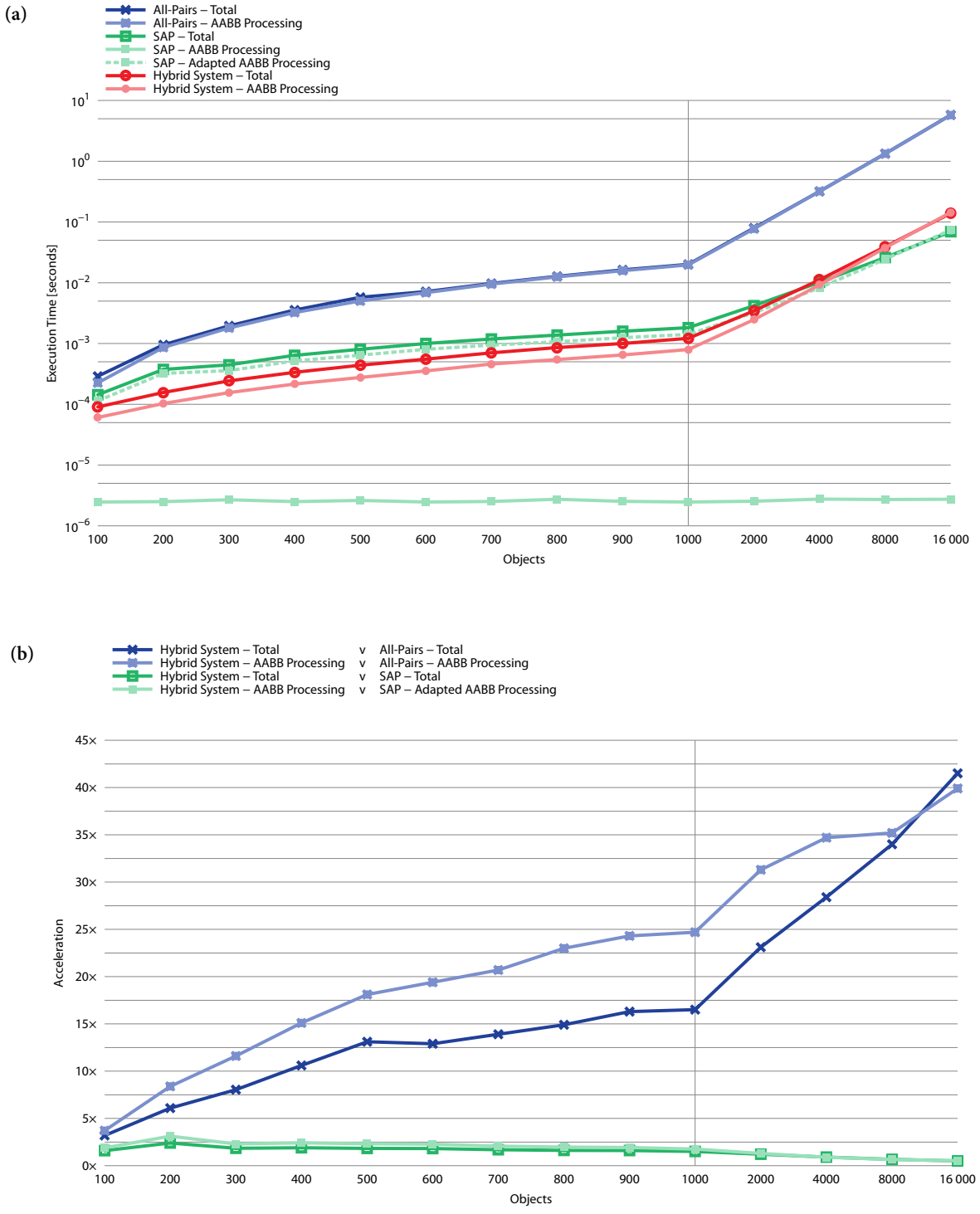


Figure 5.16: *Results for the 'Software-Based Execution Time Experiment' of Section 5.3.* **(a)** *Execution times. The equivalent linear-scale graph is Figure D.10.* **(b)** *Accelerations. The data are tabulated in Table D.7*

- **Angular Velocity:** Uniform probability distribution
  - **Minimum:** $(-2.5, -2.5, -2.5)$ rad/s
  - **Maximum:** $(2.5, 2.5, 2.5)$ rad/s

Each of the sixteen benchmarks had a different quantity of objects. All unspecified parameters were left at the defaults outlined in Section 3.6.2. For the first ten benchmarks, the number of objects increased by 100 over the previous benchmark. The remaining benchmarks doubled the number of objects from the previous benchmark. This lead to a change of scale in each of the graphs after the first ten benchmarks. This change of scale was indicated using a vertical line at the 1000-object mark.

The execution times exhibit the same trends as for the previous implementations, although the many-object implementation has slightly longer execution times than the area-efficient implementation, as it performs fewer comparisons in parallel. The execution times extend to 0.139 seconds for 16 000 objects. Unfortunately, this is not fast enough to achieve 30 fps. The accelerations also exhibit similar behaviour to the previous implementations. An acceleration of up to 39.9× is achieved for every benchmark when compared to all-pairs. In contrast, SAP achieves its maximum acceleration of 3.12× for 200 objects, after which its acceleration factor declines somewhat. This decline continues until a deceleration is evident in the benchmarks with 4000 or more objects. This is partially attributable to the reduced parallelism of the many-object microarchitecture, which enables the $O(n^2)$ computational complexity of the serial algorithm to play a greater role in the overall performance. It is also a consequence of the greater object sizes that can be simulated using the microarchitecture, as larger quantities are affected more significantly by the computational complexity. As the previous microarchitectures were limited to smaller simulations, the undesirable effects of the computational complexity were less apparent.

The deceleration for 4000 or more objects is an unfortunate side effect of using all-pairs for the microarchitecture. Other algorithms with more desirable computational complexities were analysed for microarchitecture implementation in Section 3.2. However, all were found to insufficiently exploit parallelism, making any microarchitecture implementation of these algorithms unlikely to gain an appreciable acceleration. As a result, any solution is restricted to the current microarchitecture.

Good performance is attained when simulating the smaller benchmarks using the microarchitecture. The key, therefore, is to divide larger benchmarks into smaller ones. This can be done using spatial partitioning, as outlined in Section 3.4, which was also previously discussed as a means of circumventing microarchitecture object limits. To use spatial partitioning effectively, the cell size should be small enough to enable the microarchitecture to provide a significant acceleration, but large enough that the communication overhead does not obliterate this acceleration. In this case, a cell size of 2048 could be used so that a 4000-object benchmark would be processed in two iterations. The overall goal of spatial partitioning is to reduce the complexity of the hybrid system to either $O(n)$ or $O(n \log n)$. Spatial partitioning is considered in detail in Section 5.4.

Collisions Experiment Figure 5.17

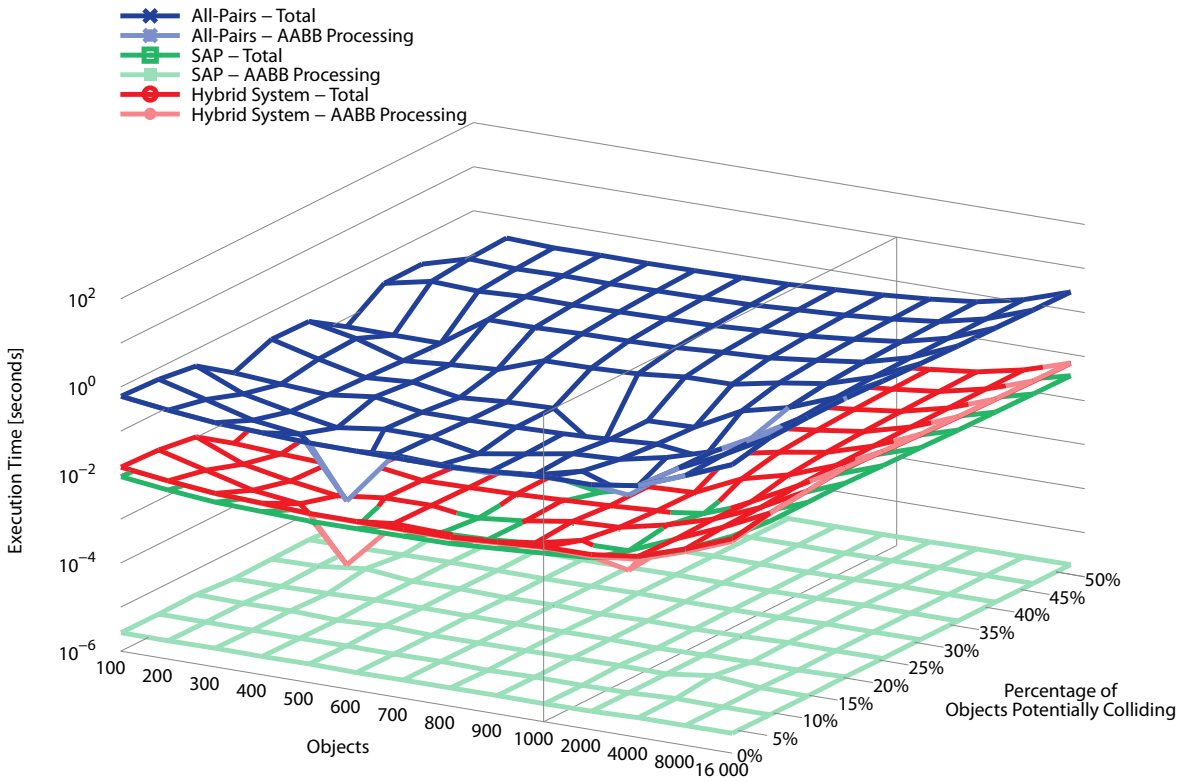


Figure 5.17: *Results for the 'Collisions Experiment' of Section 5.3. The equivalent linear-scale graph is Figure D.11.*

The results show that, for the benchmark with 16 000 objects, between the least and most potential collisions, all-pairs exhibits a 76% increase in the total broad phase execution time and a 78% increase in the AABB processing execution time. In contrast, SAP exhibits a decrease of 17% in the total but an increase of 1% in AABB processing. The hybrid system exhibits a deterioration in performance, with an increase of 134% in the total and of 164% in AABB processing. Variability between adjacent values is significantly lower than for either software algorithm, making it simpler to achieve a constant frame rate.

The increase in the AABB processing execution time is lower than for 1000 objects with the Xilinx Virtex-5 implementation of the area-efficient microarchitecture. However, the increase is still significant and it would be desirable to reduce this value. As for the previous implementation, it was determined that the increase is primarily due to the behaviour of the DRC Accelium AC2030's memories. However, these are constrained by the FPGA module and cannot be altered. Consequently, there is no solution to this problem with the current implementation.

Hardware-Based Execution Time Experiment Figure 5.18

Both the number of clock cycles and the accelerations exhibit a similar trend to those of the previous implementations, but it is evident that the gains are reduced since fewer collision tests are performed in parallel. When compared to all-pairs, the microarchitecture achieves a maximum acceleration of 95.604×, while a comparison to SAP results in a maximum acceleration of 36.774×. However, there is a noticeable decline in the acceleration relative to SAP as the number of objects increases. In particular, an acceleration of only 1.23× is achieved for 16 000 objects. This is unlikely to be considered sufficiently fast to warrant use of the microarchitecture. The cause of this performance degradation can again be attributed to the O (n^2) computational complexity of the microarchitectures. The solution to this degradation is the use of spatial partitioning.

CPU-Based Execution Time Experiment Figure 5.19

As for the Xilinx Virtex-5 implementation of the area-efficient microarchitecture, the 2 GHz Quad-Core AMD Opteron 2350 [AMD, 2007] was used. The cache accesses and cycle penalties were computed using the same cache details. These are tabulated in Table 5.6.

Objects	Cache Accesses	Cycle Penalty
100	38	1368
200	75	2700
300	113	4068
400	150	5400
500	188	6768
600	225	8100
700	263	9468
800	300	10 800
900	338	12 168
1000	375	13 500
2000	750	27 000
4000	1500	54 000
8000	3000	108 000
16 000	6000	216 000

Table 5.6: *The cycle penalties resulting from modelling the L2 cache for the 'CPU-Based Execution Time Experiment' of Section 5.3.*

From the results, it is clear that the overall pattern of execution times and accelerations remains similar to those of the previous implementations. The longest execution time is 0.004 seconds when the L2 cache is disregarded and 0.0041 seconds when the L2 cache is considered. Both execution times occur for the 16 000-object benchmark. Focusing on the accelerations, these extend to 1434.056× when compared to all-pairs and to 552.058× when compared to SAP, when disregarding the effect of the L2 cache. Considering the L2 cache reduces the acceleration to 1396.418× when compared to all-pairs and to 155.668× when compared to SAP. Overall, each acceleration has increased by 15× when the effect

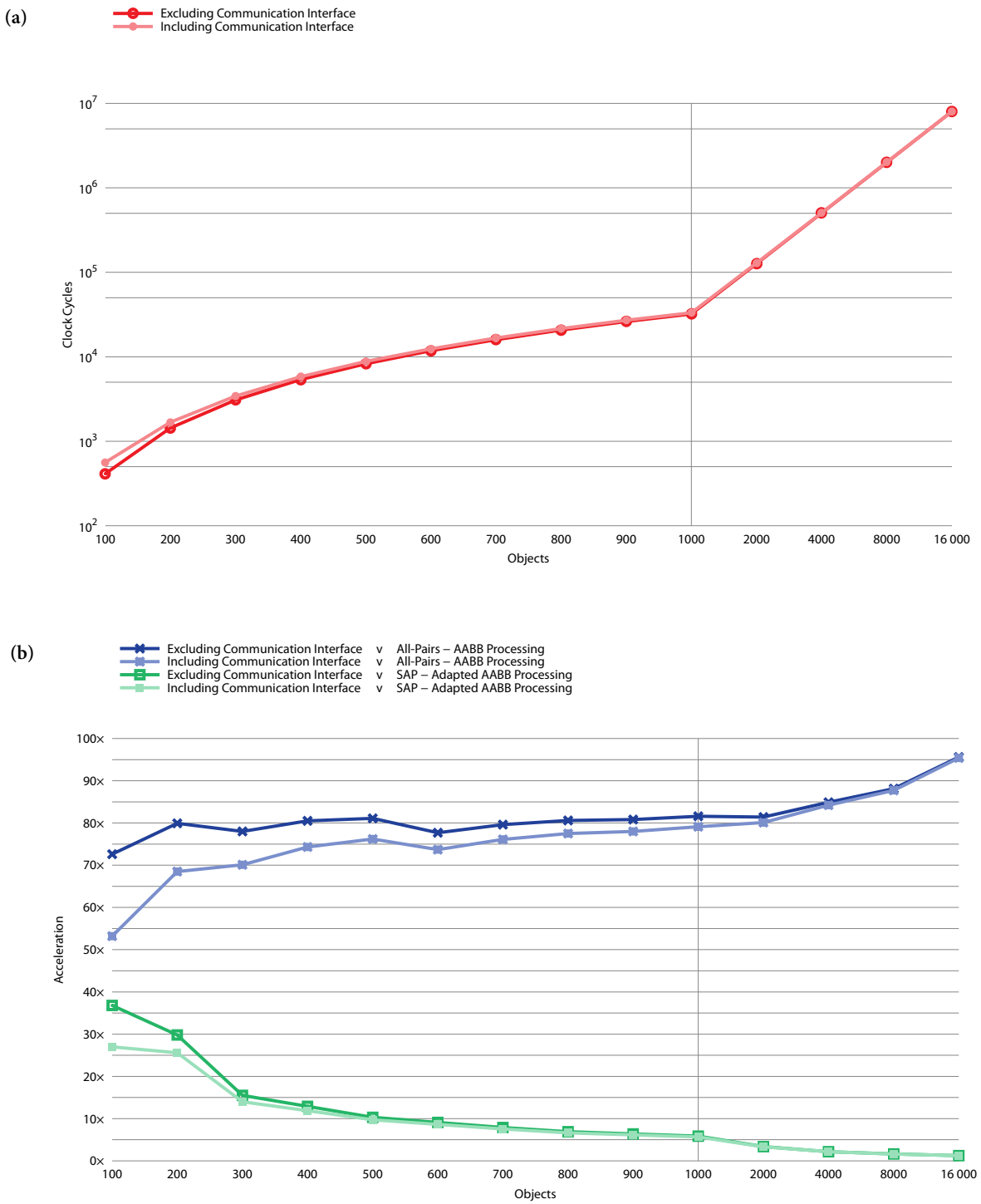


Figure 5.18: *Results for the 'Hardware-Based Execution Time Experiment' of Section 5.3.* **(a)** *Clock cycles. The equivalent linear-scale graph is Figure D.12.* **(b)** *Accelerations. The data are tabulated in Table D.8.*

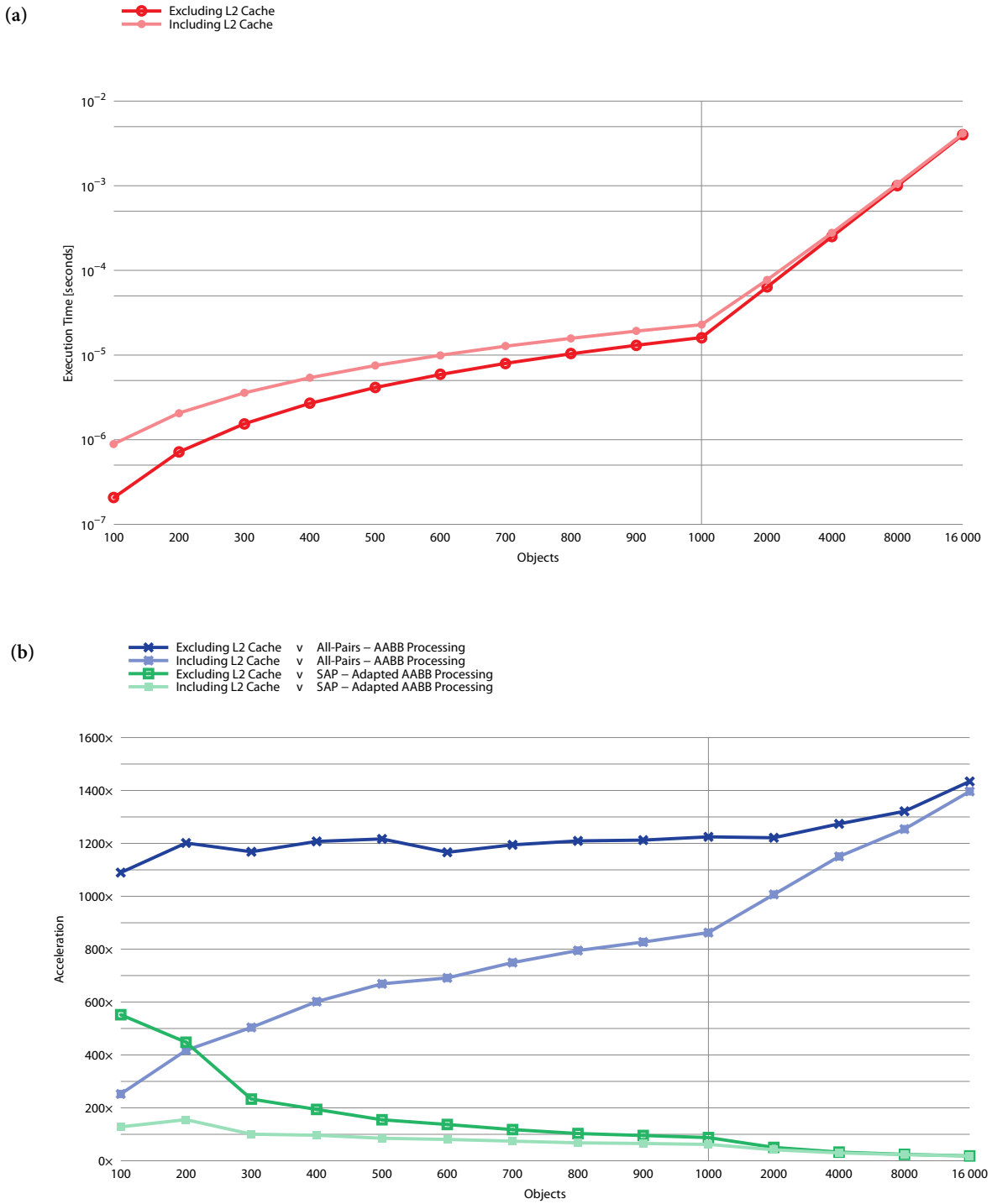


Figure 5.19: *Results for the 'CPU-Based Execution Time Experiment' of Section 5.2.2.* **(a)** *Execution times. The equivalent linear-scale graph is Figure D.14.* **(b)** *Accelerations. The data are tabulated in Table D.9.*

of the L2 cache is disregarded. The NVIDIA GeForce GTX 285 would provide an acceleration of 11× over the FPGA-clock-cycle–based accelerations. Despite the significant acceleration evident in all of the benchmarks, the effect of the O (n^2) computational complexity is also apparent in this experiment. However, unlike the previous Xilinx Virtex-5 many-object microarchitecture experiments, the significant acceleration provided to all of the benchmarks means this is of little concern.

Evaluation All of these experiments demonstrated that significant accelerations are achievable using the hybrid system with the Xilinx Virtex-5 implementation of the many-object microarchitecture. Compared to all-pairs, the accelerations extended to 39.9× when measured from a software perspective, to 95.604× when measured on the FPGA, to 1396.418× when measured on a theoretical CPU and to 1434.056× when measured on a theoretical CPU with the effect of the L2 cache disregarded. In contrast, compared to SAP, the accelerations extended to 3.12× when measured from a software perspective, to 36.774× when measured on the FPGA, to 128.018× when measured on a theoretical CPU and to 552.058× when measured on a theoretical CPU with the effect of the L2 cache disregarded.

Despite the accelerations achieved, benchmarks with 4000 or more objects were decelerated relative to SAP when considered from a software perspective. Moreover, consideration from a hardware perspective resulted in relatively poor performance for some of the benchmarks. This poor performance results from the undesirable O (n^2) computational complexity of the microarchitecture, which was also highlighted by the previous implementations, albeit to a lesser extent. Clearly, this issue needs to be addressed. The proposed solution is spatial partitioning, which can divide the inefficient large simulations into a number of smaller simulations that can be processed with a high degree of acceleration. This solution is considered in detail in the following section.

The previous issue of the communication interface consuming a significant proportion of execution time was also highlighted by the experiments performed on the current system. However, as before, this is a relatively insignificant issue and can be effectively ignored. Moreover, there is no clear solution to the issue as the interface is almost entirely constrained by the choice of FPGA module.

5.4 Spatial Partitioning

Only minimal logic is required to adapt the previous microarchitectures to allow them to process multiple microarchitecture cells. Consequently, this logic was included in the implementations described previously and the resource utilisation metrics, clock frequencies and memory throughputs are those already listed. In addition, the effective bandwidths and effective unidirectional latencies remain the same.

5.4.1 Area-Efficient Microarchitecture, Xilinx Virtex-II

Spatial partitioning was not tested using the hybrid system with the Xilinx Virtex-II implementation of the area-efficient microarchitecture. This was because the Xilinx Virtex-5 implementations were sufficient to demonstrate the benefit of spatial partitioning. In addition, the Xilinx Virtex-5 was closer

to what would be used in a practical system, making the results more meaningful. These Xilinx Virtex-5 implementations should be sufficient to deduce the effect of spatial partitioning on the performance of any practical system.

5.4.2 Area-Efficient Microarchitecture, Xilinx Virtex-5

A cell size of 1024 was selected for spatial partitioning using the hybrid system with the Xilinx Virtex-5 implementation of the area-efficient microarchitecture. This reflected the object limitation of this implementation. Moreover, the results for the equivalent hybrid system without spatial partitioning suggested that reduction of this cell size would not result in significantly better performance.

All of the benchmarks for the hybrid system were executed on 'Computer 1', comprising a Quad-Core AMD Opteron 2350 (2 GHz) CPU [AMD, 2007] with 8 GiB RAM running Ubuntu GNU/Linux 8.04 LTS Desktop (64-bit) [Canonical, 2008]. A complete list of the computer's specifications can be located in Section B.3.

Software-Based Execution Time Experiment Figure 5.20

The experiment utilised four benchmarks based on the benchmarks from the Xilinx Virtex-5 many-object 'Software-Based Execution Time Experiment':

- **Objects – Quantity:** 2000, 4000, 8000, 16 000
- **Objects – Type:** Uniform probability distribution
- **Objects – Location:** Uniform probability distribution
- **Size:** Uniform probability distribution
  - **Minimum:** 25 m
  - **Maximum:** 50 m
- **Acceleration:** Constant $(0, 0, 0)$ m/s^2
- **Linear Velocity:** Uniform probability distribution
  - **Minimum:** $(-25, -25, -25)$ m/s
  - **Maximum:** $(25, 25, 25)$ m/s
- **Angular Velocity:** Uniform probability distribution
  - **Minimum:** $(-2.5, -2.5, -2.5)$ rad/s
  - **Maximum:** $(2.5, 2.5, 2.5)$ rad/s

Each of the four benchmarks had a different quantity of objects. All unspecified parameters were left at the defaults outlined in Section 3.6.2.

The benchmarks were executed for 1000 cycles using the hybrid system with the uniform grid and AABB tree spatial partitioning algorithms, as well as the Bullet Physics SDK's all-pairs and SAP. Each cycle's broad phase execution time was recorded and the mean of the cycle execution times was subsequently computed. The execution time metrics, corresponding to Figure 5.20, were:

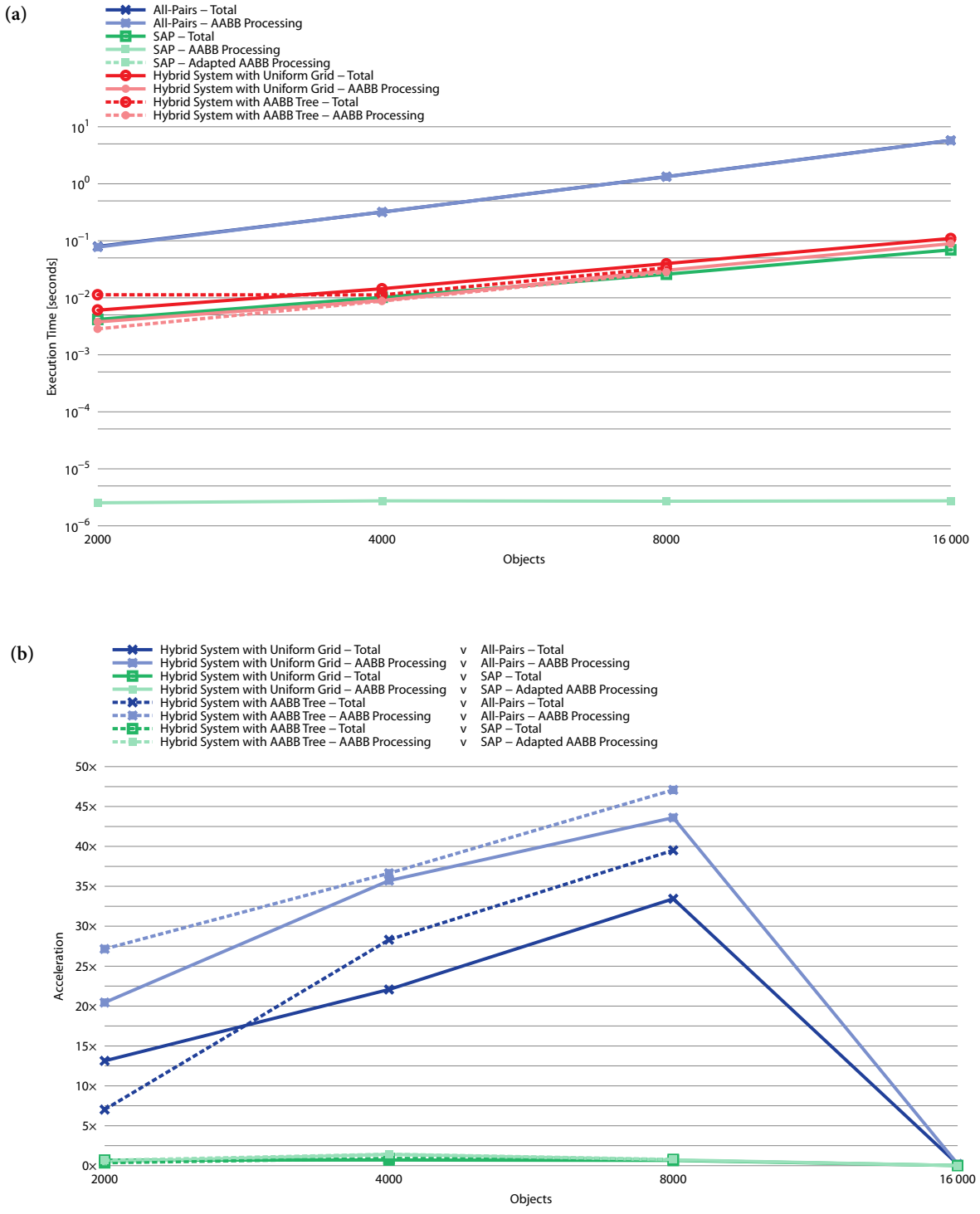


Figure 5.20: *Results for the 'Software-Based Execution Time Experiment' of Section 5.4.2.* **(a)** *Execution times. The equivalent linear-scale graph is Figure D.16.* **(b)** *Accelerations. The data are tabulated in Table D.10.*

All-Pairs – Total The total time consumed by the Bullet Physics SDK's all-pairs. This comprises AABB updating, which involves recomputing the AABBs to reflect updated object positions, and AABB processing, which involves the execution of the collision tests on the AABB pairs.

All-Pairs – AABB Processing The time consumed by the Bullet Physics SDK's all-pairs during AABB processing.

SAP – Total The total time consumed by the Bullet Physics SDK's SAP.

SAP – AABB Processing The time consumed by the Bullet Physics SDK's SAP during AABB processing.

SAP – Adapted AABB Processing The total time consumed by the Bullet Physics SDK's SAP less the time consumed by all-pairs' AABB updating. This metric was computed because SAP's AABB processing does not reflect the true value of processing, since most of it is performed in tandem with updating due to the nature of the algorithm. In previous experiments, this metric was computed by subtracting the hybrid system's AABB updating, but this was not possible for this experiment as the hybrid system exhibited highly variable times.

Hybrid System with Uniform Grid – Total The total time consumed by the hybrid system using the uniform grid spatial partitioning algorithm. This includes the software-based AABB updating, uniform grid construction and preparation of data for transfer, as well as the microarchitecture-based AABB processing. The two data transfers are also included.

Hybrid System with Uniform Grid – AABB Processing The time consumed by the hybrid system using the uniform grid spatial partitioning algorithm during AABB processing and uniform grid construction. This includes the software-based uniform grid construction and preparation of data for transfer, as well as the microarchitecture-based AABB processing. The two data transfers are also included.

Hybrid System with AABB Tree – Total The total time consumed by the hybrid system using the AABB tree spatial partitioning algorithm. This includes the software-based AABB updating, AABB tree construction and preparation of data for transfer, as well as the microarchitecture-based AABB processing. The two data transfers are also included.

Hybrid System with AABB Tree – AABB Processing The time consumed by the hybrid system using the AABB tree spatial partitioning algorithm during AABB processing and AABB tree construction. This includes the software-based AABB tree construction and preparation of data for transfer, as well as the microarchitecture-based AABB processing. The two data transfers are also included.

Based on these recorded execution times, a variety of acceleration metrics was computed. These metrics directly compared pairs of the previously described execution metrics.

The uniform grid could be used for all of the benchmarks. However, it was not possible to use the AABB tree for the benchmark with 16 000 objects. With this benchmark, tree construction reached a condition whereby some cells contained more than 1024 objects and further axis splitting was unable to reduce cell size. This occurred because all of the objects lay either on or to one side of the spatial median of all three axes. As a result, one or both of the resultant cells contained the same number of objects as the parent cell. This meant that there was no benefit in continuing to partition space and the algorithm was terminated as this point. The problem primarily resulted from the density of objects in the benchmark. This issue could occur for any spatial partitioning algorithm that adapts to scene complexity.

To resolve this issue, the benchmark could be redesigned to reduce object density. This would minimise the number of objects split during partitioning, which would lower the possibility of undividable cells exceeding 1024 objects. However, redesigning the benchmark is not a desirable solution as it may not always be possible to redesign a scenario to avoid algorithm difficulties. Therefore, a more sophisticated solution would be to redesign the AABB tree. In particular, a redesign should focus on alterative partitioning strategies as it is during partitioning that the problem is encountered. A promising alternative is to partition at the object median instead of the spatial median. Using the object median involves sorting all of a cell's objects along a specific axis and subsequently splitting so that each new cell contains half of the objects. However, sorting the objects is inefficient unless the tree is refitted between collision detection cycles. Therefore, using this metric would require significant changes to the tree. Alternatively, more sophisticated hierarchical structures such as the k-d tree or the BSP tree could be utilised. The BSP tree is particularly promising as these trees have more flexibility in their choice of partitioning axis, which provides the possibility of reducing the number of straddling objects. BSP trees are rarely used for moving objects in interactive environments due to the complexity of their construction. However, unlike most applications, it should be relatively easy to construct the trees required by the microarchitectures, as partitioning would not extend to the level of individual objects. Another interesting hierarchy would be an appropriate adaptation of the Bullet Physics SDK's DBVT, which achieved good results when analysed with the benchmarking framework. Overall, there exist a variety of ways in which this problem could be rectified, but it is clear that it will be challenging to find an efficient solution.

Focusing on the results for the uniform grid, a significant acceleration is achieved for up to 8000 objects when the execution times are compared to all-pairs. However, this turns into a deceleration for 16 000 objects. In addition, all benchmarks achieve a deceleration relative to SAP, except for the 4000-object benchmark. The uniform grid provides an acceleration of up to 43.592× when compared to all-pairs and of up to 1.406× when compared to SAP. This results in a greater acceleration factor for the 4000- and 8000-object benchmarks than is available using the many-object microarchitecture.

Turning attention to the AABB tree, a similar pattern of results is attained, although the lack of a 16 000 benchmark means that all of the benchmarks achieve an acceleration when compared to all-pairs. The acceleration factor is up to 47.081× when compared to all-pairs and is up to 1.442× when compared to SAP. As for the uniform grid, this results in a greater acceleration factor for the 4000-

and 8000-object benchmarks than is available using the many-object microarchitecture. Overall, it is also clear that the AABB tree outperforms the uniform grid.

The greater accelerations for the 4000- and 8000-object benchmarks demonstrate the validity of the spatial partitioning approach. However, the negative results suggest that there are some issues with the current implementation. Some of these problems can be attributed to the costs associated with constructing complex spatial partitioning data structures, and with converting the PCS indices from per-cell indices to global indices. However, these costs would not entirely explain the behaviour encountered. Therefore, it is vital to perform more experiments to clarify the source of these issues.

Cells Experiment Figure 5.21

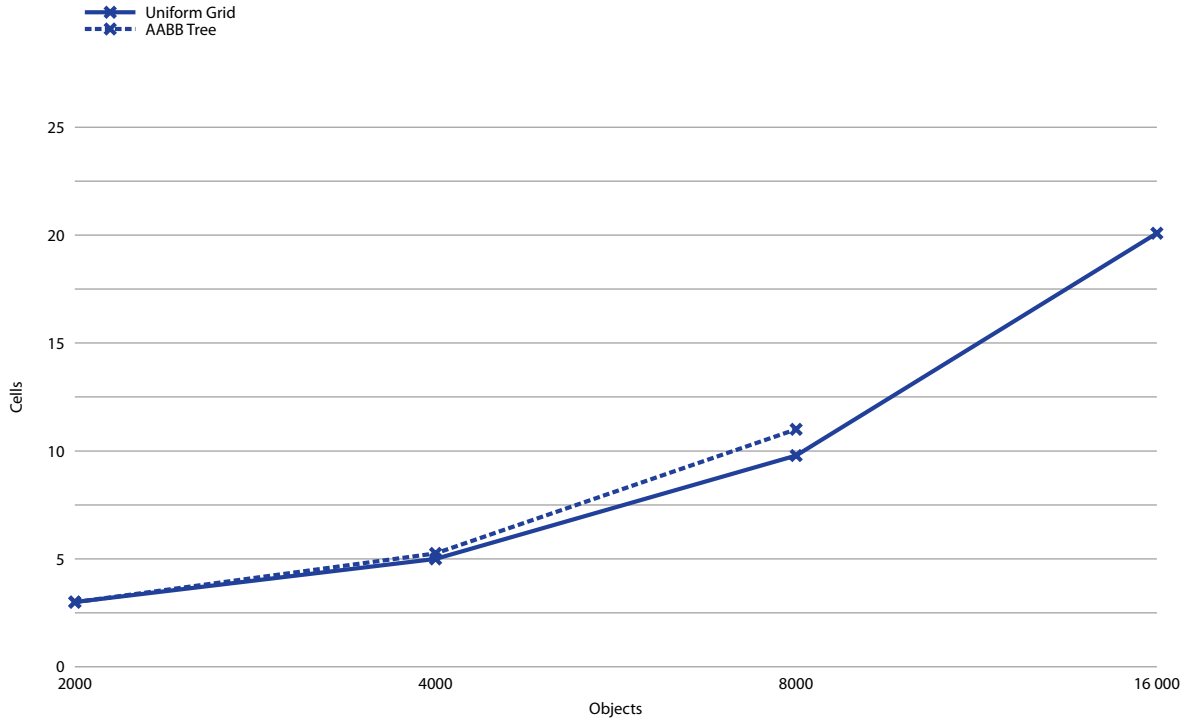


Figure 5.21: *Cells for the 'Cells Experiment' of Section 5.4.2.*

The purpose of this experiment was to determine the number of cells created by the spatial partitioning algorithms. This would allow the efficacy of the spatial partitioning algorithms to be checked, by determining whether superfluous cells were being created.

The number of cells created by the spatial partitioning algorithms was recorded for the four benchmarks from the 'Software-Based Execution Time Experiment'. The mean was computed across the 1000 collision detection cycles.

Every benchmark has a minimum number of cells that must be created. This minimum can be determined by taking the ceiling of the number of objects divided by the cell size. However, the number of cells can significantly exceed the minimum, since objects straddling multiple cells will be placed in

each cell straddled. This effectively increases the number of objects. It is particularly problematic when objects are densely packed and the division of cells results in myriad overlapping objects. These overlapping objects mean that space must be divided finely to sufficiently minimise the number of objects per cell. This ultimately results in large numbers of cells.

Examining the results, it is evident that far larger numbers of cells that the minimum are created by both spatial partitioning algorithms. It is also clear that the number of cells increases according to $O(n^2)$. Both findings can be partially explained by the relatively densely populated benchmarks used. The uniform grid and AABB tree have relatively equal results, although the uniform grid creates slightly fewer cells. Performance could be significantly increased if the number of cells were minimised. Therefore, the number of cells created is one source of the undesirable performance that was measured with the 'Software-Based Execution Time Experiment'.

Objects per Cell Experiment Figure 5.22

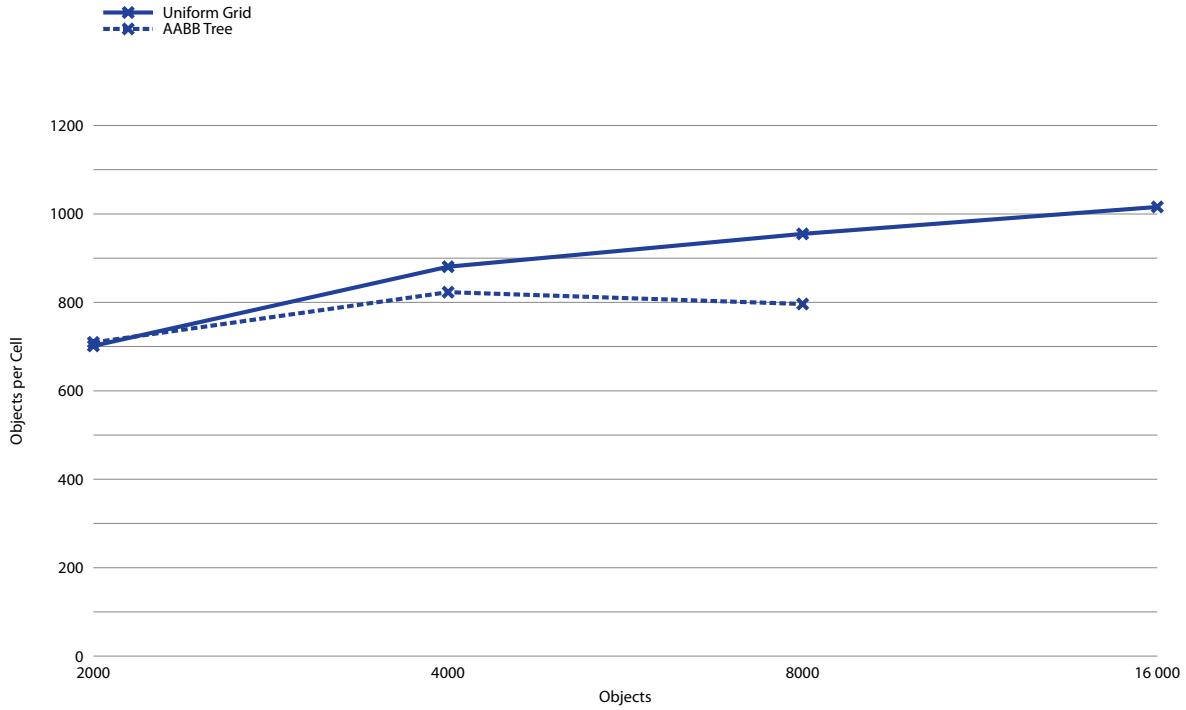


Figure 5.22: *Objects per cell for the 'Objects per Cell Experiment' of Section 5.4.2.*

The purpose of this experiment was to determine the number of objects per cell created by the spatial partitioning algorithms. This experiment, therefore, continued from the 'Cells Experiment' by examining the cells created by the spatial partitioning algorithms.

The number of objects per cell created by the spatial partitioning algorithms was recorded for the four benchmarks from the 'Software-Based Execution Time Experiment'. The mean number of objects per cell was computed for all of the cells in each collision detection cycle. These means were subsequently used to compute a mean across the 1000 collision detection cycles.

The goal of the spatial partitioning algorithm is to maximise the number of objects per cell within cell size limits. This can often be challenging as cell splitting can create cells with far fewer objects than the maximum. Underpopulated cells work inefficiently with the microarchitecture, as their small size makes it difficult to amortise the significant communication costs. Moreover, software-based cell construction and index translation create an additional per-cell overhead. In certain cases where cell sizes are optimal, this optimality may have been achieved due to duplication of objects caused by inefficient splitting. In these cases, the duplicate objects reduce the acceleration achievable. Therefore, it is vital to achieve optimal cell sizes and to avoid duplication of objects where possible.

The results reveal that the cells of each benchmark up to 8000 objects contain significantly fewer objects than the maximum, for both spatial partitioning algorithms. This occurs because the algorithms have to create relatively small cells to keep the number of objects within cell size limits. In addition, the AABB tree typically contains fewer objects per cell than the uniform grid. In this case, this is advantageous because fewer objects are partitioned by the AABB tree as it adapts to the complexity of the environment. Therefore, the cells are smaller because fewer objects are processed. The situation changes for the 16 000-object benchmark with the uniform grid, as the cells contain significantly more objects. This is not desirable in this case as the large cells result from the repeated splitting of objects, leading to the duplication of these objects across multiple cells. This also reflects the difficulties in creating the AABB tree for the 16 000-object benchmark. In all cases, performance could be significantly improved if the number of objects per cell were increased and if the number of divided objects were decreased. Overall, the low number of objects per cells is another source of the undesirable performance that was measured with the 'Software-Based Execution Time Experiment'.

Collisions Experiment Figure 5.23

The purpose of this experiment was to determine the number of potential collisions recorded by the spatial partitioning algorithms, in comparison with the Bullet Physics SDK. This would help in understanding how the replication of objects across multiple cells leads to duplicate pairs and collisions. This experiment, therefore, continued from the 'Cells Experiment' and the 'Objects per Cell Experiment' by examining the cells created by the spatial partitioning algorithms.

The number of potential collisions reported by the broad phase was recorded for the four benchmarks from the 'Software-Based Execution Time Experiment'. Both spatial partitioning algorithms and the Bullet Physics SDK's all-pairs and SAP were used. The two Bullet Physics SDK algorithms were found to return the same number of potential collisions. The mean was computed across the 1000 collision detection cycles.

Results for the hybrid systems with no spatial partitioning have shown that the number of potential collisions negatively influences execution time. Spatial partitioning can replicate objects and pairs of objects across multiple cells. As a result, the same potential collision could be detected in multiple cells, reducing execution speed. The goal is to achieve as few extra potential collisions as possible, ideally detecting the same number as would be detected by a typical software algorithm.

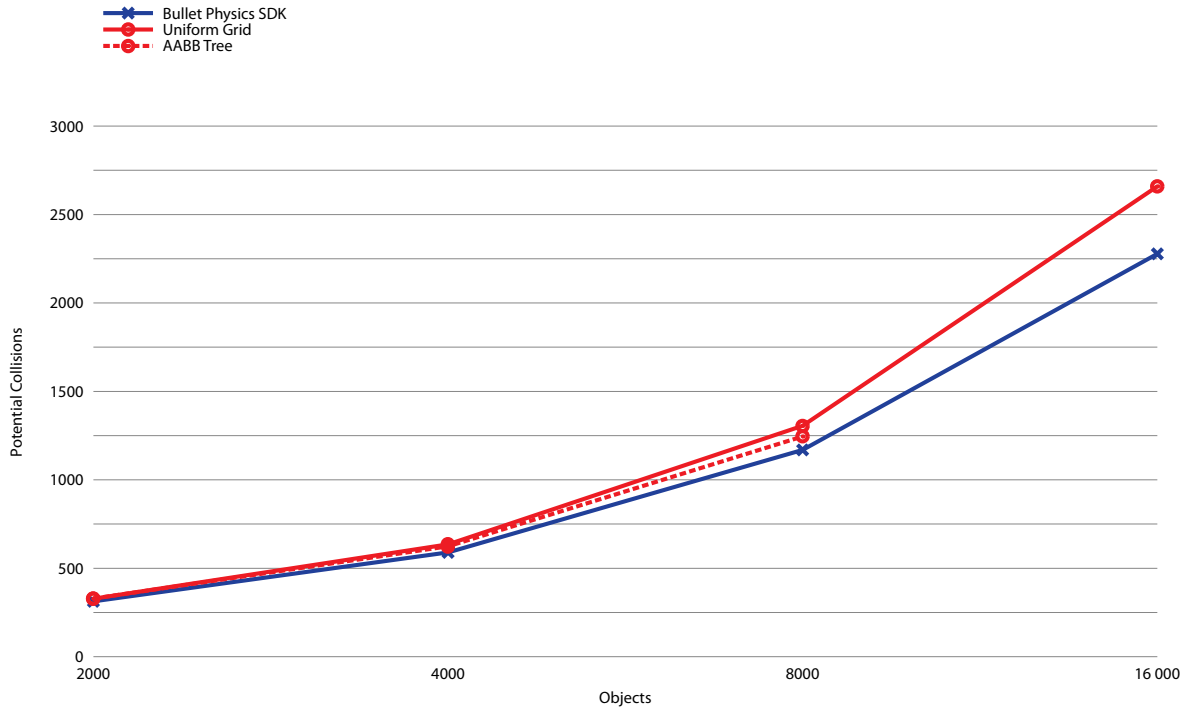


Figure 5.23: *Potential collisions for the 'Collisions Experiment' of Section 5.4.2.*

Analysing the results, one feature that becomes apparent is that the number of potential collisions detected by the hybrid system with either spatial partitioning algorithm noticeably exceeds those detected by the Bullet Physics SDK. Moreover, the rate of increase surpasses that of the Bullet Physics SDK, so that the performance degradation is more acute with larger numbers of objects. The situation is marginally better with the AABB tree than with the uniform grid. This is another source of the undesirable performance that was measured with the 'Software-Based Execution Time Experiment'.

From the 'Cells Experiment', 'Objects per Cell Experiment' and 'Collisions Experiment', it can be seen that the spatial partitioning algorithms are splitting many objects. This leads to excess cells with relatively low object counts, which inefficiently exploit the microarchitecture. In addition, the number of potential collisions detected is increased as potential collisions can occur in multiple cells. This increases execution time, although its effect is less than that of the inefficient cells. The problems highlighted by the three experiments are the major ones affecting the performance of the hybrid systems with spatial partitioning. If these could be rectified, performance would improve.

Performance can be improved by using either a more sophisticated splitting metric or a more sophisticated spatial partitioning data structure. For example, the spatial median splitting metric could be changed to the object median metric as previously outlined. Alternatively, tree structures such as k-d trees, BSP trees or DBVTs could be used instead of the AABB tree. Some or all of these ideas should be sufficient for ameliorating the issues evident in the hybrid system.

Hardware-Based Execution Time Experiment Figure 5.24

The number of clock cycles consumed by the microarchitecture was recorded for the four benchmarks from the 'Software-Based Execution Time Experiment' using clock cycle counters embedded within the microarchitecture. The effect of all software-based computation was disregarded. The benchmarks were executed for 1000 cycles and the mean of the clock cycle counts was subsequently computed. The clock cycle metrics, corresponding to Figure 5.24, were:

Excluding Communication Interface with Uniform Grid The clock cycles consumed by the microarchitecture excluding all communication interface logic, when using the uniform grid.

Including Communication Interface with Uniform Grid The clock cycles consumed by the microarchitecture including all communication interface logic, when using the uniform grid.

