Virtual Machine Showdown: Stack versus Registers

by

Yunhe Shi, BSc. MSc.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Doctor of Philosophy

University of Dublin, Trinity College

November 2007

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Yunhe Shi

September 1, 2007

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Yunhe Shi

September 1, 2007

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr David Gregg. His encouragement, patience, understanding, guidance, and financial support ensured the completion of this dissertation. Since English is not my native language, he has spent a considerable time and effort to correct countless grammatical and spelling errors in my papers and thesis . Without him, it would not have been possible for me to finish my PhD.

I would like to thank Griffith College Dublin for half of the PhD fee sponsorship. I would like to thank the staff of the computing faculty at Griffith College Dublin for their kind supports. Particular gratitude to Eamonn Nolan, Tony Mullins, Kevin Hely, and Waseem Akhtar.

I would also like to thank the members of the Computer Architecture Group who have been great company and of great assistance through the years. Particular thanks to Kevin Casey and Andrew Beatty from the group, with whom I have worked on many issues relating to the Java Virtual Machine. Thanks also go to Nicholas Nash and Paul Biggar, who spent their precious time proof-reading the draft of this dissertation.

Special thanks are also due to M. A. Ertl of the Technical University, Vienna, for his unselfish assistance and ready availability to offer advice throughout the duration of this project.

Finally, my greatest appreciation is reserved for my parents for their support during my study for the PhD.

Yunhe Shi

University of Dublin, Trinity College
November 2007

Virtual Machine Showdown: Stack versus Registers

Publication No. _____

Yunhe Shi, Ph.D.
University of Dublin, Trinity College, 2007

Supervisor: Dr. David Gregg

Virtual machines (VMs) enable the distribution of programs in an architecture-neutral format, which can easily be interpreted or compiled. The most popular VMs, such as the Java virtual machine (JVM), use a virtual stack architecture, rather than the register architecture that are most popular in real processors. A long-running question in the design of VMs is whether a stack architecture or register architecture can be implemented more efficiently with an interpreter. On the one hand, stack architectures allow smaller VM code so less code must be fetched per VM instruction executed. On the other hand, stack machines require more VM instructions for a given computation, each of which requires an expensive (usually unpredictable) indirect branch for VM instruction dispatch.

This dissertation extends existing work on comparing virtual stack and virtual register architectures in three ways. Firstly, we generate very high quality register code. The result is that our register code has 46% fewer executed VM instructions compared to optimized JVM stack code, with the bytecode size of the register machine being only 26% larger than that of the corresponding stack code. Secondly we present a fully functional virtual-register implementation of the Java virtual machine (JVM), which supports Intel, AMD64, PowerPC and Alpha processors. This register VM supports inline-threaded, direct-threaded, token-threaded, and switch dispatch. Thirdly, we present experimental results on a range of additional optimizations such as register allocation and elimination of redundant heap loads. On the AMD64 architecture the register machine using switch dispatch achieves an average speedup of 1.48 over the corresponding stack machine. Even using the more efficient inline-threaded dispatch, the register VM achieves a speedup of 1.15 over the equivalent stack-based VM.

The performance of VM interpreters is much affected by indirect branches and during the course of the work on VM interpreters we identified a strong interaction between the indirect branch predictor and the trace cache. The dissertation investigates the related phenomenon, and shows that the interaction between the two components results in significant improvements in indirect branch prediction. This is particularly true for codes with many indirect branches, such as VM interpreters.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Virtual machines (VMs) enable the distribution of programs in an architecture-neutral format, which can easily be interpreted or compiled. The most popular VMs, such as the Java virtual machine (JVM) and Microsoft .NET's common language runtime (CLR), use a virtual stack architecture rather than the register architecture that dominates in real processors.

Interpreters are frequently used to implement virtual machines because they have several practical advantages over native code compilers. Interpreters are much slower than the native code produced by just-in-time compilers (even the fastest interpreters are currently about 5–10 times slower), but they are nonetheless widely used for lightweight language implementations. If written in a high-level language, interpreters are portable; they can simply be recompiled for a new architecture, whereas a just-in-time (JIT) compiler requires considerable porting effort. Interpreters also require little memory: the interpreter itself is typically much smaller than a JIT compiler [RVJS00], and the interpreted bytecode is usually a fraction of the size of the corresponding executable native code. For this reason, interpreters are commonly found in embedded systems. Furthermore, interpreters avoid the compilation overhead in JIT compilers. For rarely executed code, interpreting may be much faster than JIT compilation. The Hotspot JVMs [SM01] take advantage of this by using a hybrid interpreter/JIT system. Code is initially interpreted, saving the time and space of JIT compilation, and

only if a section of code is executed frequently is it JIT compiled. Interpreters are also dramatically simpler than compilers; they are easy to construct, and easy to debug. Finally, it is easy to provide tools such as debuggers and profilers when using an interpreter because it is easy to insert additional code into an interpreter loop. Providing such tools for native code is much more complex. Interpreters provide a range of attractive features for language implementation. In particular, most scripting languages are implemented using interpreters.

A long-running question in the design of VMs is whether a stack architecture or a register architecture can be implemented more efficiently with an interpreter. Stack architectures allow smaller VM code so less code must be fetched per VM instruction executed. However, stack machines require more VM instructions for a given computation, each of which requires an expensive (usually unpredictable) indirect branch for VM instruction dispatch. Several authors have discussed the issue [Mye77, SM77, MB99, WP97] and presented small examples where each architecture performs better, but no general conclusions can be drawn without a larger study.

1.2 Our Thesis

The main thesis of this work is that register architectures can be implemented to be significantly faster than stack architectures when building a virtual machine interpreter. The main reason is that stack architectures need to shuffle values onto the stack before they can be operated upon, and results must be stored from the stack to variables. In contrast, a register architecture allows VM instructions to manipulate local variables directly. This allows the same functionality to be implemented on a register architecture using far fewer VM instructions. Given that dispatching VM instructions is expensive due to the high cost of real-machine indirect branches, the result is that interpreter-based VMs for register architectures are significantly faster.

1.3 Contributions

This dissertation extends previous work on comparing register and stack machine and we believe, answers the question of the relative strengths and weaknesses of stack and

register machines. We have made a number of contributions.

- *Better analysis of the features of register and stack code*

In previous work by Davis et al. [DBC+03, GBC+05] register and stack code were compared by translating optimized stack code into register code. Although the results were interesting, the quality of the generated register code was poor, and the benefits of register code were underestimated. We use a much more sophisticated scheme to generate our register code (although we also follow the route of translation from optimized stack code) and the result is that our register code requires far fewer VM instructions to implement the same benchmark programs than either corresponding stack code, or register code generated using Davis et al.'s method.