Excluding Communication Interface with AABB Tree The clock cycles consumed by the microarchitecture excluding all communication interface logic, when using the AABB tree.

Including Communication Interface with AABB Tree The clock cycles consumed by the microarchitecture including all communication interface logic, when using the AABB tree.

The clock cycle counts were subsequently converted into execution times using the microarchitecture's clock frequency. The construction costs of the uniform grid and AABB tree were added to these as appropriate. Based on these execution times, a variety of acceleration metrics was computed. These metrics compared the metrics from this experiment with those from the Bullet Physics SDK algorithms from the 'Software-Based Execution Time Experiment'.

The overall pattern of execution times and accelerations remains the same as for the 'Software-Based Execution Time Experiment', although the accelerations are greatly improved. For the uniform grid, the acceleration extends to 113.119× compared to all-pairs, while it extends to 2.39× compared to SAP. Compared to the many-object microarchitecture without spatial partitioning, the 8000-object benchmark gains a greater acceleration relative to all-pairs while the 4000- and 8000-object benchmarks gain greater accelerations relative to SAP. For the AABB tree, the acceleration extends to 366.091× compared to all-pairs, while it extends to 10.443× compared to SAP. All three benchmarks gain accelerations greater than the many-object microarchitecture without spatial partitioning, when compared against either all-pairs or SAP. Using either spatial partitioning algorithm, all benchmarks achieve an appreciable acceleration when compared to SAP, except for the 16 000-object benchmark.

CPU-Based Execution Time Experiment Figure 5.25

To compute these results, the clock cycles counts from the 'Hardware-Based Execution Time Experiment' were used. In addition, the construction costs of the spatial partitioning algorithms were added to the converted clock cycles, as this construction must always be performed in software. The four metrics, corresponding to Figure 5.25, were:

Excluding L2 Cache with Uniform Grid The 'Excluding Communication Interface with Uniform Grid' clock cycle counts from the 'Hardware-Based Execution Time Experiment' converted to

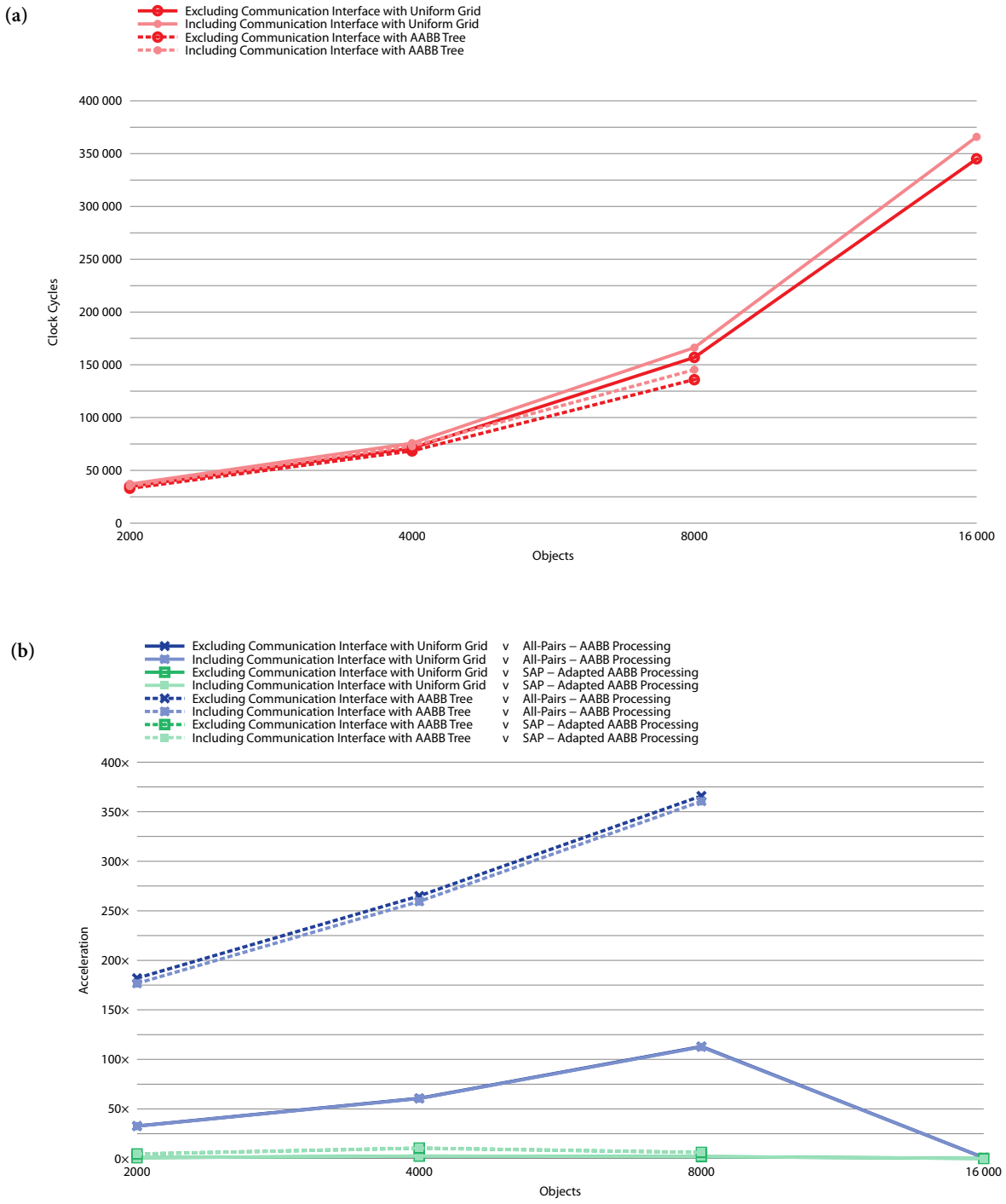


Figure 5.24: *Results for the 'Hardware-Based Execution Time Experiment' of Section 5.4.2.* **(a)** *Clock cycles.* **(b)** *Accelerations. The data are tabulated in Table D.11.*

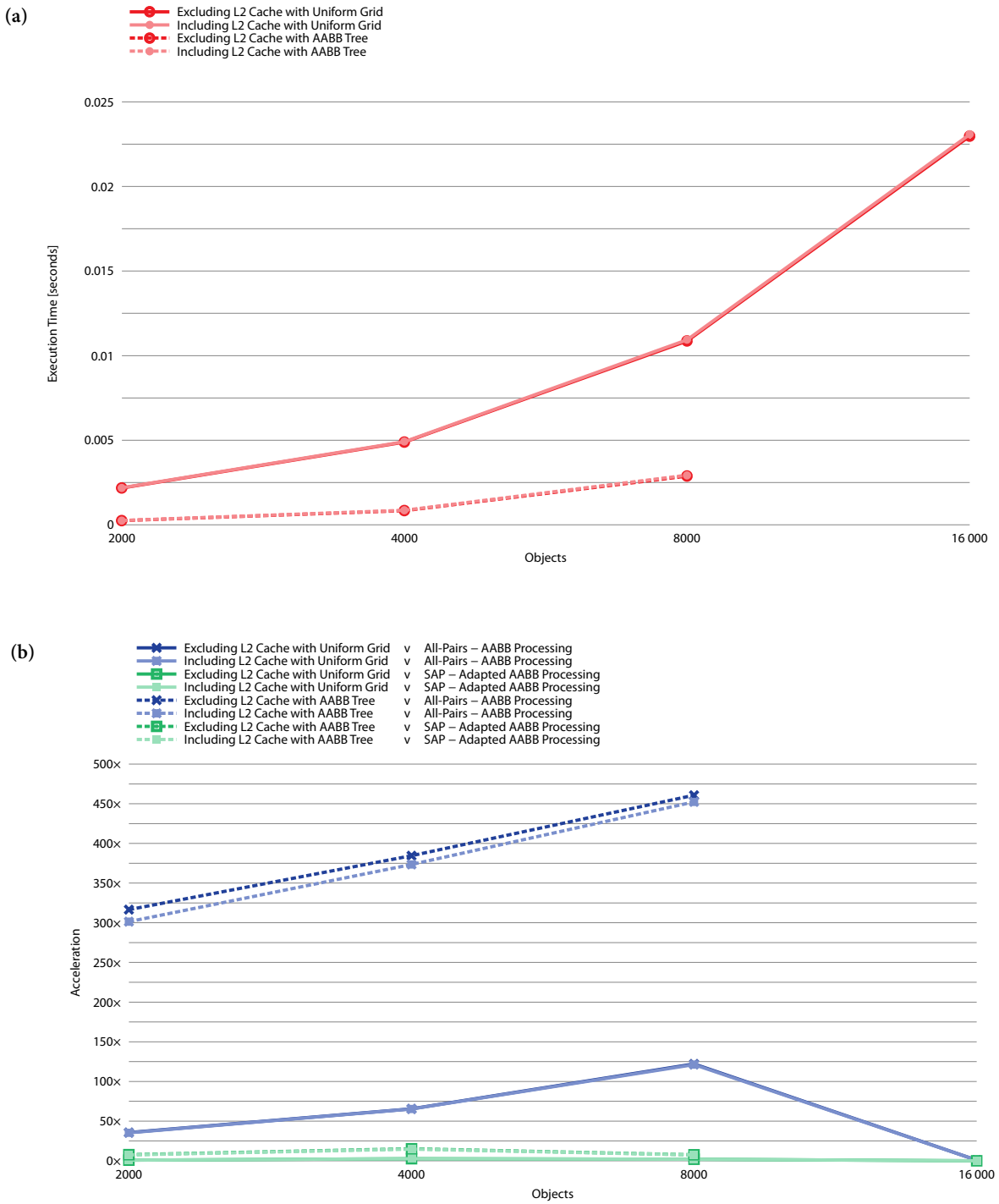


Figure 5.25: *Results for the 'CPU-Based Execution Time Experiment' of Section 5.4.2.* **(a)** *Execution times.* **(b)** *Accelerations. The data are tabulated in Table D.12.*

execution times using the 2 GHz clock frequency of the host system's CPU. The uniform grid construction costs were included in this metric.

Including L2 Cache with Uniform Grid The 'Excluding L2 Cache with Uniform Grid' metric with the estimated effect of the L2 cache taken into account. The effect of the cache was computed as described for the many-object microarchitecture without spatial partitioning. The uniform grid construction costs were included in this metric.

Excluding L2 Cache with AABB Tree The 'Excluding Communication Interface with AABB Tree' clock cycle counts from the 'Hardware-Based Execution Time Experiment' converted to execution times using the 2 GHz clock frequency of the host system's CPU. The AABB tree construction costs were included in this metric.

Including L2 Cache with AABB Tree The 'Excluding L2 Cache with AABB Tree' metric with the estimated effect of the L2 cache taken into account. The effect of the cache was computed as described for the many-object microarchitecture without spatial partitioning. The AABB tree construction costs were included in this metric.

Based on these execution times, a variety of acceleration metrics was computed. These metrics compared the metrics from this experiment with those from the Bullet Physics SDK algorithms from the 'Software-Based Execution Time Experiment'.

The overall pattern of execution times and accelerations remains the same as for the previous experiments. Considering the uniform grid, the execution time is 0.02 seconds for the 16 000-object benchmark when the effect of the L2 cache is disregarded. The acceleration achieved is up to 122.087× when compared to all-pairs and up to 2.58× when compared to SAP. Considering the L2 cache increases the execution time for the 16 000-object benchmark to 0.023 seconds and reduces the acceleration to 121.289× for all-pairs and to 2.565× for SAP. However, these accelerations are less than those achieved by the many-object microarchitecture without spatial partitioning in the 'CPU-Based Execution Time Experiment'. This results from adding the spatial partitioning data structure construction costs to the adjusted clock cycle times. As the construction costs do not lessen when the theoretical CPU is considered, because the construction is always performed in software, these costs form a constant overhead that cannot be amortised. This reduces the acceleration factor. The only solution to this problem is to optimise the spatial partitioning data structures using some of the ideas outlined previously.

Switching attention to the AABB tree, the execution time for 8000 objects is 0.0028 seconds when the effect of the L2 cache is disregarded. The acceleration is up to 461.439× when compared to all-pairs and up to 15.2× when compared to SAP. Including the cache increases the execution time for 8000 objects to 0.0029 seconds and decreases the acceleration to 452.248× when compared to all-pairs and to 14.675× when compared to SAP. Overall, it is clear than the acceleration factors are significantly higher with the AABB tree than with the uniform grid. However, as for the uniform grid, all of the accelerations are less that those achieved by the many-object microarchitecture without spatial partitioning in the 'CPU-Based Execution Time Experiment'.

Evaluation All of the experiments have demonstrated that significant accelerations are achievable using the hybrid system with spatial partitioning. Comparing the performance of the uniform grid implementation to all-pairs results in accelerations of up to 43.592× when measured from a software perspective, of up to 113.119× when measured on the FPGA, of up to 121.289× when measured on a theoretical CPU and of up to 122.087× when measured on a theoretical CPU with the effect of the L2 cache disregarded. In contrast, compared to SAP, the accelerations extended to 1.406× when measured from a software perspective, to 2.39× when measured on the FPGA, to 2.565× when measured on a theoretical CPU and to 2.58× when measured on a theoretical CPU with the effect of the L2 cache disregarded.

Switching to the AABB tree implementation, the accelerations relative to all-pairs extended to 47.081× when measured from a software perspective, to 366.091× when measured on the FPGA, to 452.248× when measured on a theoretical CPU and to 461.439× when measured on a theoretical CPU with the effect of the L2 cache disregarded. In contrast, compared to SAP, the accelerations extended to 1.442× when measured from a software perspective, to 10.443× when measured on the FPGA, to 14.675× when measured on a theoretical CPU and to 15.2× when measured on a theoretical CPU with the effect of the L2 cache disregarded.

Many of the accelerations achieved with the current hybrid system are higher than those achieved using the many-object microarchitecture without spatial partitioning. Consequently, spatial partitioning has been demonstrated as a means to overcome both the O (n^2) computational complexity of the microarchitecture as well as the microarchitecture's object limitations. In general, the AABB tree performs much more efficiently than the uniform grid, as it adapts to the locations of the objects instead of rigidly dividing the environment. However, the AABB tree is unable to divide the environment finely enough to support the 16 000-object benchmark. Therefore, it is recommended to use the AABB tree if possible and to use the uniform grid if problems are encountered.

Despite the gains achieved using spatial partitioning, accelerations tend to drop off for larger numbers of objects. Experimentation has shown that this drop-off results from inefficient splitting of cells using the spatial partitioning algorithms. This inefficiency leads to myriad cells of non-optimal size in addition to duplicate potential collisions. If the splitting could be improved, performance would similarly improve and the hybrid system should perform efficiently for very large quantities of objects.

As the drop-off results from inefficient splitting of cells, the obvious solution is to find a better cell splitting metric. For example, the object median could be used instead of the spatial median. This attempts to balance child cells, which should greatly improve the results. However, it requires that the tree be refitted, instead of reconstructed, between collision detection cycles, if efficiency is to be maintained. An alternative solution to changing the splitting metric would be to use an alternative spatial partitioning data structure. For example, k-d trees, BSP trees or DBVTs may provide better results than AABB trees. In particular, BSP trees appear promising due to their ability to split along an axis that is not aligned to the world axes. This ability should help to prevent splitting of objects. BSP trees have typically proven to be inefficient, raising questions over their applicability. However, there is significant scope to believe that these inefficiencies would not manifest themselves in the context of the

hybrid systems, due to the differing nature of these systems relative to existing systems. Therefore, it is reasonable to assume that a significant acceleration can be gained using a BSP tree or similar. Moreover, it is evident that either a new splitting metric or data structure would allow spatial partitioning to provide an acceleration for all of the benchmarks.

5.4.3 Many-Object Microarchitecture

As the implementation of the many-object microarchitecture supported up to 16 384 objects, there was less need to use spatial partitioning to overcome the object limitations. However, previous results showed that the microarchitecture decelerated the benchmarks with 4000 or more objects. Therefore, it was hypothesised that the spatial partitioning would be of utility for reducing the computational complexity of the many-object microarchitecture. To test this hypothesis, a variety of cell sizes less than 16 384 was trialled. These cell sizes were 1024, 2048, 4096 and 8192. The accelerations from the 'Software-Based Execution Time Experiment' and the 'CPU-Based Execution Time Experiment' for each of the cell sizes are plotted in Figures 5.26, 5.27 and 5.28, and are tabulated in Table 5.4. All of the other experiments performed using the hybrid system comprising the area-efficient microarchitecture and spatial partitioning were repeated using the hybrid systems comprising the many-object microarchitecture and spatial partitioning. However, the results from these experiments were extremely similar to those attained using the previous system. Therefore, the results have been excluded from this chapter; they can be located in Appendix D.

The accelerations from the 'Software-Based Execution Time Experiment' and the 'CPU-Based Execution Time Experiment' follow an extremely similar pattern to those of the hybrid system comprising the area-efficient microarchitecture with spatial partitioning. When compared to either all-pairs or SAP, many instances of the 4000- and 8000-object benchmarks achieve accelerations exceeding those attained by the many-object microarchitecture without spatial partitioning. However, all instances of the 16 000-object benchmark are decelerated. This remains true even for those results adapted to reflect their execution time if implemented on the CPU. It is also evident that the acceleration factor decreases as cell size increases. This implies that it would be best to select a cell size of 1024 or, possibly, less. Finally, the AABB tree outperforms the uniform grid for all of the benchmarks.

As for the hybrid system comprising the area-efficient microarchitecture with spatial partitioning, it is evident that the spatial partitioning provides a means of overcoming the $O(n^2)$ computational complexity of the microarchitecture. However, some benchmarks are decelerated by the hybrid systems due to the previously outlined shortcomings in the cell splitting strategies of the spatial partitioning algorithms. If these shortcomings were to be rectified, using the previously outlined suggestions of more sophisticated splitting metrics or data structures, a high degree of acceleration could be achieved for all of the benchmarks.

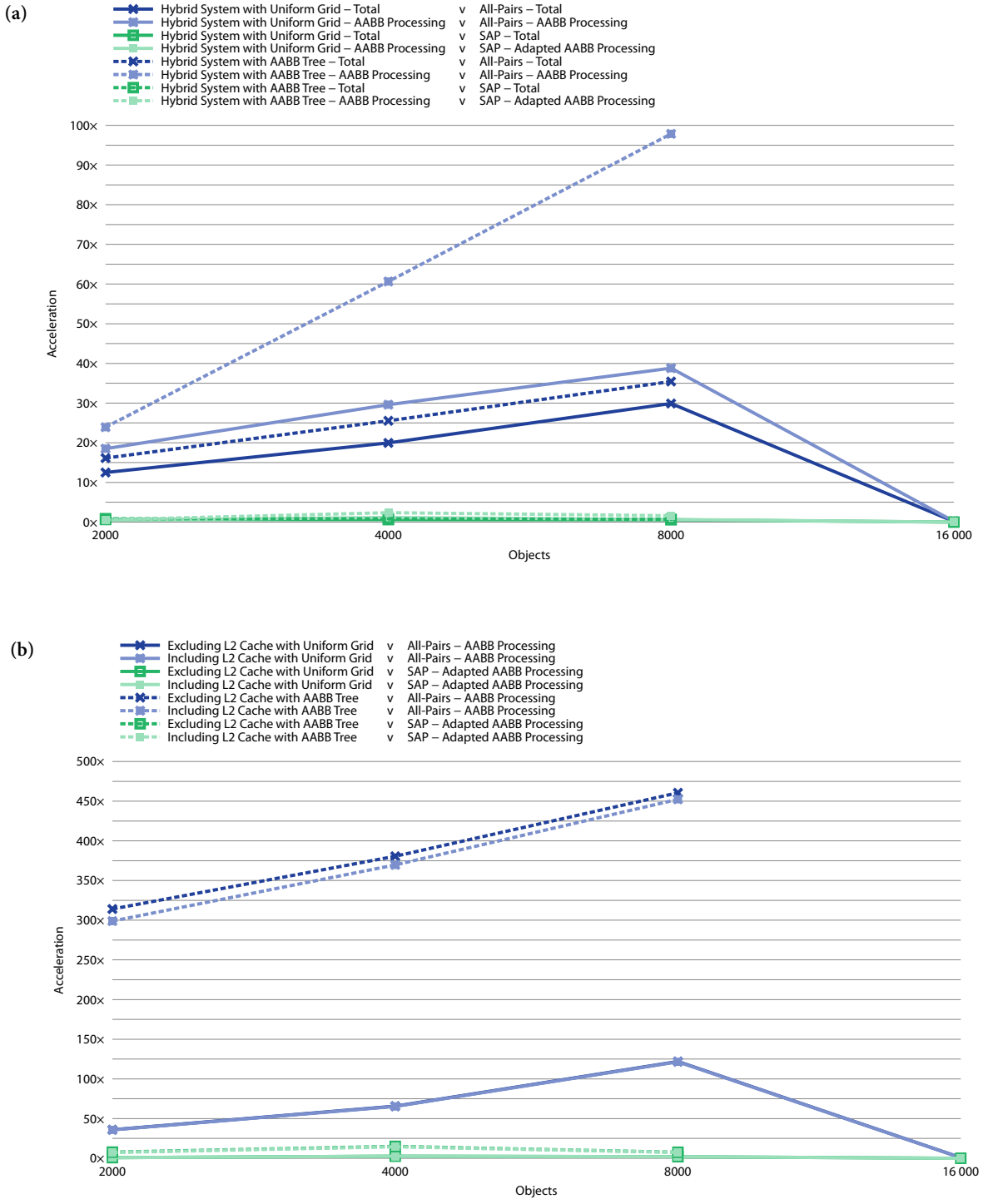


Figure 5.26: *Results for the experiments of Section 5.4.3 with a cell size of 1024.* **(a)** *Accelerations for the 'Software-Based Execution Time Experiment'. The data are tabulated in Table D.13.* **(b)** *Accelerations for the 'CPU-Based Execution Time Experiment'. The data are tabulated in Table D.15.*

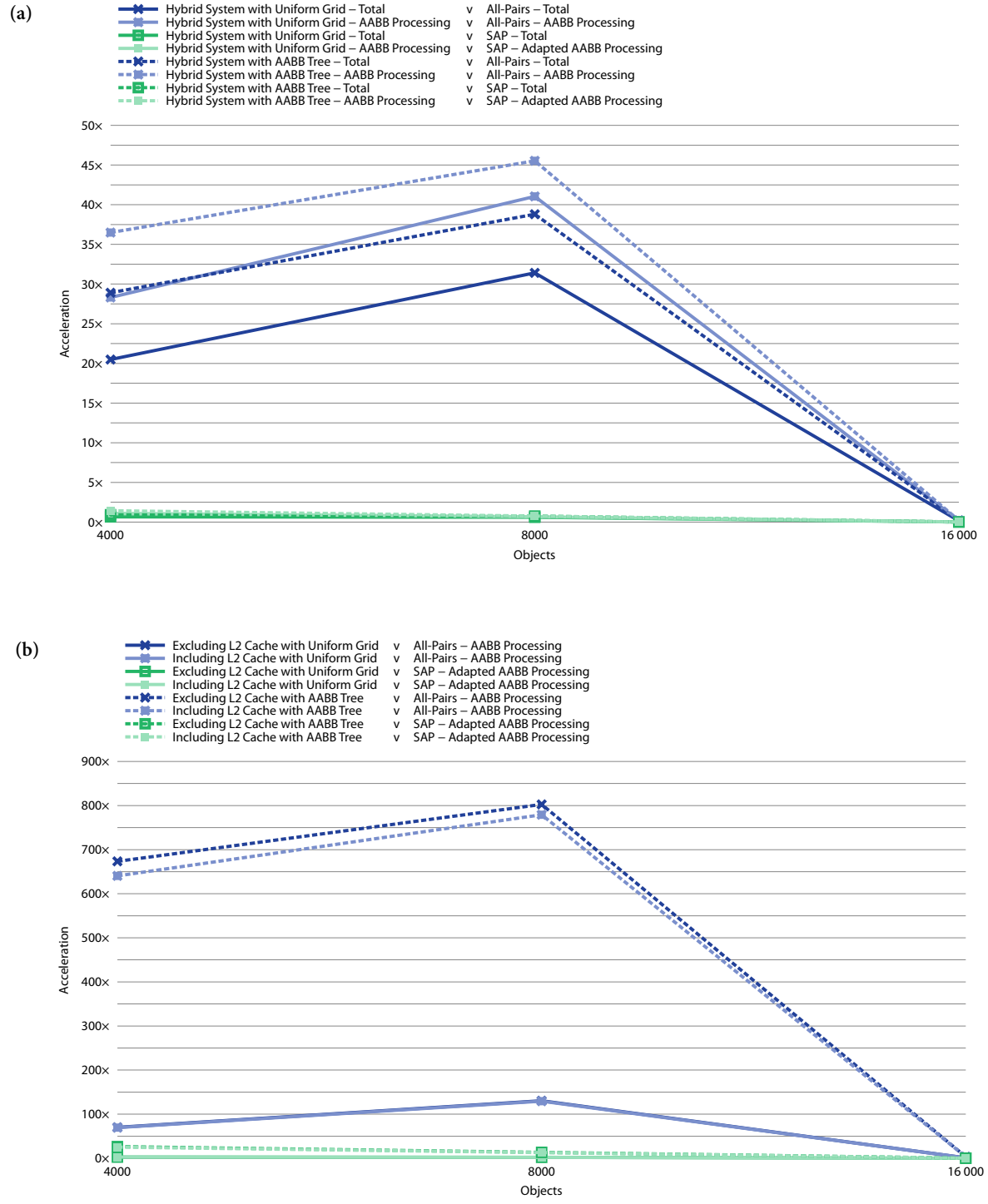


Figure 5.27: *Results for the experiments of Section 5.4.3 with a cell size of 2048.* **(a)** *Accelerations for the 'Software-Based Execution Time Experiment'. The data are tabulated in Table D.16.* **(b)** *Accelerations for the 'CPU-Based Execution Time Experiment'. The data are tabulated in Table D.18.*

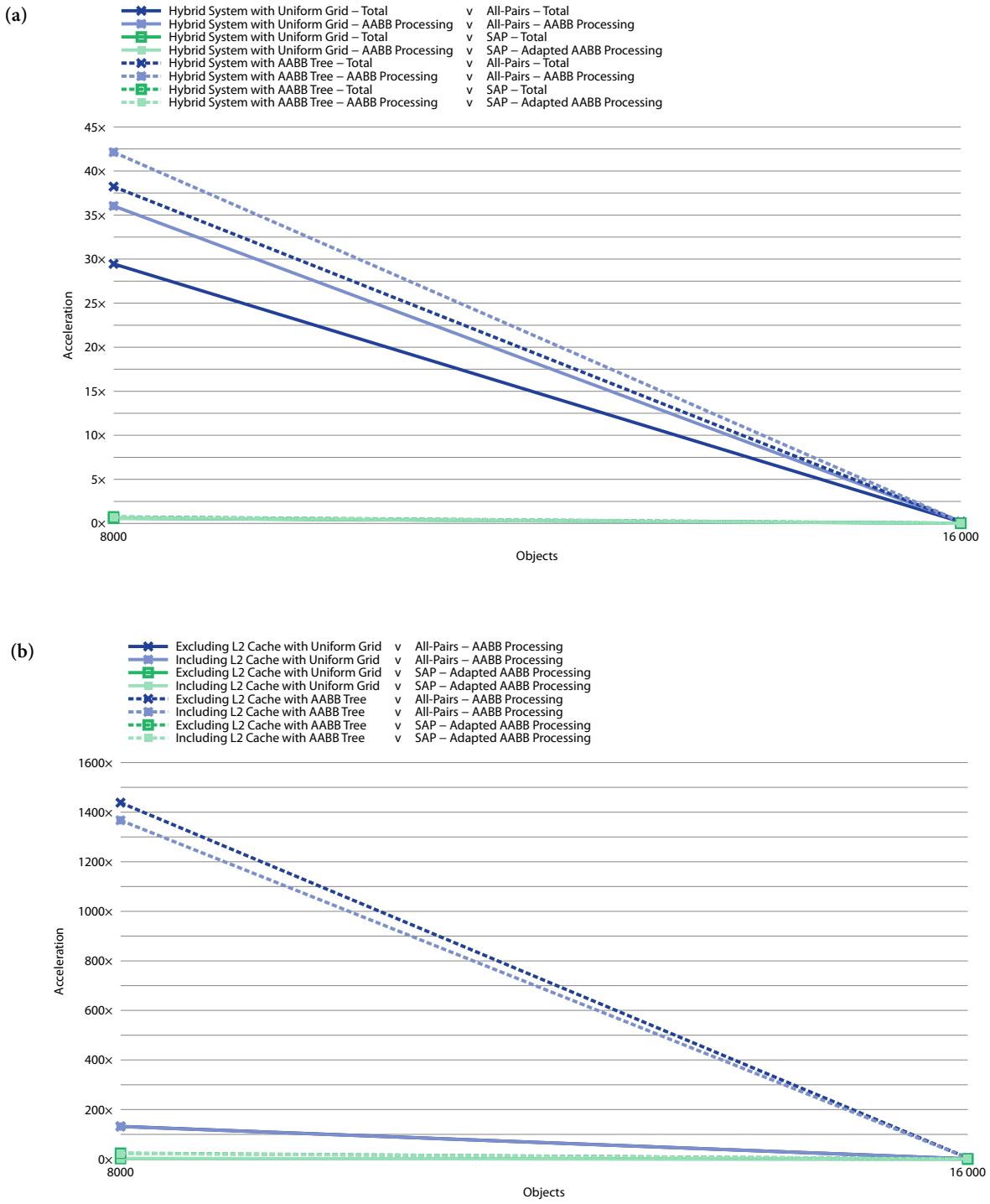


Figure 5.28: *Results for the experiments of Section 5.4.3 with a cell size of 4096.* **(a)** *Accelerations for the 'Software-Based Execution Time Experiment'. The data are tabulated in Table D.19.* **(b)** *Accelerations for the 'CPU-Based Execution Time Experiment'. The data are tabulated in Table D.21.*

(a)

	All-Pairs		SAP	
Objects	Total	AABB Processing	Total	Adapted AABB Processing
		Uniform Grid		
16 000	0.158×	0.182×	0.014×	0.013×
		AABB Tree		
16 000	0.174×	0.188×	0.016×	0.014×

(b)

	Excluding L2 Cache		Including L2 Cache	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
		Uniform Grid		
16 000	0.955×	0.071×	0.95×	0.07×
		AABB Tree		
16 000	6.896×	0.509×	6.645×	0.491×

Table 5.4: *Results for the experiments of Section 5.4.3 with a cell size of 8192.* **(a)** *Accelerations for the 'Software-Based Execution Time Experiment'.* **(b)** *Accelerations for the 'CPU-Based Execution Time Experiment'.*

5.5 Evaluation

After thoroughly measuring and analysing the performance of the different hybrid system implementations, it is beneficial to determine the relative performance of the different systems. Using this information, the systems' relative advantages and disadvantages can be deduced. This information is vital for selecting the most appropriate system for a given scenario.

The preceding experiments have shown that the Xilinx Virtex-5 implementations significantly outperform their Xilinx Virtex-II equivalent. This is because the Xilinx Virtex-5 implementations execute at a higher clock frequency, on a more sophisticated FPGA with a more sophisticated interconnect between the FPGA and CPU. Therefore, the Xilinx Virtex-5 is recommended for all scenarios.

Turning attention to the area-efficient and many-object Xilinx Virtex-5 implementations, it is evident that the area-efficient microarchitecture has faster execution times as it performs more comparisons in parallel. Consequently, it is recommended to use the area-efficient microarchitecture whenever possible. However, the area-efficient microarchitecture is limited to 1024 objects. As a result, the many-object microarchitecture would be necessary for scenarios involving larger quantities of objects. However, this microarchitecture decelerates scenarios with 4000 or more objects. Therefore, the many-object microarchitecture could be used for scenarios involving more than 1024 but less than 4000 objects.

The experiments also demonstrated how spatial partitioning could be used to overcome the microarchitectures' object limitations and undesirable computational complexities. Results indicating accelerations greater than those achievable without spatial partitioning demonstrate the validity of this approach and highlight its fundamental utility. However, not all results are satisfactory and it is clear that this initial attempt at spatial partitioning could be significantly improved. Examination of the data shows many sources contributing to the unsatisfactory performance, but it is clear that inefficient partitioning of the environment into cells is the primary cause. If this inefficient partitioning were remedied, then it would be recommended to use the Xilinx Virtex-5 implementation of the area-efficient microarchitecture with spatial partitioning for cell sizes of 4000 or more objects. In this case, a tree structure would be recommended over a grid as, despite its disadvantages, the AABB tree consistently outperformed the uniform grid.

5.6 Summary

This chapter began by considering the behaviour of the benchmarking framework using five experiments. The first experiment considered the variation of the execution time across each collision detection cycle of a benchmark. The second considered the effect of each of the benchmark's parameters and found that different parameters had a significant influence on execution time. The third was an adaptation of the second that computed the ratio of the narrow phase to the broad phase. The fourth considered the effect of creating benchmarks that maximised the number of collisions and discovered that increased collisions have a greater impact on the narrow phase than on the broad phase. The fi-

nal experiment computed the ratio of the narrow phase to the broad phase for the benchmarks of the fourth experiment. Each experiment demonstrated that SAP was the most efficient of the three broad phase Bullet Physics SDK algorithms evaluated.

Subsequently, the two microarchitectures were measured and assessed using a different set of experiments. These experiments quantified the execution time from a software perspective, that is including the communication overhead, and from a microarchitecture perspective, based on clock cycles consumed. Measurements were also provided to show the effect of increased potential collisions on the performance of the hybrid systems, as well as the performance of a theoretical implementation of the microarchitecture on the fixed-function logic of a CPU. Using these experiments, it was determined that the Xilinx Virtex-5 outperformed the Xilinx Virtex-II and that the area-efficient microarchitecture outperformed the many-object microarchitecture. These analyses also determined that an FPGA implementation could achieve an acceleration of 3.54× over a state-of-the-art software algorithm, measured from a software perspective, or of 80.893× measured from a microarchitecture perspective. Retargeting the microarchitecture for placement on a CPU would result in accelerations of up to 1565.546×.

Having investigated the microarchitectures, spatial partitioning was considered. Numerous experiments were performed to quantify and evaluate the effect of spatial partitioning. These included all of the aforementioned experiments coupled with a variety of new experiments designed to clarify the behaviour of the spatial partitioning algorithms. These analyses showed an acceleration of up to 2.387× when measured from a software perspective and of up to 10.443× when measured from a microarchitecture perspective. The implementation targeted for CPU placement provided an acceleration of up to 26.6×. These accelerations proved the validity of the spatial partitioning approach and demonstrated that significant accelerations could result from its use. However, it was also discovered that the performance was affected by deficiencies in the current spatial partitioning implementations, which created splitting planes that divided many objects and placed them in multiple cells. It was clear that these performance issues could be rectified using more sophisticated spatial partitioning strategies. Therefore, spatial partitioning showed significant promise and it was evident that much greater accelerations could be attained using the spatial partitioning concept.

Chapter 6

Conclusion and Future Work

The vast possibilities of our great future will become realities only if we make ourselves responsible for that future.

—Gifford Pinchot, US politician (1865–1946)

In this thesis, the concept of a hybrid software and hardware system for efficiently performing collision detection has been presented. The system consists of a custom microarchitecture for broad phase collision detection coupled with a software-based narrow phase. The microarchitecture is based on the all-pairs algorithm, which was selected for its parallelisation potential. Parallelism is exploited using a plurality of memories accessed in a specific sequence that allows all objects in the system to be compared. Two variants of the microarchitecture were described: the first occupies a minimum of hardware resources and is suitable for highly constrained hardware platforms, while the second supports more objects at the cost of increased resources and decreased parallelism.

The two variants of the microarchitecture were implemented on FPGAs. The performance of the resultant hybrid systems was measured and the results were analysed. These measurements demonstrated an acceleration of up to 3.54× over a state-of-the-art software implementation, when the significant communication overhead in such a system was taken into account. Removal of the communication overhead increased the acceleration to 80.893×. Since the FPGA is essentially a prototyping platform, the performance of a theoretical system was also analysed. In this theoretical system, the microarchitecture was implemented as part of the fixed-function logic of the CPU. This implementation demonstrated an acceleration of up to 1565.546× over the software implementation due to the higher clock frequency of the CPU and the elimination of the communication overhead.

To overcome the undesirable computational complexity of all-pairs as well as the object limitations imposed by restrictions on the size of internal memories, spatial partitioning was proposed. This spatial partitioning executes in software prior to the execution of the microarchitecture and splits the environment into cells of a specified size. Two algorithms were investigated in this context. The first

was the standard uniform grid and the second was an adaptation of the AABB tree that could effectively partition the environment.

The resultant systems were investigated. After the major accelerations demonstrated by the previous systems, the results from the hybrid systems comprising spatial partitioning were somewhat unsatisfactory. Accelerations of up to 2.387× were achieved with the FPGA implementation including the communication overhead, while accelerations of up to 10.443× were achieved when the communication overhead was removed. Retargeting the microarchitecture for implementation on a CPU resulted in an acceleration of 26.6×. However, many results showed a deceleration for both platforms. An analysis of the results highlighted that this deceleration resulted from inefficient spatial partitioning, which divided many objects and resulted in large numbers of cells. Therefore, it became clear that spatial partitioning showed considerable promise but more sophisticated algorithms would be necessary if a significant acceleration were to be achieved.

Finally, a framework for benchmarking collision detection was outlined. This framework consists of a single generic benchmark that can be adapted using a set of parameters to create a wide range of practical benchmarks. This use of parameters overcomes many of the problems that plague previous attempts at collision detection benchmarks, which typically focus on a single fixed scenario. In addition, the framework uses a physics engine to accurately recreate scenarios from interactive applications. The framework is beneficial to collision detection algorithm developers, who can quickly sweep through a wide test space to determine which scenarios are unfavourable to their algorithm, with a view to rectifying these algorithm deficiencies. It is also beneficial to application developers, who can recreate their application scenarios and quickly sweep through a variety of algorithms to determine those that provide the best performance for their applications. The framework was analysed using the hybrid systems and the three broad phase algorithms from the Bullet Physics SDK.

6.1 Future Work

Prior to this research, little research had been performed in the area of hybrid software and hardware systems for accelerating collision detection. Therefore, there is significant scope for further improvements and alternative considerations of the research presented in this thesis.

Firstly, a number of minimal optimisations could improve the overall performance of the microarchitecture. For example, it was mentioned in Section 4.2 that the current design maximises dataflow through the microarchitecture at the expense of slightly increased data transfer costs. Since the data transfer costs were found to be high, it would be interesting to examine the effect of reducing these costs, although any advantages gained are likely to be minimal. Another suggested improvement would be to use a duplicate buffer in the microarchitecture and to switch between the buffers during loading and processing, as suggested in Section 3.4. Section 4.3 noted that there were sufficient BRAMs to do this for the many-object microarchitecture implemented on the Xilinx Virtex-5 FPGA, although this was not carried out due to its potential to negatively influence the achievable clock frequency. Despite this, it would be interesting to quantify the effect of this optimisation, particularly since it may

improve the performance of spatial partitioning. This would be particularly interesting if the microarchitecture were implemented on a platform other than an FPGA. A third improvement would be to use the global indices instead of the per-cell indices within the microarchitecture, by transferring these global indices with the AABBs. They would be stored in internal memories with the AABBs and synchronously transferred throughout the microarchitecture pipelines with the AABBs. The algorithm for emitting the AABBs from the memories would remain the same, using the previously outlined per-cell indices; the global indices would only be used in the final stage to convert the internal microarchitecture indices into software indices for outputting as part of the PCSes. This adaptation would preclude the requirement to store the index translation tables and perform PCS index conversion in software, thereby reducing memory utilisation and processing time. However, it would reduce the value of m that could be achieved by the microarchitecture by up to 50%, as more memories would be required for storing the indices. Therefore, such a modification would only work effectively in a system with large quantities of memory resources. Overall, such optimisations would have minimal impact but it would be important to analyse and quantify each to gain the maximum acceleration from the system.

As stated in Section 5.1, the Bullet Physics SDK's all-pairs uses a short-circuited evaluation of the object-object test, as do the majority of software all-pairs algorithms. Short-circuited evaluation eliminates the need to perform redundant collision testing, so it would be beneficial to use an equivalent concept with the microarchitecture. The adapted microarchitecture would use either the area-efficient or the many-object microarchitecture to process the first axis. Afterwards, the second axis of each AABB corresponding to the objects in the PCS would be read from external memory and processed. This sequence would continue for the third axis. While processing the first axis should be relatively straightforward using the previously outlined microarchitectures, it is less clear how the second and third axes should be processed, since the microarchitectures' access patterns would not work with a list of nonconsecutive objects. Clearly, the many-object scheme is entirely unsuitable as the non-replication of data significantly curtails the way data can be accessed. The area-efficient scheme is much more flexible but the control algorithm would require significant adaptation. This adaptation is likely to result in a more complex algorithm, which would need to be switched with the original algorithm for processing the first axis. This switching would require significant logic resources to implement, which is likely to have a detrimental impact on clock frequency. For an FPGA, this would be of major consequence, but if a faster target platform were used, the impact is likely to be negligible. The advantage of this adaptation is that the value of m could be increased by three, since only a single axis would be processed at a time. However, it would result in additional accesses to potentially inefficient external memories. Therefore, it is unclear what practical gains could be achieved. During these adaptations, it would also be beneficial to explore the order in which the axes are processed, since it may often be possible to eliminate large numbers of collisions along certain axes. It may even be possible to eliminate the vast majority of objects by processing just one or two axes. In this case, the PCS resulting from processing these axes could be returned to the software for narrow phase processing, since there would be little benefit in using the microarchitecture to process all of the axes.

In Section 5.4.2, it was stated that the spatial partitioning showed promise, although the results

were not entirely satisfactory. Consequently, more research must be performed into adaptations to the spatial partitioning algorithms. The results demonstrated that the current implementation partitioned many objects and added these to multiple cells, which adversely affected performance. Therefore, the focus of this research must be on alterative partitioning strategies. The AABB tree nodes could, for instance, be partitioned along the object median instead of the spatial median. In addition, the tree could be refitted instead of reconstructed on each collision detection cycle. Alternatively, more sophisticated hierarchical structures such as the k-d tree or the BSP tree could be utilised. The BSP tree is particularly interesting as these trees have greater flexibility in their choice of partitioning axis, which provides the possibility of reducing the number of straddling objects. Of significant note is that the partitioning axis does not require alignment with a world axis. BSP trees are rarely used for moving objects in interactive environments due to the complexity of their construction. However, unlike most applications, it should be relatively simple to construct the BSP trees required by the microarchitectures, as partitioning would not extend to the level of individual objects. Another interesting hierarchy would be an appropriate adaptation of the Bullet Physics SDK's DBVT, which achieved good results when analysed with the benchmarking framework. Overall, it is obvious that a significant acceleration would be achieved if the undesirable aspects of the current implementation were rectified.

With the aforementioned developments, the hybrid system should offer very significant accelerations in excess of those achieved by the current implementation. However, the focus of these accelerations is the broad phase, as this is the novel part of the hybrid system. Therefore, it would be useful to consider the narrow phase by implementing it on the microarchitecture. It is likely that a narrow phase microarchitecture would be of lesser benefit than the broad phase microarchitecture, as Section 3.1 found that narrow phase algorithms tend to be less amenable to microarchitecture implementation than their broad phase counterparts. However, there is still significant scope for acceleration. When considering the microarchitecture, there is a very wide variety of narrow phase algorithms and it is unclear which would be most appropriate for this design. Alternatively, the microarchitecture designed by Raabe and Zavelberg [2008] could be used, although their differing design decisions mean the microarchitecture would necessitate adaptation. It would also be interesting to investigate the construction of narrow phase hierarchies in the microarchitecture, as it appears that this has not been done previously, although the sequential, memory-bound nature of hierarchy construction means it is likely to be challenging to achieve an acceleration.

The hybrid system currently exploits the properties of two platforms – software and a custom microarchitecture. Another platform, the GPU, is gaining in popularity as a platform for the implementation of myriad parallelisable algorithms. It would be interesting to investigate how a GPU could be used within the hybrid system to provide further accelerations. Perhaps the narrow phase collision detection could be implemented on a GPU instead of a microarchitecture. However, it is probable that the microarchitecture would achieve better performance as the microarchitecture designer would have the ability to determine the layout of the hardware components and could use this ability to create a configuration that provided greater acceleration. Alternatively, other stages of the computer game program loop could be analysed for GPU implementation.

After finalising and optimising every aspect of the microarchitecture, it would be appropriate to implement it on a higher-performance platform than an FPGA. This would allow for a thorough practical performance evaluation, instead of the theoretical CPU evaluation presented in this thesis. One appropriate platform would be an ASIC. If this were used, it would be best to complement the ASIC with a custom PCB whose hosted memories are connected to the ASIC in a way that maximises execution speed. Alternatively, it would be interesting to request a CPU or GPU manufacturer to include the microarchitecture as part of their fixed-function logic, although such requests are likely to be unsuccessful due to the substantial manufacturing and testing costs associated with the required small fabrication run. For all of these platforms, the TATs are extremely high, so they can only be seriously investigated once the microarchitecture is finalised and it is clear that everything works correctly and as efficiently as possible. Such implementations would be interesting to investigate as even more significant accelerations could be recorded and the design would become a step closer to a practical realisation that could be used within commodity computers.

Another topic that could be investigated, but which is independent of the collision detection microarchitectures, is the implementation of the generalised microarchitectures proposed in Appendix A. These generalisations are generic designs that can be applied to the acceleration of a wide variety of applications with characteristics similar to broad phase collision detection. Appropriate applications include ray-tracing and N-body simulations, such as astrophysics, MD, plasma physics, fluid mechanics and semiconductor device characterisation. The generalisations could initially be specialised for a particular application, similar to how the broad phase collision detection microarchitectures are essentially such specialisations. If many specialisations were implemented, it would be interesting to switch between them using the reconfigurability of an FPGA. A platform such as a hybrid ASIC-FPGA [Zuchowski et al, 2002] would open the possibility of a more sophisticated means of dealing with multiple specialisations. Those parts of the microarchitectures that do not change between specialisations could be implemented in the fixed-function logic, while those that change could exploit the FPGA reconfigurability. Such an implementation would necessitate a thorough analysis of the changes demanded by the different specialisations. Finally, the most sophisticated solution would be to implement the generalised microarchitectures directly in hardware with no specialisation. This would require the creation of a sophisticated software driver that could adapt the functionality of the microarchitecture as required by the application. However, it would allow multiple algorithms to be accelerated using a high-speed platform such as an ASIC. All of the aforementioned concepts would allow for a determination of the appropriateness of the outlined microarchitectures to the acceleration of other applications.

6.1.1 Benchmarking Framework

As for the hybrid systems, there is significant scope for advancing the outlined benchmarking framework. Initially, a wide range of interactive applications utilising collision detection could be thoroughly analysed to create a set of minimal but complete parameters that can recreate the majority of scenarios. Modifying the framework to include these parameters would expand its flexibility and enable it to perform more detailed testing that the current version. Therefore, this proposal should make the

benchmarking framework of greater utility to algorithm and application developers. It is also possible that these analyses could help to reduce the complexity of the framework by indicating that certain parameters are superfluous, although this currently appears unlikely. It is more likely that a new parameter would supersede the functionality of an existing one.

After discussing the topic of parameters for the benchmarking framework with computer game industry professionals, some scenarios that would be beneficial to test were suggested.

Object Motion Objects could be tested moving smoothly and uniformly. In addition, they could be tested moving randomly over distances of considerable length, as well as randomly appearing and disappearing throughout the duration of the benchmark. The two latter scenarios may perform inefficiently with coherence-based algorithms, such as SAP, as these do not work well with random motion and have a cost associated with adding and removing objects.

The first case of smooth and uniform motion is already covered by the benchmarking framework, for example when using the default parameters outlined in Section 3.6.2. Random motion can be approximated by turning off gravity and increasing the size of the objects in order to increase collisions. In this case, the distance can be increased by extending the execution time of the benchmark.

Object Size Large objects that span the scene could be tested, along with scenarios involving a mixture of small and large objects. These scenarios may perform inefficiently with algorithms, such as single axis SAP, that work well with many uniformly sized objects.

The current benchmarking framework could simulate the mixture of small and large objects by setting the 'Size' parameter to a probability distribution and selecting the appropriate parameters to achieve this mixture.

Object Distribution Benchmarks with a variety of object distributions could be created. In particular, extremely large environments with relatively small regions of densely packed objects would be interesting, as these are inefficient with uniform grids and related algorithms. In addition, an environment where every object aligns along a single axis could be created, as these may not work well with single axis algorithms. Such environments are typical in interactive applications, such as computer games, where many objects rest on the ground and share the same y value. This proposed environment could be replicated so that each of the three axes is tested. Finally, these environments should be adapted to make the aligned objects oscillate slightly, as the changing overlaps would make collision testing difficult.

The current benchmarking framework can approximate large environments with densely packed objects by reducing the size of the objects and using the normal probability distribution to cluster the objects in the centre. However, the clustering would be fairly loose and this would only serve as an approximation of the environment. In addition, environments where all of the objects align along a given axis can be created by specifying an acceleration operating in the direction of a specific axis-aligned plane. The benchmark can then be executed until all of the

objects align. However, this solution is non-optimal as the effect of the objects moving towards the axis would need to be filtered out.

Timing Lengthy benchmarks could also be created as some algorithms, such as those that refit trees, degrade over time.

These are already entirely supported by the benchmarking framework using the 'Cycles' parameter.

External Queries In addition, it would be interesting to consider the external queries for which a collision detection algorithm could be used. For example, AABB overlap, ray-casting and frustum checking could all be requested by parts of the program loop.