- *Design, implementation and measurement of the register VM*

Previous work made quantitative measures of the stack and register code, but it did not compare corresponding stack and register VM implementations. This dissertation presents the design and implementation of a register machine that corresponds closely to the stack-based Java VM. We present extensive measurements of both machines, using various interpreter optimization options, and running on several different hardware architectures. Our results include measurements from hardware performance counters that allow us to investigate the effect of using a register rather than stack VM on the microarchitectural behaviour of the interpreter.

- *Analysis of the effect of the trace cache on indirect branch prediction*

In addition to the core work on virtual machine design, this dissertation also addresses the closely related problem of indirect branch prediction. Indirect branch prediction has a big impact on VM interpreters because interpreters typically use indirect branches to dispatch the execution of VM instructions. A particularly interesting interaction arises with the trace cache which was designed to increase the fetch bandwidth for a superscalar processor. One unintended effect of the trace cache is that it can provide extra context information, which helps indirect branch prediction, particularly when executing programs with a lot of indirect branches, such as interpreters.

1.4 Collaborations

During my PhD study, I collaborated with several colleagues. The papers published during my PhD research illustrate our collaborations.

- *Yunhe Shi, Emre Özer, and David Gregg. Analyzing effects of trace cache configurations on the prediction of indirect branches. The Journal of Instruction-Level Parallelism, Volume 8, 2006.*

Emre Özer assisted with the research in the trace cache. He provided his expertise in processor microarchitectures and helped to give direction to the experiments. He also contributed considerably to organizing and correcting English in the paper.

- *Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: stack versus registers. In ACM/SIGPLAN Conference on Virtual Execution Environments, pages 153-163, Chicago, Illinois, June 2005. ACM Press.*

The work in this paper presented our initial results comparing register and stack VMs. The implementation for this paper was based on "CVM", an implementation of the JVM from Sun Microsystems.

Andrew Beatty started an initial implementation of a register VM using Sun's CVM, but abandoned the work before it was complete. He helped me to get started with his implementation and provided useful advice.

Anton Ertl helped to review my earlier draft of the paper and provided some useful insights into optimizing the VM.

- *Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: stack versus registers. In ACM Transactions on Architecture and Code Optimization (TACO). Forthcoming issue. ACM Press.*

For this paper I designed and built an entirely new implementation of the register machine, based around Cacao, an open source JVM. Cacao is much faster and more portable than CVM, reducing unnecessary overheads that might impact the results, and allowing comparisons to be made on a wide variety of architectures.

Kevin Casey gave me a lot of assistance in the early stage of my PhD, particularly in learning to understand the behaviour of VM interpreters. He assisted to correct and rewrite parts of the TACO journal paper and helped in replying to reviewers' comments.

Anton Ertl built the original, stack-based version of Cacao, and provided a lot of expertise in `vmgen` and `Cacao` through numerous e-mail exchanges, in order to guide me through a lot of technical problems.

Lastly, David Gregg supervised and guided me during my entire PhD research. His invaluable contributions (and countless corrections of English) are recognized in his joint authorship of all three papers. He also helped considerably in structuring, correcting, and rewriting sections of this dissertation.

1.5 Overview

The remainder of this thesis is structured as follows:

Chapter 2 This chapter examines the concept of a virtual machine and its implementation. Some aspects of modern processor architecture are introduced, such as cache memory, pipelining and branch prediction.

Chapter 3 This chapter reviews the interpreter-based virtual machine research. A major component of interpreter execution overhead is dispatch cost. The main approaches to reducing such dispatch costs are examined. Another component of execution overhead in a stack-based VM is accessing operands from the operand stack. Stack caching is introduced as an optimization technique to tackle the problem. Current research in stack-based versus register-based VMs is introduced. Current research in indirect branch prediction and the trace cache is also reviewed.

Chapter 4 In this chapter, our experimental research into how the trace cache can influence the indirect branch prediction is presented. In order to do this, we vary the trace cache configuration to identify the optimal one for indirect branch prediction.

Chapter 5 In this chapter, we compare a stack-based and register-based architecture from the viewpoint of an interpreter. Then we introduce the way in which VM stack code is translated into VM register code. Finally the various optimizations are presented.

Chapter 6 This chapter presents the experimental results comparing stack and register machine. We compare the static and dynamic (run-time) code behavior of the stack and register machines. We examine the effects of using four different VM instruction dispatch methods, and results are presented for five different hardware platforms. These results are further investigated using hardware performance counters on the AMD64 processor. Finally, other possible optimizations for a register-based VM are investigated, such as the potential for very aggressive common subexpression elimination, the use of static superinstructions, and two-address register instruction formats.

Chapter 7 In the last chapter, the results of the thesis are summarized, highlighting some of the most notable contributions. Finally, we identify some interesting aspects arising from the work that warrant further research.

Chapter 2

Background

2.1 Introduction

In this chapter, we give background information on virtual machines and branch prediction, which is related to the research in this dissertation.

2.2 Virtual Machines

Virtual machines (VMs) exist in a variety of forms in computer systems. Smith and Nair [SN05] classified virtual machines into process VMs and system VMs (See Figure 2.1). Process VMs can be further divided into multi-programmed systems and dynamic binary optimizer when the same instruction set architectures (ISA) are used in VMs as the hosted hardware platforms; process VMs can be dynamic translators when VM ISAs are different from the hosted platforms, one type of which can be high-level language (HLL) VMs. System VMs can be classic OS VMs and hosted VMs when the same VM ISA is used as the hosted hardware platforms; System VMs can also be whole system VMs, including co-designed VMs, when VM ISAs are different from hosted hardware platforms.

2.2.1 High-Level Language VMs

One clear objective of process VMs is portability. Instead of writing a VM to simulate a conventional architecture on another one on a case by case basis, a new virtual machine

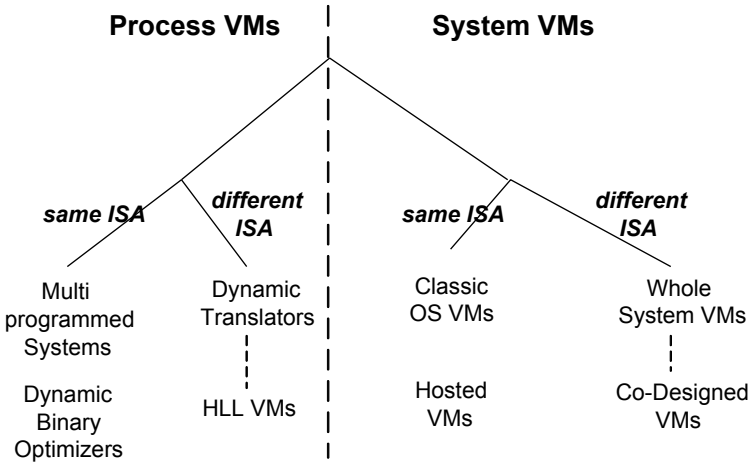

Figure 2.1: Virtual machine taxonomy. Within the general categories of process and system VMs, ISA simulation is the major basis of differentiation. Source: Smith & Nair [SN05]

(HLL VM) development and execution environment can be designed with a high-level programming language, a new portable/virtual ISA (V-ISA), a compiler, programming API and runtime environment. The portable V-ISA is not limited by or tied to any specific hardware platforms.