Support for the outlined scenarios was not added to the benchmarking framework as they do not entirely reflect its raison d'être. For example, many of the described environments solely attempt to uncover inefficient behaviour in collision detection algorithms. This differs from the outlined benchmarking framework, which attempts to recreate application scenarios and to test the algorithms within the context of these scenarios. Moreover, many of the described scenarios cannot easily be decomposed into a set of user-configurable parameters. Consequently, their implementation would reduce the benchmarking framework to a set of fixed scenarios instead of the current parameterisable framework. Therefore, there would be significant problems with implementing the proposed ideas. However, the ideas are of benefit and it would be interesting to implement them in some way. This could be done by significantly overhauling the benchmarking framework or by creating a derivative that tests specific pre-defined scenes. This proposal, like the previous, would enhance the flexibility of the benchmarking framework.

Flexibility could also be enhanced by expanding the object types supported. The framework currently supports rigid bodies modelled using polygonal representations, reflecting the object type and representation used in the vast majority of interactive applications. However, some applications use alternative object types and representations and it would be relatively easy to adapt the framework to support these. For example, articulated rigid bodies, deformable bodies and particle systems could join the list of supported object types. Similarly, parametric surfaces, implicit surfaces, voxel objects and constructive solid geometry (CSG) could join the list of supported object representations. Although these would be trivial to add to the framework, their addition raises some interesting questions such as whether they would necessitate modification of the parameters to allow their properties to be adjusted.

The benchmarking framework currently offers a wide range of parameters, which can be used to create an extremely wide range of benchmarks. If the aforementioned suggestions were implemented, the number of parameters could increase further. The flexibility resulting from these parameters is the framework's primary advantage, although the parameters also create a significant disadvantage as they make it impossible to sweep the entire test space. A sophisticated approach to managing the complexity is clearly demanded.

One promising approach for managing this complexity is interactive steering with visualisation, which is a technique designed to manage the complexity that results from having a scenario with my-

riad parameters. The idea is to concentrate initially on varying a small subset of the scenario's parameters, while leaving the remaining parameters at ideal or typical values. This results in a set of graphs showing the effect of the variation, which enables optimal values to be selected for the simulated subset of parameters. Using these optimal values, the next subset of parameters can be varied. This process continues until all parameters have been considered. Although it is similar to what could be performed manually, the proposed automated technique provides a quicker solution. Interactive steering with visualisation has previously been successfully used in a variety of applications including CAE [Matković et al, 2008]. Although the technique is typically used to find optimal parameter values, it could instead be used to find those parameter values that cause the scenario to perform inefficiently. This would be of use to algorithm developers attempting to improve the performance of their algorithm. It could expose problematic areas that were not uncovered through a manual, systematic analysis of the various parameters. For this reason, it would be greatly beneficial to expand the framework to include interactive steering techniques that guide a user's search of the available test space.

Finally, it would be interesting to create a set of predefined benchmarks that recreate common scenarios from interactive applications. Creating such benchmarks means that developers would not need to define custom benchmarks. Therefore, this would provide an alternative solution for reducing the complexity of the benchmarking framework. It would additionally benefit algorithm developers, as they would be able to publish the performance of their algorithm while executing one or more of these predefined benchmarks. Moreover, the predefined benchmarks could be exported as a set of object paths in order to increase repeatability, as outlined in Section 3.6.3. However, the creation of these predefined benchmarks creates problems for the benchmarking framework as it violates the framework's philosophy. In particular, predefined benchmarks prevent algorithm developers from locating every weak point in their algorithm. In addition, application developers would not be able to sweep through a range of collision detection algorithms in recreated application scenarios. Therefore, it would be necessary to maintain the original framework and its set of parameters even if a set of predefined benchmarks were created. However, creating the predefined benchmarks is still worthwhile as it provides a means of reducing the framework's complexity. All of the aforementioned techniques would greatly improve the utility and flexibility of the framework, thereby making it of incalculable benefit to both algorithm and application developers.

6.2 Conclusion

The thesis statement, from Section 1.3, is:

> Collision detection can be significantly accelerated using a hybrid software and hardware system. By appropriately leveraging the properties of both parts of the system, it is possible to make performance gains that would be unattainable using software alone.

This section will analyse whether this statement has been adequately proved by the content of this thesis.

To determine the statement's veracity, it is first necessary to analyse the benefits of the hybrid systems demonstrated throughout the thesis. Novel microarchitecture designs were outlined in addition to a scheme for dividing the collision detection algorithms across software and hardware. Using FPGA implementations of the microarchitectures, significant accelerations were demonstrated over a state-of-the-art software algorithm. When considered from a software perspective, the accelerations did not continue past 4000 objects. However, this limitation was easily overcome by using spatial partitioning to divide large quantities of objects into smaller ones that could be rapidly processed by the microarchitectures. Using spatial partitioning, accelerations were demonstrated for up to 8000 objects.

Next, it is important to analyse any weaknesses in the content in order to determine whether these have a significant effect. However, it is clear that the only real weakness is the suboptimal performance of the spatial partitioning algorithms. This suboptimality resulted in an inability to accelerate the 16 000-object benchmark. This is undesirable as it would be greatly beneficial if the hybrid systems could support very large object quantities that software is currently unable to handle.

Despite this weakness, the current implementation of the hybrid systems could be used directly in a practical context for the acceleration of collision detection. However, it would be better if the inefficiencies in the spatial partitioning algorithms were removed through the proposals outlined in Section 6.1. This should enable the hybrid systems to achieve an acceleration for simulations involving 16 000 or more objects. After this modification, the practical hybrid system could then be constructed from the area-efficient microarchitecture. Although this system replicates data, it is faster than the many-object microarchitecture due to its higher degree of parallelism. Moreover, if the spatial partitioning performed efficiently, partitioning could be used for object quantities greater than 1024, making the data replication unproblematic. This should result in a hybrid system offering very significant advantages over the state-of-the-art. However, even greater performances would be attainable using a higher performance platform such as an ASIC. This is not strictly necessary for use in a practical system but it would provide much better performance. The vast majority of application accelerators use ASICs for this reason.

To conclude, this thesis has proved the validity of the thesis statement. In other words, the outlined research has shown that collision detection can be significantly accelerated using a hybrid software and hardware system. By appropriately leveraging the properties of both parts of the system, it has been demonstrated that it is possible to make performance gains that would be unattainable using software alone. The research outlined in this thesis is sufficient to prove the statement in its entirety. Although some weaknesses are present, these do not take away from the veracity of the statement, and it is evident that if these were rectified, the validity of the statement could be extended.

Appendix A

Generalised Microarchitectures

> *…the world will not stop and think – it never does, it is not its way; its way is to generalize from a single sample.*
>
> —Mark Twain (*The United States of Lyncherdom*), writer and lecturer (1835–1910)

Generalisation of research designs is beneficial for facilitating the widespread adoption of useful concepts across a variety of research areas. This appendix generalises the previously outlined area-efficient and many-object microarchitectures used to perform broad phase collision detection. This generalisation allows the microarchitectures to be repurposed for the execution of a variety of alternative algorithms. Two generalisations of the microarchitectures, covering the generation of pairs of data from either a single or multiple source sets, are discussed. The discussion continues with a possible implementation option. The appendix concludes with a brief discussion of the advantages provided by these microarchitectures.

A.1 Design

Although the microarchitectures outlined in Chapter 3 are restricted to performing broad phase collision detection, it is clear that, at their core, these microarchitectures essentially generate pairs of objects and perform a comparison on each pair. The generation of pairs is useful for the microarchitecture implementation of myriad other algorithms. Therefore, it is beneficial to generalise the microarchitectures to facilitate their adoption in these other algorithms. The first generalisation generates pairs of objects from a single source, while the second takes the generalisation further and uses two sources.

A.1.1 One-Source

The essence of the microarchitectures essentially consists of a pipeline implementing two primary operations – buffer and procedure – for the area-efficient microarchitecture, and three primary operations

– buffer, reorder and procedure – for the many-object microarchitecture, with 'procedure' replacing 'compare' from the broad phase collision detection microarchitectures. The primary operations are illustrated in Figure A.1 and detailed schematics are provided in Figure A.2. These microarchitectures perform the 'procedure' on pairs of objects generated from a single source.

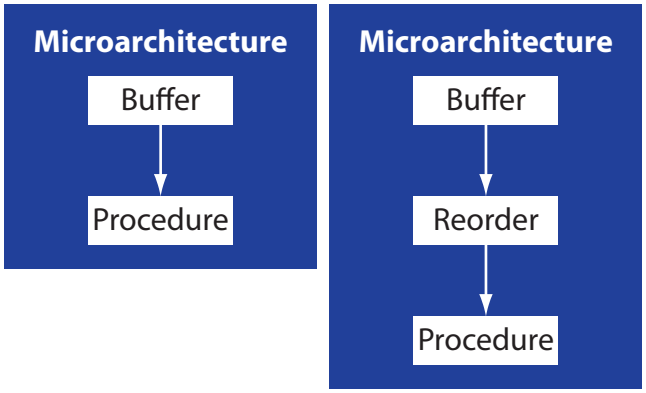


Figure A.1: *Overview of the one-source generalised microarchitectures showing the connection of the primary operations.* **(a)** *Area-efficient.* **(b)** *Many-object.*

As a first step towards generalisation, the number of coordinate axes could be adjusted from the current three. This modification is relatively straightforward but requires adjustments to all of the microarchitectures' operations. In Figure A.2, for example, the number of coordinate axes has been reduced to one. Alternatively, the use of $2m$ memories per coordinate axis could be adjusted so that a different multiplier of m is used. Figure A.2 uses a multiplier of one. A more substantial change would be to adjust the compare operation, by substituting it with any procedure. If the procedure became a functional unit, the buffer and reorder operations would essentially function as a register file supplying data to the functional unit, which would effectively create an instruction set processor (ISP). The primary benefit of such a register file is the large storage space offered, coupled with the high degree of parallelism. However, the access pattern would be significantly constrained so that the range of applications that could utilise such an ISP would be limited. Figure A.2 demonstrates the most generic implementation of the procedure, but this can be adapted in myriad ways depending on the algorithm under consideration.

By making some of the outlined changes, the microarchitecture could also be extended to execute various N-body simulations. As for the broad phase collision detection algorithm, the data would typically use three coordinate axes but the multiplier of m would vary between different N-body simulations. As each set of particles is traversed, the procedure would perform force accumulation by considering the effect of the mass and acceleration of each of the set's particles on each other. The procedure would also be response for performing a complex mathematical formula determined by the nature of the N-body simulation under consideration. This formula would typically be applied in a second iteration of the microarchitecture, after force accumulation has been completed.

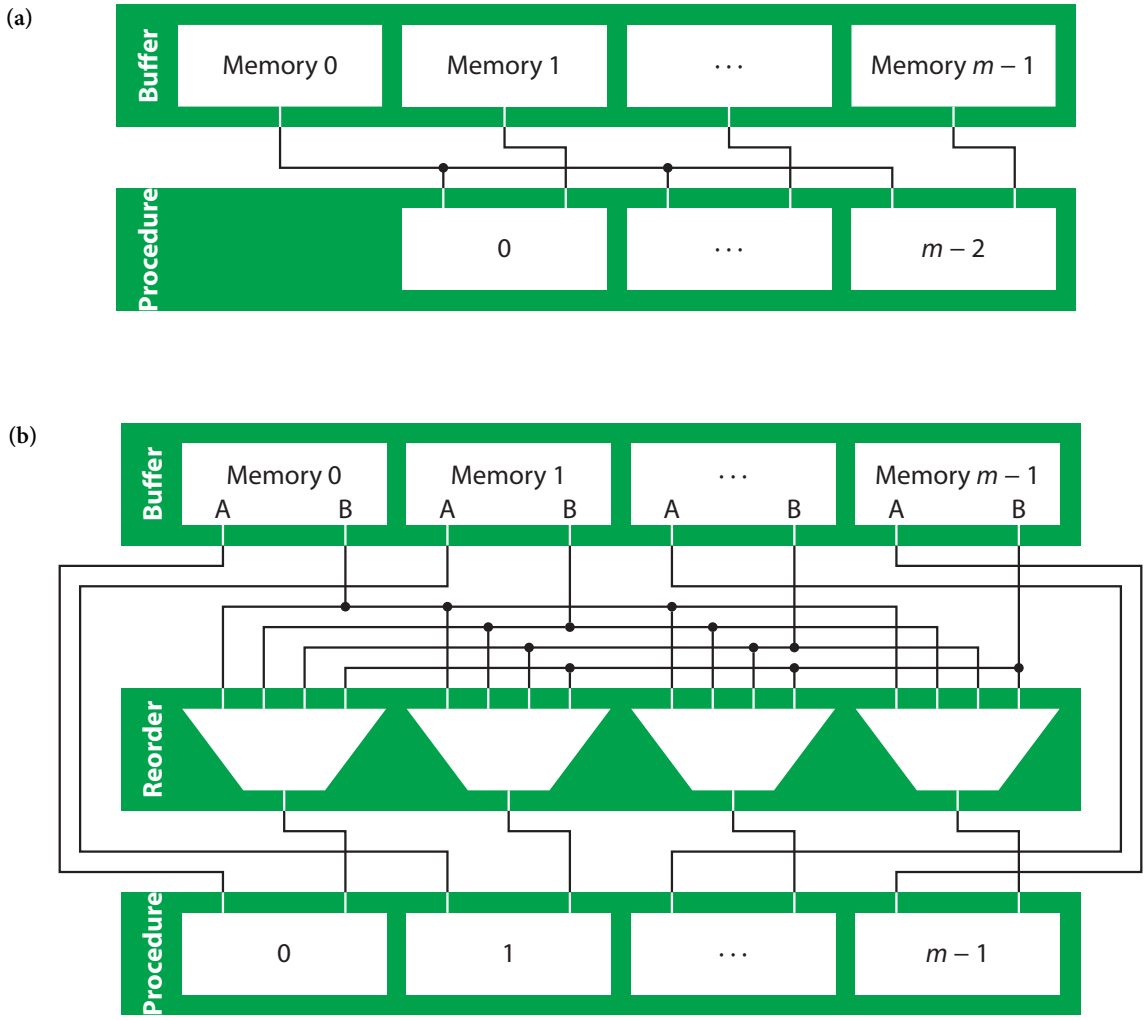


Figure A.2: *Detailed schematics of the one-source generalised microarchitectures.* **(a)** *Area-efficient.* **(b)** *Many-object.*

These generalisations create microarchitectures akin to the design proposed by Lawrie and Vora [1982]. Their design uses a parallel collection of memories connected to a processor, designed to maximise the level of achievable parallelism. The description uses discrete components, although it could be directly implemented as a microarchitecture. However, it differs from the generalised microarchitectures as certain memory locations must be left unutilised and it only works with a prime number of memories, which results in complex indexing logic. These restrictions are imposed as a result of the system being targeted towards matrix computations. Deb [1996] solves all of these problems using a 'multiskewing' technique to determine where each datum should be located in memory. Despite their similarities to the generalised microarchitectures, both designs are unsuitable for use in applications such as broad phase collision detection or N-body simulations, as they do not facilitate the generation of pairs. Instead, their memory layouts have been designed to facilitate various matrix operations.

A.1.2 Two-Source

All of the previous microarchitectures generate pairs from a single source. This results in

$$\frac{n\left(n-1\right)}{2}$$

pairs. For many algorithms, this is the appropriate behaviour. However, many other algorithms require the generation of pairs from two source sets, where each pair contains one datum from the first set and one datum from the second set. This results in $n_0 n_1$ pairs where $n_i, i \in \{0, 1\}$ is the number of data in the ith set. To facilitate these algorithms, the microarchitectures must be modified to permit the generation of pairs where the second index is less than or equal to the first, as such pairs are eliminated by the one-source microarchitecture.

In this generalisation, features from both the area-efficient and many-object microarchitectures are combined. This microarchitecture consists of a pipeline implementing two primary operations – buffer and procedure – as illustrated in Figure A.3. A detailed schematic is provided in Figure A.4. First, the buffer from the previous microarchitectures is replicated so that two buffers are created. Each buffer uses m_i single-port memories but a multiplier can be applied to m_i as appropriate. As for the many-object microarchitecture, the relevant data are stored across the m_i memories so that no data are duplicated, as illustrated in Table A.1. This structure and data layout creates a division between the memory address and the index of the datum being retrieved but the index can be computed using $a_i m_i + j_i$.

The data are outputted from the buffer in a specific sequence to perform the necessary procedure. Originally, the address inputs of each buffer are initialised to 0. On the subsequent cycle, the address inputs of the first buffer retain their values, while the address inputs of the second buffer are each incremented to 1. This sequence continues until the address inputs of the second buffer select

$$\left\lceil \frac{n_1}{m_1} \right\rceil - 1,$$

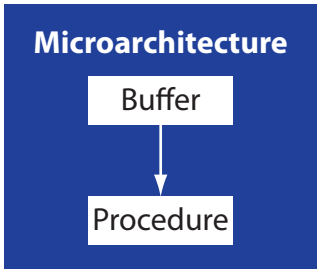


Figure A.3: *Overview of the two-source generalised microarchitecture showing the connection of the two primary operations.*

	Buffer_0 Memory		Buffer_1 Memory			
Address	0	1	0	1	2	3
0	0_0	1_0	0_1	1_1	2_1	3_1
1	2_0	3_0	4_1	5_1	6_1	7_1
2	4_0	–	8_1	9_1	–	–

Table A.1: *Buffer layouts for the two-source generalised microarchitecture with $m_0 = 2$, $m_1 = 4$, $n_0 = 5$ and $n_1 = 10$. Locations containing no data are marked '–'.*

which is the address corresponding to datum $n_1 - 1$. At this stage, the address inputs of the first buffer are each incremented by 1, while the address inputs of the second buffer are reset to 0. The sequence continues until the address inputs of the first buffer select

$$\left\lceil \frac{n_0}{m_0} \right\rceil - 1,$$

which is the address corresponding to datum $n_0 - 1$ and the address inputs of the second buffer select

$$\left\lceil \frac{n_1}{m_1} \right\rceil - 1.$$

This permits data after $n_i - 1$ to be accessed; these must be subsequently removed. The pairs generated are exemplified in Table A.2, while the entire sequence is exemplified in Table A.3 and Algorithm A.1. On each cycle $m_0 m_1$ pairs are emitted, corresponding to each item emitted from the first buffer combined with each item emitted from the second buffer.

Although the design uses single-port memories, it is equally amenable to memories with p ports, where $p > 1$. In these cases, the sequence can be trivially extended across the extra ports by considering each port as a separate memory. Extra ports should result in increased performance with decreased resource utilisation. Similarly, it is possible to extend the microarchitecture to support more than two sources.

The maximum value of n_i is $d_i m_i$. As for the other microarchitectures, multiple memories may be combined to create a deeper memory, but this may limit the value of m_i, which may decrease overall

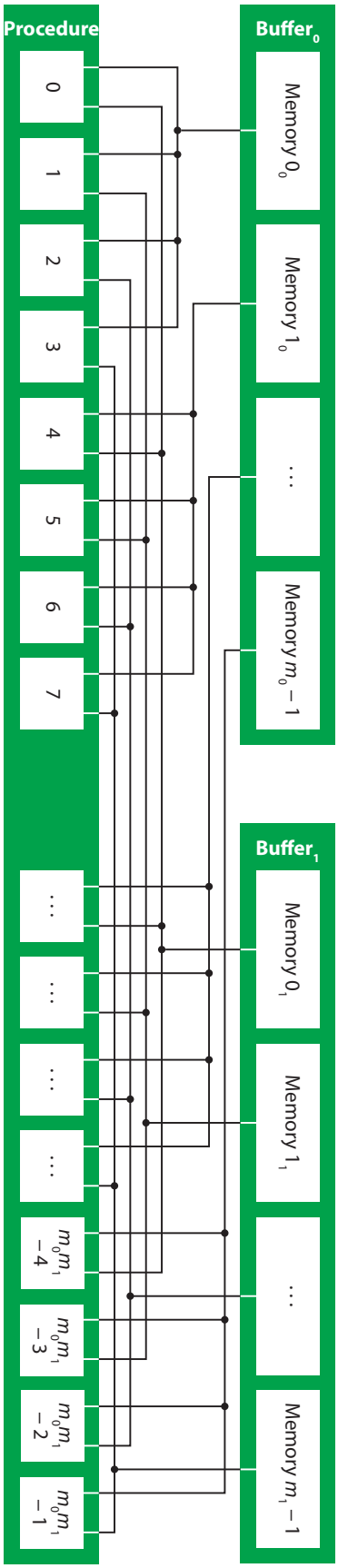


Figure A.4: *Detailed schematic of the two-source generalised microarchitecture.*

	Memory															
Cycle	0_0	0_1	0_0	1_1	0_0	2_1	0_0	3_1	1_0	0_1	1_0	1_1	1_0	2_1	1_0	3_1
0	0_0	0_1	0_0	1_1	0_0	2_1	0_0	3_1	1_0	0_1	1_0	1_1	1_0	2_1	1_0	3_1
1	0_0	4_1	0_0	5_1	0_0	6_1	0_0	7_1	1_0	4_1	1_0	5_1	1_0	6_1	1_0	7_1
2	0_0	8_1	0_0	9_1	0_0	–	0_0	–	1_0	8_1	1_0	9_1	1_0	–	1_0	–
3	2_0	0_1	2_0	1_1	2_0	2_1	2_0	3_1	3_0	0_1	3_0	1_1	3_0	2_1	3_0	3_1
4	2_0	4_1	2_0	5_1	2_0	6_1	2_0	7_1	3_0	4_1	3_0	5_1	3_0	6_1	3_0	7_1
5	2_0	8_1	2_0	9_1	2_0	–	2_0	–	3_0	8_1	3_0	9_1	3_0	–	3_0	–
6	4_0	0_1	4_0	1_1	4_0	2_1	4_0	3_1	–	0_1	–	1_1	–	2_1	–	3_1
7	4_0	4_1	4_0	5_1	4_0	6_1	4_0	7_1	–	4_1	–	5_1	–	6_1	–	7_1
8	4_0	8_1	4_0	9_1	4_0	–	4_0	–	–	8_1	–	9_1	–	–	–	–

Table A.2: *Sequence of pairs generated for the two-source generalised microarchitecture with $m_0 = 2$, $m_1 = 4$, $n_0 = 5$ and $n_1 = 10$. Invalid indices are marked '–'.*

	Buffer 0				Buffer 1							
	Address		Index		Address				Index			
Cycle	0	1	0	1	0	1	2	3	0	1	2	3
0	0	0	0_0	1_0	0	0	0	0	0_1	1_1	2_1	3_1
1	0	0	0_0	1_0	1	1	1	1	4_1	5_1	6_1	7_1
2	0	0	0_0	1_0	2	2	2	2	8_1	9_1	–	–
3	1	1	2_0	3_0	0	0	0	0	0_1	1_1	2_1	3_1
4	1	1	2_0	3_0	1	1	1	1	4_1	5_1	6_1	7_1
5	1	1	2_0	3_0	2	2	2	2	8_1	9_1	–	–
6	2	2	4_0	–	0	0	0	0	0_1	1_1	2_1	3_1
7	2	2	4_0	–	1	1	1	1	4_1	5_1	6_1	7_1
8	2	2	4_0	–	2	2	2	2	8_1	9_1	–	–

Table A.3: *Sequence of addresses and indices for the buffer of the two-source generalised microarchitecture with $m_0 = 2$, $m_1 = 4$, $n_0 = 5$ and $n_1 = 10$. Invalid indices are marked '–'.*

procedure TwoSetControl(n_0, n_1, m_0, m_1)
  for $i \leftarrow 0$ **to** $\lceil \frac{n_0}{m_0} \rceil - 1$ **do**
    $address_0 \leftarrow i$
    $index_0^0 \leftarrow im_0$
    $index_0^1 \leftarrow im_0 + 1$
    $\vdots$
    $index_0^{m_0-1} \leftarrow im_0 + m_0 - 1$
    for $j \leftarrow 0$ **to** $\lceil \frac{n_1}{m_1} \rceil - 1$ **do**
      $address_1 \leftarrow j$
      $index_1^0 \leftarrow jm_1$
      $index_1^1 \leftarrow jm_1 + 1$
      $\vdots$
      $index_1^{m_1-1} \leftarrow jm_1 + m_1 - 1$
      wait
    end for
  end for
end procedure

Algorithm A.1: *Algorithm to generate control signals for the buffer of the two-source generalised microarchitecture.* '**wait**' *indicates that the current control signals should be maintained for a single clock cycle.*

performance. The memory bandwidth is

$$f_i m_i p_i w_i \text{ bit/s}.$$

The number of cycles required to generate the sequence is

$$\left\lceil \frac{n_0}{m_0 p_0} \right\rceil \times \left\lceil \frac{n_1}{m_1 p_1} \right\rceil.$$

The reorder operation is unnecessary with the two-source generalisation, as the data do not need to be rearranged. The procedure and the communication interface require no modification to accommodate the adapted buffer, other than straightforward adjustment for the number of pairs generated in a single cycle.

One potential use for this generalisation of the microarchitecture is to implement the ray-tracing algorithm [Appel, 1968; Whitted, 1980], which is gaining interest among the graphics community as a possible replacement for rasterisation. For this algorithm, the three-dimensional scheme used for the broad phase collision detection microarchitecture is required. In addition, the first buffer is resized to $2m_0$ to contain the rays, which consist of two points specifying the origin and a normalised direction, and the second buffer is resized to $3m_1$ to contain the triangles, which consist of three points specifying the vertices. The ray-intersection test is performed as the procedure. During processing, the microarchitecture traverses the lists of rays and triangles and performs the ray intersection test on each pair. Microarchitectures for ray-tracing have previously been designed [Woop et al, 2005] but these have tended to mimic software-based algorithms.

A.2 Implementation

None of the generalised microarchitectures were implemented, as such implementations would be extremely generic and of little benefit unless a specific algorithm were under consideration. Moreover, the validity of these generalisations has already been proved by the collision detection microarchitectures. If the generalisations were implemented, an interesting target platform would be a hybrid ASIC-FPGA [Zuchowski et al, 2002]. Using this platform, the buffer and reorder operations could be implemented in the fixed-function ASIC component, while the procedure could be implemented in the FPGA component as required. This would permit a single platform to perform different procedures, while leveraging many of the benefits of ASICs.

A.3 Advantages

Although these generalised microarchitectures have not been directly applied to any specific algorithm, there is substantial reason to believe that they would be advantageous for the microarchitecture implementation of many algorithms. Firstly, they provide a means to generate pairs of objects in hardware,

which is useful for the microarchitecture implementation of many different algorithms, including those that do not necessarily generate pairs in their software implementations. Previously, there had been no standard design for performing this operation and hardware designers often had to invent ad hoc solutions for different designs. These generalisations, therefore, should facilitate hardware designers by allowing them to focus on the design and optimisation of other parts of their microarchitecture. In addition, these microarchitectures facilitate the creation of a generic hardware component that can accelerate many different algorithms. Secondly, these microarchitectures effectively exploit parallelism, which is the primary means of gaining acceleration in a microarchitecture. Finally, it can be envisaged that the microarchitectures will exhibit similar benefits to their broad phase collision detection counterparts. They are also likely to exhibit gains greater than the microarchitectures proposed by Lawrie and Vora [1982] and Deb [1996], as the generalised microarchitectures can also be thought of as generalisations of these microarchitectures. In summation, despite the lack of a design specialised to an algorithm outside broad phase collision detection, it is clear that there is significant evidence to indicate that these microarchitectures would provide substantial benefits to the microarchitecture implementation of myriad algorithms.

A.4 Summary

This appendix began by explaining the rationale for generalising the previously outlined broad phase collision detection microarchitectures. The discussion continued with descriptions of two generalisations. The first generalisation used a single source set, akin to the collision detection microarchitectures. It could be constructed from either the area-efficient or many-object microarchitecture. The second generalisation used two source sets to facilitate algorithms such as ray-tracing. This generalisation combined concepts from the two earlier microarchitectures. Following these descriptions, a brief account of how the generalisations could be used within a practical system was provided, with a hybrid ASIC-FPGA suggested as an interesting target platform. The appendix concluded with a discussion of the advantages of utilising the microarchitectures.

Appendix B

Settings and Specifications

We think in generalities, but we live in detail.

—Alfred North Whitehead, mathematician and philosopher (1861–1947)

To facilitate independent verification of results, as per the scientific method, it is necessary to have the complete details of all implementations. This appendix expands on the brief compiler and synthesiser settings and specifications found in previous chapters to provide all germane information. This includes a list of the settings used for synthesising, translating, mapping, placing and routing the microarchitectures, followed by the software compiler settings together with a list of the version numbers of the different libraries. The appendix concludes with the details of the different computers used in the evaluation of the system.

B.1 Hardware

Xilinx ISE 10.1.03 [Xilinx, 2008] was used for compiling the microarchitecture implementations. The majority of the code detailing the microarchitectures was written in VHDL [IEEE, 2009] with some Verilog [IEEE, 2006] used for interfacing with the DRC Accelium AC2030.

Xilinx Virtex-II Xilinx ISE 10.1.03 was used for the Xilinx Virtex-II [Xilinx, 2007b] designs, since this was the last version of the tool to support the Xilinx Virtex-II. The settings used were:

- **Synthesize – XST**
  - **Synthesis Options**
    - **Optimization Effort:** High
    - **Global Optimization Goal:** Maximum Delay
    - **Cross Clock Analysis:** True

        * **Other XST Command Line Options:** `-read_cores optimize`
    - **HDL Options**
        * **Safe Implementation:** Yes
        * **Case Implementation Style:** Full-Parallel
        * **Automatic BRAM Packing:** True
        * **Resource Sharing:** False
    - **Xilinx Specific Options**
        * **Equivalent Register Removal:** False
        * **Register Balancing:** Yes
        * **Optimize Instantiated Primitives:** True
- **Implement Design**
    - **Map Properties**
        * **Perform Timing-Driven Packing and Placement:** True
        * **Map Effort Level:** High
        * **Extra Effort:** Normal
        * **Combinatorial Logic Optimization:** True
        * **Register Duplication:** True
        * **Allow Logic Optimization Across Hierarchy:** True
        * **Map to Input Functions:** 8
        * **Optimization Strategy (Cover Mode):** Speed
        * **Pack I/O Registers/Latches into IOBs:** For Inputs and Outputs
    - **Place & Route Properties**
        * **Place & Route Effort Level (Overall):** High
        * **Extra Effort (Highest PAR level only):** Normal

All unspecified settings were left at their defaults. Values were selected to achieve an optimum clock frequency and were based on those recommended by Alpha Data [2009].

Xilinx Virtex-5 Xilinx ISE 10.1.03 was also used for the Xilinx Virtex-5 [Xilinx, 2009a] designs, since this was the latest version of the tool supported by the DRC Accelium AC2030. The settings used were:

- **Synthesize – XST**
    - **Synthesis Options**
        * **Other XST Command Line Options:** `-read_cores optimize`
    - **HDL Options**
        * **Safe Implementation:** Yes
        * **Resource Sharing:** False

    - **Xilinx Specific Options**
        * **Add I/O Buffers:** False
        * **Equivalent Register Removal:** False
- **Implement Design**
    - **Map Properties**
        * **Placer Effort Level:** High
        * **Placer Extra Effort:** Normal
        * **Combinatorial Logic Optimization:** True
        * **Register Duplication:** True
        * **Optimization Strategy (Cover Mode):** Speed
    - **Place & Route Properties**
        * **Place & Route Effort Level (Overall):** High
        * **Extra Effort (Highest PAR level only):** Normal

All unspecified settings were left at their defaults. Values were selected to achieve an optimum clock frequency and were based on those recommended by DRC Computer [2009]. Less aggressive optimisation was used than for the Xilinx Virtex-II as some optimisations conflicted with the HyperTransport core that was supplied with the DRC Accelium AC2030.

B.2 Software

G++ and Microsoft Visual C++ were used for compiling the software implementations, with the choice of compiler corresponding to the OS. All implementations were written in C++ [ISO and IEC, 2003] with extensive use made of the STL and TR1 additions [ISO and IEC, 2005].

G++ G++ 4.2.4 [Free Software Foundation, 2008] was used for compiling the software for execution under GNU/Linux. The optimisations were:

- `-O2`
- `-march=native`
- `-fomit-frame-pointer`

Microsoft Visual C++ Microsoft Visual C++ 2008 Service Pack 1 with Feature Pack [Microsoft, 2009a] was used for compiling the software for execution under Microsoft Windows. The optimisations used were the default 'Release' optimisations.

Libraries All libraries were compiled using their default 'Release' optimisation levels, in both G++ and Microsoft Visual C++. The version numbers of each of the libraries used were:

- ADM-XRC SDK 2.9.3 [Alpha Data, 2009]
- Boost C++ Libraries 1.36.0 (GNU/Linux) [Boost, 2008], 1.40.0 (Microsoft Windows) [Boost, 2009]
- Bullet Physics SDK 2.75 [Coumans, 2009]
- DRC Coprocessor System Development Kit 3.1.1 [DRC Computer, 2009]
- freeglut 2.4.0-6 (GNU/Linux) [freeglut Programming Consortium, 2003], GLUT for Win32 3.7.6 (Microsoft Windows) [Robins, 2001]
- wxWidgets 2.8.7.1 (GNU/Linux) [Smart et al, 2007], 2.8.10 (Microsoft Windows) [Smart et al, 2009]

The library versions used under GNU/Linux typically corresponded to those supplied with the latest updates to the system's OS.

B.3 Computers

Results were acquired using two different computers, referred to as 'Computer 1' and 'Computer 2' throughout this thesis.

Computer 1 Computer 1 was the primary system used for measuring results. It hosted the DRC Accelium AC2030 [DRC Computer, 2008b] along with the following components:

- **System:** DRC Computer Accelium AD2311 [DRC Computer, 2008a]
- **Operating System (OS):** Ubuntu GNU/Linux 8.04 LTS Desktop (64-bit) using kernel 2.6.24-23-generic [Canonical, 2008]
- **CPU:** Quad-Core AMD Opteron 2350 (2 GHz) [AMD, 2007]
- **RAM:** 8 GiB PC2-5300 double data rate 2 SDRAM (DDR2 SDRAM)
- **Graphics Card:** NVIDIA GeForce 8400 GS with 256 MiB RAM [NVIDIA, 2009a]

Computer 2 Computer 2 was the secondary system used for measuring results. It hosted the Alpha Data ADM-XRC-II [Alpha Data, 2002b] along with the following components:

- **System:** Dell PowerEdge 600SC [Dell, 2002]
- **Operating System (OS):** Microsoft Windows XP Professional Service Pack 3 (32-bit) [Microsoft, 2009b]
- **CPU:** 1.8 GHz Intel Pentium 4 [Intel, 2002]
- **RAM:** 2 GiB PC-2700 double data rate SDRAM (DDR SDRAM)
- **Graphics Card:** HIS ATI Radeon X1300 with 256 MiB RAM [HIS, 2008]

Appendix C

Source Code

We're given a code to live our lives by. We don't always follow it, but it's still there.

—Gary Oldman, actor (1958–)

Source code is the logic that underpins the implementation of a design. This appendix details selected VHDL source code from the various implementations. It begins by listing the buffer control logic from the area-efficient microarchitecture before listing the equivalent logic from the many-object microarchitecture. Both sets of buffer control logic support spatial partitioning, but can be used without spatial partitioning as outlined in Section 3.4. This code corresponds to the primary contribution of the microarchitectures in terms of algorithm development.

C.1 Coding Style

All of the code is written in the two-process structured Gaisler design style [Gaisler, 1997], which is intended to simplify the error-prone process of writing hardware description language (HDL) code, through an increase in the abstraction level. This increased abstraction also aids readability and maintainability. At its core, the Gaisler style uses one process for combinatorially executing a VHDL entity's logic and assigning the computed values to a set of variables. The values of these variables are then sequentially assigned to the appropriate signals in a second process. In addition, records are used to group signals in order to reduce complexity. A block diagram of the Gaisler style is provided in Figure C.1.

In addition to using the Gaisler style, the majority of each combinational process has been written using finite-state machines (FSMs). These are a standard construct in HDL code for assigning values to variables in a specific sequence. FSMs were used as the majority of HDL synthesis tools contain specialised routines designed to optimise these constructs. Consequently, FSMs are often the optimal approach for implementing a HDL design.

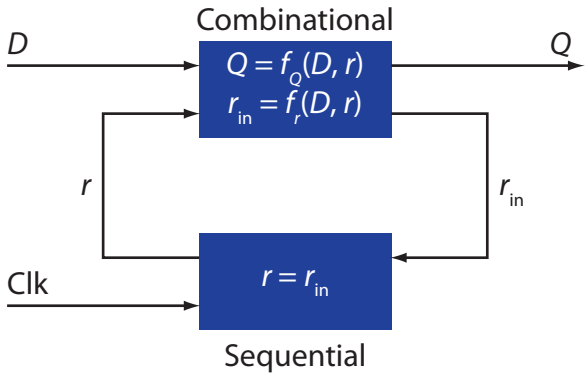


Figure C.1: *Block diagram showing the two processes of the structured Gaisler design style.*

C.2 Area-Efficient Microarchitecture

The buffer control source code for the area-efficient microarchitecture roughly corresponds to Algorithm 3.2. In addition to the logic outlined in Algorithm 3.2, the VHDL source code is responsible for loading data into the internal memories. However, the control logic for the reorder operation, which was included in the earlier algorithm, is excluded from this source code. Both the Xilinx Virtex-II and Xilinx Virtex-5 use the same implementation. In the code, `parallelism` is equivalent to m.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

use work.functions.all;
use work.parameters.all;
use work.types.all;

entity buffer_control is
  port (
    clk   : in  std_logic;
    reset: in  std_logic;
    d0    : in  control_in_type;
    d1    : in  serialise_control_out_type;
    q     : out buffer_control_out_type
  );
end entity;

architecture rtl of buffer_control is
  type state_type is (initialisation_0, initialisation_1,
     initialisation, read_data_first, read_data, waiting);

  subtype parallelism_type is natural range 0 to parallelism - 1;
```

```
  type parallelism_type_array is array(0 to parallelism - 1) of
     parallelism_type;

  type reg_type is
  record
    -- State
    state                     : state_type;
    condition0                : std_logic;
    condition1                : std_logic;
    last_address_decrement    : std_logic_vector(address_width downto 0);
    -- Outputs
    valid                     : std_logic;
    finished                  : std_logic;
    last_address              : std_logic_vector(address_width downto 0);
    write_enable_minimum_x    : std_logic_vector(parallelism - 1 downto
       0);
    write_enable_minimum_y    : std_logic_vector(parallelism - 1 downto
       0);
    write_enable_minimum_z    : std_logic_vector(parallelism - 1 downto
       0);
    write_enable_maximum_x    : std_logic_vector(parallelism - 1 downto
       0);
    write_enable_maximum_y    : std_logic_vector(parallelism - 1 downto
       0);
    write_enable_maximum_z    : std_logic_vector(parallelism - 1 downto
       0);
    buffer_address_minimum_x_a: buffer_address_extended_array;
    buffer_address_minimum_y_a: buffer_address_extended_array;
    buffer_address_minimum_z_a: buffer_address_extended_array;
    buffer_address_maximum_x_a: buffer_address_extended_array;
    buffer_address_maximum_y_a: buffer_address_extended_array;
    buffer_address_maximum_z_a: buffer_address_extended_array;
    buffer_address_b          : buffer_address_extended_array;
    address_a                 : address_extended_array;
    address_b                 : address_extended_array;
    start_record              : std_logic;
  end record;
  signal r, rin: reg_type;
begin
  comb: process(reset, d0, d1, r)
    variable v: reg_type;
  begin
    v := r;
```

```
-- Read data into the memories.
if d0.data_minimum_x_valid = '1' then
  v.write_enable_minimum_x := (others => '1');
  for i in 0 to parallelism - 1 loop
    v.buffer_address_minimum_x_a(i) :=
        r.buffer_address_minimum_x_a(i) + 1;
  end loop;
  v.address_a(0) := r.address_a(0) + 1;
else
  v.write_enable_minimum_x := (others => '0');
end if;
if d0.data_minimum_y_valid = '1' then
  v.write_enable_minimum_y := (others => '1');
  for i in 0 to parallelism - 1 loop
    v.buffer_address_minimum_y_a(i) :=
        r.buffer_address_minimum_y_a(i) + 1;
  end loop;
else
  v.write_enable_minimum_y := (others => '0');
end if;
if d0.data_minimum_z_valid = '1' then
  v.write_enable_minimum_z := (others => '1');
  for i in 0 to parallelism - 1 loop
    v.buffer_address_minimum_z_a(i) :=
        r.buffer_address_minimum_z_a(i) + 1;
  end loop;
else
  v.write_enable_minimum_z := (others => '0');
end if;
if d0.data_maximum_x_valid = '1' then
  v.write_enable_maximum_x := (others => '1');
  for i in 0 to parallelism - 1 loop
    v.buffer_address_maximum_x_a(i) :=
        r.buffer_address_maximum_x_a(i) + 1;
  end loop;
else
  v.write_enable_maximum_x := (others => '0');
end if;
if d0.data_maximum_y_valid = '1' then
  v.write_enable_maximum_y := (others => '1');
  for i in 0 to parallelism - 1 loop
    v.buffer_address_maximum_y_a(i) :=
        r.buffer_address_maximum_y_a(i) + 1;
  end loop;
```

```
    else
      v.write_enable_maximum_y := (others => '0');
    end if;
    if d0.data_maximum_z_valid = '1' then
      v.write_enable_maximum_z := (others => '1');
      for i in 0 to parallelism - 1 loop
        v.buffer_address_maximum_z_a(i) :=
            r.buffer_address_maximum_z_a(i) + 1;
      end loop;
    else
      v.write_enable_maximum_z := (others => '0');
    end if;

    -- Output data from the memories.
    case r.state is
      when initialisation_0 =>
        if d0.finished = '1' then
          v.state := initialisation_1;
        end if;
      when initialisation_1 =>
        if d0.finished = '0' then
          v.state := initialisation;
        end if;
      when initialisation =>
        if d0.finished = '1' then
          v.state        := read_data_first;
          v.last_address := r.address_a(0);
        end if;
      when read_data_first =>
        if d1.almost_full = '0' then
          v.state := read_data;
          v.valid := '1';
          for i in 0 to parallelism - 1 loop
            v.buffer_address_minimum_x_a(i) := conv_std_logic_vector(2 *
                i, buffer_address_width + 1);
            v.buffer_address_minimum_y_a(i) := conv_std_logic_vector(2 *
                i, buffer_address_width + 1);
            v.buffer_address_minimum_z_a(i) := conv_std_logic_vector(2 *
                i, buffer_address_width + 1);
            v.buffer_address_maximum_x_a(i) := conv_std_logic_vector(2 *
                i, buffer_address_width + 1);
            v.buffer_address_maximum_y_a(i) := conv_std_logic_vector(2 *
                i, buffer_address_width + 1);
```

```
        v.buffer_address_maximum_z_a(i) := conv_std_logic_vector(2 *
            i, buffer_address_width + 1);
        v.buffer_address_b(i)           := conv_std_logic_vector((2
            * i) + 1, buffer_address_width + 1);
        v.address_a(i)                  := conv_std_logic_vector(2 *
            i, address_width + 1);
        v.address_b(i)                  := conv_std_logic_vector((2
            * i) + 1, address_width + 1);
      end loop;
    else
      v.valid := '0';
    end if;
    v.start_record := '1';
  when read_data =>
    if d1.almost_full = '0' then
      if r.condition0 = '1' then
        if r.condition1 = '1' then
          v.state := waiting;
          v.valid := '0';
        else
          v.valid                          := '1';
          v.buffer_address_minimum_x_a(0) :=
              r.buffer_address_minimum_x_a(0) + 1;
          v.buffer_address_minimum_y_a(0) :=
              r.buffer_address_minimum_y_a(0) + 1;
          v.buffer_address_minimum_z_a(0) :=
              r.buffer_address_minimum_z_a(0) + 1;
          v.buffer_address_maximum_x_a(0) :=
              r.buffer_address_maximum_x_a(0) + 1;
          v.buffer_address_maximum_y_a(0) :=
              r.buffer_address_maximum_y_a(0) + 1;
          v.buffer_address_maximum_z_a(0) :=
              r.buffer_address_maximum_z_a(0) + 1;
          v.buffer_address_b(0)           :=
              r.buffer_address_minimum_x_a(0) + 2;
          v.address_a(0)                  := r.address_a(0) + 1;
          v.address_b(0)                  := r.address_a(0) + 2;
          for i in 1 to parallelism - 1 loop
            v.buffer_address_minimum_x_a(i) :=
                r.buffer_address_minimum_x_a(0) + 1 + (2 * i);
            v.buffer_address_minimum_y_a(i) :=
                r.buffer_address_minimum_y_a(0) + 1 + (2 * i);
            v.buffer_address_minimum_z_a(i) :=
                r.buffer_address_minimum_z_a(0) + 1 + (2 * i);
```

```
            v.buffer_address_maximum_x_a(i) :=
                r.buffer_address_maximum_x_a(0) + 1 + (2 * i);
            v.buffer_address_maximum_y_a(i) :=
                r.buffer_address_maximum_y_a(0) + 1 + (2 * i);
            v.buffer_address_maximum_z_a(i) :=
                r.buffer_address_maximum_z_a(0) + 1 + (2 * i);
            v.buffer_address_b(i)           :=
                r.buffer_address_minimum_x_a(0) + 2 + (2 * i);
            v.address_a(i)                  := r.address_a(0) + 1 +
                (2 * i);
            v.address_b(i)                  := r.address_a(0) + 2 +
                (2 * i);
          end loop;
        end if;
      else
        v.valid               := '1';
        v.buffer_address_b(0) := r.buffer_address_b(0) + (2 *
            parallelism) - 1;
        v.address_b(0)        := r.address_b(0) + (2 * parallelism)
            - 1;
        for i in 1 to parallelism - 1 loop
          v.buffer_address_minimum_x_a(i) :=
              r.buffer_address_minimum_x_a(i) + (2 * parallelism) - 1;
          v.buffer_address_minimum_y_a(i) :=
              r.buffer_address_minimum_y_a(i) + (2 * parallelism) - 1;
          v.buffer_address_minimum_z_a(i) :=
              r.buffer_address_minimum_z_a(i) + (2 * parallelism) - 1;
          v.buffer_address_maximum_x_a(i) :=
              r.buffer_address_maximum_x_a(i) + (2 * parallelism) - 1;
          v.buffer_address_maximum_y_a(i) :=
              r.buffer_address_maximum_y_a(i) + (2 * parallelism) - 1;
          v.buffer_address_maximum_z_a(i) :=
              r.buffer_address_maximum_z_a(i) + (2 * parallelism) - 1;
          v.buffer_address_b(i)           := r.buffer_address_b(i) +
              (2 * parallelism) - 1;
          v.address_a(i)                  := r.address_a(i) + (2 *
              parallelism) - 1;
          v.address_b(i)                  := r.address_b(i) + (2 *
              parallelism) - 1;
        end loop;
      end if;
    else
      v.valid := '0';
    end if;
```

```
      v.start_record := '0';
    when waiting =>
      if d0.finished = '0' then
        v.state                      := initialisation;
        v.finished                   := '0';
        v.buffer_address_minimum_x_a := (others => (others => '1'));
        v.buffer_address_minimum_y_a := (others => (others => '1'));
        v.buffer_address_minimum_z_a := (others => (others => '1'));
        v.buffer_address_maximum_x_a := (others => (others => '1'));
        v.buffer_address_maximum_y_a := (others => (others => '1'));
        v.buffer_address_maximum_z_a := (others => (others => '1'));
        v.address_a                  := (others => (others => '1'));
      else
        v.finished := '1';
      end if;
    when others =>
      v.state := initialisation_0;
  end case;

  v.last_address_decrement := r.last_address - 1;
  if v.address_b(parallelism - 1) >= r.last_address then
    v.condition0 := '1';
  else
    v.condition0 := '0';
  end if;
  if v.address_a(0) = r.last_address_decrement then
    v.condition1 := '1';
  else
    v.condition1 := '0';
  end if;

  -- Reset
  if reset = '1' then
    v.state                    := initialisation_0;
    v.condition0               := '0';
    v.condition1               := '0';
    v.last_address_decrement   := (others => '0');
    v.valid                    := '0';
    v.finished                 := '0';
    v.last_address             := (others => '0');
    v.write_enable_minimum_x   := (others => '0');
    v.write_enable_minimum_y   := (others => '0');
    v.write_enable_minimum_z   := (others => '0');
    v.write_enable_maximum_x   := (others => '0');
```

```
      v.write_enable_maximum_y     := (others => '0');
      v.write_enable_maximum_z     := (others => '0');
      v.buffer_address_minimum_x_a := (others => (others => '1'));
      v.buffer_address_minimum_y_a := (others => (others => '1'));
      v.buffer_address_minimum_z_a := (others => (others => '1'));
      v.buffer_address_maximum_x_a := (others => (others => '1'));
      v.buffer_address_maximum_y_a := (others => (others => '1'));
      v.buffer_address_maximum_z_a := (others => (others => '1'));
      v.buffer_address_b           := (others => (others => '0'));
      v.address_a                  := (others => (others => '1'));
      v.address_b                  := (others => (others => '0'));
      v.start_record               := '0';
    end if;

    rin                          <= v;
    q.valid                      <= r.valid;
    q.finished                   <= r.finished;
    q.last_address               <= r.last_address;
    q.write_enable_minimum_x     <= r.write_enable_minimum_x;
    q.write_enable_minimum_y     <= r.write_enable_minimum_y;
    q.write_enable_minimum_z     <= r.write_enable_minimum_z;
    q.write_enable_maximum_x     <= r.write_enable_maximum_x;
    q.write_enable_maximum_y     <= r.write_enable_maximum_y;
    q.write_enable_maximum_z     <= r.write_enable_maximum_z;
    q.buffer_address_minimum_x_a <= r.buffer_address_minimum_x_a;
    q.buffer_address_minimum_y_a <= r.buffer_address_minimum_y_a;
    q.buffer_address_minimum_z_a <= r.buffer_address_minimum_z_a;
    q.buffer_address_maximum_x_a <= r.buffer_address_maximum_x_a;
    q.buffer_address_maximum_y_a <= r.buffer_address_maximum_y_a;
    q.buffer_address_maximum_z_a <= r.buffer_address_maximum_z_a;
    q.buffer_address_b           <= r.buffer_address_b;
    q.address_a                  <= r.address_a;
    q.address_b                  <= r.address_b;
    q.start_record               <= r.start_record;
  end process;

  regs: process(reset, clk)
  begin
    if reset = '1' then
      r.state                      <= initialisation_0;
      r.condition0                 <= '0';
      r.condition1                 <= '0';
      r.last_address_decrement     <= (others => '0');
      r.valid                      <= '0';
```