Figure 2.2 shows the difference between a conventional platform-specific compilation environment and an HLL VM environment. In a conventional system, shown in Figure 2.2(a), high-level programming language source code is first compiled into intermediate code, which is then translated into platform-dependent object code. The object code is distributed and executed on a targeted platform.

In a HLL VM, as shown in Figure 2.2(b), the portable code (V-ISA) is generated from the high-level source code by a compiler, which is platform-independent. The portable V-ISA code is loaded into the virtual machine, which can be either interpreted or translated into native code. To support a new platform, only the HLL virtual machine and its library needs to be ported. Some examples of HLL VMs are the Pascal P-Code virtual machine, Sun's Java virtual machine (JVM) and Microsoft.NET common-language runtime (CLR).

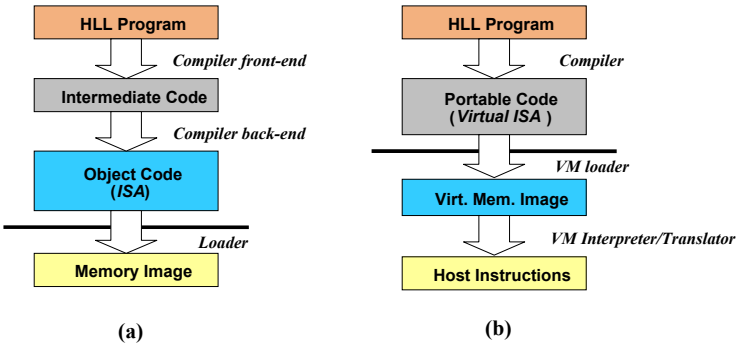

Figure 2.2: High-level-language environments. (a) Conventional environment where platform-dependent object code is distributed. (b) HLL VM environment where a platform-dependent VM executes portable intermediate code. Source: Smith & Nair [SN05]

2.2.2 The Pascal P-Code Virtual Machine

One of the virtual machines, which had a very big influence on later generations of virtual machines, is the Pascal P-Code virtual machine. The Pascal source code was compiled to P-code, which is a stack-oriented instruction set. Then the P-code ran on a virtual machine. The most popular implementation was the P4 [PD82]. P-code was the first really successful virtual machine, and it helped establish the concept as a real alternative for language implementations. Later, a stack architecture was also chosen for the virtual machine in the Smalltalk programming environment [Kay93, Kra83]. Since then, stack architectures have been used as the intermediate representations for several popular virtual machines including the Java VM and .NET VM.

2.3 The Java Virtual Machine

The Java Virtual Machine (JVM) is a process HLL VM, which is designed to support the Java programming language [GJS96]. The Java programming language is a general-purpose object-oriented concurrent language. The Java platform consists of the Java programming language, compiler, class library (API), and virtual machine.

At compile-time, as shown in Figure 2.3, Java source code is compiled into intermediate V-ISA (bytecode) with meta-information in a class file format. Those class files are moved locally or transported over a network. In a runtime environment, a Java

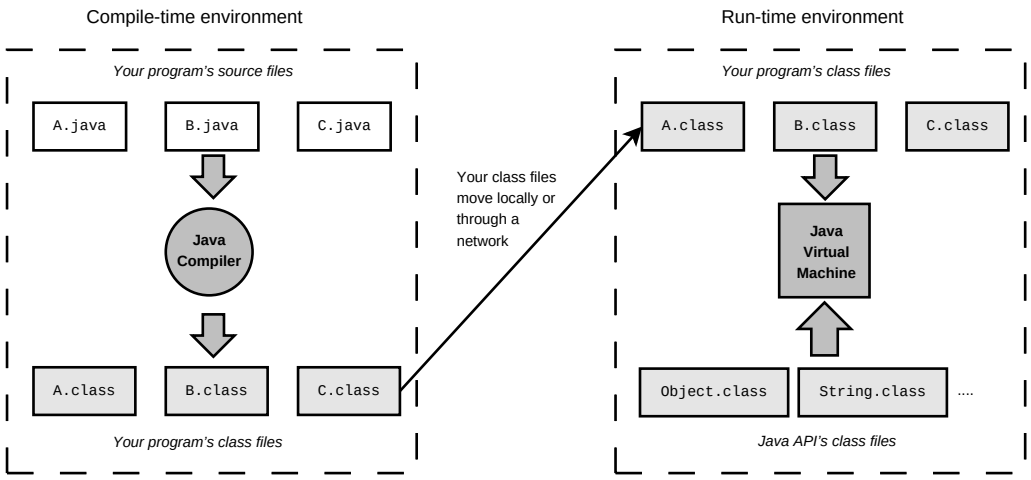

Figure 2.3: The Java programming environment. Source: Bill Venners [Ven99]

virtual machine instance loads the Java application class files and classes from the Java class library which are used in the application code to run the application.

2.3.1 The Internal Architecture of a Java Virtual Machine

The Java Virtual Machine specification [LY99] defines an abstract stack-based virtual machine, which can load the Java application classes and API classes and execute bytecodes. The JVM can be implemented in different ways. One way to implement the JVM, as shown in Figure 2.4, is on top of an operating system. The class loader loads application classes and library classes and creates in-memory representations of classes and bytecodes. The execution engine executes bytecodes on a hosted operating system and hardware platform.

The Java Virtual Machine specification also defines the standard class file format, which has cross-platform portability. As long as a Java virtual machine is implemented on a hosted platform, Java class files can be loaded and bytecodes can be executed.

The internal architecture of a Java virtual machine, as shown in Figure 2.5, shows the run-time data areas, which include a method area, a heap, Java stacks, PC registers, and the native method stack.

The method area is the data structure used to store class information and Java bytecodes. The heap is the dynamic memory area where objects are allocated. Java stacks are used to store information about method calls. Each method call results in

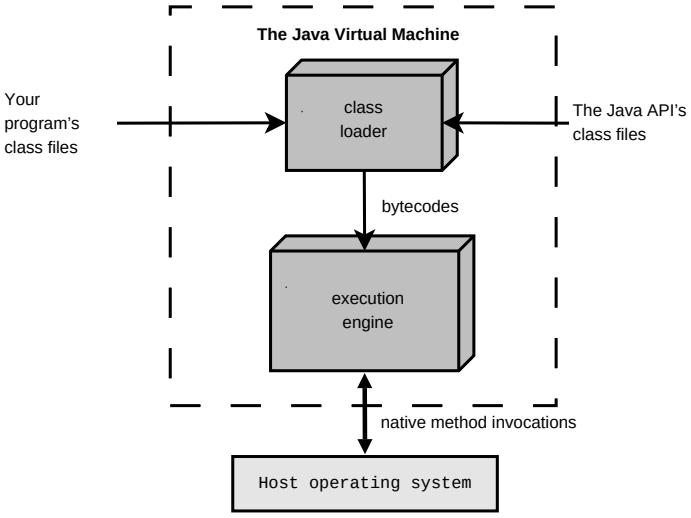

Figure 2.4: A Java Virtual Machine implemented on top of a host operating system. Source: Bill Venners [Ven99]

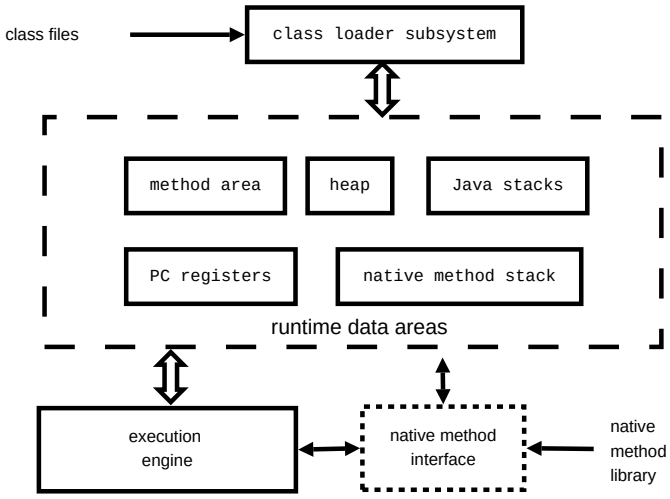

Figure 2.5: The internal architecture of the Java Virtual Machine Source: Bill Venners [Ven99]

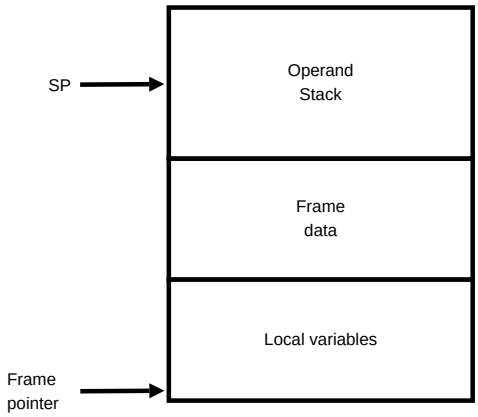

Figure 2.6: Java frame data structure on the Java stacks.

a new Java stack frame being pushed to store the local information and the operand stack for that method. When the method completes, the Java stack frame is popped. PC registers store the virtual program counter, which points to the code being executed and the stack pointer of the operand stack for the execution of a thread. The native method stack is used to record the native method call information. The native methods are used to implement platform dependent functions.

The Java stack records the history of method calls for the execution engine. For each method call, there exists a Java frame, which includes local variables for the method, frame data, and an operand stack for the computation (see Figure 2.6). The frame data may include program counters (PC) & stack pointer (SP) of the called method and/or PC and SP of current method.

2.3.2 Execution Engine

The core of a Java virtual machine is the execution engine, which executes the Java bytecode. The execution engine can be implemented with an interpreter, or a just-in-time compiler, or both an interpreter and a just-in-time (JIT) compiler (mixed mode).

When an execution engine is implemented by a JIT compiler, it will translate the Java bytecodes into native code for a processor and then execute the native code. An execution engine based on an interpreter will carry out the function of each bytecode instruction by jumping to the segments of the code in an interpreter loop, which implements the bytecode instruction.

In this dissertation, we only consider the interpreter for the Java Virtual Machine implementation. When the execution engine is implemented using an interpreter, indirect branches are needed in order to jump to their corresponding segments inside an interpreter loop which implement the functions of bytecode instructions. In modern hardware processors, the indirect jumps are poorly predicted [EG03], which leads to poor performance of a Java interpreter.

2.3.3 Java Bytecode Instruction Set

The Java Virtual Machine (JVM) is stack-based and its instructions manipulate the operand stack of a Java stack frame on the Java stack. For a stack-based instruction set, the operands of an instruction are implicit and any computation has to be done through a stack.

Java bytecode instructions can be put into the following categories:

- Stack load/store instructions - includes all instructions which load values from the local variables onto the operand stack and instructions which store values from the operand stack to the local variables.
- Constant instructions - load constants onto the operand stack.
- Flow control instructions - includes *if*, *switch* and unconditional branch instructions.
- Arithmetic/logical instructions - includes different types of arithmetic instructions.
- Object access instructions - includes array access instructions and class/object field access instructions
- Method call and return instructions - includes all *invoke* static (class) and object method call and return instructions.
- Stack manipulation instructions - includes all those instructions which duplicate stack items on the operand stack and stack pointer manipulation instructions
- Exception handling instructions - includes all those instructions which support Java exception handling

- Threading support instructions - includes all those instructions which support Java threads, such as thread synchronization.
- Type testing/casting

2.4 Modern Processor Architecture

Modern processors' performance has been improved dramatically since the first general purpose computer was created. This performance improvement came from the advancement of the technology used to build the processors and innovative computer design. In this section, we are going to introduce some of the concepts and terms related to this dissertation.