```
      r.finished                   <= '0';
      r.last_address               <= (others => '0');
      r.write_enable_minimum_x     <= (others => '0');
      r.write_enable_minimum_y     <= (others => '0');
      r.write_enable_minimum_z     <= (others => '0');
      r.write_enable_maximum_x     <= (others => '0');
      r.write_enable_maximum_y     <= (others => '0');
      r.write_enable_maximum_z     <= (others => '0');
      r.buffer_address_minimum_x_a <= (others => (others => '1'));
      r.buffer_address_minimum_y_a <= (others => (others => '1'));
      r.buffer_address_minimum_z_a <= (others => (others => '1'));
      r.buffer_address_maximum_x_a <= (others => (others => '1'));
      r.buffer_address_maximum_y_a <= (others => (others => '1'));
      r.buffer_address_maximum_z_a <= (others => (others => '1'));
      r.buffer_address_b           <= (others => (others => '0'));
      r.address_a                  <= (others => (others => '1'));
      r.address_b                  <= (others => (others => '0'));
      r.start_record               <= '0';
    elsif rising_edge(clk) then
      r <= rin;
    end if;
  end process;
end architecture;
```

C.3 Many-Object Microarchitecture

The buffer control source code for the many-object microarchitecture roughly corresponds to Algorithm 3.3. In addition to the logic outlined in Algorithm 3.3, the VHDL source code is responsible for loading data into the internal memories. However, the control logic for the reorder operation, which was included in the earlier algorithm, is excluded from this source code. In the code, `parallelism` is equivalent to m.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

use work.functions.all;
use work.parameters.all;
use work.types.all;

entity buffer_control is
  port (
    clk   : in  std_logic;
```

```
    reset : in  std_logic;
    d0    : in  control_in_type;
    d1    : in  serialise_control_out_type;
    q     : out buffer_control_out_type);
end;

architecture rtl of buffer_control is
  type state_type is (initialisation_0, initialisation_1,
     initialisation, read_data_first, read_data, waiting);

  subtype parallelism_type is natural range 0 to parallelism - 1;
  type parallelism_type_array is array(0 to parallelism - 1) of
     parallelism_type;

  type reg_type is record
    -- State
    state                       : state_type;
    condition0                  : std_logic;
    condition1                  : std_logic;
    last_address_parallelism    : std_logic_vector(address_width downto
       0);
    last_address_decrement      : std_logic_vector(address_width downto
       0);
    write_enable_minimum_x_index: parallelism_type;
    write_enable_minimum_y_index: parallelism_type;
    write_enable_minimum_z_index: parallelism_type;
    write_enable_maximum_x_index: parallelism_type;
    write_enable_maximum_y_index: parallelism_type;
    write_enable_maximum_z_index: parallelism_type;
    last_updated                : parallelism_type;
    increment                   : parallelism_type_array;
    -- Outputs
    valid                       : std_logic;
    finished                    : std_logic;
    last_address                : std_logic_vector(address_width downto
       0);
    write_enable_minimum_x      : std_logic_vector(parallelism - 1
       downto 0);
    write_enable_minimum_y      : std_logic_vector(parallelism - 1
       downto 0);
    write_enable_minimum_z      : std_logic_vector(parallelism - 1
       downto 0);
    write_enable_maximum_x      : std_logic_vector(parallelism - 1
       downto 0);
```

```
    write_enable_maximum_y      : std_logic_vector(parallelism - 1
        downto 0);
    write_enable_maximum_z      : std_logic_vector(parallelism - 1
        downto 0);
    buffer_address_minimum_x_a  : buffer_address_extended_array;
    buffer_address_minimum_y_a  : buffer_address_extended_array;
    buffer_address_minimum_z_a  : buffer_address_extended_array;
    buffer_address_maximum_x_a  : buffer_address_extended_array;
    buffer_address_maximum_y_a  : buffer_address_extended_array;
    buffer_address_maximum_z_a  : buffer_address_extended_array;
    buffer_address_b            : buffer_address_extended_array;
    address_a                   : address_extended_array;
    address_b                   : address_extended_array;
    start_record                : std_logic;
  end record;
  signal r, rin: reg_type;
begin
  comb: process(reset, d0, d1, r)
    variable v: reg_type;
  begin
    v := r;

    -- Read data into the memories.
    v.write_enable_minimum_x(r.write_enable_minimum_x_index) := '0';
    v.write_enable_minimum_y(r.write_enable_minimum_y_index) := '0';
    v.write_enable_minimum_z(r.write_enable_minimum_z_index) := '0';
    v.write_enable_maximum_x(r.write_enable_maximum_x_index) := '0';
    v.write_enable_maximum_y(r.write_enable_maximum_y_index) := '0';
    v.write_enable_maximum_z(r.write_enable_maximum_z_index) := '0';
    if d0.data_minimum_x_valid = '1' then
      if r.write_enable_minimum_x_index = (parallelism - 1) then
        v.write_enable_minimum_x_index  := 0;
        v.buffer_address_minimum_x_a(0) :=
            r.buffer_address_minimum_x_a(0) + 1;
      else
        v.write_enable_minimum_x_index := r.write_enable_minimum_x_index
            + 1;
      end if;
      v.write_enable_minimum_x(v.write_enable_minimum_x_index) := '1';
    end if;
    if d0.data_minimum_y_valid = '1' then
      if r.write_enable_minimum_y_index = (parallelism - 1) then
        v.write_enable_minimum_y_index  := 0;
```

```
      v.buffer_address_minimum_y_a(0) :=
          r.buffer_address_minimum_y_a(0) + 1;
    else
      v.write_enable_minimum_y_index := r.write_enable_minimum_y_index
          + 1;
    end if;
    v.write_enable_minimum_y(v.write_enable_minimum_y_index) := '1';
  end if;
  if d0.data_minimum_z_valid = '1' then
    if r.write_enable_minimum_z_index = (parallelism - 1) then
      v.write_enable_minimum_z_index  := 0;
      v.buffer_address_minimum_z_a(0) :=
          r.buffer_address_minimum_z_a(0) + 1;
    else
      v.write_enable_minimum_z_index := r.write_enable_minimum_z_index
          + 1;
    end if;
    v.write_enable_minimum_z(v.write_enable_minimum_z_index) := '1';
  end if;
  if d0.data_maximum_x_valid = '1' then
    if r.write_enable_maximum_x_index = (parallelism - 1) then
      v.write_enable_maximum_x_index  := 0;
      v.buffer_address_maximum_x_a(0) :=
          r.buffer_address_maximum_x_a(0) + 1;
    else
      v.write_enable_maximum_x_index := r.write_enable_maximum_x_index
          + 1;
    end if;
    v.write_enable_maximum_x(v.write_enable_maximum_x_index) := '1';
  end if;
  if d0.data_maximum_y_valid = '1' then
    if r.write_enable_maximum_y_index = (parallelism - 1) then
      v.write_enable_maximum_y_index  := 0;
      v.buffer_address_maximum_y_a(0) :=
          r.buffer_address_maximum_y_a(0) + 1;
    else
      v.write_enable_maximum_y_index := r.write_enable_maximum_y_index
          + 1;
    end if;
    v.write_enable_maximum_y(v.write_enable_maximum_y_index) := '1';
  end if;
  if d0.data_maximum_z_valid = '1' then
    if r.write_enable_maximum_z_index = (parallelism - 1) then
      v.write_enable_maximum_z_index  := 0;
```

```
      v.buffer_address_maximum_z_a(0) :=
          r.buffer_address_maximum_z_a(0) + 1;
    else
      v.write_enable_maximum_z_index := r.write_enable_maximum_z_index
          + 1;
    end if;
    v.write_enable_maximum_z(v.write_enable_maximum_z_index) := '1';
  end if;

  -- Output data from the memories.
  case r.state is
    when initialisation_0 =>
      if d0.finished = '1' then
        v.state := initialisation_1;
      end if;
    when initialisation_1 =>
      if d0.finished = '0' then
        v.state := initialisation;
      end if;
    when initialisation =>
      if d0.finished = '1' then
        v.state        := read_data_first;
        v.last_address := r.address_a(r.write_enable_minimum_x_index);
      end if;
    when read_data_first =>
      if d1.almost_full = '0' then
        v.state := read_data;
        v.valid := '1';

        v.increment(0) := parallelism - 1;
        for i in 1 to parallelism - 1 loop
          v.increment(i) := i - 1;
        end loop;

        v.write_enable_minimum_x_index  := parallelism - 1;
        v.write_enable_minimum_y_index  := parallelism - 1;
        v.write_enable_minimum_z_index  := parallelism - 1;
        v.write_enable_maximum_x_index  := parallelism - 1;
        v.write_enable_maximum_y_index  := parallelism - 1;
        v.write_enable_maximum_z_index  := parallelism - 1;
        v.buffer_address_minimum_x_a(0) := (others => '0');
        v.buffer_address_minimum_y_a(0) := (others => '0');
        v.buffer_address_minimum_z_a(0) := (others => '0');
        v.buffer_address_maximum_x_a(0) := (others => '0');
```

```
      v.buffer_address_maximum_y_a(0) := (others => '0');
      v.buffer_address_maximum_z_a(0) := (others => '0');
      v.buffer_address_b(0)           := conv_std_logic_vector(1,
          buffer_address_width + 1);
      for i in 1 to parallelism - 1 loop
        v.buffer_address_b(i) := (others => '0');
      end loop;
    else
      v.valid := '0';
    end if;
    v.start_record := '1';
  when read_data =>
    if d1.almost_full = '0' then
      if r.condition0 = '1' then
        if r.condition1 = '1' then
          v.state := waiting;
        else
          v.valid        := '1';
          v.increment(0) := parallelism - 1;
          for i in 1 to parallelism - 1 loop
            v.increment(i) := i - 1;
          end loop;

          v.buffer_address_minimum_x_a(0) :=
              r.buffer_address_minimum_x_a(0) + 1;
          v.buffer_address_minimum_y_a(0) :=
              r.buffer_address_minimum_y_a(0) + 1;
          v.buffer_address_minimum_z_a(0) :=
              r.buffer_address_minimum_z_a(0) + 1;
          v.buffer_address_maximum_x_a(0) :=
              r.buffer_address_maximum_x_a(0) + 1;
          v.buffer_address_maximum_y_a(0) :=
              r.buffer_address_maximum_y_a(0) + 1;
          v.buffer_address_maximum_z_a(0) :=
              r.buffer_address_maximum_z_a(0) + 1;
          v.buffer_address_b(0)           :=
              r.buffer_address_minimum_x_a(0) + 2;
          for i in 1 to parallelism - 1 loop
            v.buffer_address_b(i) := r.buffer_address_minimum_x_a(0)
                + 1;
          end loop;
        end if;
      else
        v.valid := '1';
```

```
        for i in 0 to parallelism - 1 loop
          if r.increment(i) = 0 then
            v.increment(i)        := parallelism - 1;
            v.last_updated        := i;
            v.buffer_address_b(i) := r.buffer_address_b(i) + 1;
          else
            v.increment(i) := r.increment(i) - 1;
          end if;
        end loop;
      end if;
    else
      v.valid := '0';
    end if;
    v.start_record := '0';
  when waiting =>
    if d0.finished = '0' then
      v.state                          := initialisation;
      v.finished                       := '0';
      v.buffer_address_minimum_x_a(0) := (others => '1');
      v.buffer_address_minimum_y_a(0) := (others => '1');
      v.buffer_address_minimum_z_a(0) := (others => '1');
      v.buffer_address_maximum_x_a(0) := (others => '1');
      v.buffer_address_maximum_y_a(0) := (others => '1');
      v.buffer_address_maximum_z_a(0) := (others => '1');
    else
      v.valid    := '0';
      v.finished := '1';
    end if;
  when others =>
    v.state := initialisation_0;
end case;

v.last_address_parallelism := r.last_address + parallelism - 1;
v.last_address_decrement   := r.last_address - 2;
if r.address_b(r.last_updated) = r.last_address_parallelism then
  v.condition0 := '1';
else
  v.condition0 := '0';
end if;
if r.address_a(parallelism - 1) >= r.last_address_decrement then
  v.condition1 := '1';
else
  v.condition1 := '0';
end if;
```

```
-- Address converter
for i in 1 to parallelism - 1 loop
  v.buffer_address_minimum_x_a(i) := v.buffer_address_minimum_x_a(0);
  v.buffer_address_minimum_y_a(i) := v.buffer_address_minimum_y_a(0);
  v.buffer_address_minimum_z_a(i) := v.buffer_address_minimum_z_a(0);
  v.buffer_address_maximum_x_a(i) := v.buffer_address_maximum_x_a(0);
  v.buffer_address_maximum_y_a(i) := v.buffer_address_maximum_y_a(0);
  v.buffer_address_maximum_z_a(i) := v.buffer_address_maximum_z_a(0);
end loop;
for i in 0 to parallelism - 1 loop
  v.address_a(i) := multiply(v.buffer_address_minimum_x_a(0),
      parallelism, address_width + 1) + i;
  v.address_b(i) := multiply(v.buffer_address_b(i), parallelism,
      address_width + 1) + i;
end loop;

-- Reset
if reset = '1' then
  v.state                         := initialisation_0;
  v.condition0                    := '0';
  v.condition1                    := '0';
  v.last_address_parallelism      := (others => '0');
  v.last_address_decrement        := (others => '0');
  v.write_enable_minimum_x_index  := parallelism - 1;
  v.write_enable_minimum_y_index  := parallelism - 1;
  v.write_enable_minimum_z_index  := parallelism - 1;
  v.write_enable_maximum_x_index  := parallelism - 1;
  v.write_enable_maximum_y_index  := parallelism - 1;
  v.write_enable_maximum_z_index  := parallelism - 1;
  v.last_updated                  := 0;
  v.increment                     := (others => 0);
  v.valid                         := '0';
  v.finished                      := '0';
  v.last_address                  := (others => '0');
  v.write_enable_minimum_x        := (others => '0');
  v.write_enable_minimum_y        := (others => '0');
  v.write_enable_minimum_z        := (others => '0');
  v.write_enable_maximum_x        := (others => '0');
  v.write_enable_maximum_y        := (others => '0');
  v.write_enable_maximum_z        := (others => '0');
  v.buffer_address_minimum_x_a    := (others => (others => '1'));
  v.buffer_address_minimum_y_a    := (others => (others => '1'));
  v.buffer_address_minimum_z_a    := (others => (others => '1'));
```

```
      v.buffer_address_maximum_x_a   := (others => (others => '1'));
      v.buffer_address_maximum_y_a   := (others => (others => '1'));
      v.buffer_address_maximum_z_a   := (others => (others => '1'));
      v.buffer_address_b             := (others => (others => '0'));
      v.address_a                    := (others => (others => '0'));
      v.address_b                    := (others => (others => '0'));
      v.start_record                 := '0';
    end if;

    rin                          <= v;
    q.valid                      <= r.valid;
    q.finished                   <= r.finished;
    q.last_address               <= r.last_address;
    q.write_enable_minimum_x     <= r.write_enable_minimum_x;
    q.write_enable_minimum_y     <= r.write_enable_minimum_y;
    q.write_enable_minimum_z     <= r.write_enable_minimum_z;
    q.write_enable_maximum_x     <= r.write_enable_maximum_x;
    q.write_enable_maximum_y     <= r.write_enable_maximum_y;
    q.write_enable_maximum_z     <= r.write_enable_maximum_z;
    q.buffer_address_minimum_x_a <= r.buffer_address_minimum_x_a;
    q.buffer_address_minimum_y_a <= r.buffer_address_minimum_y_a;
    q.buffer_address_minimum_z_a <= r.buffer_address_minimum_z_a;
    q.buffer_address_maximum_x_a <= r.buffer_address_maximum_x_a;
    q.buffer_address_maximum_y_a <= r.buffer_address_maximum_y_a;
    q.buffer_address_maximum_z_a <= r.buffer_address_maximum_z_a;
    q.buffer_address_b           <= r.buffer_address_b;
    q.address_a                  <= r.address_a;
    q.address_b                  <= r.address_b;
    q.start_record               <= r.start_record;
  end process;

  regs: process(reset, clk)
  begin
    if reset = '1' then
      r.state                        <= initialisation_0;
      r.condition0                   <= '0';
      r.condition1                   <= '0';
      r.last_address_parallelism     <= (others => '0');
      r.last_address_decrement       <= (others => '0');
      r.write_enable_minimum_x_index <= parallelism - 1;
      r.write_enable_minimum_y_index <= parallelism - 1;
      r.write_enable_minimum_z_index <= parallelism - 1;
      r.write_enable_maximum_x_index <= parallelism - 1;
      r.write_enable_maximum_y_index <= parallelism - 1;
```

```
      r.write_enable_maximum_z_index <= parallelism - 1;
      r.last_updated                 <= 0;
      r.increment                    <= (others => 0);
      r.valid                        <= '0';
      r.finished                     <= '0';
      r.last_address                 <= (others => '0');
      r.write_enable_minimum_x       <= (others => '0');
      r.write_enable_minimum_y       <= (others => '0');
      r.write_enable_minimum_z       <= (others => '0');
      r.write_enable_maximum_x       <= (others => '0');
      r.write_enable_maximum_y       <= (others => '0');
      r.write_enable_maximum_z       <= (others => '0');
      r.buffer_address_minimum_x_a   <= (others => (others => '1'));
      r.buffer_address_minimum_y_a   <= (others => (others => '1'));
      r.buffer_address_minimum_z_a   <= (others => (others => '1'));
      r.buffer_address_maximum_x_a   <= (others => (others => '1'));
      r.buffer_address_maximum_y_a   <= (others => (others => '1'));
      r.buffer_address_maximum_z_a   <= (others => (others => '1'));
      r.buffer_address_b             <= (others => (others => '0'));
      r.address_a                    <= (others => (others => '0'));
      r.address_b                    <= (others => (others => '0'));
      r.start_record                 <= '0';
    elsif rising_edge(clk) then
      r <= rin;
    end if;
  end process;
end architecture;
```

Appendix D

Supplemental Results

Since time is the one immaterial object which we cannot influence – neither speed up nor slow down, add to nor diminish – it is an imponderably valuable gift.

—Maya Angelou, poet (1928–)

Results are one of the most significant outcomes of any research endeavour, but there are often vast quantities of results generated throughout a research project. This appendix provides supplemental results for the hybrid systems, to permit a greater insight into their behaviour. It commences with results for the benchmark used for the different experiments. This is followed by myriad results for the Xilinx Virtex-II and Xilinx Virtex-5 implementations of the area-efficient microarchitecture and for the Xilinx Virtex-5 implementation of the many-object microarchitecture. The appendix concludes with results corresponding to spatial partitioning used with the Xilinx Virtex-5 implementations of the area-efficient and many-object microarchitectures.

D.1 Standard Benchmark

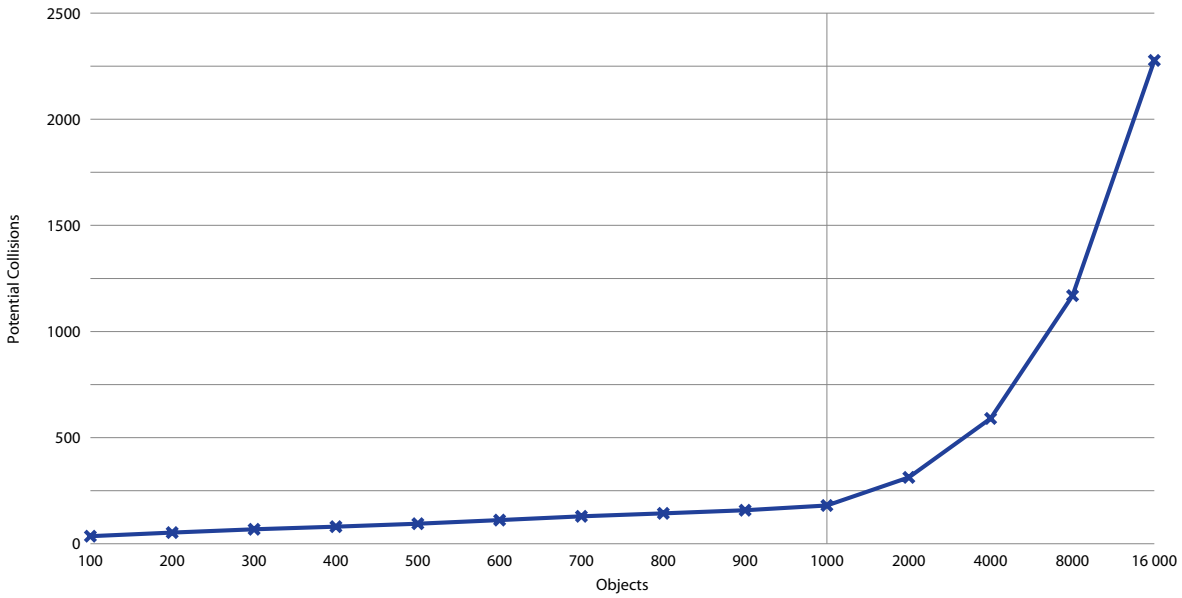


Figure D.1: *Potential collisions for the benchmarks of Sections 5.2, 5.3 and 5.4.*

D.2 Area-Efficient Microarchitecture

D.2.1 Xilinx Virtex-II

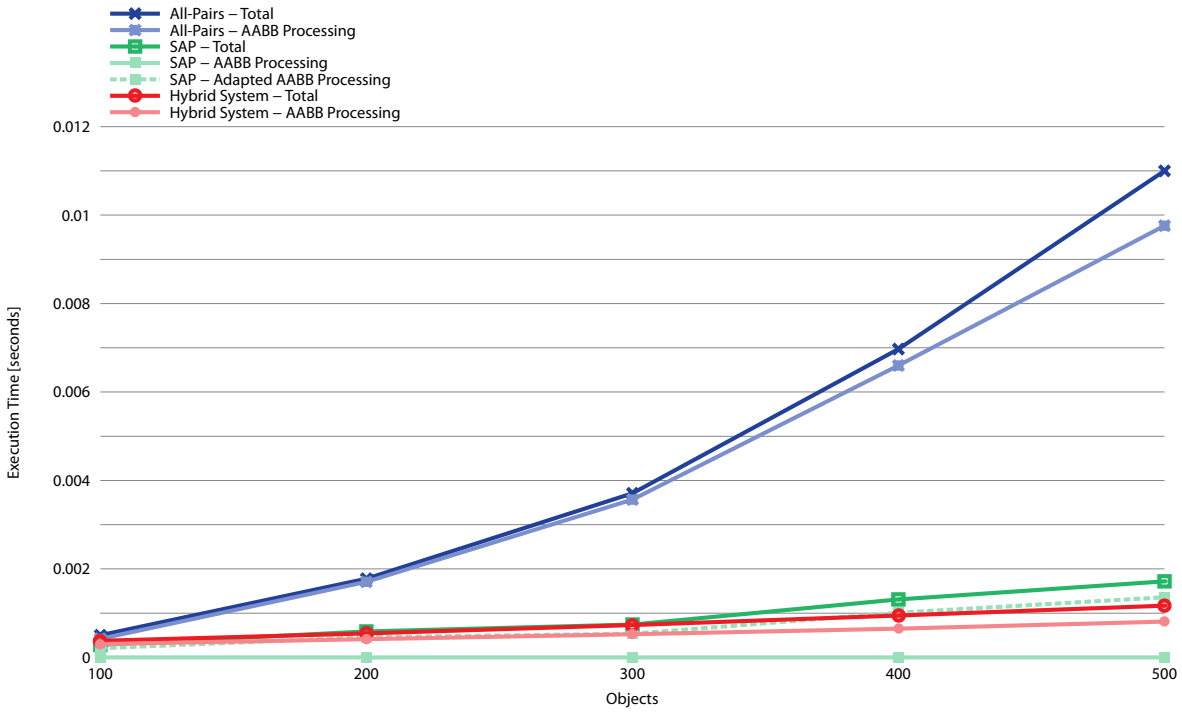


Figure D.2: *Execution times for the 'Software-Based Execution Time Experiment' of Section 5.2.1. The equivalent logarithmic-scale graph is Figure 5.8.*

	All-Pairs		SAP	
Objects	Total	AABB Processing	Total	Adapted AABB Processing
100	1.34×	1.39×	0.752×	0.69×
200	3.27×	4.15×	1.08×	1.11×
300	5.08×	6.8×	1.01×	1.02×
400	7.36×	10.2×	1.38×	1.56×
500	9.4×	12.1×	1.47×	1.68×

Table D.1: *Accelerations for the 'Software-Based Execution Time Experiment' of Section 5.2.1. The data are graphed in Figure 5.8.*

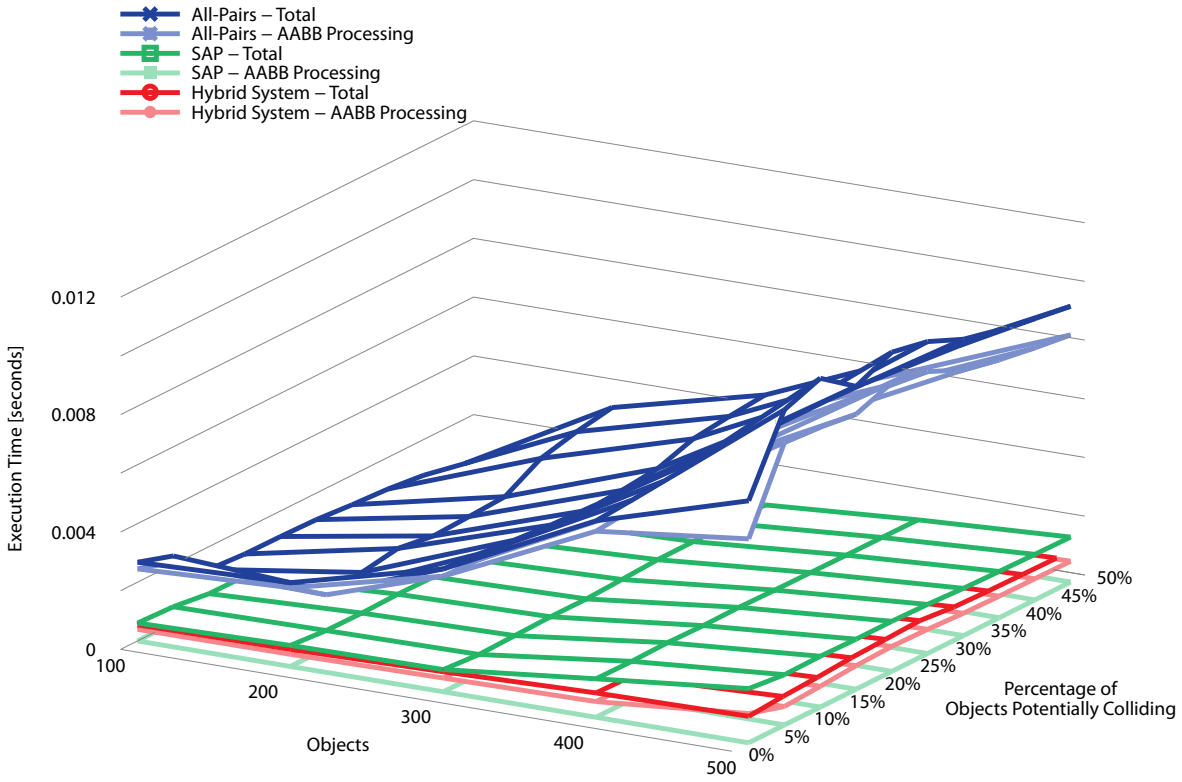


Figure D.3: *Results for the 'Collisions Experiment' of Section 5.2.1. The equivalent logarithmic-scale graph is Figure 5.9.*

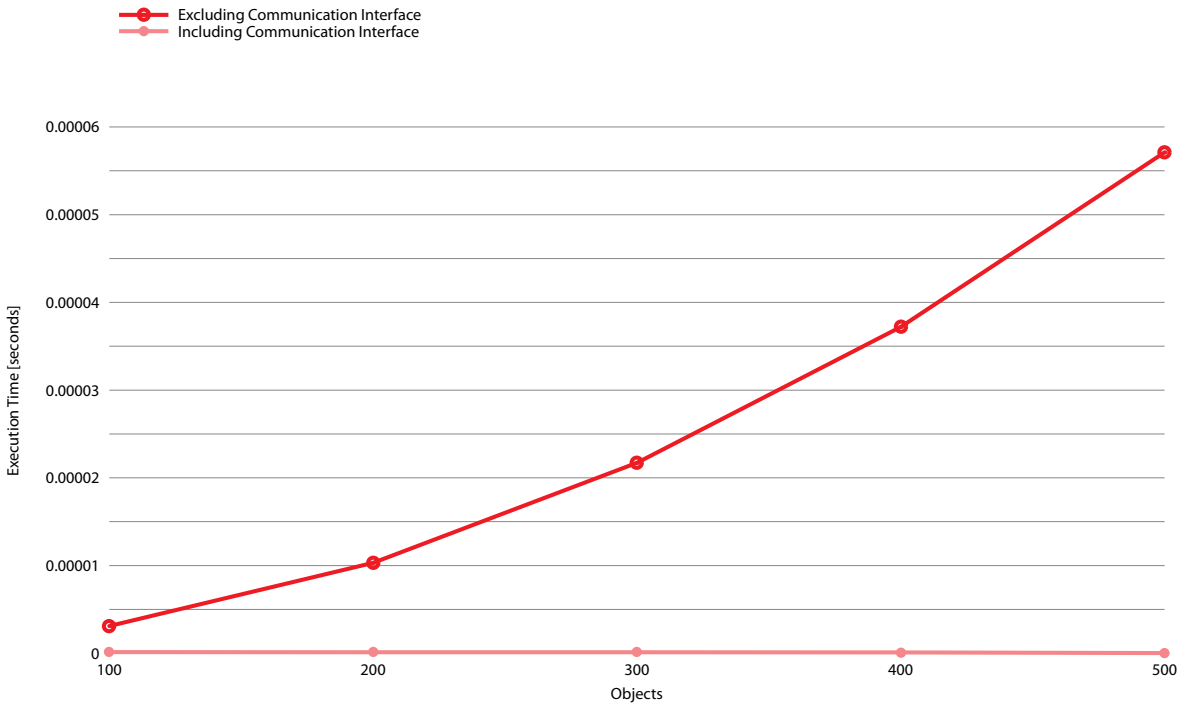


Figure D.4: *Execution times for the 'Hardware-Based Execution Time Experiment' of Section 5.2.1.*

	Excluding Communication Interface		Including Communication Interface	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
100	134.583×	66.647×	66.987×	33.173×
200	166.278×	44.343×	106.869×	28.5×
300	164.896×	24.711×	119.398×	17.893×
400	177.269×	27.127×	137.214×	20.998×
500	171.025×	23.831×	138.44×	19.291×

Table D.2: *Accelerations for the 'Hardware-Based Execution Time Experiment' of Section 5.2.1. The data are graphed in Figure 5.10.*

	Excluding L2 Cache		Including L2 Cache	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
100	3161.345×	1565.546×	595.253×	294.779×
200	3905.863×	1041.624×	1094.141×	291.788×
300	3873.418×	580.470×	1364.331×	204.459×
400	4164.038×	637.224×	1720.991×	263.364×
500	4017.379×	559.799×	1859.244×	259.075×

Table D.3: *Accelerations for the 'CPU-Based Execution Time Experiment' of Section 5.2.1. The data are graphed in Figure 5.11.*

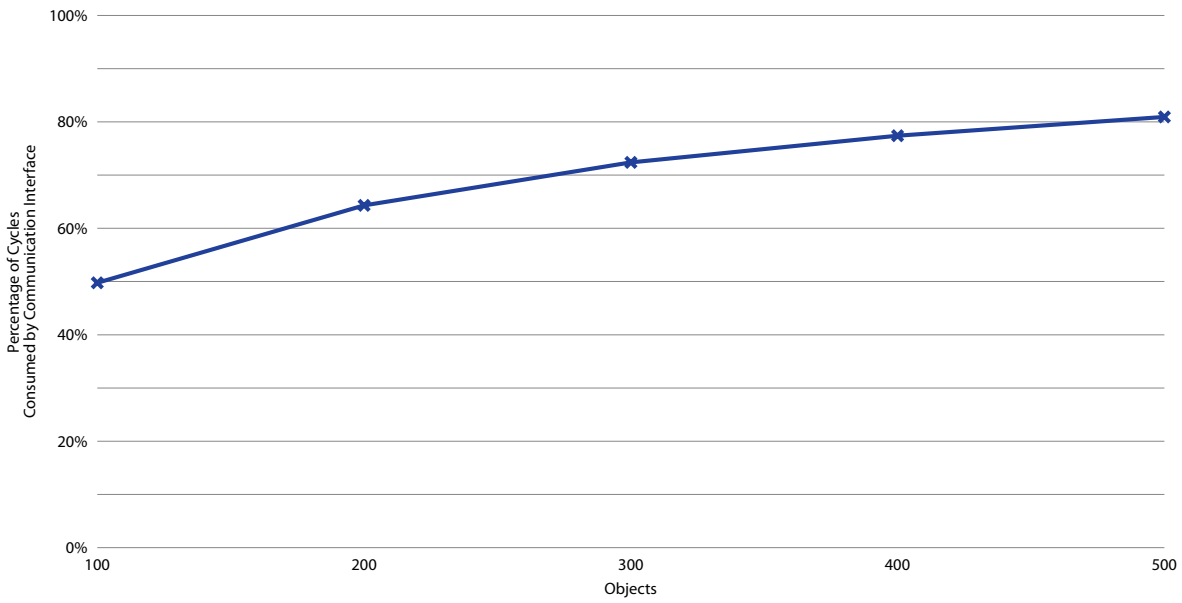


Figure D.5: *The percentage of clock cycles consumed by the communication interface relative to those consumed by the microarchitecture-based algorithm corresponding to Section 5.2.1.*

D.2.2 Xilinx Virtex-5

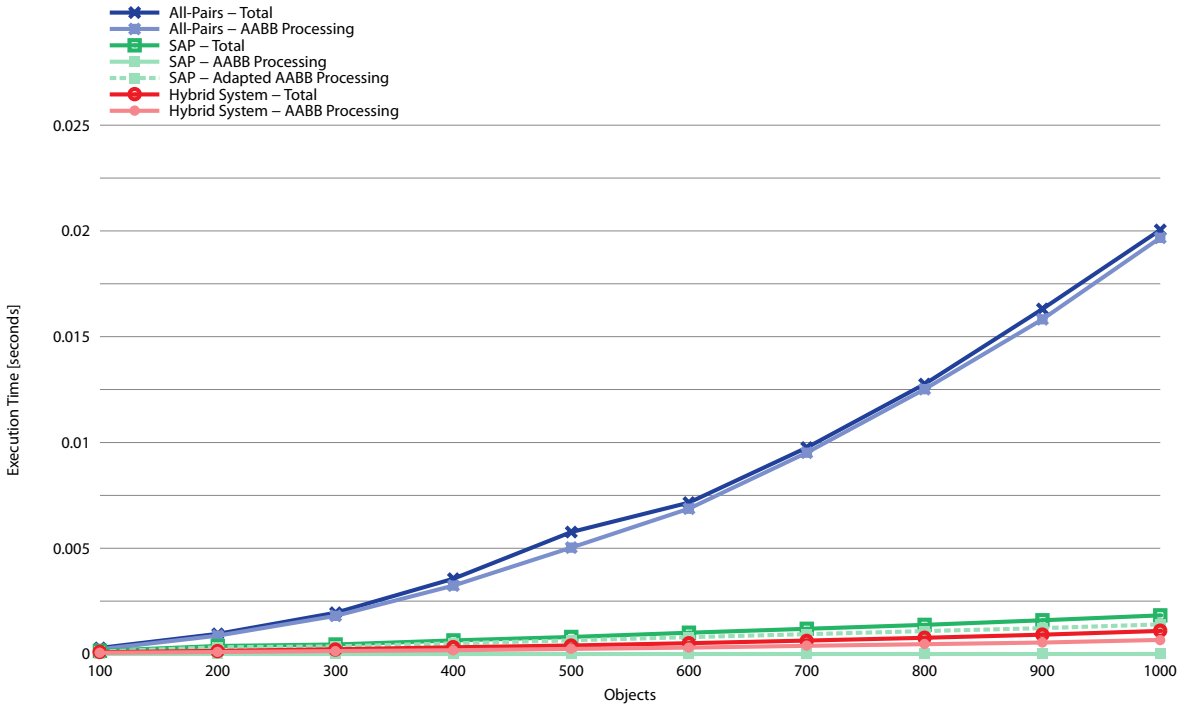


Figure D.6: *Execution times for the 'Software-Based Execution Time Experiment' of Section 5.2.2. The equivalent logarithmic-scale graph is Figure 5.12.*

	All-Pairs		SAP	
Objects	Total	AABB Processing	Total	Adapted AABB Processing
100	3.38×	3.9×	1.68×	2.01×
200	6.53×	9.56×	2.57×	3.54×
300	8.6×	13.2×	1.97×	2.61×
400	11.3×	17.8×	2.03×	2.79×
500	14.3×	20.8×	1.99×	2.66×
600	13.9×	22.5×	1.95×	2.61×
700	15.4×	24.7×	1.87×	2.44×
800	16.7×	27.2×	1.81×	2.34×
900	17.9×	29.4×	1.75×	2.27×
1000	18.4×	29.5×	1.68×	2.11×

Table D.4: *Accelerations for the 'Software-Based Execution Time Experiment' of Section 5.2.2. The data are graphed in Figure 5.12.*

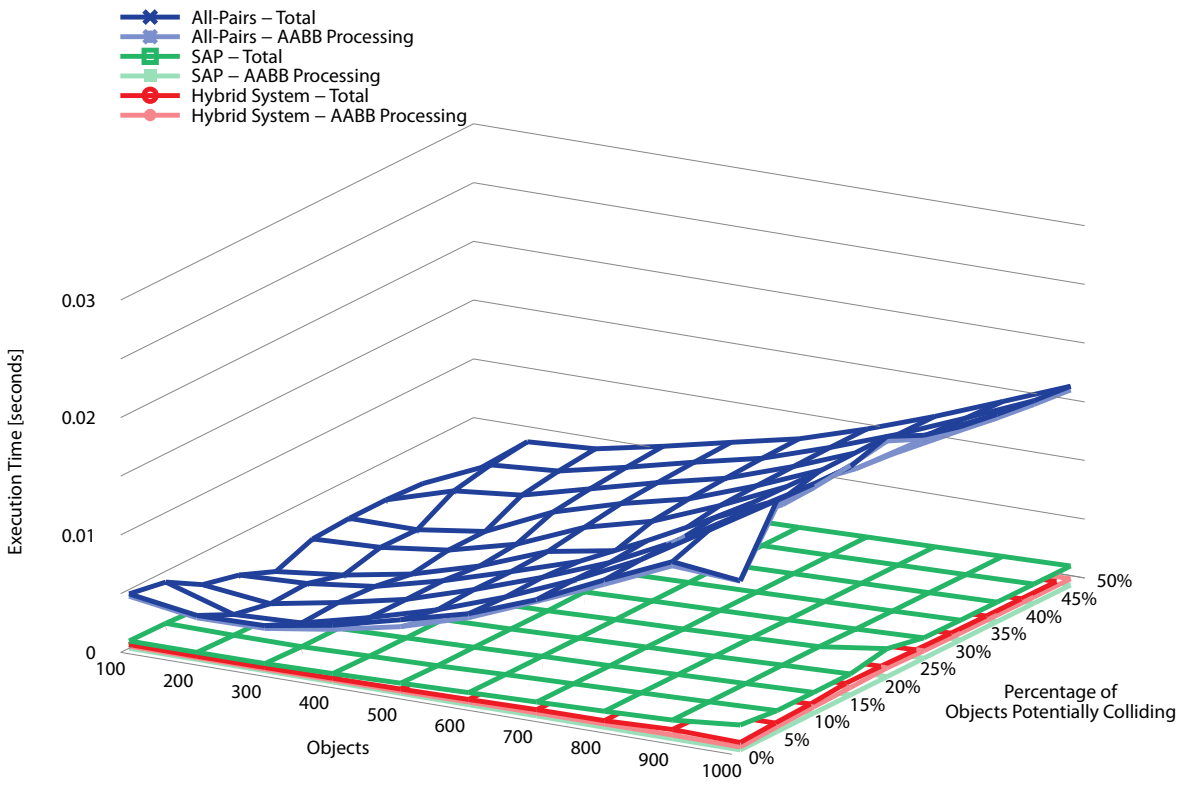


Figure D.7: *Results for the 'Collisions Experiment' of Section 5.2.2. The equivalent logarithmic-scale graph is Figure 5.13.*

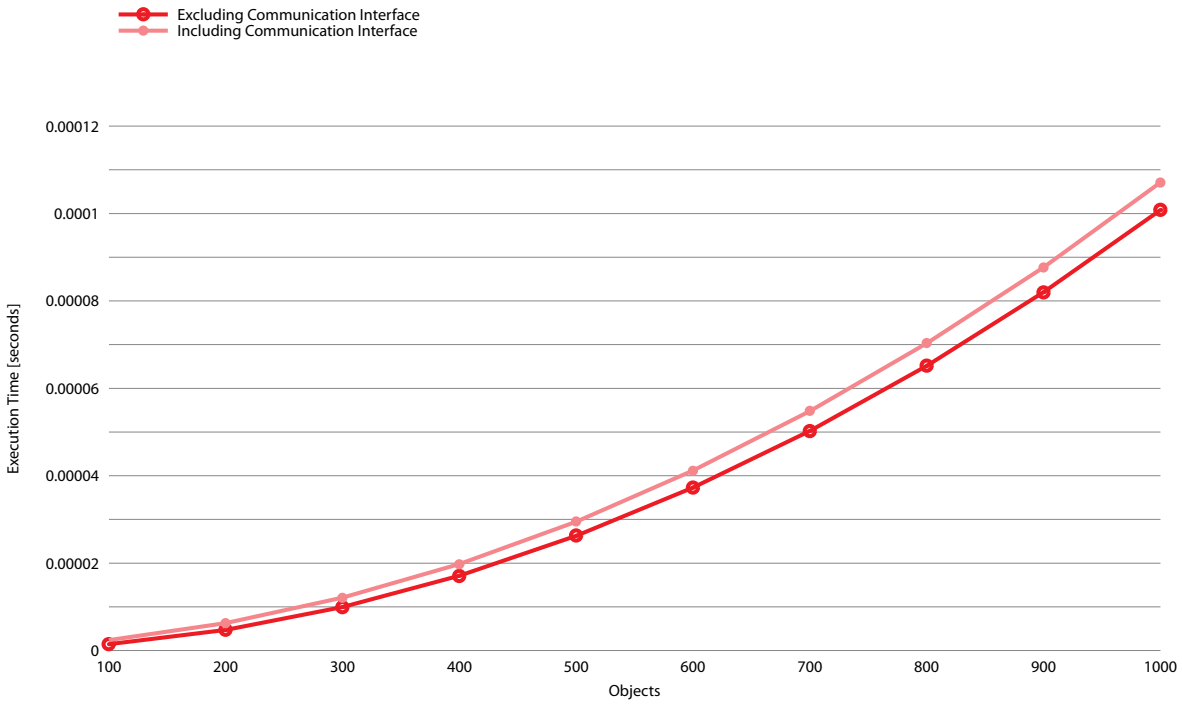


Figure D.8: *Execution times for the 'Hardware-Based Execution Time Experiment' of Section 5.2.2.*

	Including Communication Interface		Excluding Communication Interface	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
100	156.856×	80.893×	94.509×	48.739×
200	182.332×	67.596×	137.665×	51.037×
300	180.943×	35.743×	148.941×	29.421×
400	189.057×	29.549×	163.506×	25.556×
500	191.54×	24.463×	170.399×	21.763×
600	184.285×	21.319×	166.796×	19.296×
700	189.339×	18.642×	173.660×	17.099×
800	192.185×	16.575×	178.134×	15.363×
900	192.952×	14.874×	180.455×	13.911×
1000	195.212×	13.887×	183.715×	13.069×

Table D.5: *Accelerations for the 'Hardware-Based Execution Time Experiment' of Section 5.2.2. The data are graphed in Figure 5.14.*

	Excluding L2 Cache		Including L2 Cache	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
100	1882.268×	970.711×	279.939×	144.368×
200	2187.985×	811.153×	494.789×	183.434×
300	2171.313×	428.916×	629.256×	124.302×
400	2268.683×	354.59×	784.446×	122.607×
500	2298.482×	293.553×	902.312×	115.24×
600	2211.424×	255.833×	960.112×	111.072×
700	2272.069×	223.708×	1066.58×	105.015×
800	2306.225×	198.895×	1156.307×	99.723×
900	2315.421×	178.493×	1225.01×	94.435×
1000	2342.543×	166.647×	1298.904×	92.403×

Table D.6: *Accelerations for the 'CPU-Based Execution Time Experiment' of Section 5.2.2. The data are graphed in Figure 5.15.*

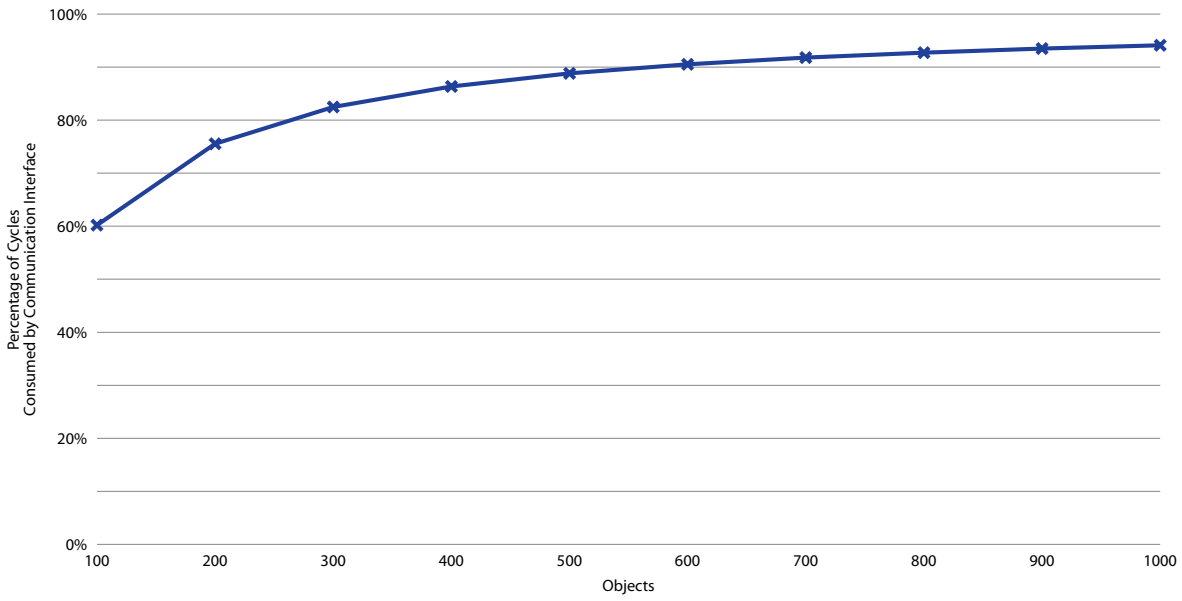


Figure D.9: *The percentage of clock cycles consumed by the communication interface relative to those consumed by the microarchitecture-based algorithm corresponding to Section 5.2.2.*

D.3 Many-Object Microarchitecture

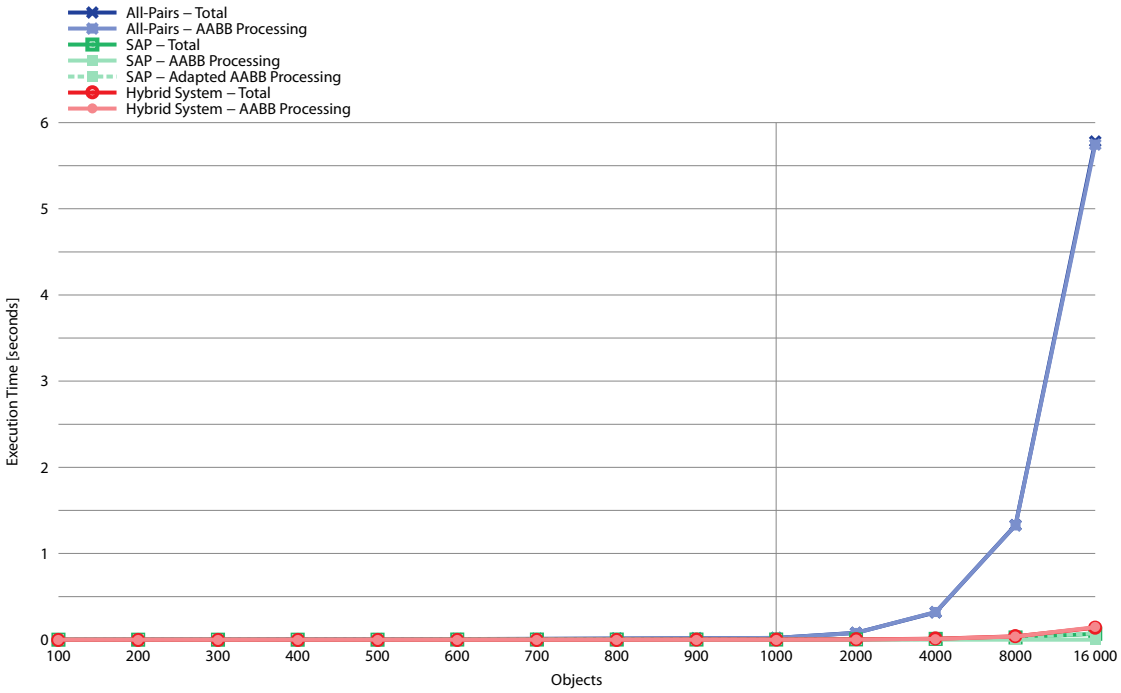


Figure D.10: *Execution times for the 'Software-Based Execution Time Experiment' of Section 5.3. The equivalent logarithmic-scale graph is Figure 5.16.*