2.4.1 Cache Memory

Cache memory is widely used in computer systems. This is a relatively small area of memory for storing duplicate information to allow faster access, such as an instruction cache, or storing a small subset of information due to resource constraints, such as a branch target buffer. The fast access to a cache item can usually be done by hashing keys to produce the (possible) location of the item. The cache can be organized in different ways. Direct-mapped caches organize the cache items as a linear list, as shown in Figure 2.7. A hash function with keys as parameters will give the exact location of a cache item. When the hash function produces the same location with different keys, a conflict (interference) occurs. Set-associative caches organize the items into N sets and each set has M items, as shown in Figure 2.8. The hash function will give the location of a set. Then an associative search will continue to match one or more keys. In this way, the set-associative cache can reduce the possibility of a conflict. Another type of cache is a fully-associative cache. It is usually not practical to implement a fully-associative cache in a computer system. A set-associative cache is a compromise between a direct-mapped cache and a fully-associative cache. In a direct mapped cache, adding a new item will generally replace an existing item. In a set-associative cache, one of the more common ways to add a new item is to replace the least recently used (LRU) existing item in a set.

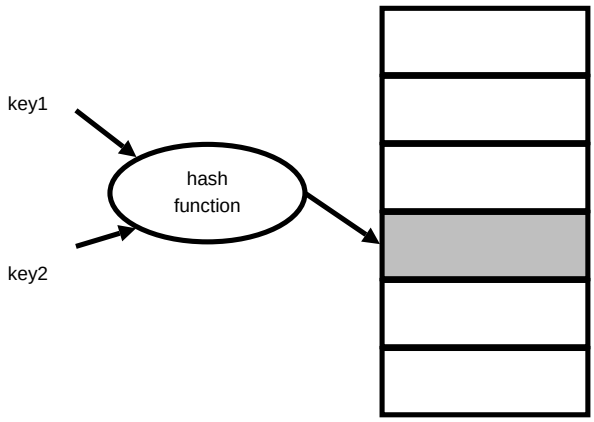

Figure 2.7: Direct-mapped cache

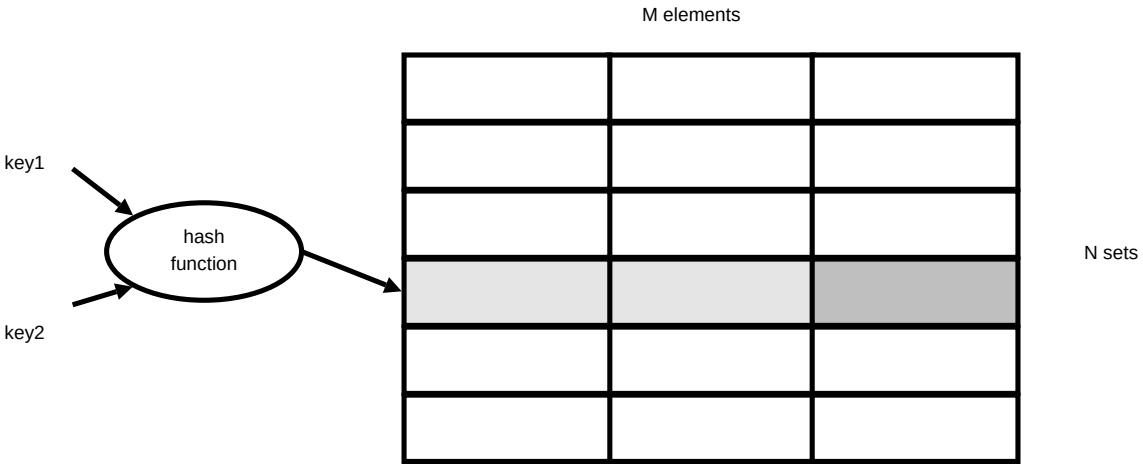

Figure 2.8: Set-associative cache

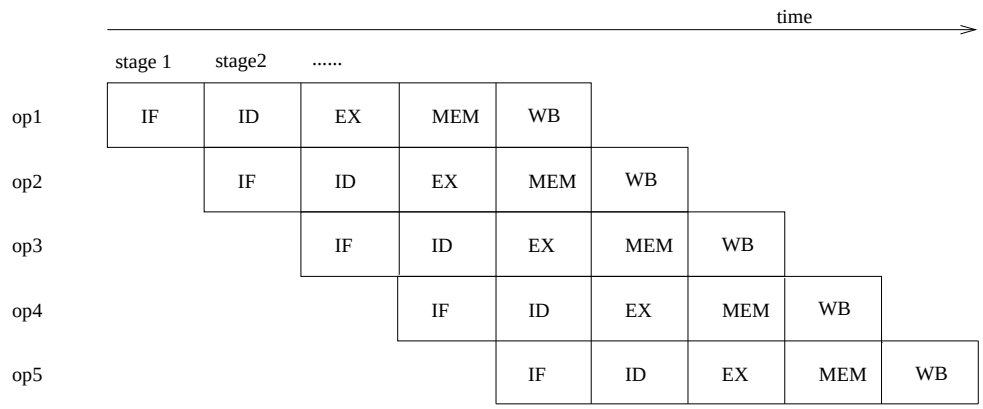

Figure 2.9: Classic processor pipeline

2.4.2 Pipelining

Even though computer processor architecture has become very complex in modern processor, the logical steps to execute an instructions remain the same:

- Fetch an instruction from the main memory
- Decode the instruction
- Fetch the needed operands (data) from memory
- Execute the instruction
- Write the results of the instruction back to the memory.

In the early days of processors, instructions were executed sequentially. There are a lot of instructions in computer programs, which are independent of each other. Those independent instruction can be executed at the same time without changing the end results of the program execution. Instruction level parallelism (ILP) can be exploited to improve the performance of processors. One computer design techniques is pipelining, which divides the instruction execution into stages and overlaps the stages of independent instruction execution, as shown in Figure 2.9. In ideal circumstances, in which all the instructions are independent of each other, only one cycle is needed to execute each instruction.

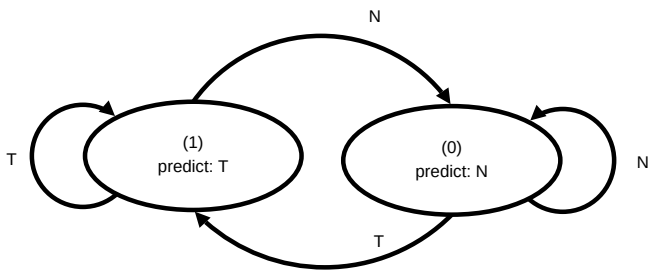

Figure 2.10: Dynamic 1-Bit Predictor. When the one-bit state is 1, the branch is predicted taken. When the state is 0, the branch is predicted to be not taken.

2.4.3 Branch Prediction

Instructions in a programs are not generally independent of each other. There exist data dependencies and control dependencies (branch effects) [USS97] between the instructions. When a branch instruction passes through the pipeline, the path to be taken is not determined until the instruction is executed, which will stall the whole pipeline. Speculative instruction execution tries to solve this problem by predicting the branch direction and continuing execution in the predicted direction. Branch prediction accuracy has a large impact on the performance of a pipelined processor.