	All-Pairs		SAP	
Objects	Total	AABB Processing	Total	Adapted AABB Processing
100	3.19×	3.71×	1.59×	1.88×
200	6.08×	8.38×	2.4×	3.12×
300	8.04×	11.6×	1.84×	2.3×
400	10.6×	15.1×	1.9×	2.41×
500	13.1×	18.1×	1.83×	2.31×
600	12.9×	19.4×	1.81×	2.27×
700	13.9×	20.7×	1.69×	2.06×
800	14.9×	23×	1.61×	1.96×
900	16.3×	24.3×	1.6×	1.92×
1000	16.5×	24.7×	1.5×	1.76×
2000	23.1×	31.3×	1.21×	1.3×
4000	28.4×	34.7×	0.909×	0.89×
8000	34×	35.2×	0.663×	0.65×
16 000	41.5×	39.9×	0.497×	0.513×

Table D.7: *Accelerations for the 'Software-Based Execution Time Experiment' of Section 5.3. The data are graphed in Figure 5.16.*

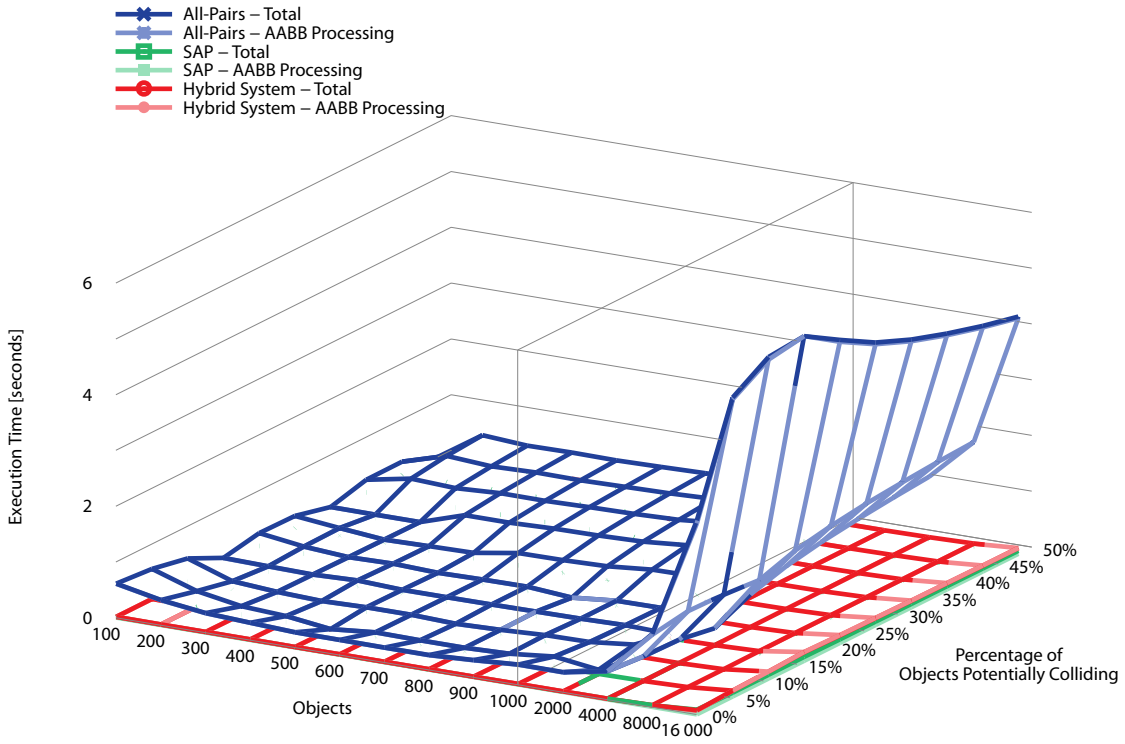


Figure D.11: *Results for the 'Collisions Experiment' of Section 5.3. The equivalent logarithmic-scale graph is Figure 5.17.*

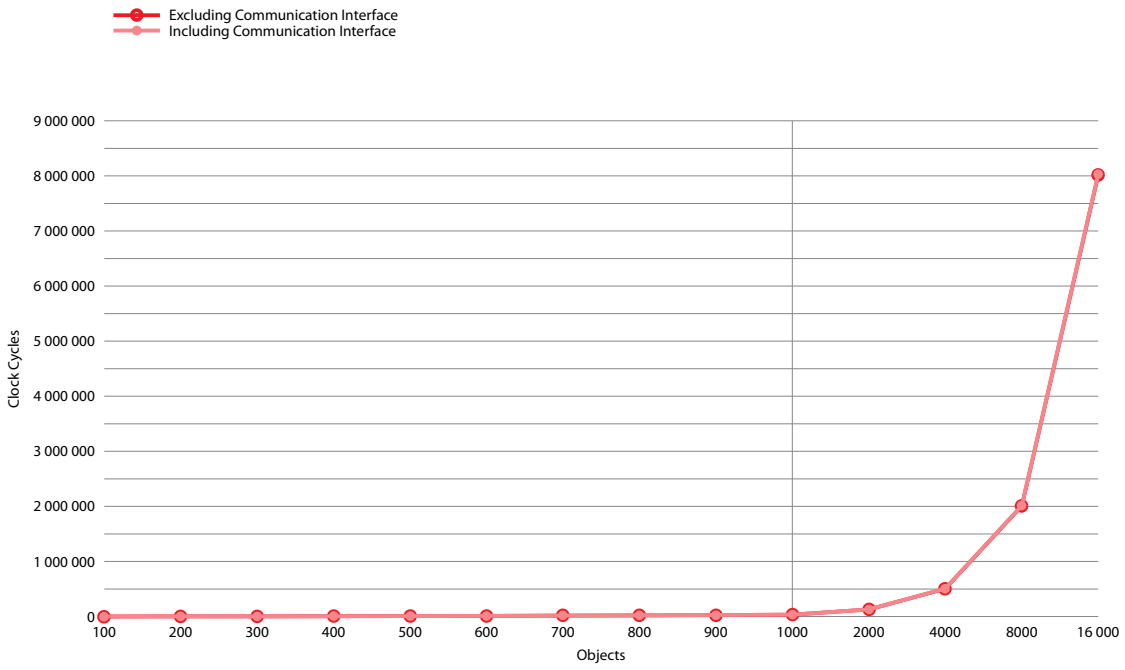


Figure D.12: *Clock cycles for the 'Hardware-Based Execution Time Experiment' of Section 5.3. The equivalent logarithmic-scale graph is Figure 5.18.*

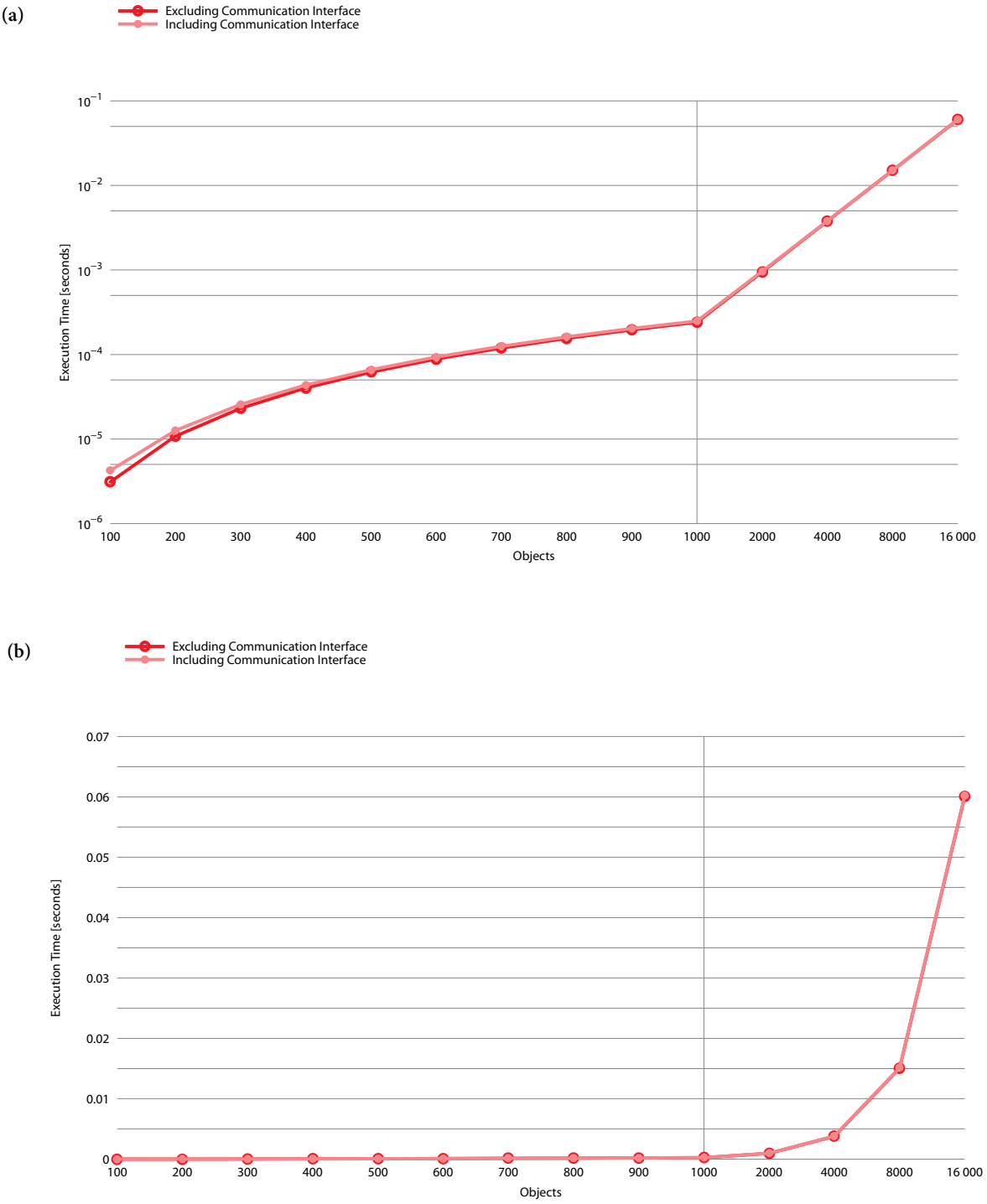


Figure D.13: *Execution times for the 'Hardware-Based Execution Time Experiment' of Section 5.3.* **(a)** *Logarithmic scale.* **(b)** *Linear scale.*

	Excluding Communication Interface		Including Communication Interface	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
100	72.558×	36.774×	53.175×	26.95×
200	79.922×	29.815×	68.505×	25.556×
300	78.017×	15.541×	70.124×	13.969×
400	80.533×	12.91×	74.253×	11.904×
500	81.077×	10.339×	76.163×	9.712×
600	77.738×	9.106×	73.734×	8.637×
700	79.628×	7.89×	76.081×	7.539×
800	80.611×	6.888×	77.46×	6.619×
900	80.807×	6.383×	77.992×	6.16×
1000	81.649×	5.85×	79.073×	5.665×
2000	81.408×	3.376×	80.117×	3.323×
4000	84.899×	2.177×	84.223×	2.159×
8000	88.09×	1.628×	87.739×	1.621×
16 000	95.604×	1.23×	95.413×	1.227×

Table D.8: *Accelerations for the 'Hardware-Based Execution Time Experiment' of Section 5.3. The data are graphed in Figure 5.18.*

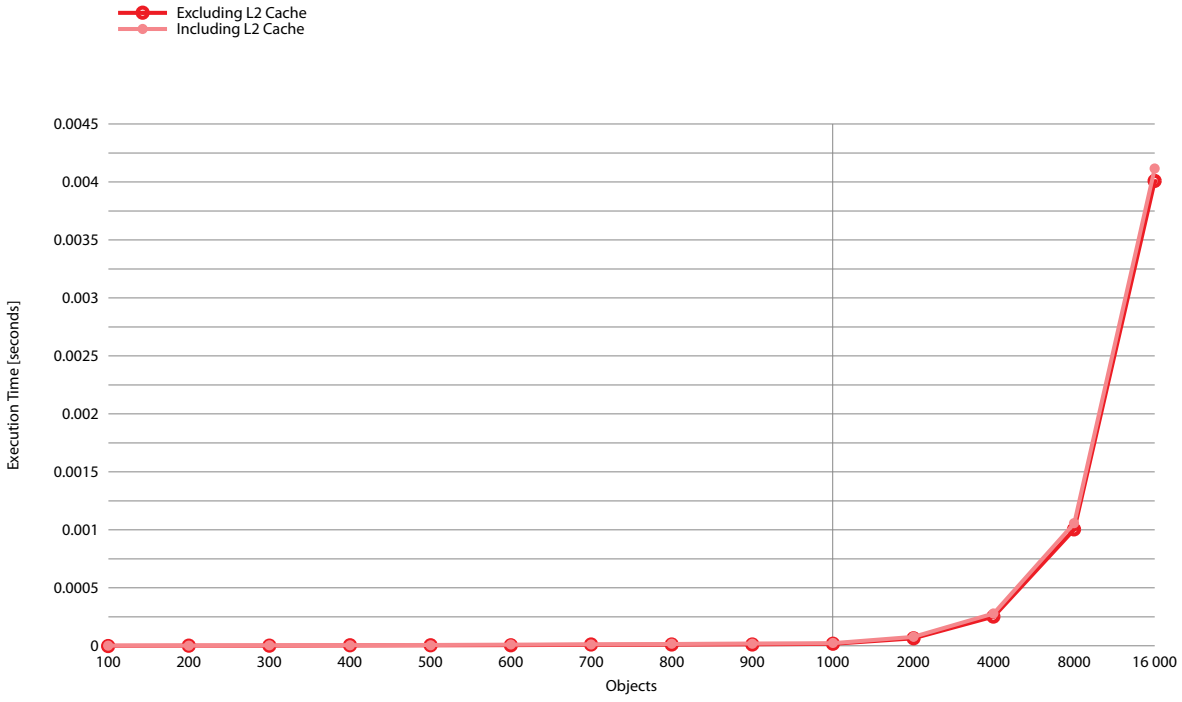


Figure D.14: *Execution times for the 'CPU-Based Execution Time Experiment' of Section 5.3. The equivalent logarithmic-scale graph is Figure 5.19.*

	Excluding L2 Cache		Including L2 Cache	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
100	1089.254×	552.058×	252.59×	128.018×
200	1201.336×	448.156×	417.288×	155.668×
300	1168.357×	232.739×	503.898×	100.378×
400	1207.313×	193.548×	601.581×	96.442×
500	1216.696×	155.149×	668.855×	85.29×
600	1166.03×	136.591×	691.105×	80.957×
700	1194.422×	118.356×	749.247×	74.243×
800	1209.173×	103.317×	794.77×	67.908×
900	1212.109×	95.738×	826.828×	65.307×
1000	1224.738×	87.749×	862.445×	61.792×
2000	1221.124×	50.646×	1006.701×	41.753×
4000	1273.488×	32.651×	1150.14×	29.488×
8000	1321.357×	24.414×	1253.884×	23.168×
16 000	1434.056×	18.443×	1396.418×	17.959×

Table D.9: *Accelerations for the 'CPU-Based Execution Time Experiment' of Section 5.3. The data are graphed in Figure 5.19.*

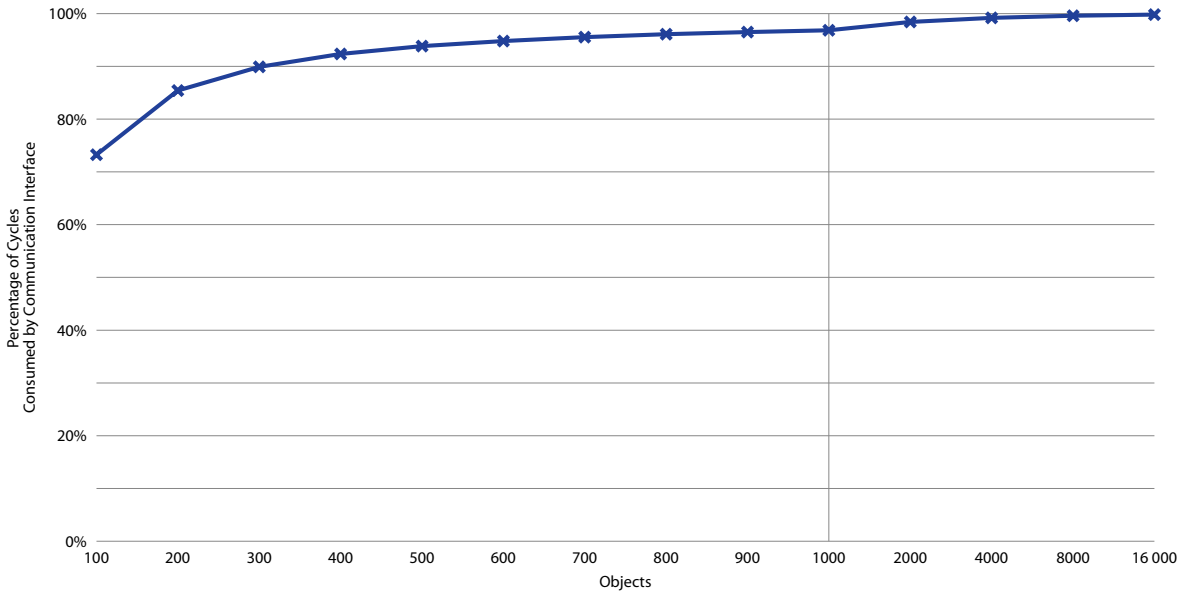


Figure D.15: *The percentage of clock cycles consumed by the communication interface relative to those consumed by the microarchitecture-based algorithm corresponding to Section 5.3.*

D.4 Spatial Partitioning

D.4.1 Area-Efficient Microarchitecture, Xilinx Virtex-5

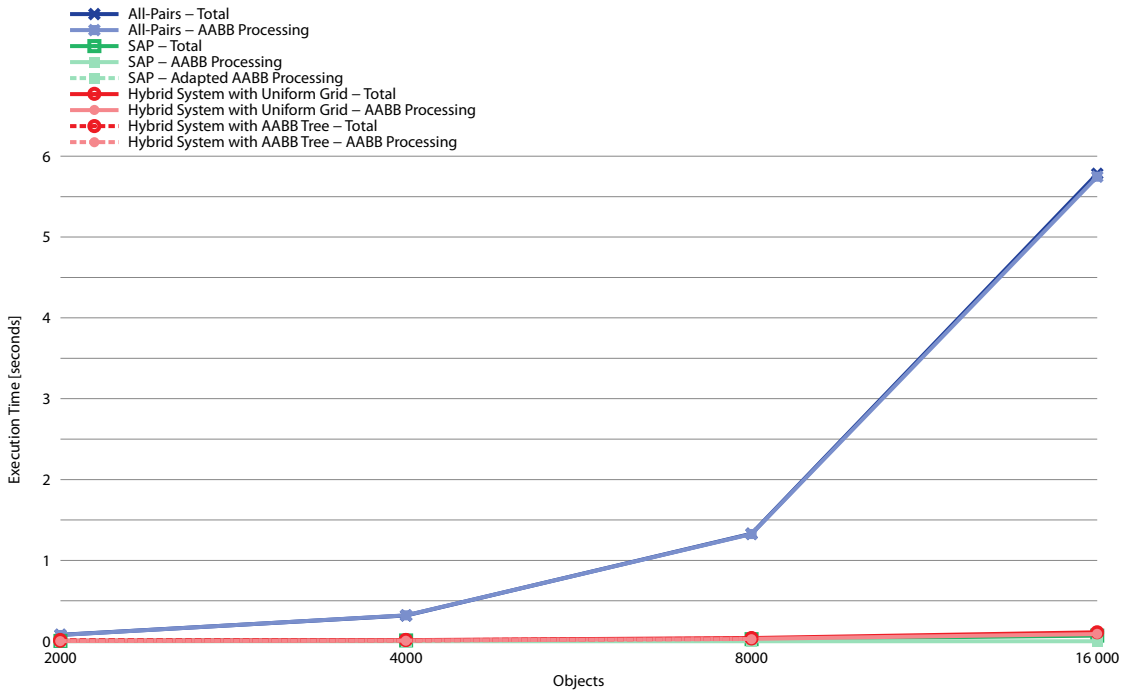


Figure D.16: *Execution times for the 'Software-Based Execution Time Experiment' of Section 5.4.2. The equivalent logarithmic-scale graph is Figure 5.20.*

	All-Pairs		SAP	
Objects	Total	AABB Processing	Total	Adapted AABB Processing
		Uniform Grid		
2000	13.125×	20.443×	0.69×	0.5×
4000	22.076×	35.709×	0.708×	1.406×
8000	33.43×	43.592×	0.653×	0.724×
16 000	0.182×	0.222×	0.017×	0.016×
		AABB Tree		
2000	7.036×	27.157×	0.37×	0.664×
4000	28.313×	36.639×	0.907×	1.442×
8000	39.516×	47.081×	0.771×	0.782×

Table D.10: *Accelerations for the 'Software-Based Execution Time Experiment' of Section 5.4.2. The data are graphed in Figure 5.20.*

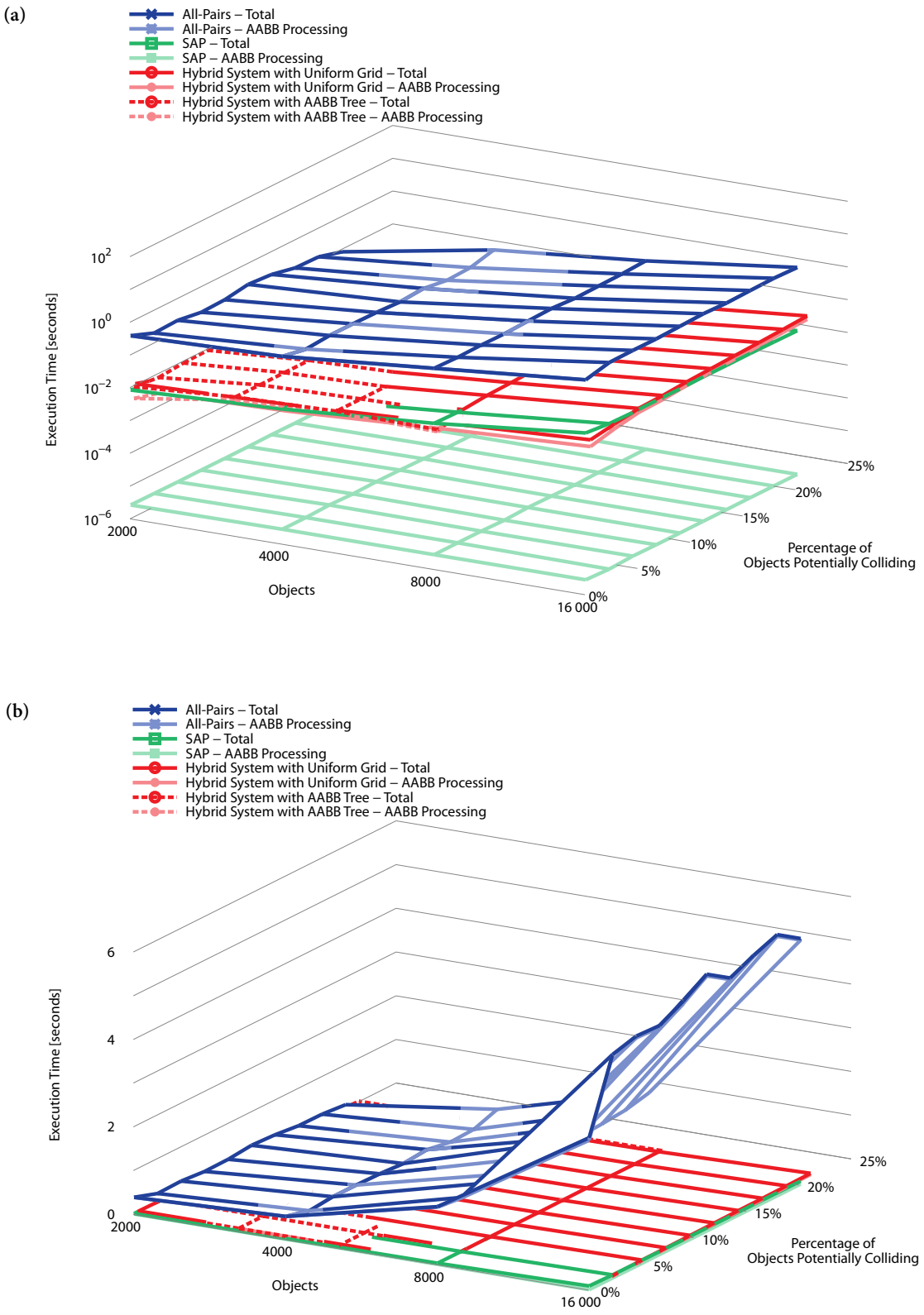


Figure D.17: *Results for the 'Collisions Experiment' of Section 5.4.2.* **(a)** *Logarithmic scale.* **(b)** *Linear scale.*

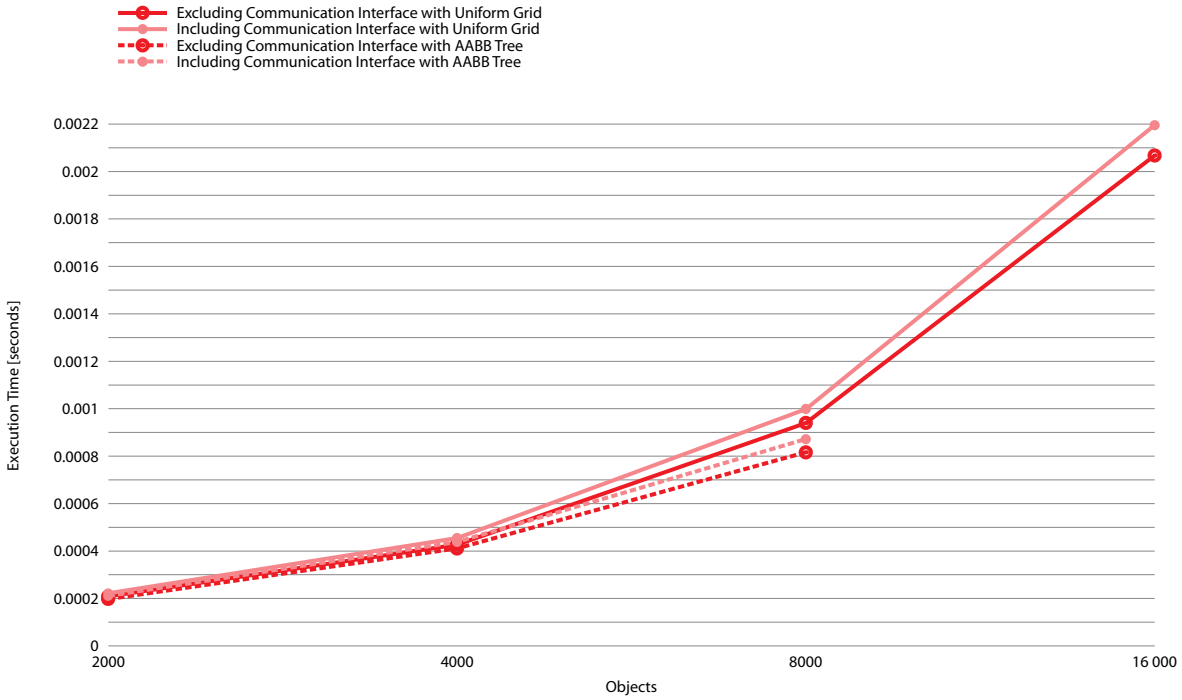


Figure D.18: *Execution times for the 'Hardware-Based Execution Time Experiment' of Section 5.4.2.*

	Excluding Communication Interface		Including Communication Interface	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
		Uniform Grid		
2000	32.718×	0.8×	32.533×	0.8×
4000	60.704×	2.39×	60.387×	2.39×
8000	113.119×	1.88×	112.556×	1.88×
16 000	0.792×	0.058×	0.788×	0.058×
		AABB Tree		
2000	182.363×	4.46×	176.721×	4.46×
4000	265.278×	10.443×	259.426×	10.443×
8000	366.091×	6.084×	360.553×	6.084×

Table D.11: *Accelerations for the 'Hardware-Based Execution Time Experiment' of Section 5.4.2. The data are graphed in Figure 5.24.*

	Excluding L2 Cache		Including L2 Cache	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
		Uniform Grid		
2000	35.599×	0.871×	35.326×	0.863×
4000	65.563×	2.58×	65.263×	2.565×
8000	122.087×	2.03×	121.289×	2.012×
16 000	0.857×	0.063×	0.853×	0.063×
		AABB Tree		
2000	318.051×	7.78×	301.347×	7.359×
4000	385.531×	15.2×	373.416×	14.675×
8000	461.439×	7.67×	452.248×	7.503×

Table D.12: *Accelerations for the 'CPU-Based Execution Time Experiment' of Section 5.4.2. The data are graphed in Figure 5.25.*

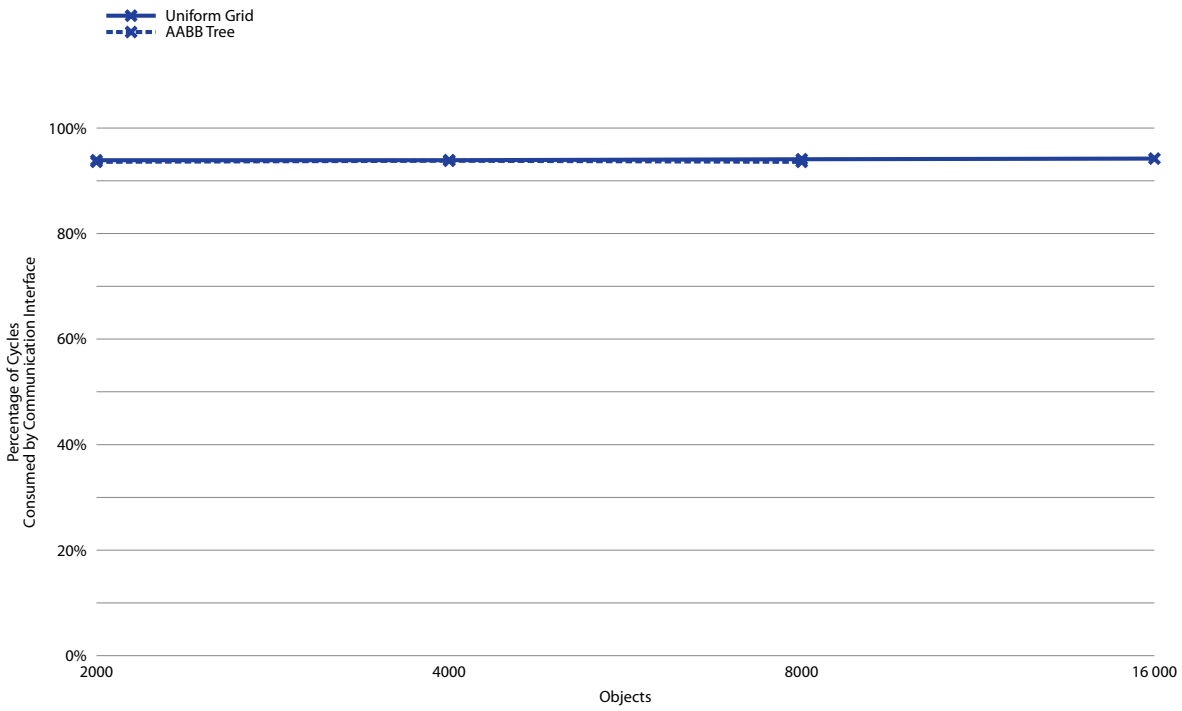


Figure D.19: *The percentage of clock cycles consumed by the communication interface relative to those consumed by the microarchitecture-based algorithm corresponding to Section 5.4.2.*

D.4.2 Many-Object Microarchitecture

Cell Size 1024

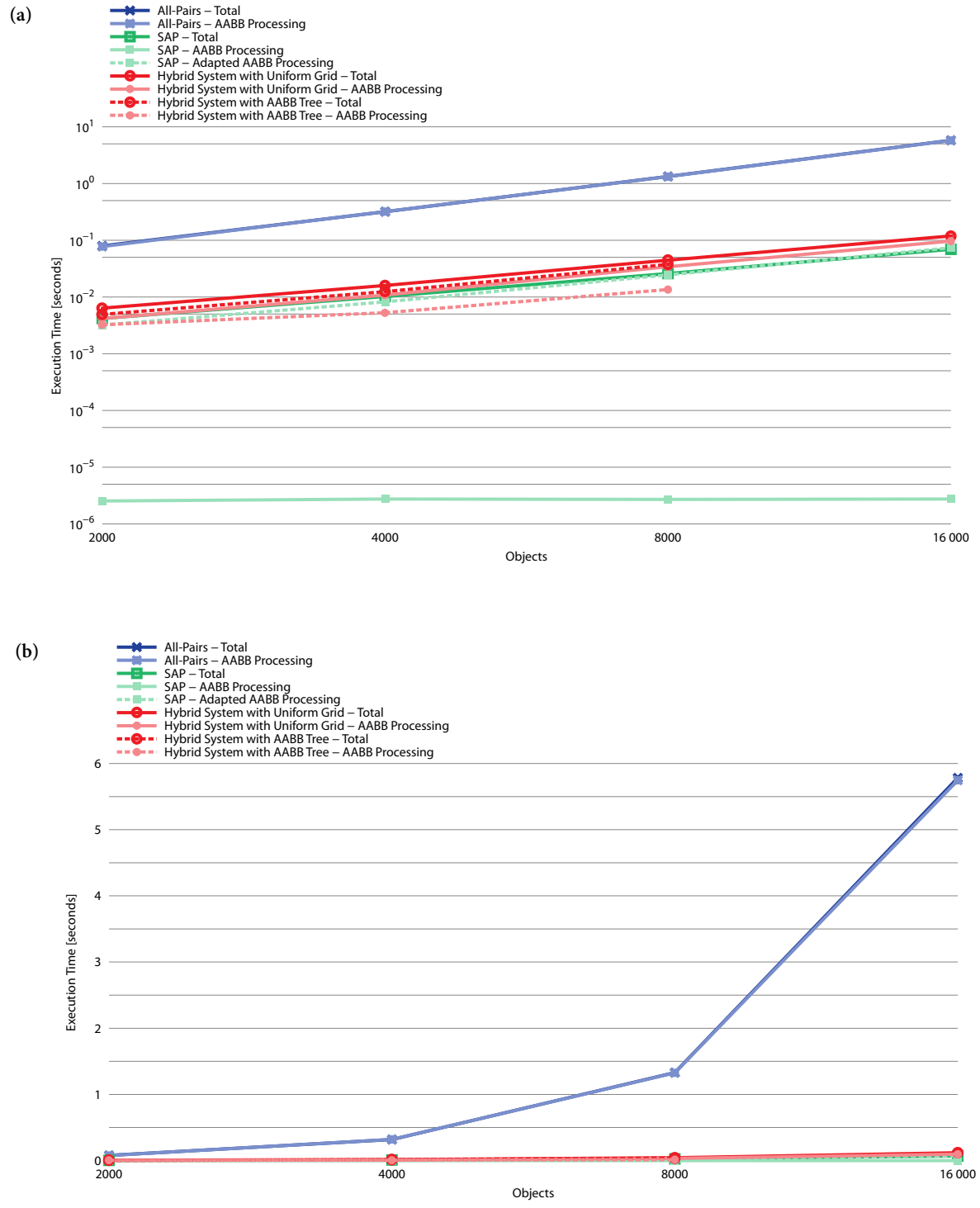


Figure D.20: *Execution times for the 'Software-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 1024.* **(a)** *Logarithmic scale.* **(b)** *Linear scale.*

	All-Pairs		SAP	
Objects	Total	AABB Processing	Total	Adapted AABB Processing
		Uniform Grid		
2000	12.495×	18.506×	0.657×	0.453×
4000	19.959×	29.593×	0.64×	1.165×
8000	29.905×	38.789×	0.584×	0.645×
16 000	0.168×	0.203×	0.015×	0.015×
		AABB Tree		
2000	16.144×	23.907×	0.848×	0.585×
4000	25.548×	60.633×	0.819×	2.387×
8000	35.446×	97.87×	0.692×	1.626×

Table D.13: *Accelerations for the 'Software-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 1024. The data are graphed in Figure 5.26.*

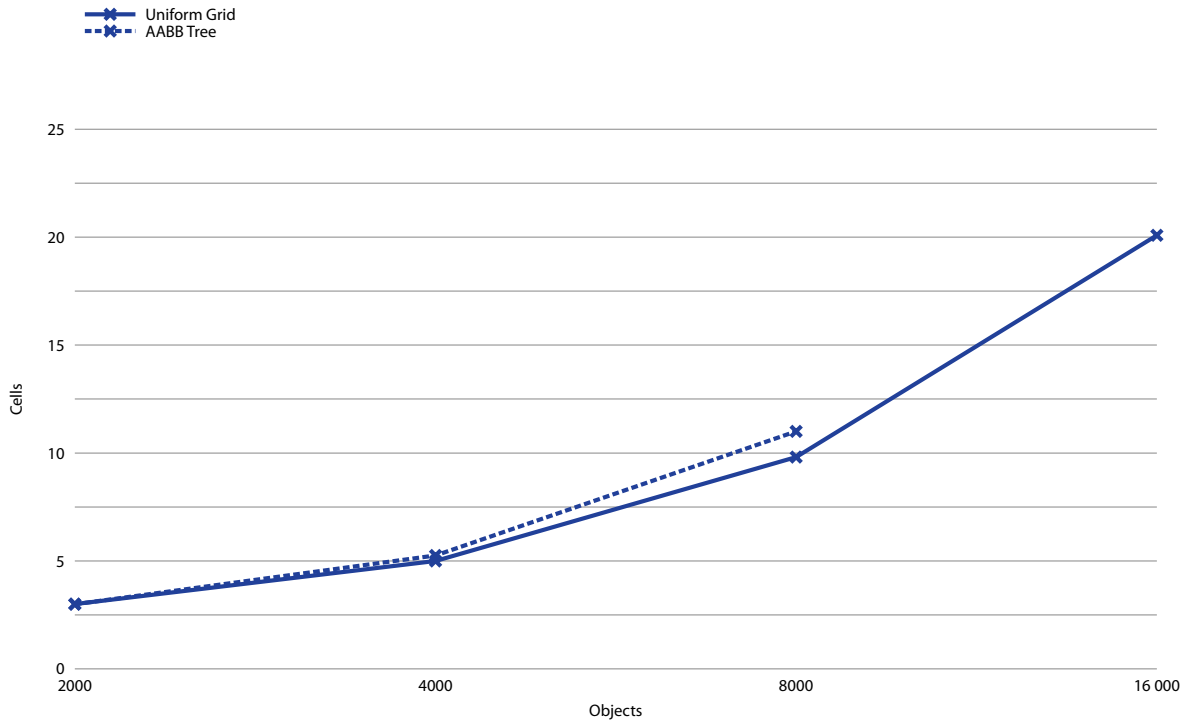


Figure D.21: *Cells for the 'Cells Experiment' of Section 5.4.3 with a cell size of 1024.*

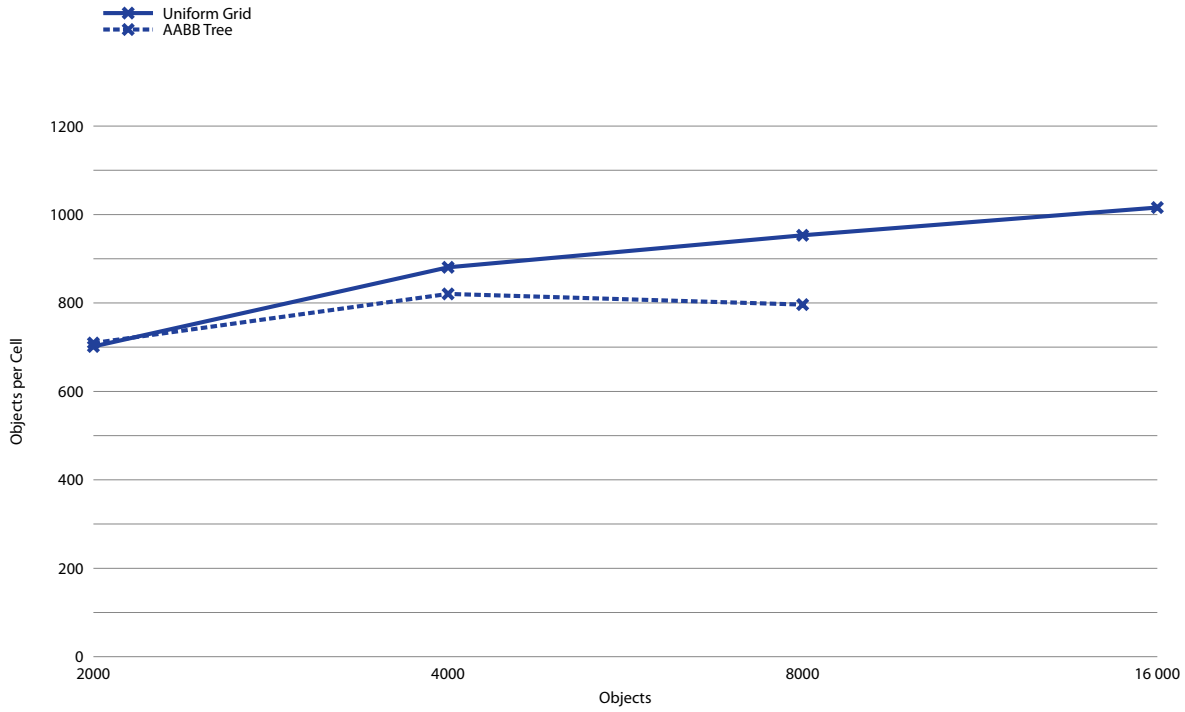


Figure D.22: *Objects per cell for the 'Objects per Cell Experiment' of Section 5.4.3 with a cell size of 1024.*

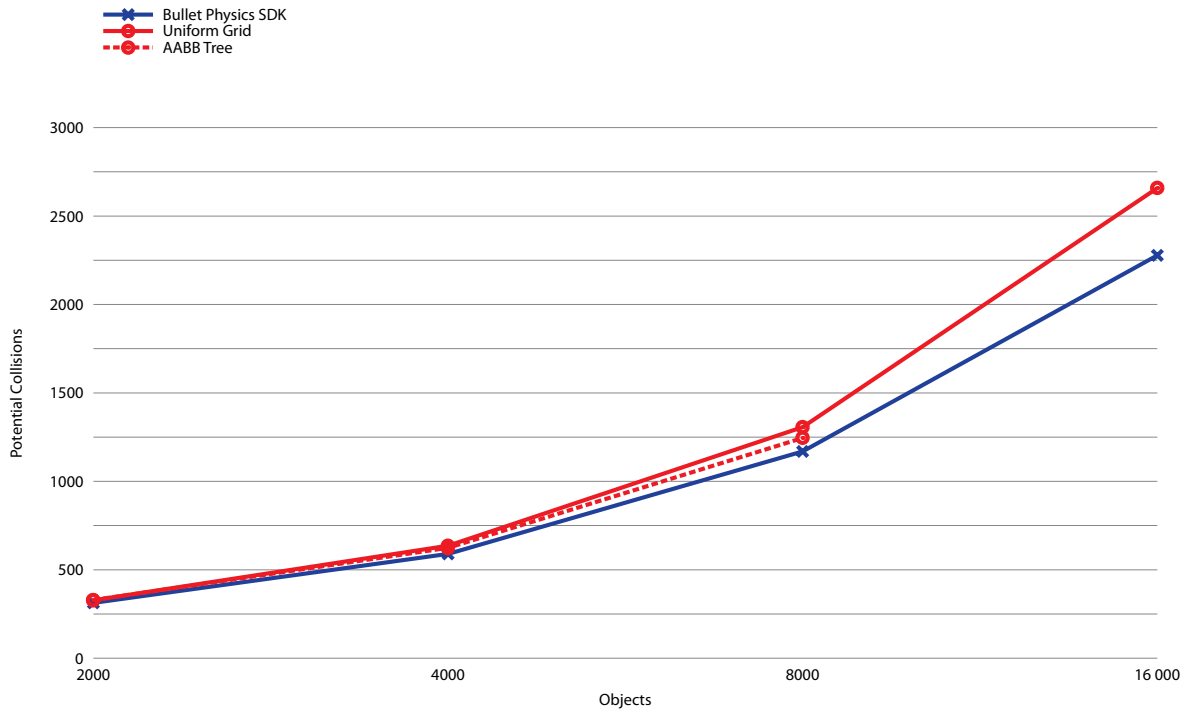


Figure D.23: *Potential collisions for the 'Collisions Experiment' of Section 5.4.3 with a cell size of 1024.*

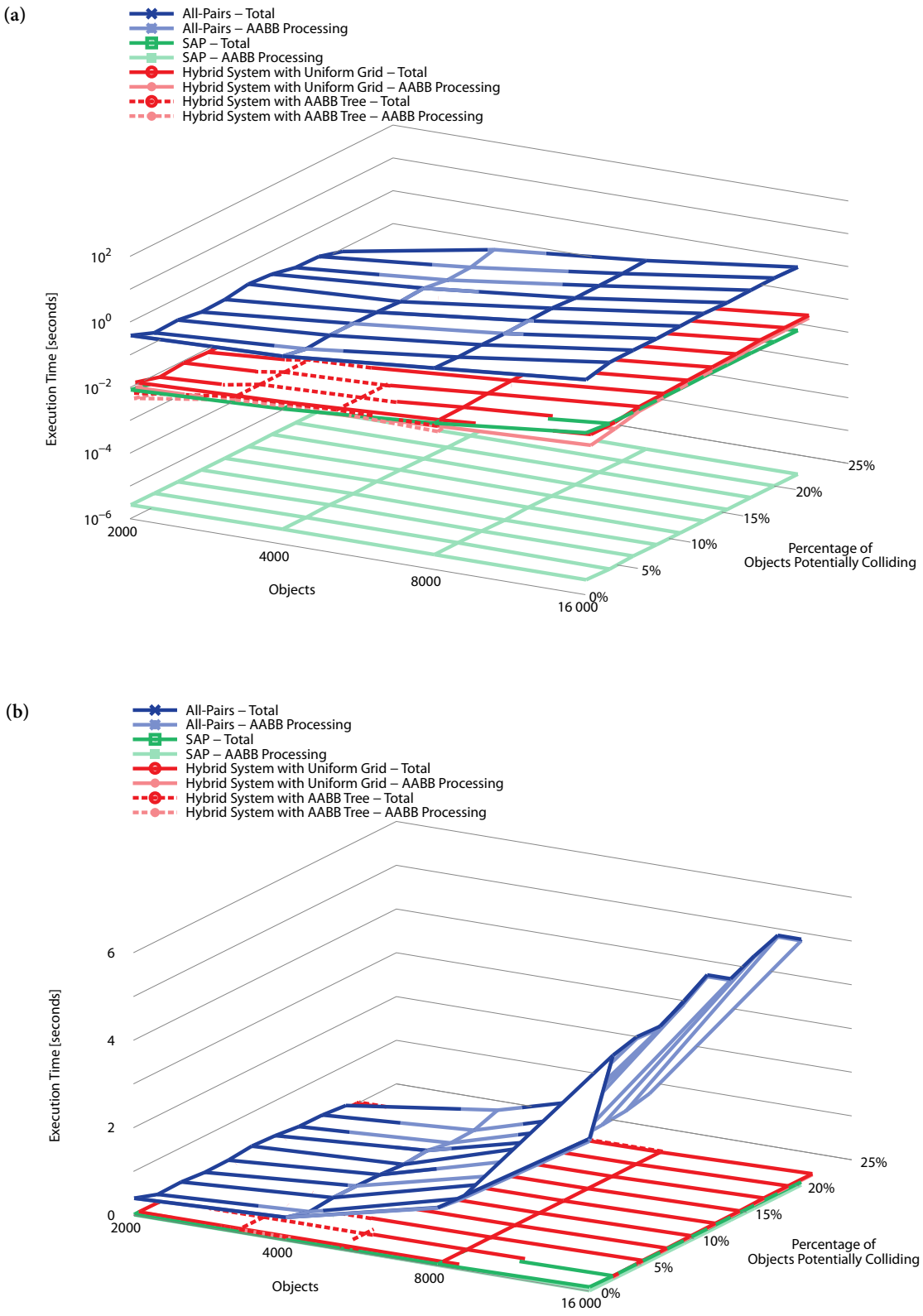


Figure D.24: *Results for the 'Collisions Experiment' of Section 5.4.3 with a cell size of 1024.* **(a)** *Logarithmic scale.* **(b)** *Linear scale.*