Branch instruction can be a conditional branch or an unconditional branch. Conditional branches have two branch targets: taken (T) or not-taken (N). Static predictors always predict the branches to be taken or not-taken, or BTFN (Backward Taken; Forward Not Taken). Dynamic predictors use the previous branch execution results to make future predictions, which allow branch predictors to adapt to the characteristics of an application. As shown in Figure 2.10, a 1-bit predictor uses one bit to represent the state of each branch instruction and uses the state to make a prediction. The bit for each state is usually stored in a table, such as a branch target buffer (BTB) and the branch instruction address is used to look up the state in the table. When the state is one, the branch is predicted to be taken. After the branch is actually resolved (taken or not taken), the state will be updated as shown in Figure 2.10. A 2-bit predictor uses 2-bits to represent the state of each branch instructions and use the state to make future predictions, as shown in Figure 2.11.

1-bit or 2-bit predictor only considers the branch's own history (taken or not taken) itself. The branch history of a branch is the execution path (T-N-T...) before the

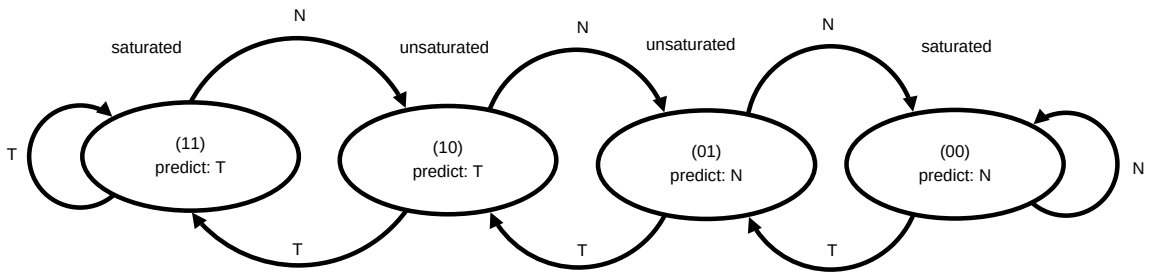

Figure 2.11: Dynamic 2-Bit Predictor. When the significant bit of the state is 1, the branch is predicted taken. When the significant bit of the state is 0, the branch is predicted to be not taken.

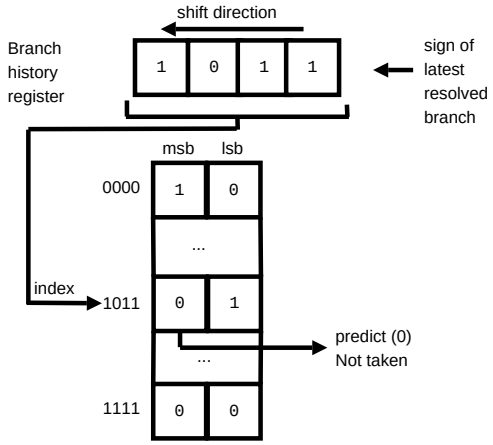

Figure 2.12: Two-level adaptive branch predictor.

branch. The two-level adaptive predictor, as shown in Figure 2.12, takes into account the branch history before the one currently being predicted. The branch history register holds the history of previously executed branches. Each time a branch's direction is resolved, its direction (taken: 1 or not taken: 0) is shifted to the least significant bit of the branch history register. The branch history register is used as an index into the branch pattern table, which holds a 2-bit counter (state) similar to Figure 2.11.

With advancements in manufacturing technology, which allows larger predictors, and research in branch prediction, the two-level adaptive branch predictor can give a prediction accuracy of over 90% [USS97].

One type of unconditional branch is the direct jump. The direct jump has only one target and transfers the execution of a program from one location to another. Another type of unconditional branch instruction is the indirect branch. These branches have

more than one target, which may depend on some computed value. Indirect branches are not very common (less than 1% [DH98]) in regular programs. However they are quite common (up to 13% [EG01]) in the implementation of interpreters. Indirect branches can be unpredictable (up to 95% [EG03]) when a branch target buffer (BTB) is used.

2.5 Conclusion

In this chapter, we have presented the concepts relating to virtual machines and examined the internal structure of the Java Virtual Machine. Branch prediction in modern pipelined processors is critical to their performance. Various predictors have been introduced.

In the next chapter, we examine existing research in these two areas.

Chapter 3

Literature Survey

3.1 Introduction

As discussed in the last chapter, an interpreter is one of the ways to implement the execution engine of a HLL VM. In this chapter, we examine the interpreter optimizations relating to the dispatch cost reduction, the stack caching for stack-based VMs, register-based VMs, indirect branch prediction, and trace cache.

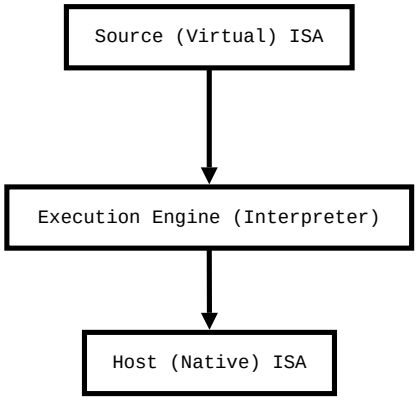

Figure 3.1: Source and target ISA for an interpreter.

3.2 Virtual Machine Interpreters

Interpreters have a long and rich history. As shown in Figure 3.1, an interpreter can understand the source (virtual) ISA (usually referred to as bytecodes) and interpret those virtual ISA on a hosted platform. The interpreter-based VM abstracts away the underlying details of hosted platforms and makes the implemented high-level programming language portable across different hardware platforms as long as the VM has been ported to them.

There are many different types of interpreters. Some interpreters (MIPSI [SA97] and SimpleScalar [ALE02b]) simulate the ISA of new hardware, which does not yet exist, or to port binary applications compiled for one hardware platforms to run on another one. Some other interpreters (Java [LY99], Perl, Tcl, Lua [IdFC05]) are used to implement higher level programming languages.

When an interpreter is used to implement a high-level language, there are two ways to convert the high-level source code into a sequence of virtual machine instructions or *bytecodes* understandable by the interpreter. The translation of the source code into VM code can be either off-line (JVM [LY99]) or during runtime (Perl and Lua [IdFC05]). VM instructions for higher-level portable languages like Java are usually designed with the intention of easing interpretation. The opcodes are usually encoded with one byte (256 possible VM instructions) in interpreters, such as Java [LY99] and Smalltalk [GR83].