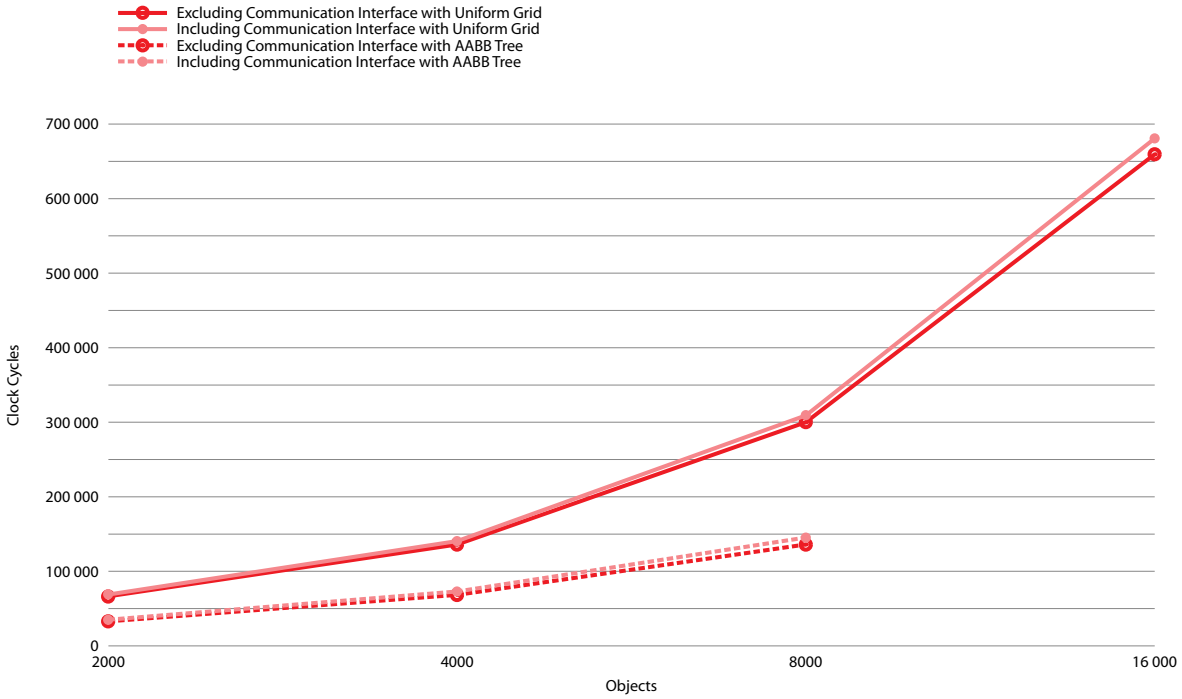


Figure D.25: *Clock cycles for the 'Hardware-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 1024.*

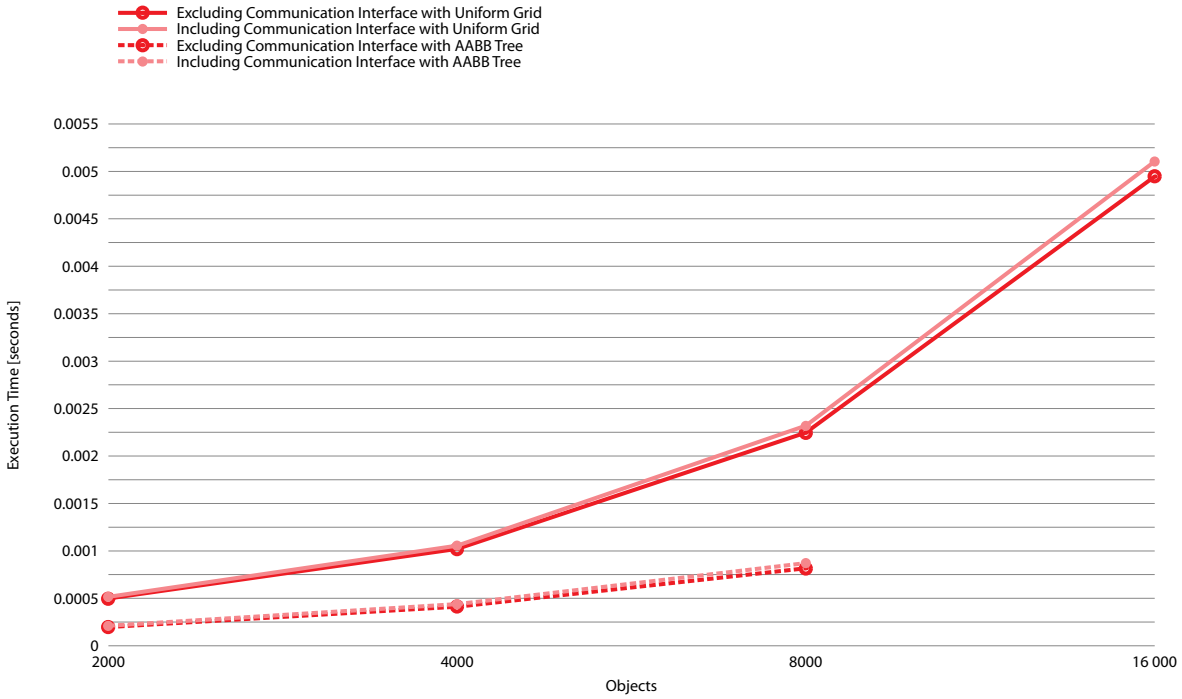


Figure D.26: *Execution times for the 'Hardware-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 1024.*

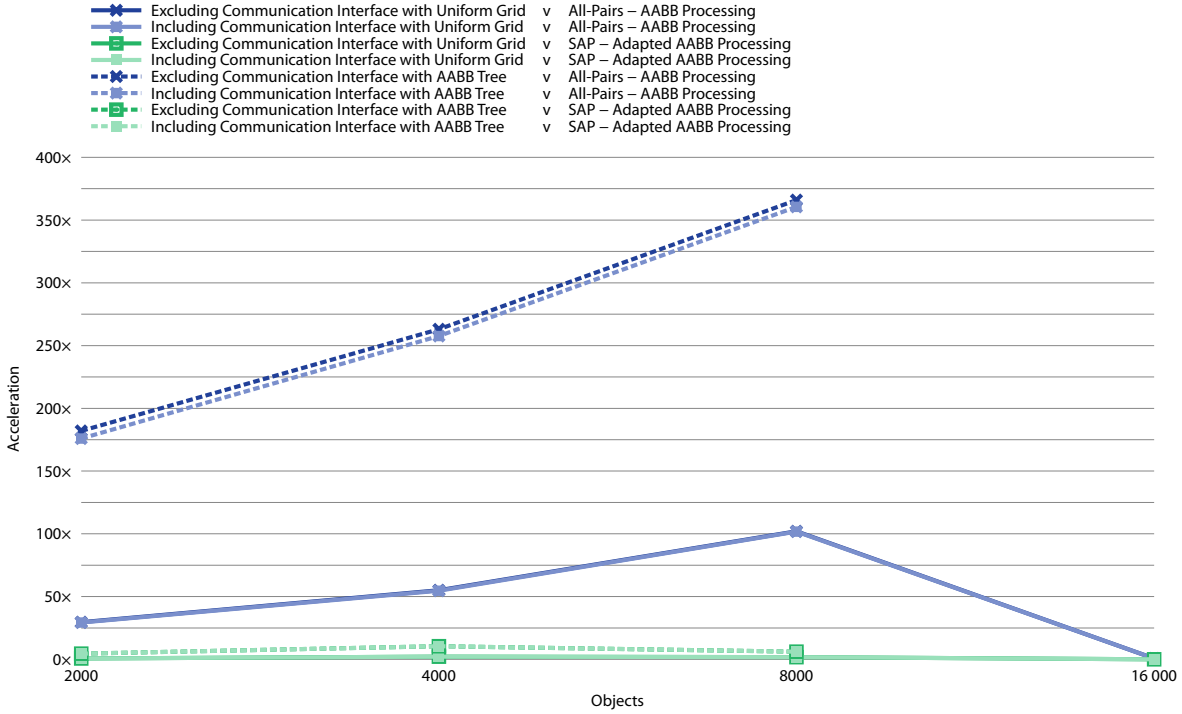


Figure D.27: *Accelerations for the 'Hardware-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 1024. The data are tabulated in Table D.14.*

	Excluding Communication Interface		Including Communication Interface	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
		Uniform Grid		
2000	29.541×	0.723×	29.355×	0.723×
4000	54.951×	2.163×	54.629×	2.163×
8000	102.285×	1.7×	101.713×	1.7×
16 000	0.713×	0.053×	0.709×	0.053×
		AABB Tree		
2000	181.525×	4.44×	175.934×	4.44×
4000	263.406×	10.369×	257.636×	10.369×
8000	366.012×	6.082×	360.477×	6.082×

Table D.14: *Accelerations for the 'Hardware-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 1024. The data are graphed in Figure D.27.*

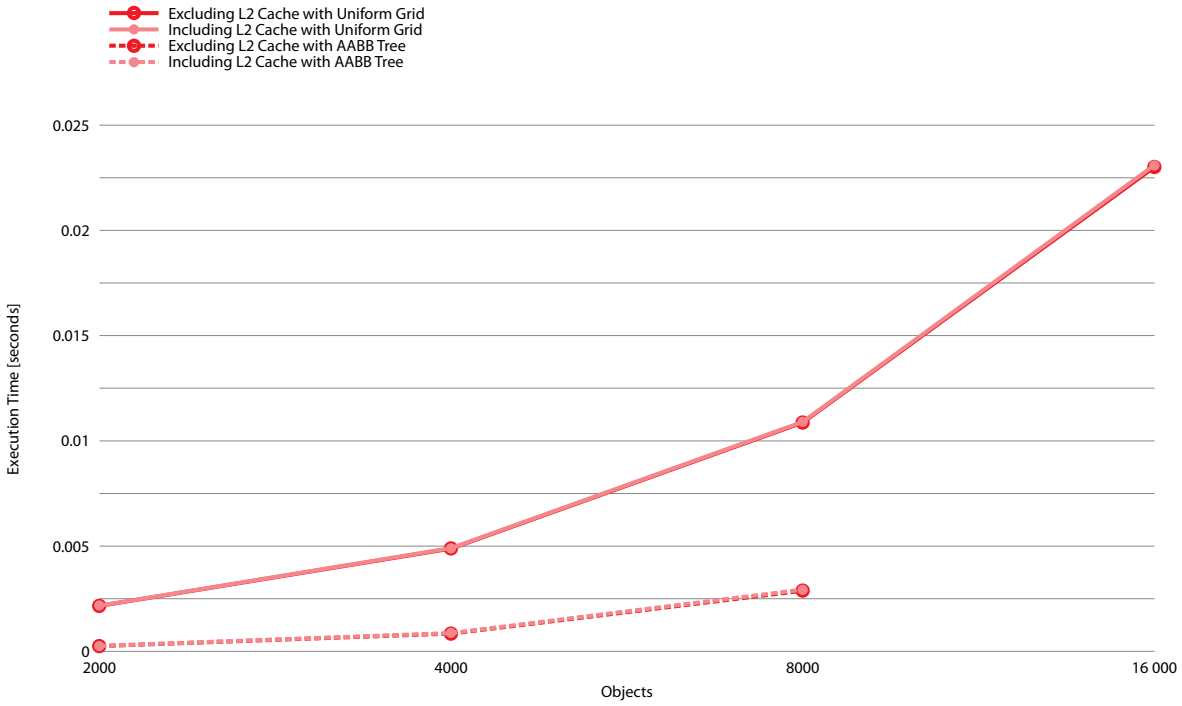


Figure D.28: *Execution times for the 'CPU-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 1024.*

	Excluding L2 Cache		Including L2 Cache	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
		Uniform Grid		
2000	35.932×	0.879×	35.719×	0.872×
4000	65.673×	2.59×	65.364×	2.569×
8000	122.021×	2.03×	121.606×	2.018×
16 000	0.856×	0.063×	0.852×	0.063×
		AABB Tree		
2000	315.511×	7.72×	298.887×	7.299×
4000	381.589×	15×	369.685×	14.529×
8000	461.314×	7.67×	452.248×	7.503×

Table D.15: *Accelerations for the 'CPU-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 1024. The data are graphed in Figure 5.26.*

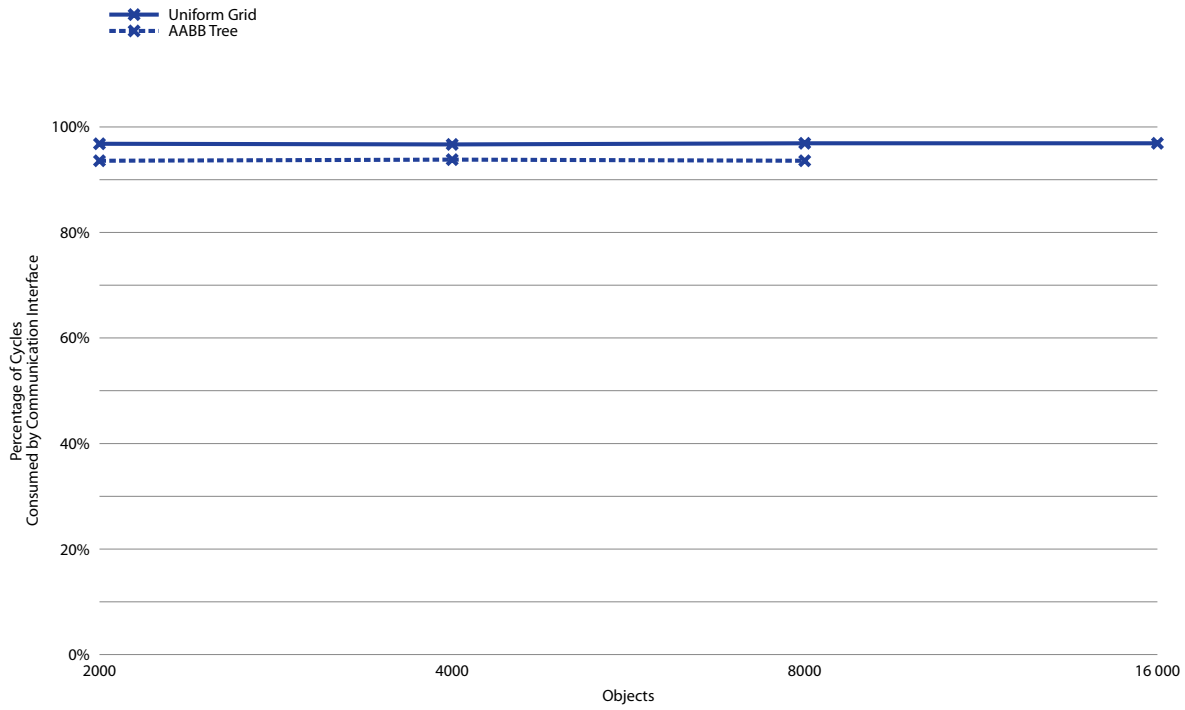


Figure D.29: *The percentage of clock cycles consumed by the communication interface relative to those consumed by the microarchitecture-based algorithm corresponding to Section 5.4.3 with a cell size of 1024.*

Cell Size 2048

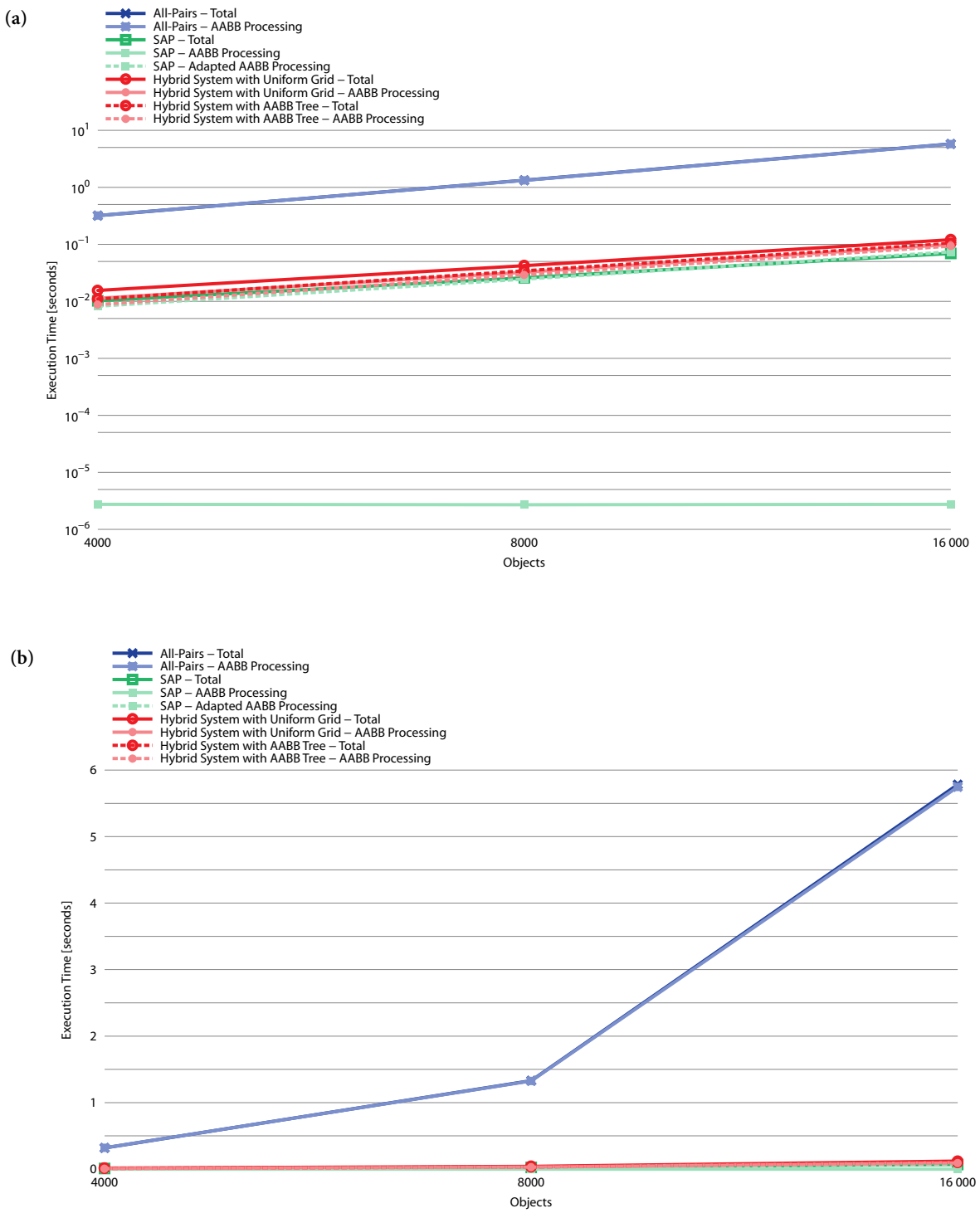


Figure D.30: *Execution times for the 'Software-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 2048.* **(a)** *Logarithmic scale.* **(b)** *Linear scale.*

	All-Pairs		SAP	
Objects	Total	AABB Processing	Total	Adapted AABB Processing
		Uniform Grid		
4000	20.495×	28.315×	0.657×	1.115×
8000	31.406×	41.037×	0.613×	0.682×
16 000	0.167×	0.198×	0.015×	0.015×
		AABB Tree		
4000	28.895×	36.493×	0.926×	1.437×
8000	38.803×	45.531×	0.758×	0.757×
16 000	0.193×	0.211×	0.018×	0.016×

Table D.16: *Accelerations for the 'Software-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 2048. The data are graphed in Figure 5.27.*

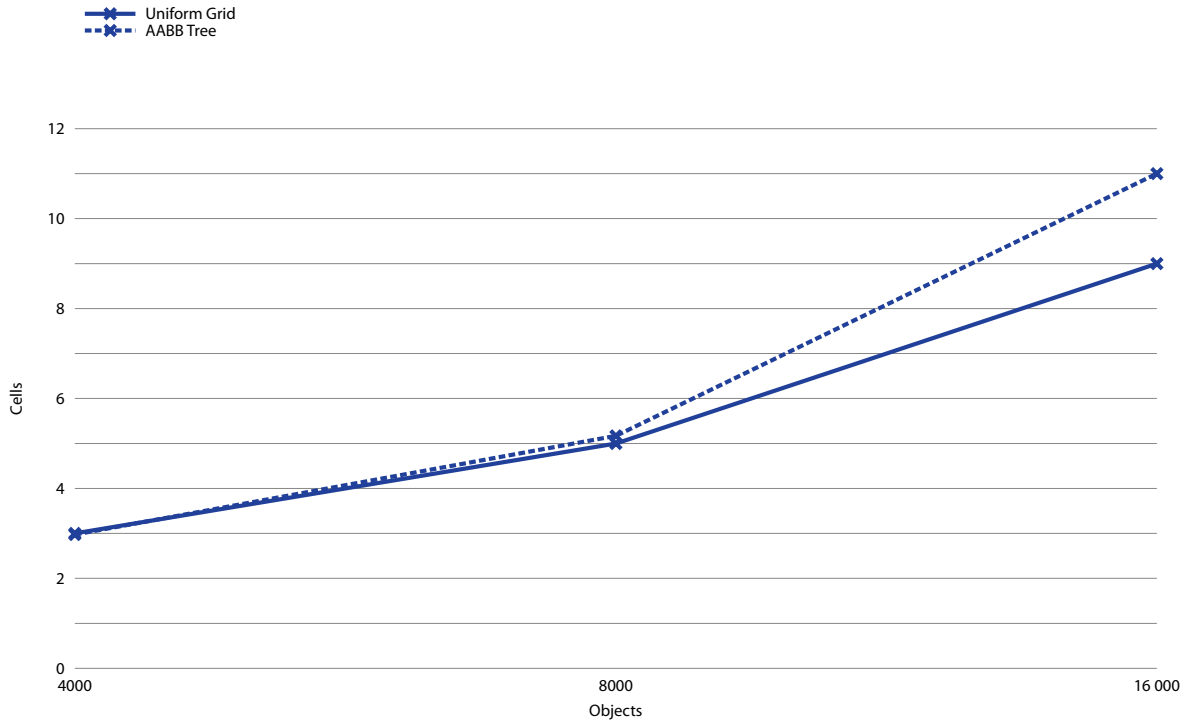


Figure D.31: *Cells for the 'Cells Experiment' of Section 5.4.3 with a cell size of 2048.*

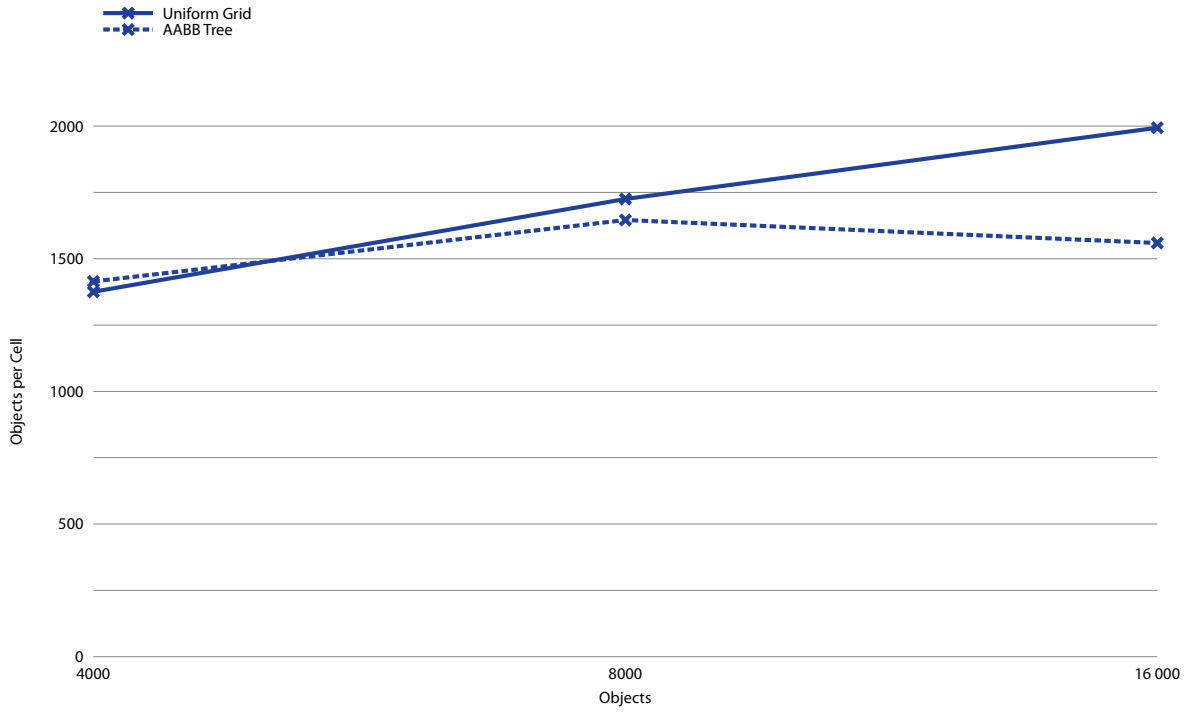


Figure D.32: *Objects per cell for the 'Objects per Cell Experiment' of Section 5.4.3 with a cell size of 2048.*

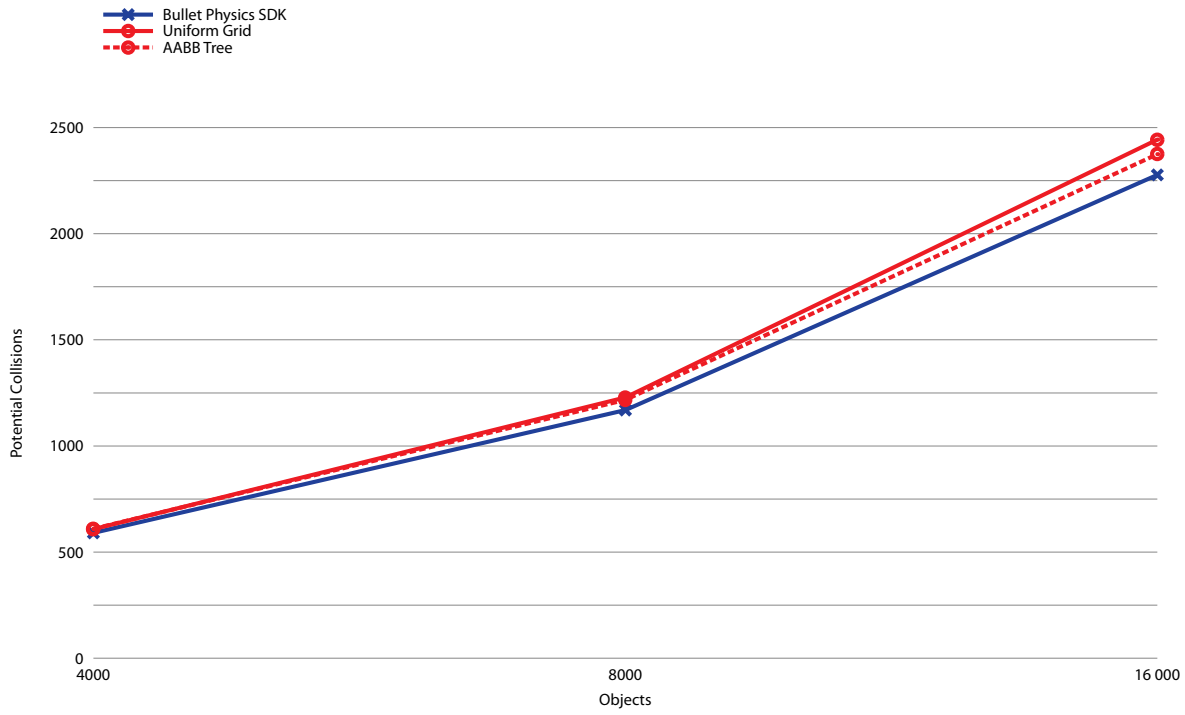


Figure D.33: *Potential collisions for the 'Collisions Experiment' of Section 5.4.3 with a cell size of 2048.*

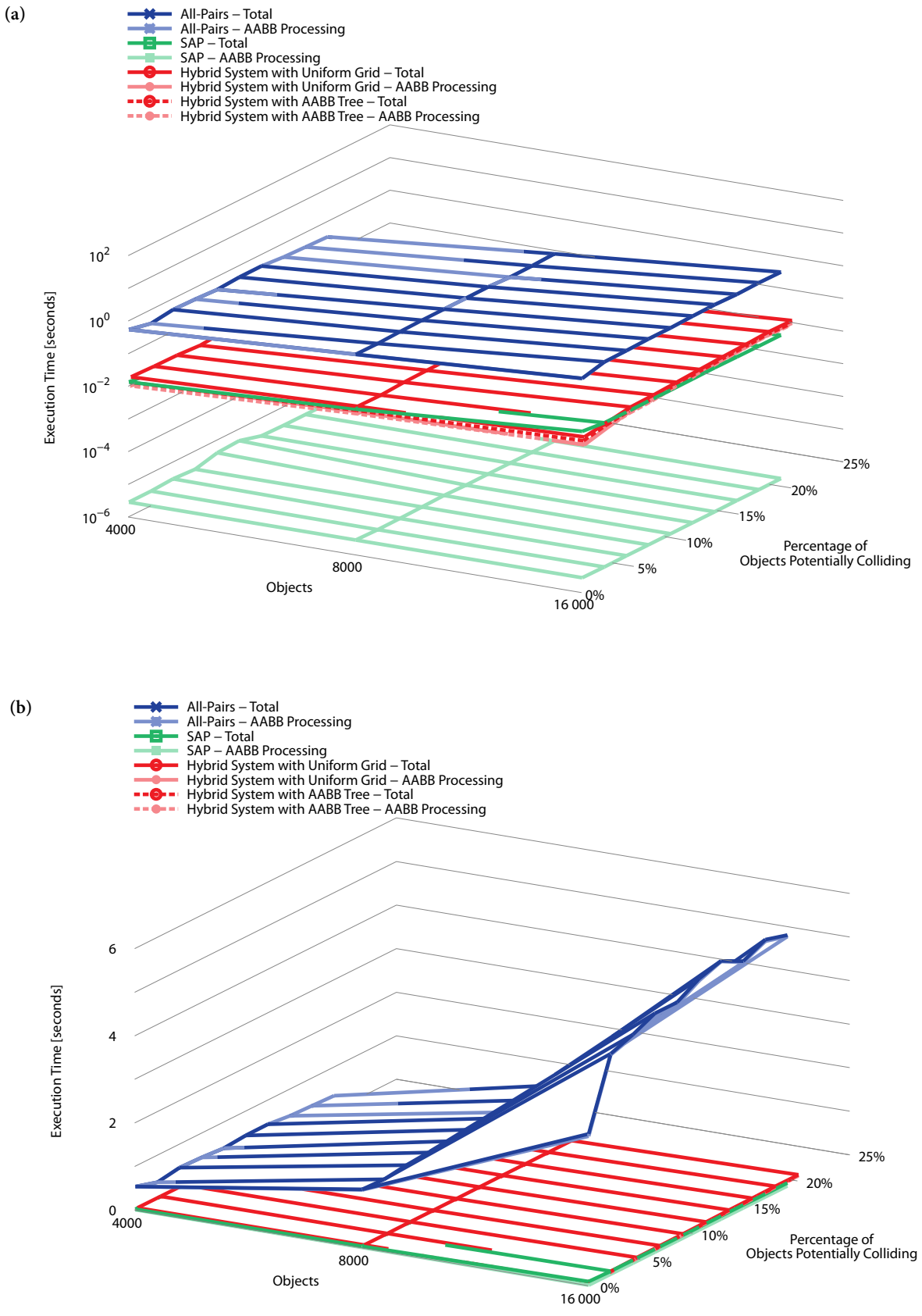


Figure D.34: *Results for the 'Collisions Experiment' of Section 5.4.3 with a cell size of 2048.* **(a)** *Logarithmic scale.* **(b)** *Linear scale.*

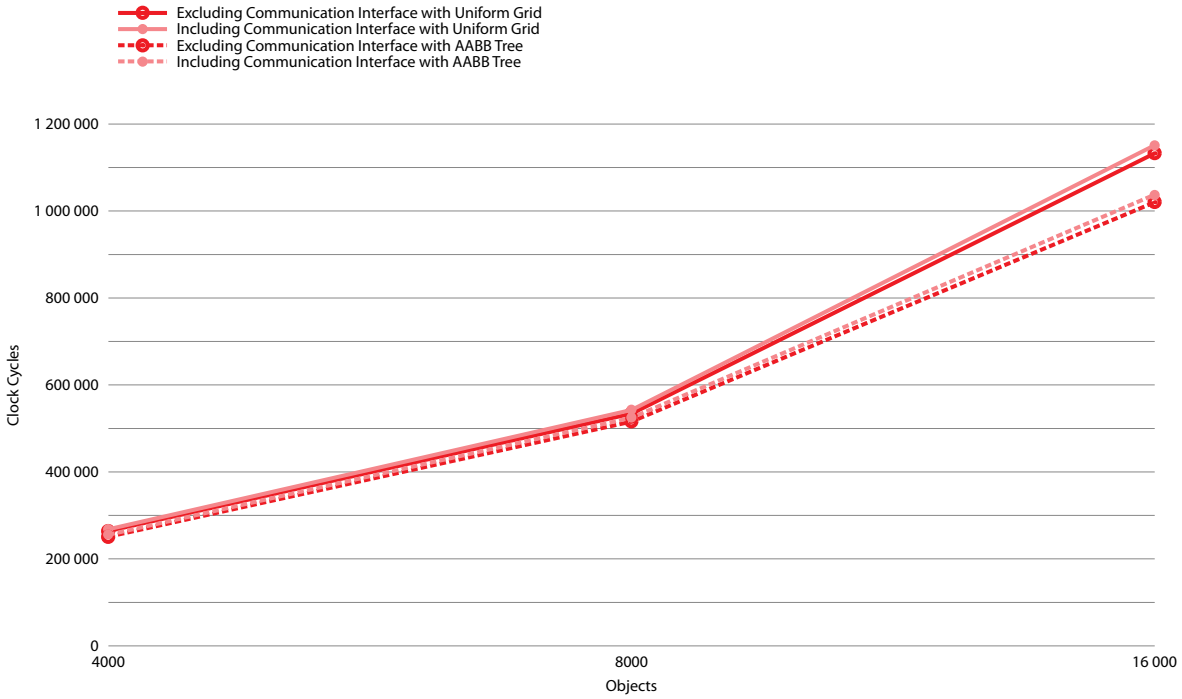


Figure D.35: *Clock cycles for the 'Hardware-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 2048.*

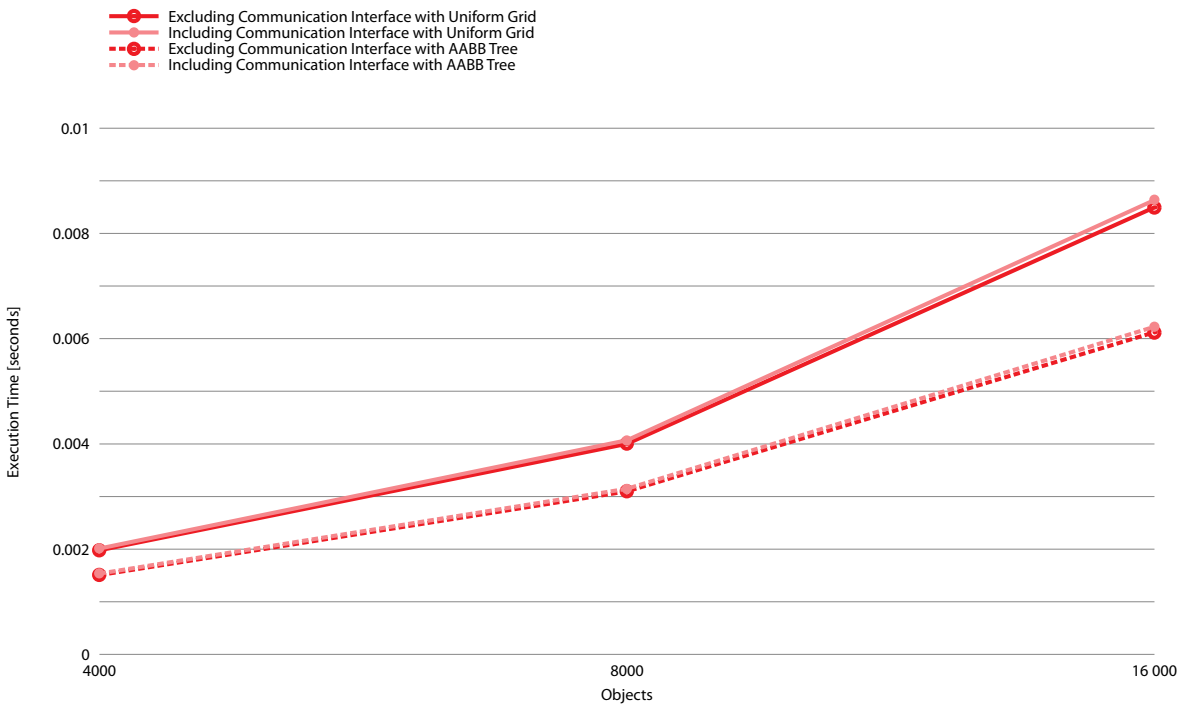


Figure D.36: *Execution times for the 'Hardware-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 2048.*

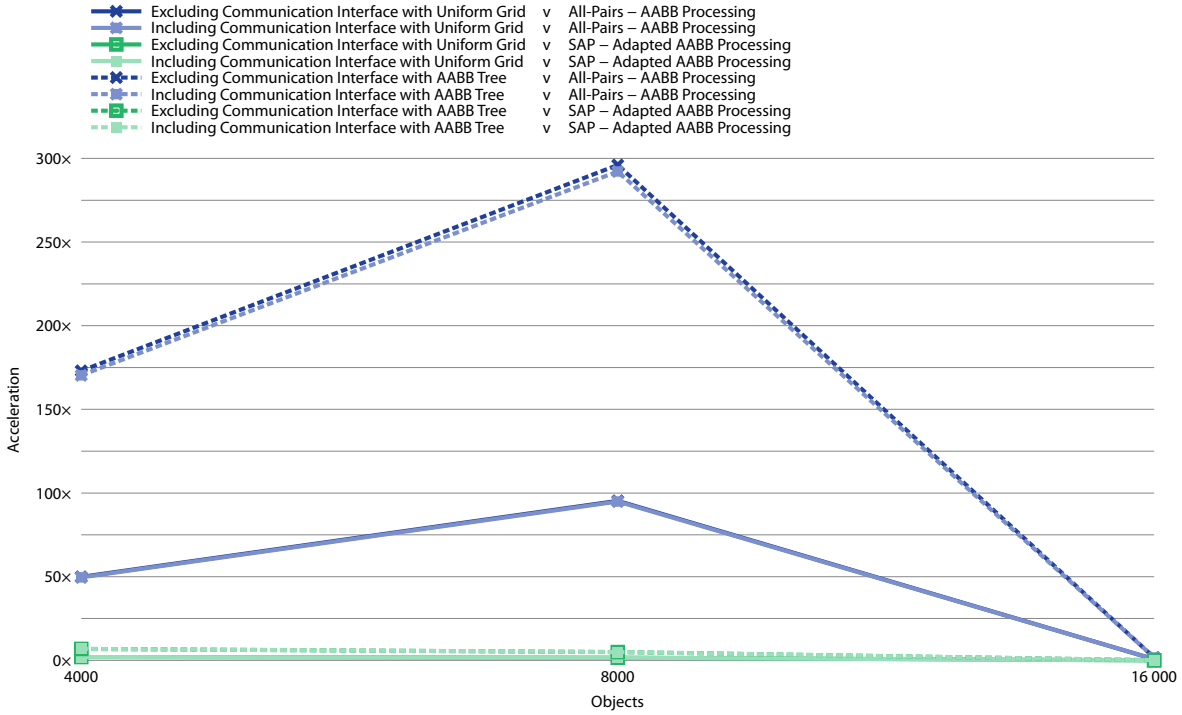


Figure D.37: *Accelerations for the 'Hardware-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 2048. The data are tabulated in Table D.17.*

	Excluding Communication Interface		Including Communication Interface	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
		Uniform Grid		
4000	49.877×	1.963×	49.631×	1.963×
8000	95.331×	1.584×	94.88×	1.584×
16 000	0.67×	0.049×	0.666×	0.049×
		AABB Tree		
4000	172.708×	6.799×	170.34×	6.799×
8000	295.535×	4.911×	292.148×	4.911×
16 000	1.696×	0.125×	1.681×	0.125×

Table D.17: *Accelerations for the 'Hardware-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 2048. The data are graphed in Figure D.37.*

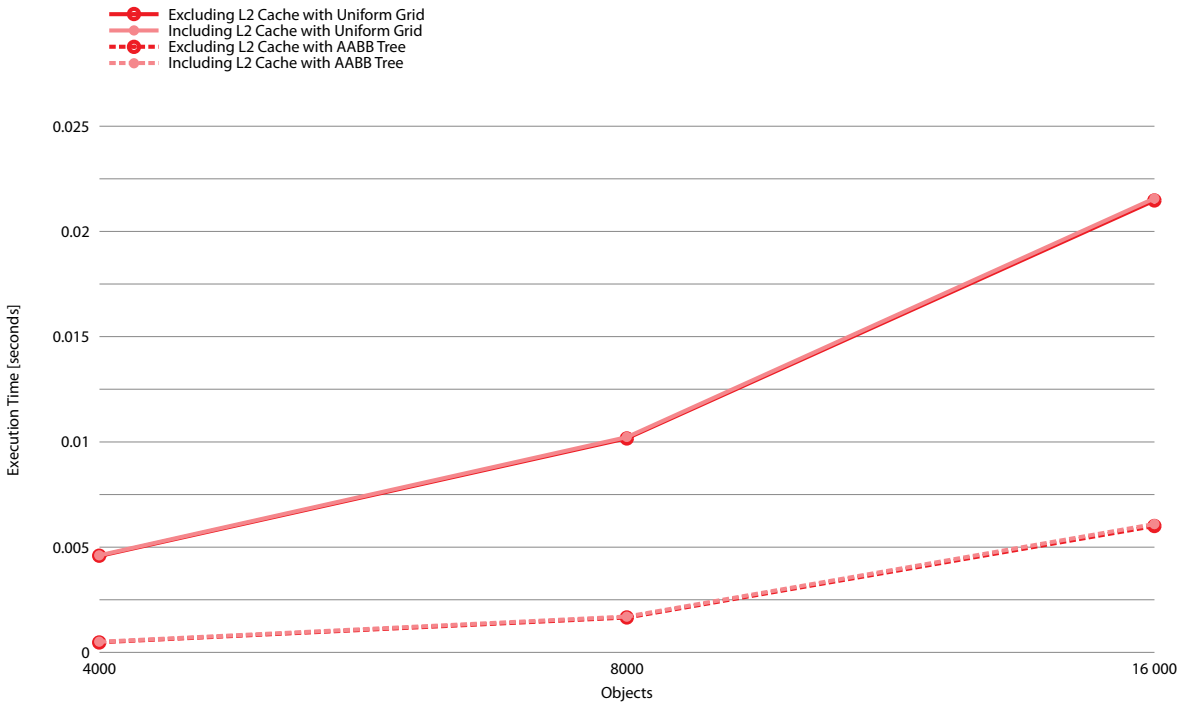


Figure D.38: *Execution times for the 'CPU-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 2048.*

	Excluding L2 Cache		Including L2 Cache	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
		Uniform Grid		
4000	69.988×	2.76×	69.562×	2.734×
8000	130.334×	2.17×	129.605×	2.150×
16 000	0.917×	0.068×	0.912×	0.067×
		AABB Tree		
4000	676.52×	26.6×	640.578×	25.175×
8000	804.593×	13.4×	779.078×	12.926×
16 000	3.284×	0.243×	3.226×	0.238×

Table D.18: *Accelerations for the 'CPU-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 2048. The data are graphed in Figure 5.27.*

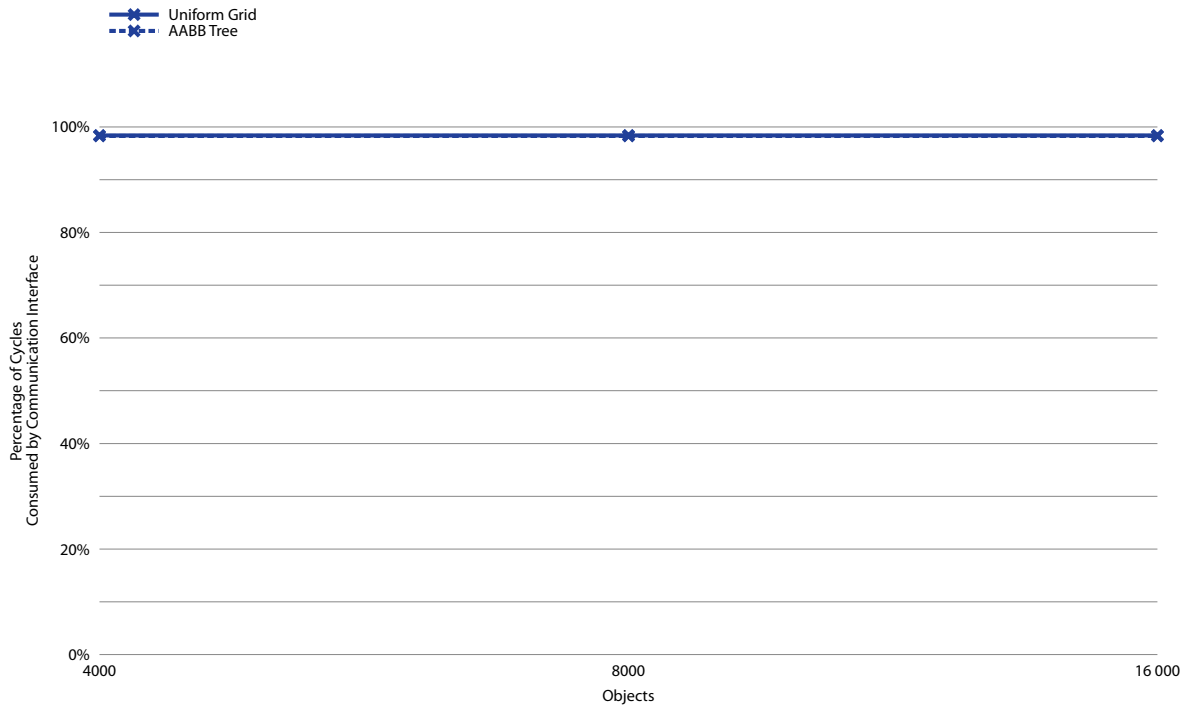


Figure D.39: *The percentage of clock cycles consumed by the communication interface relative to those consumed by the microarchitecture-based algorithm corresponding to Section 5.4.3 with a cell size of 2048.*

Cell Size 4096

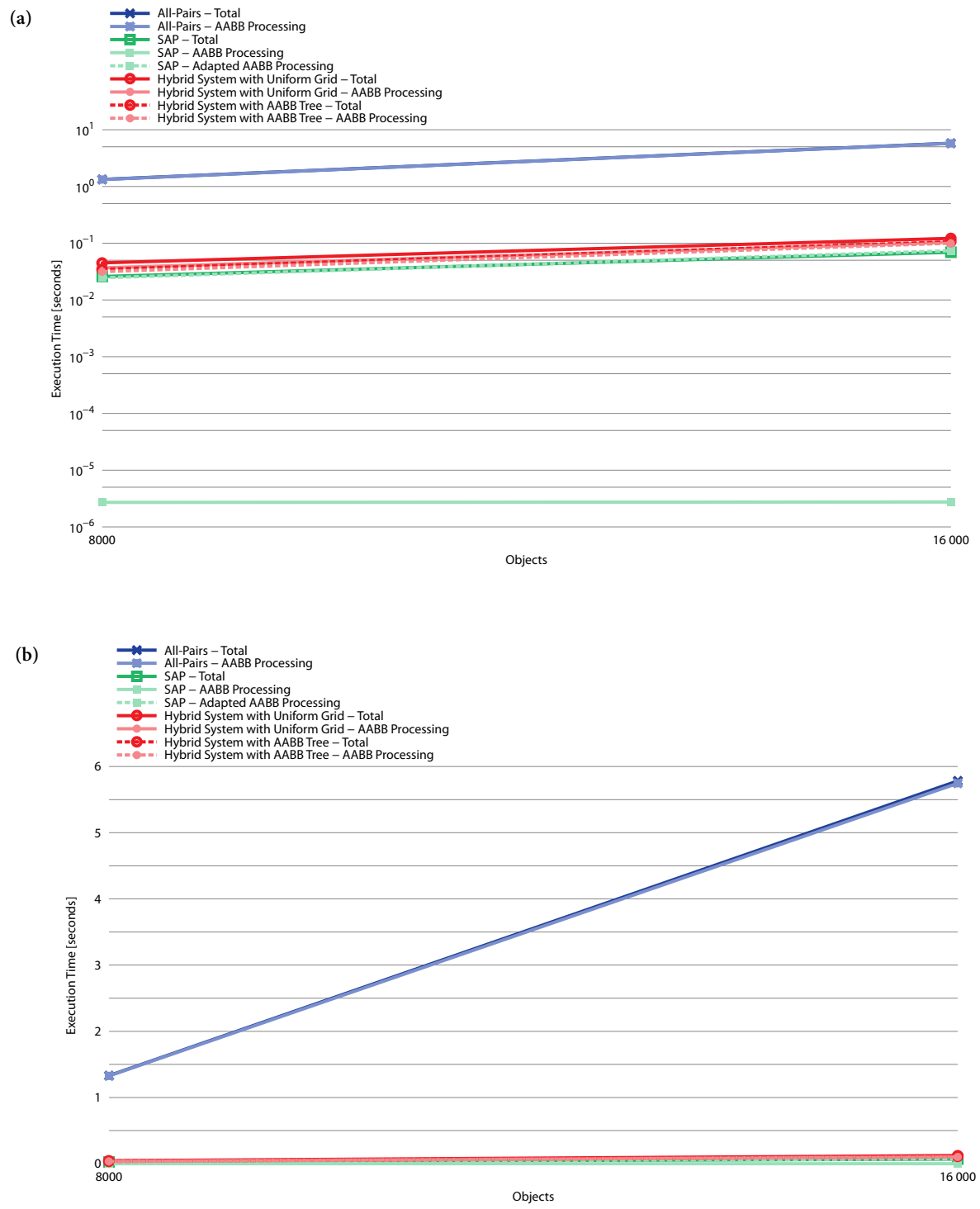


Figure D.40: *Execution times for the 'Software-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 4096.* **(a)** *Logarithmic scale.* **(b)** *Linear scale.*

	All-Pairs		SAP	
Objects	Total	AABB Processing	Total	Adapted AABB Processing
		Uniform Grid		
8000	29.455×	36.036×	0.575×	0.599×
16 000	0.164×	0.189×	0.015×	0.014×
		AABB Tree		
8000	38.223×	42.13×	0.746×	0.7×
16 000	0.189×	0.197×	0.017×	0.015×

Table D.19: *Accelerations for the 'Software-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 4096. The data are graphed in Figure 5.28.*

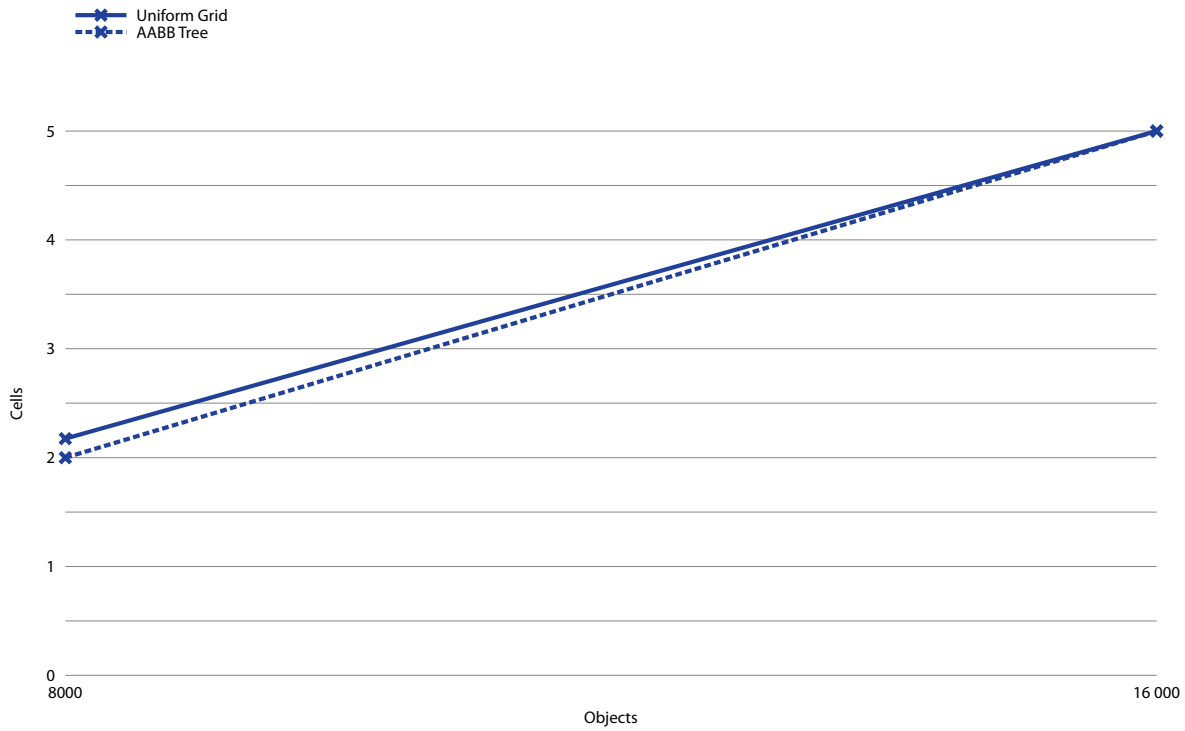


Figure D.41: *Cells for the 'Cells Experiment' of Section 5.4.3 with a cell size of 4096.*

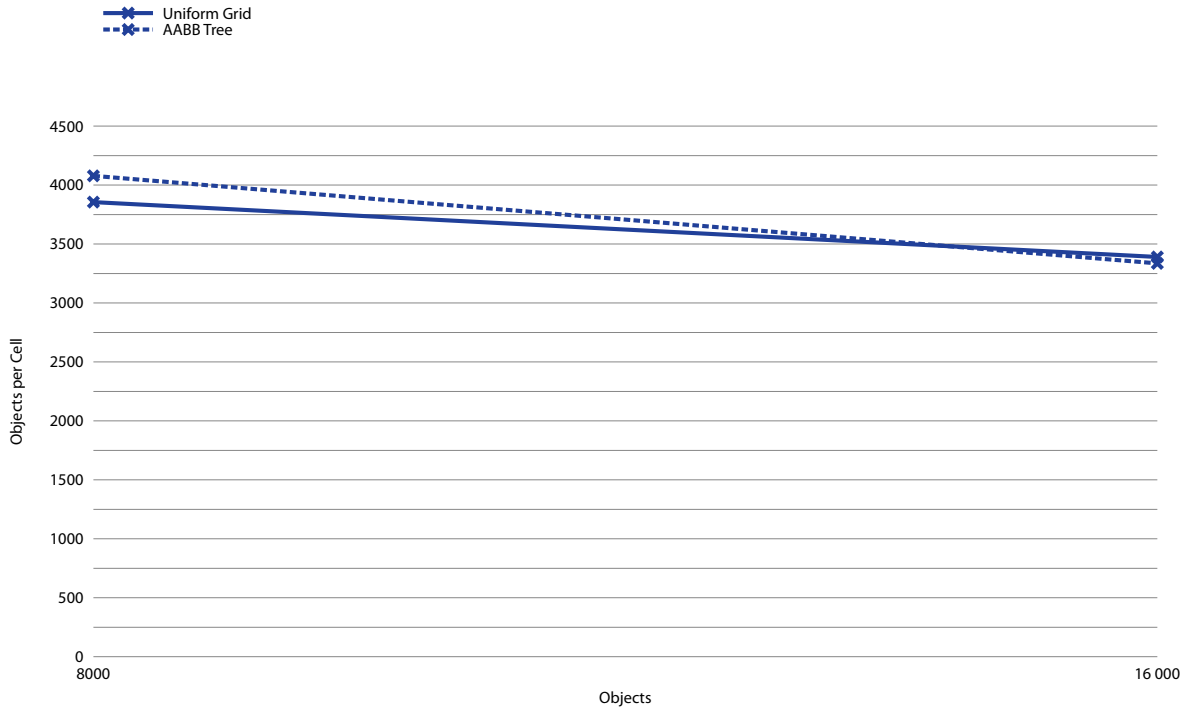


Figure D.42: *Objects per cell for the 'Objects per Cell Experiment' of Section 5.4.3 with a cell size of 4096.*

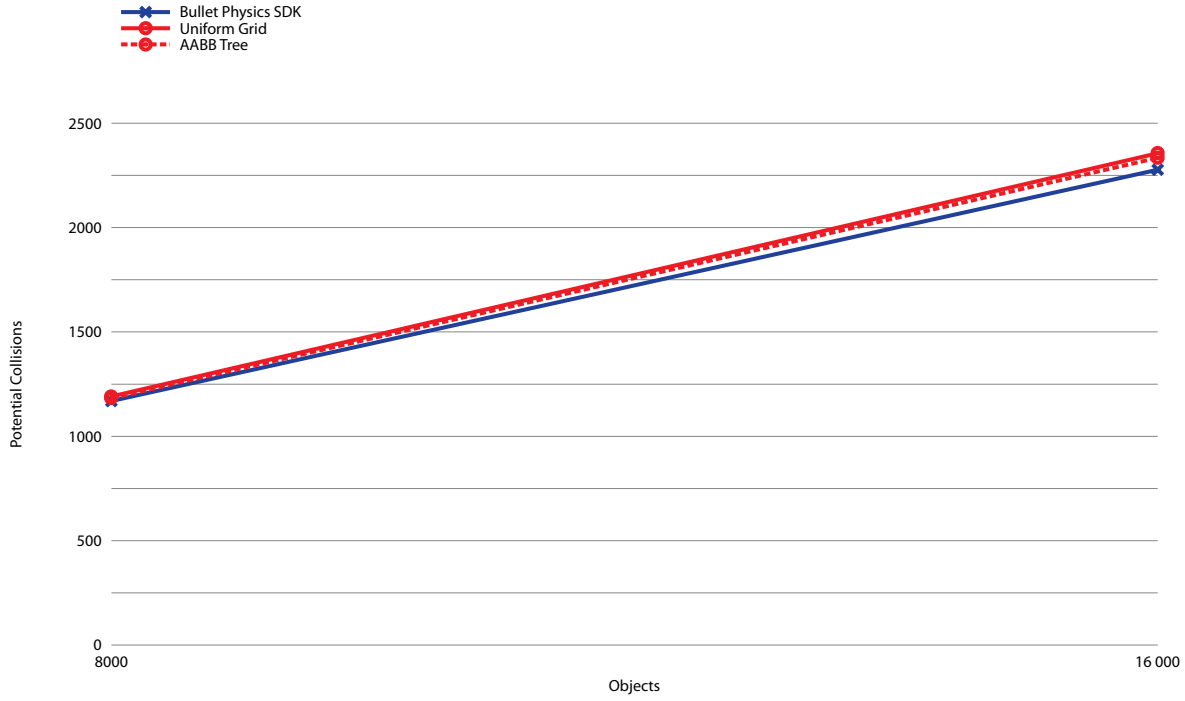


Figure D.43: *Potential collisions for the 'Collisions Experiment' of Section 5.4.3 with a cell size of 4096.*

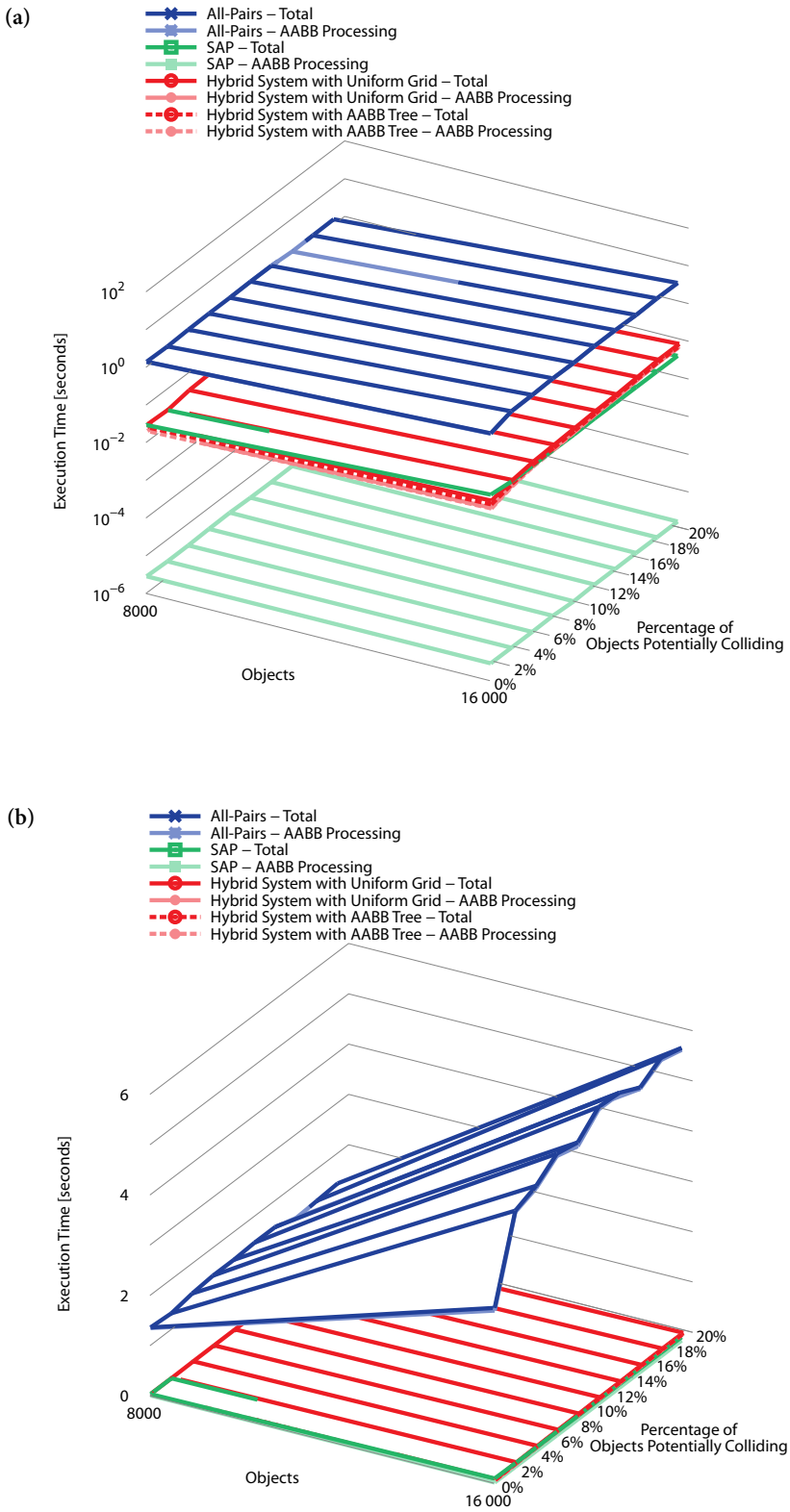


Figure D.44: *Results for the 'Collisions Experiment' of Section 5.4.3 with a cell size of 4096.* **(a)** *Logarithmic scale.* **(b)** *Linear scale.*

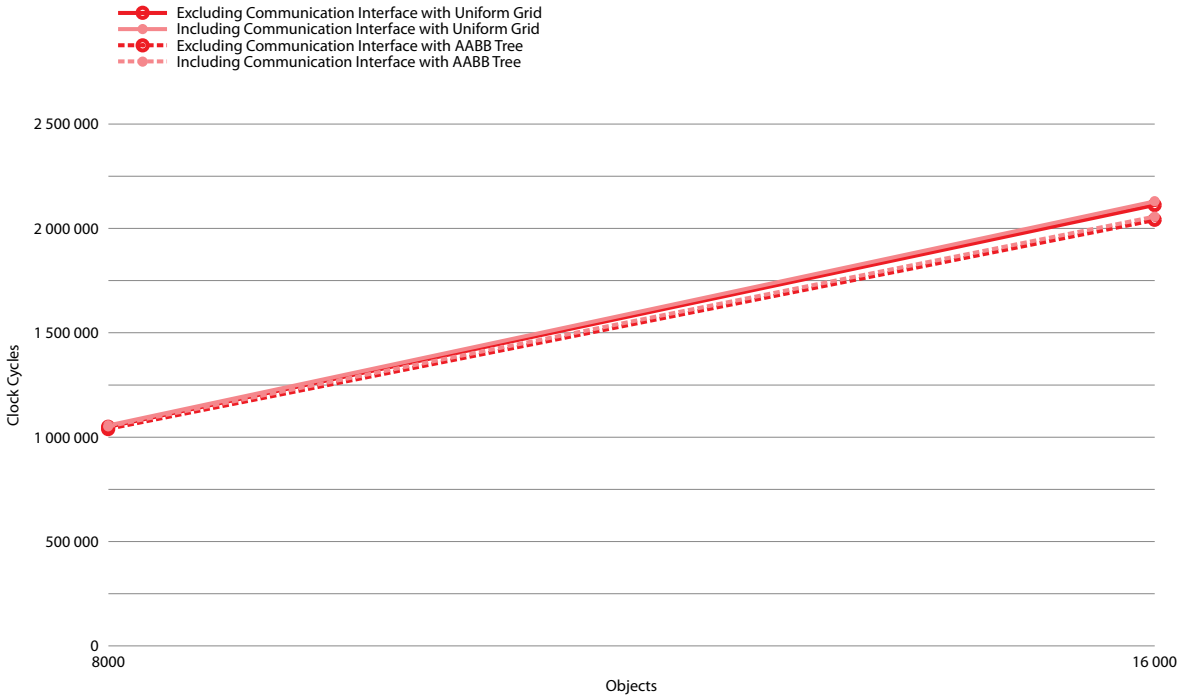


Figure D.45: *Clock cycles for the 'Hardware-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 4096.*

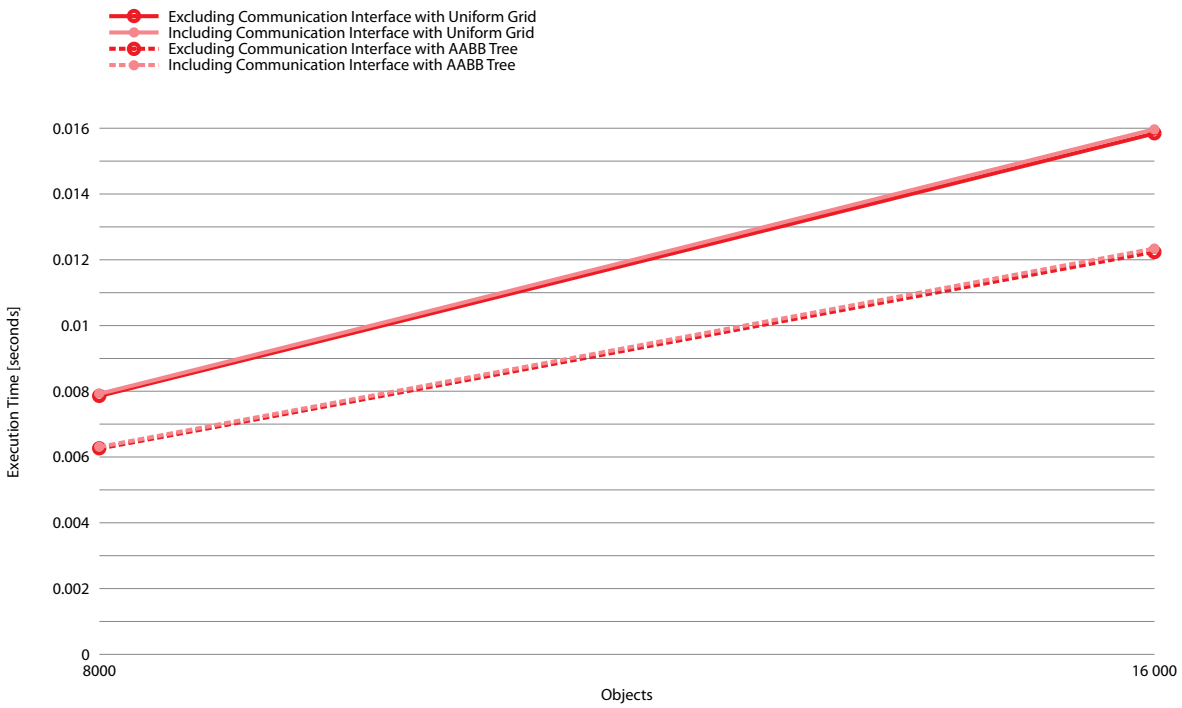


Figure D.46: *Execution times for the 'Hardware-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 4096.*

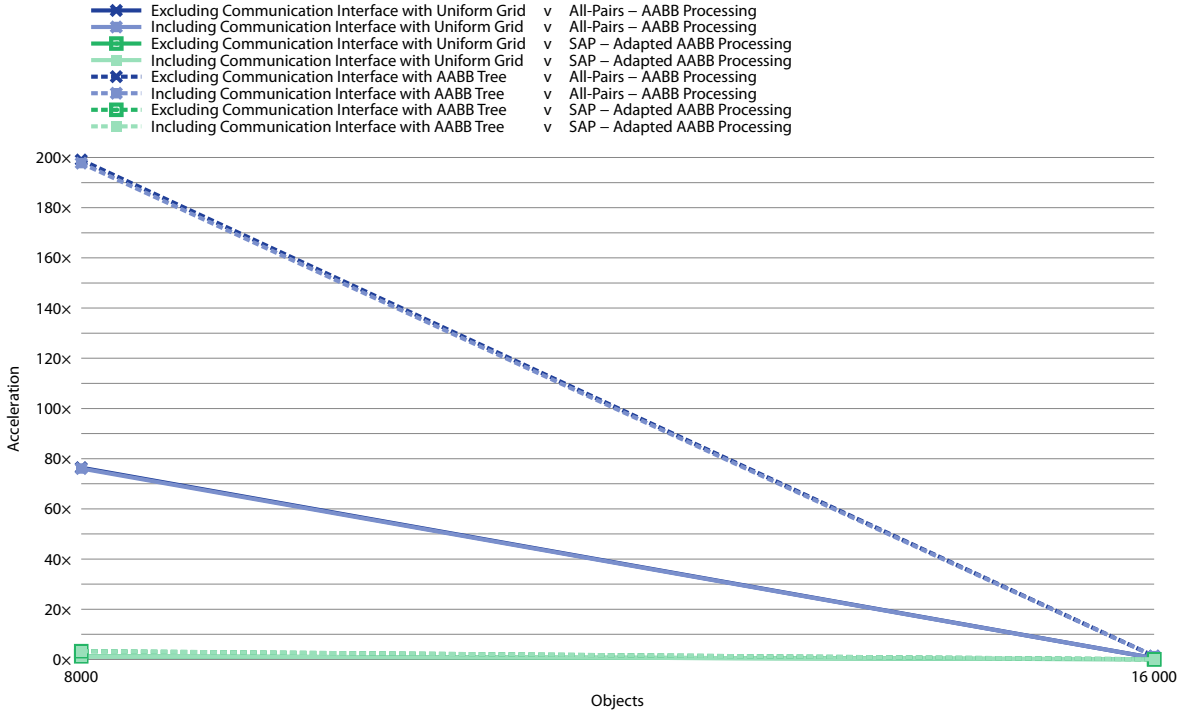


Figure D.47: *Accelerations for the 'Hardware-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 4096. The data are tabulated in Table D.20.*

	Excluding Communication Interface		Including Communication Interface	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
		Uniform Grid		
8000	76.361×	1.269×	76.089×	1.269×
16 000	0.553×	0.041×	0.551×	0.041×
		AABB Tree		
8000	199.232×	3.311×	197.765×	3.311×
16 000	1.298×	0.096×	1.29×	0.096×

Table D.20: *Accelerations for the 'Hardware-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 4096. The data are graphed in Figure D.47.*

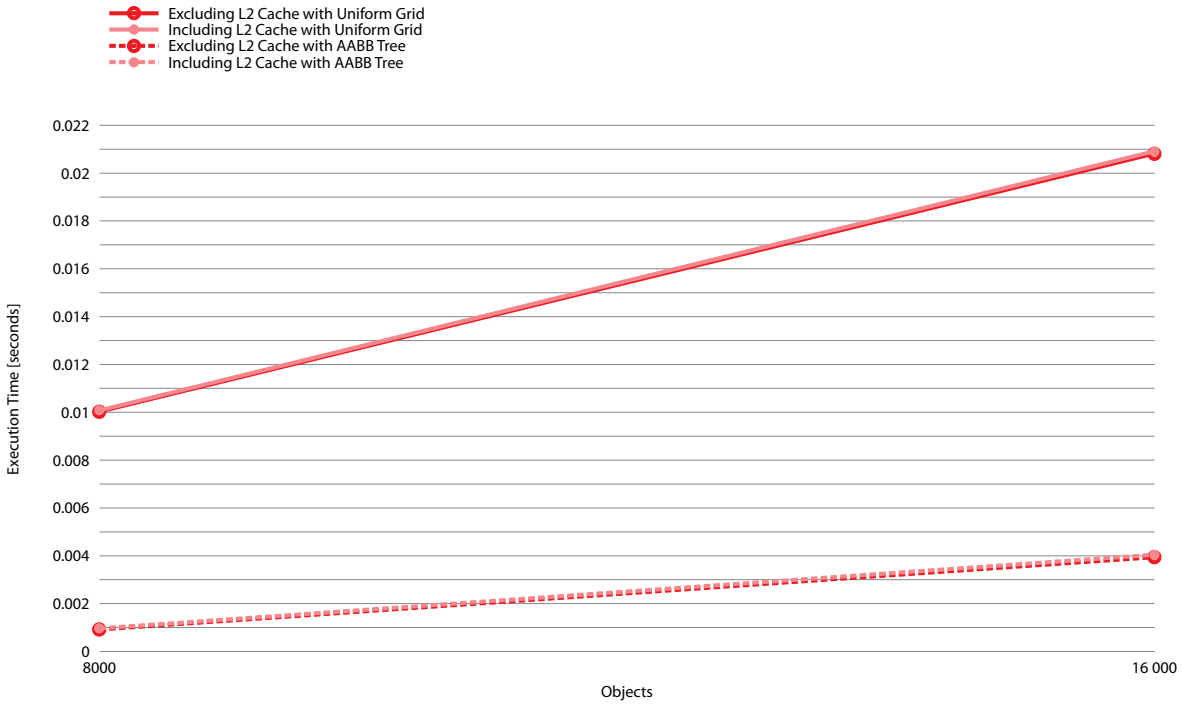


Figure D.48: *Execution times for the 'CPU-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 4096.*

	Excluding L2 Cache		Including L2 Cache	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
		Uniform Grid		
8000	132.226×	2.197×	131.56×	2.183×
16 000	0.946×	0.07×	0.941×	0.069×
		AABB Tree		
8000	1444.885×	24.01×	1367×	22.68×
16 000	4.999×	0.369×	4.865×	0.359×

Table D.21: *Accelerations for the 'CPU-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 4096. The data are graphed in Figure 5.28.*

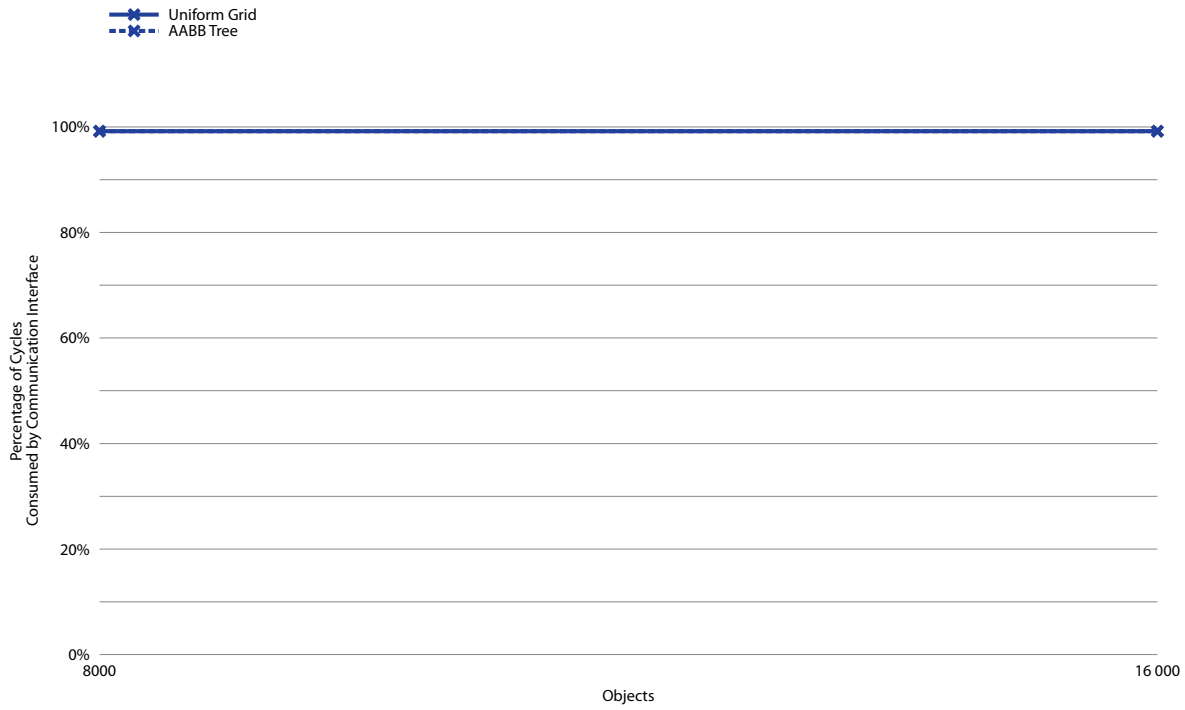


Figure D.49: *The percentage of clock cycles consumed by the communication interface relative to those consumed by the microarchitecture-based algorithm corresponding to Section 5.4.3 with a cell size of 4096.*

Cell Size 8192

Objects	Algorithm	Execution Time [seconds]
16 000	All-Pairs – Total	5.782
	All-Pairs – AABB Processing	5.746
	SAP – Total	0.069
	SAP – AABB Processing	0.000003
	SAP – Adapted AABB Processing	0.074
	Hybrid System with Uniform Grid – Total	0.127
	Hybrid System with Uniform Grid – AABB Processing	0.108
	Hybrid System with AABB Tree – Total	0.115
	Hybrid System with AABB Tree – AABB Processing	0.105

Table D.22: *Execution times for the 'Software-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 8192.*

Objects	Algorithm	Cells
16 000	Uniform Grid	2
	AABB Tree	2

Table D.23: *Cells for the 'Cells Experiment' of Section 5.4.3 with a cell size of 8192.*

Objects	Algorithm	Objects per Cell
16 000	Uniform Grid	8130.094
	AABB Tree	8120.102

Table D.24: *Objects per cell for the 'Objects per Cell Experiment' of Section 5.4.3 with a cell size of 8192.*

Objects	Algorithm	Potential Collisions
16 000	Bullet Physics SDK	2277.308
	Uniform Grid	2294.897
	AABB Tree	2295.341

Table D.25: *Potential collisions for the 'Collisions Experiment' of Section 5.4.3 with a cell size of 8192.*

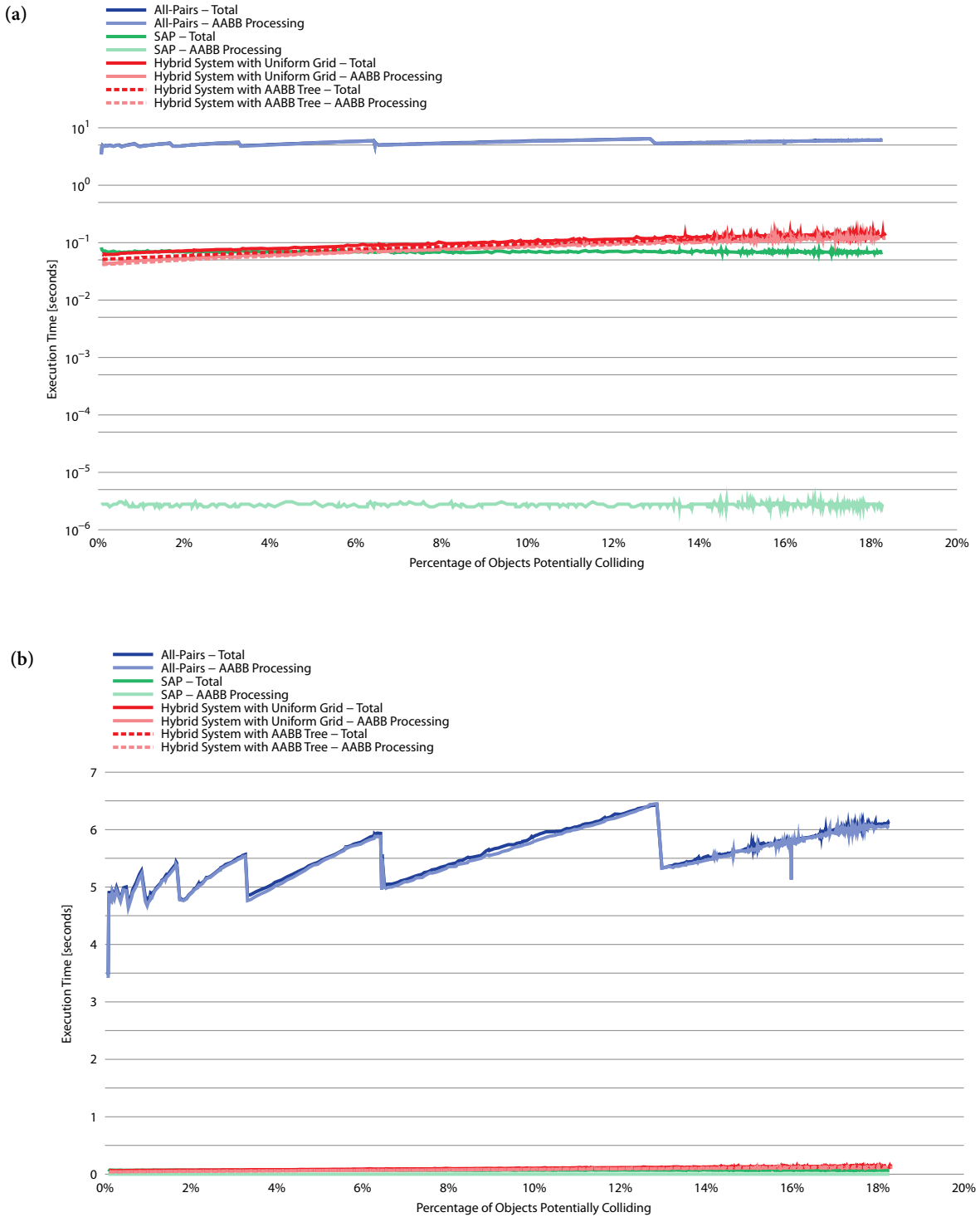


Figure D.50: *Results for the 'Collisions Experiment' of Section 5.4.3 with a cell size of 8192.* **(a)** *Logarithmic scale.* **(b)** *Linear scale.*

Objects	Algorithm	Clock Cycles
16 000	Excluding Communication Interface with Uniform Grid	4 139 501
	Including Communication Interface with Uniform Grid	4 155 838
	Excluding Communication Interface with AABB Tree	4 129 148
	Including Communication Interface with AABB Tree	4 145 493

Table D.26: *Clock cycles for the 'Hardware-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 8192.*

Objects	Algorithm	Execution Time [seconds]
16 000	Excluding Communication Interface with Uniform Grid	0.026
	Including Communication Interface with Uniform Grid	0.05
	Excluding Communication Interface with AABB Tree	0.05
	Including Communication Interface with AABB Tree	0.026

Table D.27: *Execution times for the 'Hardware-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 8192.*

	Excluding Communication Interface		Including Communication Interface	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
		Uniform Grid		
16 000	0.397×	0.029×	0.396×	0.029×
		AABB Tree		
16 000	0.767×	0.057×	0.77×	0.057×

Table D.28: *Accelerations for the 'Hardware-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 8192.*

Objects	Algorithm	Execution Time [seconds]
16 000	Excluding L2 Cache with Uniform Grid	0.02
	Including L2 Cache with Uniform Grid	0.021
	Excluding L2 Cache with AABB Tree	0.003
	Including L2 Cache with AABB Tree	0.003

Table D.29: *Execution times for the 'CPU-Based Execution Time Experiment' of Section 5.4.3 with a cell size of 8192.*

Objects	Algorithm	Percentage of Cycles Consumed by Communication Interface
16 000	Uniform Grid	99.6%
	AABB Tree	99.6%

Table D.30: *The percentage of clock cycles consumed by the communication interface relative to those consumed by the microarchitecture-based algorithm corresponding to Section 5.4.3 with a cell size of 8192.*

D.5 Evaluation

	All-Pairs		SAP	
Objects	Total	AABB Processing	Total	Adapted AABB Processing
Many-Object Microarchitecture				
2000	23.1×	31.3×	1.21×	1.3 ×
4000	28.4×	34.7×	0.909×	0.89 ×
8000	34×	35.2×	0.663×	0.65 ×
16 000	41.5×	39.9×	0.497×	0.513×
Area-Efficient Microarchitecture with Cell Size 1024 and Uniform Grid				
2000	13.125×	20.443×	0.69×	0.5 ×
4000	22.076×	**35.709**×	0.708×	**1.406**×
8000	33.43×	**43.592**×	0.653×	**0.724**×
16 000	0.182×	0.222×	0.017×	0.016×

Table continued on following page.

Table continued from previous page.

Area-Efficient Microarchitecture with Cell Size 1024 and AABB Tree				
2000	7.036×	27.157×	0.37×	0.664×
4000	28.313×	**36.639**×	0.907×	**1.442**×
8000	**39.516**×	**47.081**×	**0.771**×	**0.782**×
Many-Object Microarchitecture with Cell Size 1024 and Uniform Grid				
2000	12.495×	18.506×	0.657×	0.453×
4000	19.959×	29.593×	0.64×	**1.165**×
8000	29.905×	**38.789**×	0.584×	0.645×
16 000	0.168×	0.203×	0.015×	0.015×
Many-Object Microarchitecture with Cell Size 1024 and AABB Tree				
2000	16.144×	23.907×	0.848×	0.585×
4000	25.548×	**60.633**×	0.819×	**2.387**×
8000	**35.446**×	**97.87**×	**0.692**×	**1.626**×
Many-Object Microarchitecture with Cell Size 2048 and Uniform Grid				
4000	20.495×	28.315×	0.657×	**1.115**×
8000	31.406×	**41.037**×	0.613×	**0.682**×
16 000	0.167×	0.198×	0.015×	0.015×
Many-Object Microarchitecture with Cell Size 2048 and AABB Tree				
4000	**28.895**×	**36.493**×	**0.926**×	**1.437**×
8000	**38.803**×	**45.531**×	**0.758**×	**0.757**×
16 000	0.193×	0.211×	0.018×	0.016×
Many-Object Microarchitecture with Cell Size 4096 and Uniform Grid				
8000	29.455×	**36.036**×	0.575×	0.599×
16 000	0.164×	0.189×	0.015×	0.014×
Many-Object Microarchitecture with Cell Size 4096 and AABB Tree				
8000	**38.223**×	**42.13**×	**0.746**×	**0.7**×
16 000	0.189×	0.197×	0.017×	0.015×
Many-Object Microarchitecture with Cell Size 8192 and Uniform Grid				
16 000	0.158×	0.182×	0.014×	0.013×

Table continued on following page.

Table continued from previous page.

Many-Object Microarchitecture with Cell Size 8192 and AABB Tree				
16 000	0.174×	0.188×	0.016×	0.014×

Table D.31: *Accelerations for the 'Software-Based Execution Time Experiments'. Bolded accelerations are greater than those achieved using the many-object microarchitecture without spatial partitioning.*

	Excluding Communication Interface		Including Communication Interface	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
Many-Object Microarchitecture				
2000	81.408×	3.376×	80.117×	3.323×
4000	84.899×	2.177×	84.223×	2.159×
8000	88.09×	1.628×	87.739×	1.621×
16 000	95.604×	1.23×	95.413×	1.227×
Area-Efficient Microarchitecture with Cell Size 1024 and Uniform Grid				
2000	32.718×	0.8×	32.533×	0.8 ×
4000	60.704×	**2.39**×	60.387×	**2.39**×
8000	**113.119**×	**1.88**×	**112.556**×	**1.88**×
16 000	0.792×	0.058×	0.788×	0.058×
Area-Efficient Microarchitecture with Cell Size 1024 and AABB Tree				
2000	**182.363**×	**4.46**×	**176.721**×	**4.46**×
4000	**265.278**×	**10.443**×	**259.426**×	**10.443**×
8000	**366.091**×	**6.084**×	**360.553**×	**6.084**×

Table continued on following page.

Table continued from previous page.

Many-Object Microarchitecture with Cell Size 1024 and Uniform Grid				
2000	29.541×	0.723×	29.355×	0.723×
4000	54.951×	2.163×	54.629×	**2.163**×
8000	**102.285**×	**1.7**×	**101.713**×	**1.7**×
16 000	0.713×	0.053×	0.709×	0.053×
Many-Object Microarchitecture with Cell Size 1024 and AABB Tree				
2000	**181.525**×	**4.44**×	**175.934**×	**4.44**×
4000	**263.406**×	**10.369**×	**257.636**×	**10.369**×
8000	**366.012**×	**6.082**×	**360.477**×	**6.082**×
Many-Object Microarchitecture with Cell Size 2048 and Uniform Grid				
4000	49.877×	1.963×	49.631×	1.963×
8000	**95.331**×	1.584×	**94.88**×	1.584×
16 000	0.67×	0.049×	0.666×	0.049×
Many-Object Microarchitecture with Cell Size 2048 and AABB Tree				
4000	172.708×	6.799×	170.34×	6.799×
8000	**295.535**×	**4.911**×	**292.148**×	**4.911**×
16 000	1.696×	0.125×	1.681×	0.125×
Many-Object Microarchitecture with Cell Size 4096 and Uniform Grid				
8000	76.361×	1.269×	76.089×	1.269×
16 000	0.553×	0.041×	0.551×	0.041×
Many-Object Microarchitecture with Cell Size 4096 and AABB Tree				
8000	**199.232**×	**3.311**×	**197.765**×	**3.311**×
16 000	1.298×	0.096×	1.29×	0.096×
Many-Object Microarchitecture with Cell Size 8192 and Uniform Grid				
16 000	0.397×	0.029×	0.396×	0.029×
Many-Object Microarchitecture with Cell Size 8192 and AABB Tree				
16 000	0.767×	0.057×	0.77×	0.057×

Table D.32: *Accelerations for the 'Hardware-Based Execution Time Experiments'. Bolded accelerations are greater than those achieved using the many-object microarchitecture without spatial partitioning.*

	Excluding L2 Cache		Including L2 Cache	
Objects	All-Pairs AABB Processing	SAP Adapted AABB Processing	All-Pairs AABB Processing	SAP Adapted AABB Processing
Many-Object Microarchitecture				
2000	1221.124×	50.646×	1006.701×	41.753×
4000	1273.488×	32.651×	1150.14×	29.488×
8000	1321.357×	24.414×	1253.884×	23.168×
16 000	1434.056×	18.443×	1396.418×	17.959×
Area-Efficient Microarchitecture with Cell Size 1024 and Uniform Grid				
2000	35.599×	0.871×	35.326×	0.863×
4000	65.563×	2.58×	65.263×	2.565×
8000	122.087×	2.03×	121.289×	2.012×
16 000	0.857×	0.063×	0.853×	0.063×
Area-Efficient Microarchitecture with Cell Size 1024 and AABB Tree				
2000	318.051×	7.78×	301.347×	7.359×
4000	385.531×	15.2×	373.416×	14.675×
8000	461.439×	7.67×	452.248×	7.503×
Many-Object Microarchitecture with Cell Size 1024 and Uniform Grid				
2000	35.932×	0.879×	35.719×	0.872×
4000	65.673×	2.59×	65.364×	2.569×
8000	122.021×	2.03×	121.606×	2.018×
16 000	0.856×	0.063×	0.852×	0.063×
Many-Object Microarchitecture with Cell Size 1024 and AABB Tree				
2000	315.511×	7.72×	298.887×	7.299×
4000	381.589×	15×	369.685×	14.529×
8000	461.314×	7.67×	452.248×	7.503×
Many-Object Microarchitecture with Cell Size 2048 and Uniform Grid				
4000	69.988×	2.76×	69.562×	2.734×
8000	130.334×	2.17×	129.605×	2.150×
16 000	0.917×	0.068×	0.912×	0.067×

Table continued on following page.

Table continued from previous page.

Many-Object Microarchitecture with Cell Size 2048 and AABB Tree				
4000	676.52×	26.6×	640.578×	25.175×
8000	804.593×	13.4×	779.078×	12.926×
16 000	3.284×	0.243×	3.226×	0.238×
Many-Object Microarchitecture with Cell Size 4096 and Uniform Grid				
8000	132.226×	2.197×	131.56×	2.183×
16 000	0.946×	0.07×	0.941×	0.069×
Many-Object Microarchitecture with Cell Size 4096 and AABB Tree				
8000	**1444.885**×	24.01×	**1367**×	22.68 ×
16 000	4.999×	0.369×	4.865×	0.359×
Many-Object Microarchitecture with Cell Size 8192 and Uniform Grid				
16 000	0.955×	0.071×	0.95×	0.07 ×
Many-Object Microarchitecture with Cell Size 8192 and AABB Tree				
16 000	6.896×	0.509×	6.645×	0.491×

Table D.33: *Accelerations for the 'CPU-Based Execution Time Experiments'. Bolded accelerations are greater than those achieved using the many-object microarchitecture without spatial partitioning.*

References

Agarwal, PK, Kaplan, H and Sharir, M. 2008. Kinetic and dynamic data structures for closest pairs and all nearest neighbors. *ACM Transactions on Algorithms 5,* 1 (Nov), 4:1–4:37.

Akgunduz, A, Banerjee, P and Mehrotra, S. 2005. A linear programming solution for exact collision detection. *Journal of Computing and Information Science in Engineering 5,* 1 (Mar), 48–55.

Alpha Data 2002a. *ADC-PMC2 User Manual, Revision 1.0*. Edinburgh, UK.

Alpha Data 2002b. *ADM-XRC-II PCI Mezzanine Card User Guide, Revision 1.5*. Edinburgh, UK.

Alpha Data 2009. *ADM-XRC SDK v4.9.3 User Guide*. Edinburgh, UK.

Altera 2003. *FLEX 10K Embedded Programmable Logic Device Family*. San José, California, USA. Data Sheet.

Altera 2006. *DE2 Development and Education Board User Manual, Revision 1.4*. San José, California, USA.

Altera 2008. *Cyclone II Device Family*. San José, California, USA. Data Sheet.

Amanatides, J and Woo, A. 1987. A fast voxel traversal algorithm for ray tracing. In *Proceedings of the 1987 European Computer Graphics Conference (Eurographics 1987)*. Elsevier, Amsterdam, Netherlands, 3–10.

AMD 2007. *AMD Opteron Processor*. Sunnyvale, California, USA. Data Sheet.

Amdahl, GM. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the 1967 AFIPS Spring Joint Computer Conference*. Thompson, Washington, DC, USA, 483–485.

Appel, A. 1968. Some techniques for shading machine renderings of solids. In *Proceedings of the 1968 AFIPS Spring Joint Computer Conference*. ACM, New York, USA, 37–45.

Atay, N and Bayazit, B. 2006. A motion planning processor on reconfigurable hardware. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation (ICRA 2006)*. IEEE Computer Society, Los Alamitos, California, USA, 125–132.

Atay, N, Lockwood, JW and Bayazit, B. 2005a. A collision detection chip on reconfigurable hardware. In *Proceedings of the 13th Pacific Conference on Computer Graphics and Applications (Pacific Graphics 2005)*. University of Macau, Macau, China. Poster.

Atay, N, Lockwood, JW and Bayazit, B. 2005b. A collision detection chip on reconfigurable hardware. Tech Rep WUCSE-2005-33, Department of Computer Science and Engineering, School of Engineering & Applied Science, Washington University in St Louis, St Louis, Missouri, USA. 22 July.

Avril, Q, Gouranton, V and Arnaldi, B. 2009. New trends in collision detection performance. In *Proceedings of the 2009 Virtual Reality International Conference (VRIC 2009)*. Laval Virtual, Laval, France, 53–62.

Azizi, N, Kuon, I, Egier, A, Darabiha, A and Chow, P. 2004. Reconfigurable molecular dynamics simulator. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004)*. IEEE Computer Society, Los Alamitos, California, USA, 197–206.

Baciu, G, Wong, WS-K and Sun, H. 1998. RECODE: An image-based collision detection algorithm. In *Proceedings of the 6th Pacific Conference on Computer Graphics and Applications (Pacific Graphics 1998)*. IEEE Computer Society, Los Alamitos, California, USA, 125–133.

Bailey, D, Barszcz, E, Barton, J, Browning, D, Carter, R, Dagum, L, Fatoohi, R, Fineberg, S, Frederickson, P, Lasinski, T, Schreiber, R, Simon, H, Venkatakrishnan, V and Weeratunga, S. 1994. The NAS parallel benchmarks. Tech Rep RNR-94-007, NASA Ames Research Center, Moffett Field, California, USA. Mar.

Banta, D, Aldine, B and Bishop, B. 2006. Development and analysis of an interactive physical modeling benchmark set. In *Proceedings of the 2006 International Conference on Modeling, Simulation and Visualization Methods (MSV 2006)*. CSREA, Las Vegas, Nevada, USA, 8–14.

Baraff, D. 1992. Dynamic simulation of non-penetrating rigid bodies. PhD thesis, Cornell University, Ithaca, New York, USA.

Baraff, D. 1995. Interactive simulation of solid rigid bodies. *IEEE Computer Graphics and Applications 15,* 3 (May), 63–75.

Barzel, R, Hughes, JF and Wood, DN. 1996. Plausible motion simulation for computer graphics animation. In *Proceedings of the 7th Eurographics Workshop on Computer Animation and Simulation (EGCAS 1996)*. Springer, New York, USA, 183–197.

Basch, J, Erickson, J, Guibas, LJ, Hershberger, J and Zhang, L. 2004. Kinetic collision detection between two simple polygons. *Computational Geometry 27,* 3 (Mar), 211–235.

Beechick, A, Casselman, S and Yarbrough, L. 1996. Internal sorting and FPGA. *Proceedings of SPIE – High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic 2914*, 66–71.

Beeckler, JS and Gross, WJ. 2005. FPGA particle graphics hardware. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005)*. IEEE Computer Society, Los Alamitos, California, USA, 85–94.