An interpreter is an attractive option for VM implementation because it is easy to implement and port to different platforms. However, an interpreter suffers from the drawback of low performance when compared to native code compiled directly from the source programming language. Many researchers [RLV+96, EG03, BVZB05, PR98] have studied ways to improve the performance of an interpreter. We will focus on the two categories of improvement which are relevant to our research in this dissertation. The first category is related to interpreter implementations, such as the dispatch mechanism. The second category is related to the VM instruction architecture design choices, such as the choice between virtual register machine instruction format or virtual stack machine one.

The core of a virtual machine (VM) is an execution engine, which behaves like a real processor. The execution engine, which can be implemented with an interpreter,

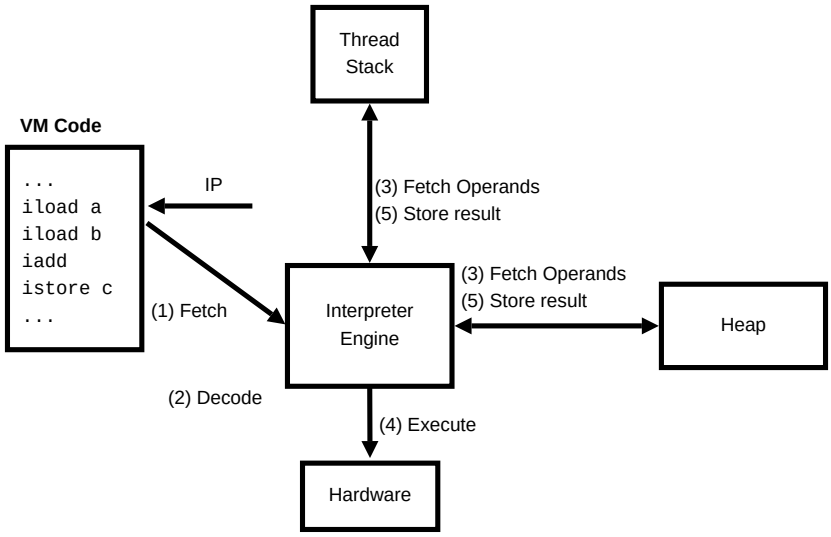

Figure 3.2: The execution cycle of a VM instruction by an interpreter. (1) Fetch an instruction pointed by IP, (2) Decode the VM instruction by finding out its implementation, (3) Fetch the source operands for the instruction, (4) Execute the implementation of the instruction inside the interpreter loop, (5) Store the results back. The thread stack holds the virtual register and the operand stack for a particular method call. The heap will store more global data structures like an object representation

fetches, decodes and executes VM instructions, as shown in Figure 3.2. Inside a virtual machine, an interpreter (execution engine) has a virtual instruction pointer to the VM code currently being executed. In order to execute a VM instruction, an interpreter first fetches an instruction by using the instruction pointer, decodes the instruction (find the segment of code which implements the VM instruction in the interpreter loop), and then executes the code in the segment to carry out the function of the VM instruction. The last step includes the fetching the operands of the instruction and storing any results. There are two types of operand locations for VM instructions. The first type of operand location is virtual registers or an operand stack, which are typically implemented as an array in the memory. The second type of operand location can be some data structures, such as an object representation, in the heap in the run-time data areas. From the point view of hardware, the interpreter itself is the only code executed. The VM application code, the operands in local variables and on the operand stack, and object representation on the heap are just inputs to the code (interpreter) executed natively on the hardware platform.

```
typedef enum {
  add /* ... */
} Opcode;
void engine()
{
  static Inst program[] = { add /* ... */ };
  Inst *ip = program;
  int *sp;
  for (;;)
    switch (*ip++) {
    case add:
      sp[1]=sp[0]+sp[1];
      sp++;
      break;
    /* ... */
    }
}
```

Figure 3.3: `switch` interpreter dispatch. Source: [EG03]

3.3 Dispatch Cost Reduction Techniques

Interpreter *instruction dispatch* involves extracting the opcode of an instruction and finding the corresponding interpreter segment which implements the instruction. Instruction dispatch is an overhead of executing VM instructions. For fine-grained VM instruction set architecture like Java bytecodes [LY99], more than 40% [RLV+96] of executed native instructions can be related to instruction dispatch. Moreover, instruction dispatches perform a large number of indirect branches (3.2% - 13% of all executed native instructions) and the high misprediction of indirect branches are very expensive (62% of execution time without a predictor) [EG03]. Much research [Bel73, EG03, BVZB05, PR98] has been carried out to minimize the cost of dispatch.

3.3.1 *switch* Dispatch

The most common and easy way to implement interpreter dispatch is by using a big *switch* statement inside a loop, as shown in Figure 3.3, with one *switch* label for each VM instruction inside the loop, such as `add`.

For the *switch* based interpreter shown in Figure 3.3, the dispatch native MIPS

```
$L2: #for (;;)
 lw    $3,0($6)      #$6=instruction pointer
 #nop
 situ  $2,$8,$3      #check upper bound
 bne   $2,$0,$L2
 addu  $6,$6,4       #branch delay slot
 Sll   $2,$3,2       #multiply by 4
 addu  $2,$2,$7      #add switch table base ($L13)
 lw    $2,0($2)
 #nop
 j     $2

 #nop
 ...

$L13: #switch target table
 .word   $L12
 ...

$L12: #add:
 ...
 j $L2
 #nop
```

Figure 3.4: `switch` dispatch in MIPS assembly. Register n is denoted by $n, the destination operand of an instruction is the leftmost register, and comments start with #. Source: [EG03]

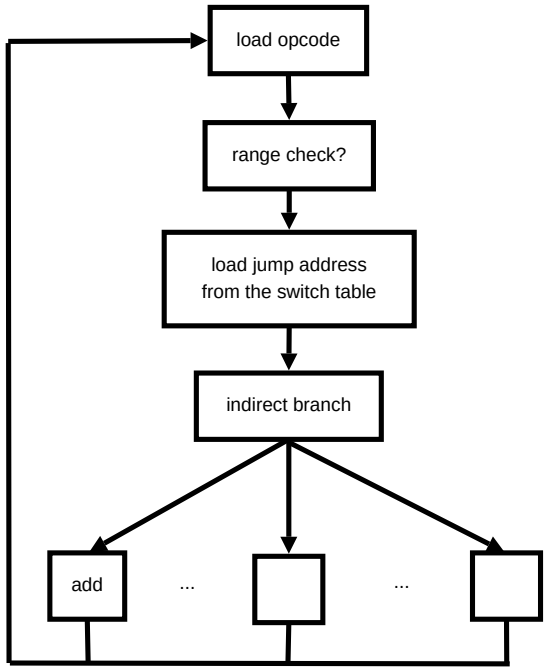

Figure 3.5: `switch`-based interpreter flow diagram

assembly code is shown in Figure 3.4. First, the opcode of a VM instruction is loaded. The opcode is checked to see if it is within the range of available switch labels. Then the opcode is used to look up the jump address (corresponding to a VM instruction label) in a switch table. Finally **one** indirect branch jumps to the VM instruction's implementation. Moreover, there will be a direct jump at the end of each case (a VM instruction implementation) to go to the beginning of the interpreter loop. Because only one indirect branch exists in the whole interpreter to jump to all the labels (instruction implementation), there will be only one entry in the branch target buffer (BTB) in most modern processor. The BTB will store the previous target of an indirect branch, which will provide the target for the following execution of the same indirect branch. Unless all instructions are the same in the VM code streams, the BTB will give incorrect predictions.