Beeckler, JS and Gross, WJ. 2008. Particle graphics on reconfigurable hardware. *ACM Transactions on Reconfigurable Technology and Systems 1,* 3 (Sept), 15:1–15:27.

Benitez, D. 2002. A quantitative understanding of the performance of reconfigurable coprocessors. *Lecture Notes in Computer Science – Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL 2002) 2438*, 976–986.

Bentley, JL. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18,* 9 (Sept), 509–517.

Bentley, JL, Stanat, DF and Hollins Williams Jr, E. 1977. The complexity of finding fixed-radius near neighbors. *Information Processing Letters 6,* 6 (Dec), 209–212.

BIPM and *Organisation intergouvernementale de la Convention du Mètre* 2006. *Le Système international d'unités (SI), Revision 8*. Sèvres, France.

Bishop, B and Kelliher, TP. 2001. Hardware acceleration for physical modeling of deformable objects. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2001): Sketches & Applications*. ACM, New York, USA, 193. Sketch.

Bishop, B and Kelliher, TP. 2003. Specialized hardware for deformable object modeling. *IEEE Transactions on Circuits and Systems for Video Technology 13,* 11 (Nov), 1074–1079.

Bishop, B, Kelliher, TP and Irwin, MJ. 2000a. Hardware/software co-design for real-time physical modeling. In *Proceedings of the 2000 IEEE International Conference on Multimedia and Expo (ICME 2000)*. Vol 3. IEEE, Piscataway, New Jersey, USA, 1363–1366.

Bishop, B, Kelliher, TP and Irwin, MJ. 2000b. SPARTA: Simulation of physics on a real-time architecture. In *Proceedings of the 10th Great Lakes Symposium on VLSI (GLSVLSI 2000)*. ACM, New York, USA, 177–182.

Boeing, A and Bräunl, T. 2007. Evaluation of real-time physics simulation systems. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia (GRAPHITE 2007)*. ACM, New York, USA, 281–288.

Boost 2008. *Boost v1.36.0 Library Documentation*.

Boost 2009. *Boost v1.40.0 Library Documentation*.

Bordes, JP, Davis, C and Hedge, M. 2006. System with PPU/GPU architecture. International Patent WO/2006/055279.

Bordes, JP and Sequeira, D. 2006. Asynchronous and parallel execution by physics processing unit. International Patent WO/2006/052750.

Brown, J, Sorkin, S, Bruyns, C, Latombe, J-C, Montgomery, K and Stephanides, M. 2001. Real-time simulation of deformable objects: Tools and application. In *Proceedings of the 14th Conference on Computer Animation (CA 2001)*. IEEE Computer Society, Los Alamitos, California, USA, 228–258.

Cameron, S. 1997. Enhancing GJK: Computing minimum and penetration distances between convex polyhedra. In *Proceedings of the 1997 IEEE International Conference on Robotics and Automation (ICRA 1997)*. Vol 4. IEEE, Piscataway, New Jersey, USA, 3112–3117.

Canonical 2008. *Documentation for Ubuntu v8.04 LTS*. London, UK.

Cao, XR. 2006. A framework for benchmarking haptic systems. Masters thesis, Simon Fraser University, Burnaby, British Columbia, Canada.

Caselli, S, Reggiani, M and Mazzoli, M. 2002. Exploiting advanced collision detection libraries in a probabilistic motion planner. *Journal of WSCG – Proceedings of the 10th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2002) 10,* 1–3 (4–8 Feb), 103–109.

Cazals, F, Drettakis, G and Puech, C. 1995. Filtering, clustering and hierarchy construction: a new solution for ray-tracing complex scenes. *Computer Graphics Forum – Proceedings of the 16th Annual Conference of the European Association for Computer Graphics (Eurographics 1995) 14,* 3 (28 Aug–1 Sept), 371–382.

Cazals, F and Puech, C. 1997. Bucket-like space partitioning data structures with applications to ray-tracing. In *Proceedings of the 13th Annual Symposium on Computational Geometry (SCG 1997)*. ACM, New York, USA, 11–20.

Che, S, Li, J, Sheaffer, JW, Skadron, K and Lach, J. 2008. Accelerating compute-intensive applications with GPUs and FPGAs. In *Proceedings of the 6th IEEE Symposium on Application Specific Processors (SASP 2008)*. IEEE Computer Society, Los Alamitos, California, USA, 101–107.

Chen, T, Raghavan, R, Dale, JN and Iwata, E. 2007. Cell broadband engine architecture and its first implementation—A performance view. *IBM Journal of Research and Development 51,* 5 (Sept), 559–572.

Chen, Y-K, Chhugani, J, Hughes, CJ, Kim, D, Kumar, S, Lee, V, Lin, A, Nguyen, AD, Sifakis, E and Smelyanskiy, M. 2007. High-performance physical simulations on next-generation architecture with many cores. *Intel Technology Journal 11,* 3 (22 Aug), 251–261.

Chung, K and Wang, W. 1996. Quick collision detection of polytopes in virtual environments. In *Proceedings of the 3rd ACM Symposium on Virtual Reality Software and Technology (VRST 1996)*. ACM, New York, USA, 125–131.

Cohen, JD, Lin, MC, Manocha, D and Ponamgi, MK. 1995. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics (SI3D 1995)*. ACM, New York, USA, 189–196.

Coombe, G, Harris, MJ and Lastra, A. 2004. Radiosity on graphics hardware. In *Proceedings of Graphics Interface 2004 (GI 2004)*. Canadian Human-Computer Communications Society, Waterloo, Ontario, Canada, 161–168.

Coumans, E. 2009. *Bullet Physics SDK v2.75 Manual.*

Crosnier, A and Rossignac, JR. 1999. Tribox bounds for three-dimensional objects. *Computers & Graphics 23,* 3 (1 June), 429–437.

Curtis, S, Tamstorf, R and Manocha, D. 2008. Fast collision detection for deformable models using representative-triangles. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games (I3D 2008)*. ACM, New York, USA, 61–69.

Davis, C, Hedge, M, Schmid, OA, Maher, M and Bordes, JP. 2005a. Method for providing physics simulation data. International Patent WO/2005/038560.

Davis, C, Hedge, M, Schmid, OA, Maher, M and Bordes, JP. 2005b. Physics processing unit. International Patent WO/2005/038559.

Davis, C, Hedge, M, Schmid, OA, Maher, M and Bordes, JP. 2005c. System incorporating physics processing unit. International Patent WO/2005/038561.

Deb, A. 1996. Multiskewing—A novel technique for optimal parallel memory access. *IEEE Transactions on Parallel and Distributed Systems 7,* 6 (June), 595–604.

Dell 2002. *Dell PowerEdge 600SC Systems User's Guide*. Round Rock, Texas, USA.

Dingliana, J and O'Sullivan, C. 2000. Graceful degradation of collision handling in physically based animation. *Computer Graphics Forum – Proceedings of the 21st Annual Conference of the European Association for Computer Graphics (Eurographics 2000) 19,* 3 (21–25 Aug), C-239–C-247.

Dobkin, DP and Kirkpatrick, DG. 1990. Determining the separation of preprocessed polyhedra – A unified approach. *Lecture Notes in Computer Science – Proceedings of the 17th International Colloquium on Automata, Programming and Languages (ICALP 1990) 443*, 400–413.

DRC Computer 2008a. *Accelium Appliance Platform*. Sunnyvale, California, USA. Data Sheet.

DRC Computer 2008b. *Accelium Coprocessors*. Sunnyvale, California, USA. Data Sheet.

DRC Computer 2009. *DRC Coprocessor System User's Guide, Revision 3.1.1*. Sunnyvale, California, USA.

Ehmann, SA and Lin, MC. 2000. Accelerated proximity queries between convex polyhedra by multi-level Voronoi marching. In *Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000)*. Vol 3. IEEE, Piscataway, New Jersey, USA, 2101–2106.

Eitz, M and Lixu, G. 2007. Hierarchical spatial hashing for real-time collision detection. In *Proceedings of the 2007 IEEE International Conference on Shape Modeling and Applications (SMI 2007)*. IEEE Computer Society, Los Alamitos, California, USA, 61–70.

Ericson, C. 2005. *Real-Time Collision Detection*. The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann, San Francisco, California, USA.

Erleben, K. 2005. Stable, robust, and versatile multibody dynamics animation. PhD thesis, University of Copenhagen, Copenhagen, Denmark.

Fares, C and Hamam, Y. 2005. Collision detection for rigid bodies: A state of the art review. In *Proceedings of the 15th International Conference on Computer Graphics and Applications (GraphiCon 2005)*. Institute of Computational Mathematics and Mathematical Geophysics SB RAS, Novosibirsk, Russia.

Foisy, A, Hayward, V and Aubry, S. 1990. The use of awareness in collision prediction. In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation (ICRA 1990)*. Vol 1. IEEE Computer Society, Los Alamitos, California, USA, 338–343.

Foley, T and Sugerman, J. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the 20th ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (GH 2005)*. ACM, New York, USA, 15–22.

Fowles, GR. 1985. *Analytical Mechanics*, fourth ed. Saunders Golden Sunburst. Saunders College, Philadelphia, Pennsylvania, USA.

Free Software Foundation 2008. *Using the GNU Compiler Collection v4.2.4*.

freeglut Programming Consortium 2003. *The Open-Source OpenGL Utility Toolkit (freeglut) v2.0.0 Application Programming Interface*.

Friedman, JH, Bentley, JL and Finkel, RA. 1977. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software 3*, 3 (Sept), 209–226.

Frisken, SF and Perry, RN. 2002. Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools 7,* 3, 1–11.

Frisken, SF, Perry, RN, Rockwood, AP and Jones, TR. 2000. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2000)*. ACM, New York, USA, 249–254.

Fuhrmann, A, Sobottka, G and Gross, C. 2005. Distance fields for rapid collision detection in physically based modeling. In *Proceedings of the 13th International Conference on Computer Graphics and Applications (GraphiCon 2003)*. Institute of Computational Mathematics and Mathematical Geophysics SB RAS, Novosibirsk, Russia.

Fujita, O and Jinya, K. 2009. Fine grained parallel processing for soft computing. *Lecture Notes in Computer Science – Proceedings of the 13th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems (KES 2009) 5712*, 767–772.

Futuremark 2009. *3DMark Vantage v1.0.1.*

Gaisler, J. 1997. *A structured VHDL design method.* Gaisler Research, Gothenburg, Sweden, 41–50.

García-Alonso, A, Serrano, N and Flaquer, J. 1994. Solving the collision detection problem. *IEEE Computer Graphics and Applications 14,* 3 (May), 36–43.

Gilbert, EG and Foo, C-P. 1990. Computing the distance between general convex objects in three-dimensional space. *IEEE Transactions on Robotics and Automation 6,* 1 (Feb), 53–61.

Gilbert, EG, Johnson, DW and Keerthi, SS. 1988. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Animation 4,* 2 (Apr), 193–203.

Gochman, S, Ronen, R, Anati, I, Berkovits, A, Kurts, T, Naveh, A, Saeed, A, Sperber, Z and Valentine, RC. 2003. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal 7,* 2 (21 May), 21–36.

Gothandaraman, A, Peterson, GD, Warren, GL, Hinde, RJ and Harrison, RJ. 2009. A Hardware-Accelerated Quantum Monte Carlo framework (HAQMC) for N-body systems. *Computer Physics Communications 180,* 12 (Dec), 2563–2573.

Gottschalk, S. 2000. Collision queries using oriented bounding boxes. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, USA.

Gottschalk, S, Lin, MC and Manocha, D. 1996. OBBTree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1996)*. ACM, New York, USA, 171–180.

Govindaraju, NK, Kabul, I, Lin, MC and Manocha, D. 2007. Fast continuous collision detection among deformable models using graphics processors. *Computers & Graphics 31,* 1 (Jan), 5–14.

Govindaraju, NK, Knott, D, Jain, N, Kabul, I, Tamstorf, R, Gayle, R, Lin, MC and Manocha, D. 2005. Interactive collision detection between deformable models using chromatic decomposition. *ACM Transactions on Graphics – Proceedings of the 32nd International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2005) 24,* 3 (31 July–4 Aug), 991–999.

Govindaraju, NK, Lin, MC and Manocha, D. 2005. Quick-CULLIDE: Fast inter- and intra-object collision culling using graphics hardware. In *Proceedings of the 2005 IEEE Conference on Virtual Reality (VR 2005)*. IEEE Computer Society, Los Alamitos, California, USA, 59–66 and 319.

Govindaraju, NK, Redon, S, Lin, MC and Manocha, D. 2003. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings of the 18th ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (GH 2003)*. Eurographics Association, Aire-la-Ville, Switzerland, 25–32.

Govindu, G, Zhuo, L, Choi, S and Prasanna, V. 2004. Analysis of high-performance floating-point arithmetic on FPGAs. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*. IEEE Computer Society, Los Alamitos, California, USA, Paper 149.

Gress, A, Guthe, M and Klein, R. 2006. GPU-based collision detection for deformable parameterized surfaces. *Computer Graphics Forum – Proceedings of the 27th Annual Conference of the European Association for Computer Graphics (Eurographics 2006) 25,* 3 (4–8 Sept), 497–506.

Gress, A and Zachmann, G. 2003. Object-space interference detection on programmable graphics hardware. In *Proceedings of the 2003 SIAM Conference on Geometric Design and Computing (GD 2003)*. Nashboro, Brentwood, Tennessee, USA, 311–328.

Guendelman, E, Bridson, R and Fedkiw, R. 2003. Nonconvex rigid bodies with stacking. *ACM Transactions on Graphics – Proceedings of the 30th International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2003) 22,* 3 (27–31 July), 871–878.

Guibas, LJ, Hsu, D and Zhang, L. 1999. *H-Walk*: Hierarchical distance computation for moving convex bodies. In *Proceedings of the 15th Annual Symposium on Computational Geometry (SCG 1999)*. ACM, New York, USA, 265–273.

Guibas, LJ, Nguyen, A and Zhang, L. 2003. Zonotopes as bounding volumes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*. SIAM, Philadelphia, Pennsylvania, USA, 803–812.

Günther, J, Popov, S, Seidel, H-P and Slusallek, P. 2007. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing 2007 (RT 2007)*. Nashboro, Brentwood, Tennessee, USA, 113–118.

Guo, Z, Najjar, W, Vahid, F and Vissers, K. 2004. A quantitative analysis of the speedup factors of FPGAs over processors. In *Proceedings of the 12th International Symposium on Field Programmable Gate Arrays (FPGA 2004)*. ACM, New York, USA, 162–170.

Guy, S and Debunne, G. 2004. Monte-Carlo collision detection. Tech Rep RR-5136, INRIA, Rocquencourt, France. Mar.

Haines, E. 1987. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications 7,* 11 (Nov), 3–5.

Hamlin, GJ, Kelley, RB and Tornero, J. 1992. Efficient distance calculation using the spherically-extended polytope (s-tope) model. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation (ICRA 1992)*. Vol 3. IEEE, Piscataway, New Jersey, USA, 2502–2507.

Hasselgren, J and Akenine-Möller, T. 2007. PCU: The programmable culling unit. *ACM Transactions on Graphics – Proceedings of the 34th International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2007) 26,* 3 (5–9 Aug), 92-1–92-10.

He, T. 1999. Fast collision detection using QuOSPO trees. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics (I3D 1999)*. ACM, New York, USA, 55–62.

Heidelberger, B, Teschner, M and Gross, M. 2003. Real-time volumetric intersections of deforming objects. In *Proceedings of the 8th International Fall Workshop on Vision, Modeling, and Visualization (VMV 2003)*. AKA, Heidelberg, Germany, 461–468.

Heidelberger, B, Teschner, M and Gross, M. 2004. Detection of collisions and self-collisions using image-space techniques. *Journal of WSCG – Proceedings of the 12th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2004) 12,* 1–3 (2–6 Feb), 145–152.

Heinzle, S, Guennebaud, G, Botsch, M and Gross, M. 2008. A hardware processing unit for point sets. In *Proceedings of the 23rd ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (GH 2008)*. Eurographics Association, Aire-la-Ville, Switzerland, 21–31.

Held, M. 1997. ERIT: A collection of efficient and reliable intersection tests. *Journal of Graphics Tools 2,* 4, 25–44.

Held, M, Klosowski, JT and Mitchell, JSB. 1995. Evaluation of collision detection methods for virtual reality fly-throughs. In *Proceedings of the 7th Canadian Conference on Computational Geometry (CCCG 1995)*. Université Laval, Laval, Québec, Canada, 205–210.

Held, M, Klosowski, JT and Mitchell, JSB. 1996. Real-time collision detection for motion simulation within complex environments. In *Visual Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1996)*. ACM, New York, USA, 151. Technical Sketch.

Hettinger, LJ and Riccio, GE. 1992. Visually induced motion sickness in virtual environments. *Presence: Teleoperators and Virtual Environments 1,* 3, 306–310.

HIS 2008. *HIS X1300 Dual DVI 256MB (64bit) DDR2 PCI*. Hong Kong. http://www.hisdigital.com/un/product2-210.shtml.

Hochgürtel, S, Raabe, A, Anlauf, JK and Zachmann, G. 2005. Collision detection for k-DOPs using SAT with error bounded fixed-point arithmetic. Tech Rep TR-TI-2005-1, Institut für Informatik II, University of Bonn, Bonn, Germany. 5 Oct.

Hopcroft, JE, Schwartz, JT and Sharir, M. 1983. Efficient detection of intersections among spheres. *The International Journal of Robotics Research 2,* 4, 77–80.

Horn, D. 2005. Stream reduction operations for GPGPU applications. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, M Pharr, Ed. Addison-Wesley, Upper Saddle River, New Jersey, USA, Chapter 36, 573–589.

Horn, DR, Sugerman, J, Houston, M and Hanrahan, P. 2007. Interactive k-D tree GPU raytracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games (I3D 2007)*. ACM, New York, USA, 167–174.

Hubbard, PM. 1993a. Interactive collision detection. In *Proceedings of the 1993 IEEE Symposium on Research Frontiers in Virtual Reality (VR 1993)*. IEEE Computer Society, Los Alamitos, California, USA, 24–32.

Hubbard, PM. 1993b. Space-time bounds for collision detection. Tech Rep CS-93-04, Department of Computer Science, Brown University, Providence, Rhode Island, USA. 27 Jan.

Hubbard, PM. 1996. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics 15,* 3 (July), 179–210.

Hudson, TC, Lin, MC, Cohen, J, Gottschalk, S and Manocha, D. 1997. V-COLLIDE: Accelerated collision detection for VRML. In *Proceedings of the 2nd Symposium on the Virtual Reality Modeling Language (VRML 1997)*. ACM, New York, USA, 117–123.

HyperTransport Consortium 2009. *HyperTransport I/O Link Specification, Revision 3.10b*. Sunnyvale, California, USA.

IEEE 2006. *Verilog Hardware Description Language, Standard 1364, Revision 2005*. New York, USA.

IEEE 2008. *Floating-Point Arithmetic, Standard 754, Revision 2005*. New York, USA.

IEEE 2009. *VHDL Language Reference Manual, Standard 1076, Revision 2008*. New York, USA.

Intel 2002. *Intel Pentium 4 Processor in the 478-Pin Package at 1.40 GHz, 1.50 GHz, 1.60 GHz, 1.70 GHz, 1.80 GHz, 1.90 GHz, and 2 GHz*. Santa Clara, California, USA. Data Sheet.

Intel 2009. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*. Santa Clara, California, USA.

ISO and IEC 2003. *Programming languages – C++, Standard 14882, Revision 2003(E)*. Geneva, Switzerland.

ISO and IEC 2005. *C++ Library Extensions, Draft Tech Rep 19768*. Geneva, Switzerland.

Jang, H-Y, Jeong, TS and Han, JH. 2007. Image-space collision detection through alternate surface peeling. *Lecture Notes in Computer Science – Proceedings of the 3rd International Symposium on Visual Computing (ISVC 2007) 4841*, 66–75.

Jevans, D and Wyvill, B. 1989. Adaptive voxel subdivision for ray tracing. In *Proceedings of Graphics Interface 1989 (GI 1989)*. CIPS, Toronto, Ontario, Canada, 164–172.

Jiménez, P, Thomas, F and Torras, C. 2001. 3D collision detection: a survey. *Computers & Graphics 25,* 2 (Apr), 269–285.

Jin, H, Liu, Z, Wu, T and Wang, Y. 2009. The research of collision detection algorithm based on spatial subdivision. In *Proceedings of the 2009 International Conference on Computer Engineering and Technology (ICCET 2009)*. Vol 2. IEEE Computer Society, Los Alamitos, California, USA, 452–455.

Jones, MW, Bærentzen, JA and Sramek, M. 2006. 3D distance fields: A survey of techniques and applications. *IEEE Transactions on Visualization and Computer Graphics 12,* 4 (July–Aug), 581–599.

Jonkman, RJT. 1994. NetSpec: Philosophy, design and implementation. Masters thesis, University of Kansas, Lawrence, Kansas, USA.

Kim, D, Heo, J-P, Huh, J, Kim, J and Yoon, S-E. 2009. HPCCD: Hybrid parallel continuous collision detection using CPUs and GPUs. *Computer Graphics Forum – Proceedings of the 17th Pacific Conference on Computer Graphics and Animation (Pacific Graphics 2009) 28,* 7 (7–9 Oct), 1791–1800.

Kim, JS, Deng, L, Mangalagiri, P, Irick, K, Sobti, K, Kandemir, M, Narayanan, V, Chakrabarti, C, Pitsianis, N and Sun, X. 2009. An automated framework for accelerating numerical algorithms on reconfigurable platforms using algorithmic/architectural optimization. *IEEE Transactions on Computers 58,* 12 (Dec), 1654–1667.

Kipfer, P, Segal, M and Westermann, R. 2004. UberFlow: A GPU-based particle engine. In *Proceedings of the 19th ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (GH 2004)*. ACM, New York, USA, 115–122.

Klein, J and Zachmann, G. 2003. ADB-trees: Controlling the error of time-critical collision detection. In *Proceedings of the 8th International Fall Workshop on Vision, Modeling, and Visualization (VMV 2003)*. AKA, Heidelberg, Germany, 37–45.

Klimaszewski, KS and Sederburg, TW. 1997. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications 17,* 1 (Jan–Feb), 42–51.

Knittel, G. 1996. A PCI-compatible FPGA-coprocessor for 2D/3D image processing. In *Proceedings of the 1996 IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE Computer Society, Los Alamitos, California, USA, 136–145.

Knittel, G. 2007. A compact shader for FPGA-based volume rendering accelerators. *Lecture Notes in Computer Science – Proceedings of the 3rd International Workshop on Applied Reconfigurable Computing (ARC 2007) 4419*, 271–282.

Knittel, G and Zachmann, G. 2003. High-performance collision detection hardware. Tech Rep CG-2003-3, Institut für Informatik II, University of Bonn, Bonn, Germany. 25 Aug.

Knott, D and Pai, DK. 2003. CInDeR: Collision and interference detection in real-time using graphics hardware. In *Proceedings of Graphics Interface 2003 (GI 2003)*. Canadian Human-Computer Communications Society, Waterloo, Ontario, Canada, 73–80.

Kockara, S, Halic, T, Iqbal, K, Bayrak, C and Rowe, R. 2007. Collision detection: A survey. In *Proceedings of the 2007 IEEE International Conference on Systems, Man and Cybernetics (SMC 2007)*. IEEE, Piscataway, New Jersey, USA, 4046–4051.

Kolb, A, Latta, L and Rezk-Salama, C. 2004. Hardware-based simulation and collision detection for large particle systems. In *Proceedings of the 19th ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (GH 2004)*. ACM, New York, USA, 123–131.

Krishnamurthy, V and Levoy, M. 1996. Fitting smooth surfaces to dense polygon meshes. In *Proceedings of the 23rd International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1996)*. ACM, New York, USA, 313–324.

Krishnan, S, Pattekar, A, Lin, MC and Manocha, D. 1998. Spherical shell: A higher order bounding volume for fast proximity queries. In *Proceedings of the 3rd Workshop on the Algorithmic Foundations of Robotics (WAFR 1998)*. AK Peters, Natick, Massachusetts, USA, 177–190.

Kumar, VBY, Joshi, S, Patkar, SB and Narayanan, H. 2009. FPGA based high performance double-precision matrix multiplication. In *Proceedings of the 22nd International Conference on VLSI Design (VLSID 2009)*. IEEE Computer Society, Los Alamitos, California, USA, 341–346.

Kuon, I and Rose, J. 2007. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 26,* 2 (Feb), 203–215.

Larsen, E, Gottschalk, S, Lin, MC and Manocha, D. 1999. Fast proximity queries with swept sphere volumes. Tech Rep TR99-018, Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, USA.

Larsen, E, Gottschalk, S, Lin, MC and Manocha, D. 2000. Fast distance queries with rectangular swept sphere volumes. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation (ICRA 2000)*. Vol 4. IEEE, Piscataway, New Jersey, USA, 3719–3726.

Larsson, T and Akenine-Möller, T. 2001. Collision detection for continuously deforming bodies. In *Proceedings of the 22nd Annual Conference of the European Association for Computer Graphics (Eurographics 2001): Short Presentations*. Eurographics Association, Aire-la-Ville, Switzerland, 325–333. Short Presentation.

Lauterbach, C, Garland, M, Sengupta, S, Luebke, D and Manocha, D. 2009. Fast BVH construction on GPUs. *Computer Graphics Forum – Proceedings of the 30th Annual Conference of the European Association for Computer Graphics (Eurographics 2009) 28,* 2 (30 Mar–3 Apr), 375–384.

Lawrie, DH and Vora, CR. 1982. The prime memory system for array access. *IEEE Transactions on Computers C-31,* 5 (May), 435–442.

Le Grand, S. 2007. Broad-phase collision detection with CUDA. In *GPU Gems 3*, H Nguyen, Ed. Addison-Wesley, Upper Saddle River, New Jersey, USA, Chapter 32, 697–721.

Lee, B-O. 1995. Wide area ATM network experiments using emulated traffic sources. Masters thesis, University of Kansas, Lawrence, Kansas, USA.

Lefebvre, S and Hoppe, H. 2006. Perfect spatial hashing. *ACM Transactions on Graphics – Proceedings of the 33rd International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2006) 25,* 3 (30 July–3 Aug), 579–588.

Lefebvre, S, Hornus, S and Neyret, F. 2005. Octree textures on the GPU. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, M Pharr, Ed. Addison-Wesley, Upper Saddle River, New Jersey, USA, Chapter 37, 595–613.

Lefohn, AE, Kniss, J, Strzodka, R, Sengupta, S and Owens, JD. 2006. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics 26,* 1 (Jan), 60–99.

Lext, J, Assarsson, U and Möller, T. 2001. A benchmark for animated ray tracing. *IEEE Computer Graphics and Applications 21,* 2 (Mar–Apr), 22–31.

Lin, MC and Canny, JF. 1991. A fast algorithm for incremental distance calculation. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation (ICRA 1991)*. Vol 2. IEEE, Piscataway, New Jersey, USA, 1008–1014.

Lin, MC and Gottschalk, S. 1998. Collision detection between geometric models: a survey. In *Proceedings of the 8th IMA Conference on the Mathematics of Surfaces*. IMA, Southend-on-Sea, Essex, UK, 37–56.

Lin, MC and Manocha, D. 2004. Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*, second ed, JE Goodman and J O'Rourke, Eds. Chapman & Hall/CRC, Boca Raton, Florida, USA, Chapter 35, 787–807.

Liu, R, Zhang, H and Busby, J. 2008. Convex hull covering of polygonal scenes for accurate collision detection in games. In *Proceedings of Graphics Interface 2008 (GI 2008)*. Canadian Human-Computer Communications Society, Waterloo, Ontario, Canada, 203–210.

Luque, RG, Comba, JLD and Freitas, CMDS. 2005. Broad-phase collision detection using semi-adjusting BSP-trees. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games (I3D 2005)*. ACM, New York, USA, 179–186.

Maher, M, Bordes, JP, Sequeira, D and Tonge, R. 2005. Physics processing unit instruction set architecture. International Patent WO/2005/111831.

Makino, J. 2006. The GRAPE project. *Computing in Science & Engineering 8,* 1 (Jan–Feb), 30–40.

Manegold, S and Boncz, P. 2000. Cache-memory and TLB calibration tool.

Matković, K, Gračanin, D, Jelović, M and Hauser, H. 2008. Interactive visual steering – Rapid visual prototyping of a common rail injection system. *IEEE Transactions on Visualization and Computer Graphics 14,* 6 (Nov–Dec), 1699–1706.

Matoušek, R, Tichý, M, Pohl, Z, Kadlec, J, Softley, C and Coleman, N. 2002. Logarithmic number system and floating-point arithmetics on FPGA. *Lecture Notes in Computer Science – Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL 2002) 2438*, 175–188.

Mendoza, C and O'Sullivan, C. 2006. Interruptible collision detection for deformable objects. *Computers & Graphics 30,* 3 (June), 432–438.

Mezger, J, Kimmerle, S and Etzmuss, O. 2003. Hierarchical techniques in collision detection for cloth animation. *Journal of WSCG – Proceedings of the 11th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2003) 11,* 1–3 (3–7 Feb), 322–329.

Microsoft 2009a. *Visual Studio 2008 Help and Support*. Redmond, Washington, USA.

Microsoft 2009b. *Windows XP*. Redmond, Washington, USA. http://www.microsoft.com/windows/windows-xp/.

Mirtich, BV. 1996. Impulse-based dynamic simulation of rigid body systems. PhD thesis, University of California at Berkeley, Berkeley, California, USA.

Mirtich, B. 1998a. Efficient algorithms for two-phase collision detection. In *Practical Motion Planning in Robotics: Current Approaches and Future Directions*, K Gupta and AP del Pobil, Eds. John Wiley & Sons, Chichester, UK, Chapter 10, 203–223.

Mirtich, B. 1998b. V-Clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics 17,* 3 (July), 177–208.

Mirtich, B. 2000. Timewarp rigid body simulation. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2000)*. ACM, New York, USA, 193–200.

Moll, L, Heirichy, A and Shand, M. 1999. Sepia: scalable 3D compositing using PCI Pamette. In *Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 1999)*. IEEE Computer Society, Los Alamitos, California, USA, 146–155.

Moll, L and Shand, M. 1997. Systems performance measurement on PCI Pamette. In *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 1997)*. IEEE Computer Society, Los Alamitos, California, USA, 125–133.

Möller, T. 1997. A fast triangle-triangle intersection test. *Journal of Graphics Tools 2,* 2, 25–30.

Munshi, A. 2008. *The OpenCL Specification, Revision 1.0.29*. Beaverton, Ohio, USA.

Myszkowski, K, Okunev, OG and Kunii, TL. 1995. Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer 11,* 9 (Nov), 497–511.

NVIDIA 2009a. *GeForce 8 Series*. Santa Clara, California, USA. http://www.nvidia.com/page/geforce8.html.

NVIDIA 2009b. *GeForce GTX 285*. Santa Clara, California, USA. http://www.nvidia.com/object/product_geforce_gtx_285_us.html.

NVIDIA 2009c. *NVIDIA CUDA v2.3 Reference Manual*. Santa Clara, California, USA.

O'Sullivan, C, Radach, R and Collins, S. 1999. A model of collision perception for real-time animation. In *Proceedings of the 1999 Eurographics Workshop on Computer Animation and Simulation (CAS 1999)*. Springer, Vienna, Austria, 67–76.

Owens, JD, Luebke, D, Govindaraju, N, Harris, M, Krüger, J, Lefohn, AE and Purcell, TJ. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26,* 1 (Mar), 80–113.

Palmer, IJ and Grimsdale, RL. 1995. Collision detection for animation using sphere-trees. *Computer Graphics Forum 14,* 2 (June), 105–116.

Patel, A, Madill, CA, Saldaña, M, Comis, C, Pomès, R and Chow, P. 2006. A scalable FPGA-based multiprocessor. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2006)*. IEEE Computer Society, Los Alamitos, California, USA, 111–120.

Preparata, FP and Shamos, MI. 1985. *Computational Geometry: An Introduction*. Texts and Monographs in Computer Science. Springer, New York, USA.

Purcell, TJ. 2004. Ray tracing on a stream processor. PhD thesis, Stanford University, Stanford, California, USA.

Purcell, TJ, Buck, I, Mark, WR and Hanrahan, P. 2002. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21,* 3 (July), 703–712.

Quinlan, S. 1994. Efficient distance computation between non-convex objects. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation (ICRA 1994)*. Vol 4. IEEE Computer Society, Los Alamitos, California, USA, 3324–3329.

Raabe, A. 2008. Describing and simulating dynamic reconfiguration in SystemC exemplified by a dedicated 3D collision detection hardware. PhD thesis, University of Bonn, Bonn, Germany.

Raabe, A, Bartyzel, B, Anlauf, JK and Zachmann, G. 2005. Hardware accelerated collision detection – An architecture and simulation results. In *Proceedings of the 2005 Design, Automation and Test in Europe Conference and Exhibition (DATE 2005)*. Vol 3. IEEE Computer Society, Los Alamitos, California, USA, 130–135.

Raabe, A, Hochgürtel, S, Anlauf, JK and Zachmann, G. 2006a. Hardware-accelerated collision detection using bounded-error fixed-point arithmetic. *Journal of WSCG – Proceedings of the 14th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2006) 14,* 1–3 (30 Jan–3 Feb), 17–24.

Raabe, A, Hochgürtel, S, Anlauf, J and Zachmann, G. 2006b. Space-efficient FPGA-accelerated collision detection for virtual prototyping. In *Proceedings of the 2006 Design, Automation and Test in Europe Conference and Exhibition (DATE 2006)*. Vol 2. European Design and Automation Association, Leuven, Belgium, 206–211.

Raabe, A, Nett, A and Niers, AH. 2008. A refinement case-study of a dynamically reconfigurable intersection test hardware. In *Proceedings of the 4th International Workshop on Reconfigurable Communication Centric Systems-on-Chips (ReCoSoC 2008)*. Polytechnic University of Catalonia, Barcelona, Spain.

Raabe, A, Tietjen, J and Anlauf, JK. 2009. EFSAT: An exact and efficient triangle intersection test hardware. In *Proceedings of the 4th International Conference on Computer Graphics Theory and Applications (GRAPP 2009)*. INSTICC, Barcelona, Spain, 355–360.

Raabe, A and Zavelberg, F. 2008. Defying the memory bottleneck in hardware accelerated collision detection. In *Proceedings of the 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2008) – Communication Papers*. UNION Agency-Science, Plzeň, Czech Republic, 25–31. Short Paper.

Raghupathi, L, Grisoni, L, Faure, F, Marchall, D, Cani, M-P and Chaillou, C. 2004. An intestine surgery simulator: Real-time collision processing and visualization. *IEEE Transactions on Visualization and Computer Graphics 10,* 6 (Nov–Dec), 708–718.

Redon, S, Lin, MC, Manocha, D and Kim, YJ. 2005. Fast continuous collision detection for articulated models. *Journal of Computing and Information Science in Engineering 5,* 2 (June), 126–137.

Rimon, E and Boyd, SP. 1992. Efficient distance computation using best ellipsoid fit. In *Proceedings of the 1992 IEEE International Symposium on Intelligent Control (ISIC 1992)*. IEEE, Piscataway, New Jersey, USA, 360–365.

Robins, N. 2001. *GLUT for Win32 v3.7.6 README*.

Rousseaux, S, Hubaux, D, Guisset, P and Legat, J-D. 2007. A high performance FPGA-based accelerator for BLAS library implementation. In *Proceedings of the 3rd Annual Reconfigurable Systems Summer Institute (RSSI 2007)*. NCSA, Urbana-Champaign, Illinois, USA, Paper 3.

Rueda, AJ and Ortega, L. 2008. Geometric algorithms on CUDA. In *Proceedings of the 3rd International Conference on Computer Graphics Theory and Applications (GRAPP 2008)*. INSTICC, Setúbal, Portugal, 107–112.

Samet, H. 1989. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, Massachusetts, USA.

Scrofano, R, Zhuo, L and Prasanna, VK. 2008. Area-efficient arithmetic expression evaluation using deeply pipelined floating-point cores. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 16,* 2 (Feb), 167–176.

Segal, M and Akeley, K. 2009. *The OpenGL Graphics System, Revision 3.2*. Beaverton, Oregon, USA.

Seiler, L, Carmean, D, Sprangle, E, Forsyth, T, Abrash, M, Dubey, P, Junkins, S, Lake, A, Sugerman, J, Cavin, R, Espasa, R, Grochowski, E, Juan, T and Hanrahan, P. 2008. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics – Proceedings of the 35th International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2008) 27,* 3 (11–15 Aug), Paper 18.

Selle, A, Su, J, Irving, G and Fedkiw, R. 2009. Robust high-resolution cloth using parallelism, history-based collisions, and accurate friction. *IEEE Transactions on Visualization and Computer Graphics 15,* 2 (Mar–Apr), 339–350.

Seugling, A and Rölin, M. 2006. Evaluation of physics engines and implementation of a physics module in a 3d-authoring tool. Masters thesis, Umeå University, Umeå, Sweden.

Shanley, T. 2005. *The Unabridged Pentium 4: IA32 Processor Genealogy*. Addison-Wesley, Boston, Massachusetts, USA.

Shevtsov, M, Soupikov, A and Kapustin, A. 2007. Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum – Proceedings of the 28th Annual Conference of the European Association for Computer Graphics (Eurographics 2007) 26,* 3 (3–7 Sept), 395–404.

Singh, S and Slous, R. 1998. Accelerating Adobe Photoshop with reconfigurable logic. In *Proceedings of the 1998 IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE Computer Society, Los Alamitos, California, USA, 236–244.

Six, HW and Wood, D. 1982. Counting and reporting intersections of d-ranges. *IEEE Transactions on Computers 31,* 3 (Mar), 181–187.

Smart, J, Roebling, R, Zeitlin, V, Dunn, R et al. 2007. *wxWidgets v2.8.7: A Portable C++ and Python GUI toolkit*.

Smart, J, Roebling, R, Zeitlin, V, Dunn, R et al. 2009. *wxWidgets v2.8.10: A Portable C++ and Python GUI toolkit*.

Smith, R. 2009. *Open Dynamics Engine v0.11.1 User Guide*.

Snyder, JM, Woodbury, AR, Fleischer, K, Currin, B and Barr, AH. 1993. Interval methods for multi-point collisions between time-dependent curved surfaces. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1993)*. ACM, New York, USA, 321–334.

So, B, Hall, MW and Ziegler, HE. 2004. Custom data layout for memory parallelism. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO 2004)*. IEEE Computer Society, Los Alamitos, California, USA, 291–302.

Spurzem, R, Berczik, P, Marcus, G, Kugel, A, Lienhart, G, Berentzen, I, Männer, R, Klessen, R and Banerjee, R. 2009. Accelerating astrophysical particle simulations with programmable hardware (FPGA and GPU). *Computer Science – Research and Development – Proceedings of the 2009 International Supercomputing Conference (ISC 2009) 23,* 3–4 (23–26 June), 231–239.

Steinbach, K, Kuffner, J, Asfour, T and Dillmann, R. 2006. Efficient collision and self-collision detection for humanoids based on sphere trees hierarchies. In *Proceedings of the 6th IEEE-RAS International Conference on Humanoid Robots (HUMANOIDS 2006)*. IEEE, Piscataway, New Jersey, USA, 560–566.

Su, C-J. 2007. Enhancing sense of reality by efficient and precise collision detection in virtual environments. *Lecture Notes in Computer Science – Proceedings of the 1st International Conference on Digital Human Modeling (ICDHM 2007) 4561*, 969–978.

Sud, A, Govindaraju, N, Gayle, R, Kabul, I and Manocha, D. 2006. Fast proximity computation among deformable models using discrete Voronoi diagrams. *ACM Transactions on Graphics – Proceedings of the 33rd International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2006) 25,* 3 (30 July–3 Aug), 1144–1153.

Swartzlander Jr, EE and Alexopoulos, AG. 1975. The sign/logarithm number system. *IEEE Transactions on Computers C-24,* 12 (Dec), 1238–1242.

Tang, M, Kim, YJ and Manocha, D. 2009. C^2A: Controlled conservative advancement for continuous collision detection of polygonal models. In *Proceedings of the 2009 IEEE International Conference on Robotics and Automation (ICRA 2009)*. IEEE Computer Society, Los Alamitos, California, USA, 849–854.

Tang, M, Manocha, D and Tong, R. 2009. Multi-core collision detection between deformable models. In *Proceedings of the 2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling (SPM 2009)*. ACM, New York, USA, 355–360. Short Paper.

Tang, TD, Bohez, ELJ and Koomsap, P. 2007. The sweep plane algorithm for global collision detection with workpiece geometry update for five-axis NC machining. *Computer-Aided Design 39,* 11 (Nov), 1012–1024.

Teschner, M, Heidelberger, B, Müller, M, Pomeranets, D and Gross, M. 2003. Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of the 2003 International Fall Workshop on Vision, Modeling, and Visualization (VMV 2003)*. AKA, Berlin, Germany, 47–54.

Teschner, M, Kimmerle, S, Heidelberger, B, Zachmann, G, Raghupathi, L, Fuhrmann, A, Cani, M-P, Faure, F, Magnenat-Thalmann, N, Strasser, W and Volino, P. 2005. Collision detection for deformable objects. *Computer Graphics Forum 24,* 1 (Mar), 61–81.

Thrane, N and Simonsen, LO. 2005. A comparison of acceleration structures for GPU assisted ray tracing. Masters thesis, University of Aarhus, Aarhus, Denmark.

Tonge, R, Zhang, L and Sequeira, D. 2005. Method and program solving LCPs for rigid body dynamics. International Patent WO/2005/074424.

Trenkel, S, Weller, R and Zachmann, G. 2007. A benchmarking suite for static collision detection algorithms. *Journal of WSCG – Proceedings of the 15th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2007) 15,* 1–3 (29 Jan–1 Feb), 105–110.

Tripp, JL, Hanson, AÅ, Gokhale, M and Mortveit, H. 2005. Partitioning hardware and software for reconfigurable supercomputing applications: A case study. In *Proceedings of the 2005 ACM/IEEE*

Conference on Supercomputing (SC 2005). IEEE Computer Society, Los Alamitos, California, USA, Paper 27.

Turk, G. 1989. Interactive collision detection for molecular graphics. Masters thesis, University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, USA.

Turk, G and Levoy, M. 1994. Zippered polygon meshes from range images. In *Proceedings of the 21st International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1994)*. ACM, New York, USA, 311–318.

Uliano, KC, Kennedy, RS and Lambert, EY. 1986. Asynchronous visual delays and the development of simulator sickness. In *Proceedings of the 30th Annual Meeting of the Human Factors and Ergonomics Society*. Human Factors and Ergonomics Society, Santa Monica, California, USA, 422–426.

Ulrich, T. 2000. Loose octrees. In *Game Programming Gems*, M DeLoura, Ed. Charles River Media, Rockland, Massachusetts, USA, 444–453.

Underwood, K. 2004. FPGAs vs. CPUs: Trends in peak floating-point performance. In *Proceedings of the 12th International Symposium on Field Programmable Gate Arrays (FPGA 2004)*. ACM, New York, USA, 171–180.

Uno, S and Slater, M. 1997. The sensitivity of presence to collision response. In *Proceedings of the 1997 IEEE Virtual Reality Annual International Symposium (VRAIS 1997)*. IEEE Computer Society, Los Alamitos, California, USA, 95–103 and 216.

Vahora, F, Temkin, B, Krummel, TM and Gorman, PJ. 1999. Development of real-time virtual reality haptic application: Real-time issues. In *Proceedings of the 12th IEEE Symposium on Computer-Based Medical Systems (CBMS 1999)*. IEEE Computer Society, Los Alamitos, California, USA, 290–295.

van den Bergen, G. 1997. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools 2,* 4, 1–14.

van den Bergen, G. 1999. A fast and robust GJK implementation for collision detection of convex objects. *Journal of Graphics Tools 4,* 2, 7–25.

van den Bergen, G. 2001. Proximity queries and penetration depth computation on 3D game objects. In *Game Developers Conference 2001 Proceedings (GDC 2001)*. CMP Media, San Francisco, California, USA, 821–837.

van den Bergen, G. 2004. *Collision Detection in Interactive 3D Environments*. The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann, San Francisco, California, USA.

van den Bergen, G. 2007. *User's Guide to the SOLID Collision Detection Library v3.5.6*.

Vaněček Jr, G. 1994. Back-face culling applied to collision detection of polyhedra. *The Journal of Visualization and Computer Animation 5,* 1 (Jan–Mar), 55–63.

Wald, I and Havran, V. 2006. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing (RT 2006)*. IEEE, Piscataway, New Jersey, USA, 61–69.

Wang, W, Wang, J and Kim, M-S. 2001. An algebraic condition for the separation of two ellipsoids. *Computer Aided Geometric Design 18,* 6 (July), 531–539.

Wang, Z and Gu, Y. 2007. A hierarchical architecture of N-port memory based on FPGA. In *Proceedings of the 2007 International Conference on Convergence Information Technology (ICCIT 2007)*. IEEE Computer Society, Los Alamitos, California, USA, 157–161.

Whitted, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23,* 6 (June), 343–349.

Wilton, SJE, Rose, J and Vranesic, ZG. 1995. Architecture of centralized field-configurable memory. In *Proceedings of the 3rd ACM International Symposium on Field-Programmable Gate Arrays (FPGA 1995)*. ACM, New York, USA, 97–103.

Wong, S, Vassiliadis, S and Hur, JY. 2005. Parallel merge sort on a binary tree on-chip network. In *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2005)*. Technologiestichting STW, Utrecht, Netherlands, 365–368.

Woop, S, Brunvand, E and Slusallek, P. 2006. Estimating performance of a ray-tracing ASIC design. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing (RT 2006)*. IEEE, Piscataway, New Jersey, USA, 7–14.

Woop, S, Schmittler, J and Slusallek, P. 2005. RPU: A programmable ray processing unit for real-time ray tracing. *ACM Transactions on Graphics – Proceedings of the 32nd International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2005) 24,* 3 (31 July–4 Aug), 434–444.

Woulfe, M and Manzke, M. 2009. A framework for benchmarking interactive collision detection. In *Proceedings of the 25th Spring Conference on Computer Graphics (SCCG 2009)*. Comenius University, Bratislava, Slovakia, 221–228.

Wu, C-H, Ogrenci Memik, S and Mahrotra, S. 2009. FPGA implementation of the interior-point algorithm with applications to collision detection. In *Proceedings of the 17th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM 2009)*. IEEE Computer Society, Los Alamitos, California, USA, 295–298. Short Paper.

Xilinx 2007a. *Xilinx Virtex-4 Family Overview*. San José, California, USA. Data Sheet.

Xilinx 2007b. *Xilinx Virtex-II Platform FPGAs*. San José, California, USA. Data Sheet.

Xilinx 2008. *Xilinx ISE Design Suite v10.1.03 Software Manuals and Help*. San José, California, USA.

Xilinx 2009a. *Xilinx Virtex-5 Family Overview*. San José, California, USA. Data Sheet.

Xilinx 2009b. *Xilinx Virtex-6 Family Overview*. San José, California, USA. Data Sheet.

Yardi, S, Bishop, B and Kelliher, TP. 2006. HELLAS: A specialized architecture for interactive deformable object modeling. In *Proceedings of the 44th Annual Southeast Regional Conference (ACM SE 2006)*. ACM, New York, USA, 56–61.

Yeh, TY, Faloutsos, P, Patel, SJ and Reinman, G. 2007. ParallAX: An architecture for real-time physics. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA 2007)*. ACM, New York, USA, 220–231.

Zachmann, G. 1998. Rapid collision detection by dynamically aligned DOP-trees. In *Proceedings of the 1998 IEEE Virtual Reality Annual International Symposium (VRAIS 1998)*. IEEE Computer Society, Los Alamitos, California, USA, 90–97.

Zachmann, G. 2000. Virtual reality in assembly simulation – Collision detection, simulation algorithms, and interaction techniques. PhD thesis, Darmstadt University of Technology, Darmstadt, Germany.

Zachmann, G and Knittel, G. 2003. An architecture for hierarchical collision detection. *Journal of WSCG – Proceedings of the 11th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2003) 11,* 1 (3–7 Feb), 149–156.

Zeller, C. 2005. Cloth simulation on the GPU. In *Proceedings of the 32nd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2005): Sketches & Applications*. ACM, New York, USA, Paper 39. Sketch.

Zhang, D and Yuen, MMF. 2000. Collision detection for clothed human animation. In *Proceedings of the 8th Pacific Conference on Computer Graphics and Animation (Pacific Graphics 2000)*. IEEE Computer Society, Los Alamitos, California, USA, 328–337.

Zhang, L, Tonge, R, Sequeira, D and Maher, M. 2009a. Method for operation of parallel LCP solver. International Patent WO/2005/074423.

Zhang, L, Tonge, R, Sequeira, D and Maher, M. 2009b. Parallel LCP solver and system incorporating same. International Patent WO/2005/074425.

Zhang, Z, Li, X, Wang, L and Ma, P. 2008. GPU-based collision detection between deformable objects. In *Proceedings of the 5th International Conference on Visual Information Engineering (VIE 2008)*. IET, Stevenage, UK, 560–565.

Zikan, K and Konečný, P. 1997. Lower bound of distance in 3D. In *Proceedings of the 5th International Conference in Central Europe on Computer Graphics and Visualization (WSCG 1997)*. University of West Bohemia, Plzeň, Czech Republic, 640–647.

Zuchowski, PS, Reynolds, CB, Grupp, RJ, Davis, SG, Cremen, B and Troxel, B. 2002. A hybrid ASIC and FPGA architecture. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2002)*. ACM, New York, USA, 187–194.