As shown in Figure 3.5, the jump to an instruction implementation segment in an interpreter loop is translated to only one indirect branch on a hosted platform when *switch* is used. On modern processors, indirect branches are very hard to predict and are a cause of performance loss in virtual machines due to pipeline stalls when

```
typedef void *Inst;
typedef enum {
  add /* ... */
} Opcode;

void engine()
{
   static Inst program[] = { add, /* ... */ };
   static Inst dispatch_table[] = { &&add /* ... */ };
   Inst *ip = program;
   int *sp;
   goto dispatch_table[ip++]; /* dispatch first VM inst. */
  add:
   sp[1]=sp[0]+sp[1];
   sp++;
   goto dispatch_table[ip++]; /* dispatch next VM inst. */
}
```

Figure 3.6: Token-threaded interpreter dispatch. Source: [EG03]

a pipelined processor speculatively executes in the wrong direction [EG03]. There is much research on restructuring the interpreter loop to improve the indirect branch prediction of an interpreter, such as direct-threaded code [Bel73, Ert93] or to reduce the number of indirect branches by reducing the number of executed instructions, such as superinstructions [Ell05, EGKP02, CGEN03].

3.3.2 Token-Threaded Dispatch

There are some programming languages (such as assembly, Fortran, and GNU C) that support labels as values. These languages allow the value (code address) of a label in a variable. In GNU C, the && operator followed by a label gives a memory location (address) of the code after the label. Token-threaded dispatch [Kli81] takes advantage of labels as values in some compilers, such as gcc, in order to restructure the interpreter loop so that the dispatch will happen after the execution of a VM instruction, as shown in Figure 3.6. A dispatch table holds all the VM instruction implementation addresses in an interpreter. The opcode of the next instruction is used to look up the address of the implementation segment in the dispatch table. One very important

characteristic of threaded code is that there is no need for a loop to execute the VM code. When the execution of one VM instruction ends, the dispatch code at the end of the instruction will dispatch to the next instruction in the code stream. There are multiple indirect branches (one for each VM instruction) in the native code compiled from the interpreter. On modern processors with a BTB, multiple entries exist in the BTB. As long as a given instruction will be followed by the same instruction next time in the code stream, a correct prediction will be made.

Compared to `switch` dispatch, the range-check[1] and the direct jump are eliminated. One big advantage of token-threaded dispatch is to keep the bytecode program unchanged while benefiting from better indirect branch prediction.

3.3.3 Direct-Threaded Dispatch

Direct-threaded dispatch [Bel73, Ert93] goes one step further to eliminate the table lookup operation in the token-threaded dispatch, as shown in Figure 3.7. In order to do this, the VM code needs to be transformed into direct-threaded code, in which the opcodes (usually one byte) for VM instructions will be changed into the addresses (usually 4 bytes on 32-bit hardware platform) of instruction implementation in the interpreter. Direct-threaded dispatch uses memory addresses as the opcodes of instructions to jump to their corresponding implementation directly. The main drawbacks of this approach are an additional translation step and the larger code size for the VM instruction representation, because the size of absolute memory addresses in a 32-bit processor is 4 bytes while it is 8 bytes on a 64-bit processor.

An inspection of MIPS assembly code (Figure 3.8) shows that the direct-threaded dispatch overhead is only four native instructions while *switch* dispatch, as shown in Figure 3.4, requires eight native instructions.

3.3.4 Indirect-Threaded Dispatch

Indirect-threaded dispatch [Dew75] uses an additonal level of indirection to achieve a space saving for VM code. Unlike direct-threaded dispatch [Bel73], the opcode of a

[1]The range check is created by the compiler to make sure that the target of the indirect branch is one of the `switch` label before a table loop-up is carried out. In many cases, the designer of an interpreter are certain that opcodes will be within the range and the check is not necessary.

```
typedef void *Inst;
void engine()
{
   static Inst program[] = { &&add /* ... */ };
   Inst *ip = program;
   int *sp;
   goto *ip++; /* dispatch first VM inst. */
  add:
   sp[1]=sp[0]+sp[1];
   sp++;
   goto *ip++; /* dispatch next VM inst. */
}
```

Figure 3.7: Direct-threaded interpreter dispatch. Source: [EG03]

```
lw   $2,0($4) #get next inst., $4=inst.ptr.
addu $4,$4,4  #advance instruction pointer
j    $2       #execute next instruction
#nop          #branch delay slot
```

Figure 3.8: Direct-threaded interpreter dispatch in MIPS assembly. Register n is denoted by $n, the destination operand of an instruction is the leftmost register, and comments start with #. Source: [EG03]

VM instruction points to a `struct`. The `struct` has an address pointer to the actual VM instruction implementation and the immediate values and/or operand(s) for the instruction. Effectively, the VM code will be translated into a list of pointers to the `struct`s. Repeating operand(s) and/or immediate value(s) will contribute to the space saving. However, because of a level of indirection, indirect-threaded dispatch is not be as efficient as direct-threaded dispatch.

3.3.5 Static Superinstructions

Static superinstruction optimization [Bad95, Hug82, Pro95, Ell05, CGE05] tries to find recurring instruction sequences and combine them together to form new superinstructions. The superinstruction carries out all the operations of the original sequence of component instructions. Thus all the dispatches within a superinstruction are eliminated. Moreover, the compiler will be in a better position to optimize the implementation segments of a superinstruction because of longer native code sequences. The main difficulty with superinstruction optimization is to find the right recurring sequences (superinstructions). Dynamic profiling examines the executed sequences of VM instructions of applications and tries to find the most frequently occurring sequences. The main problem with dynamic profiling is that the recurring sequences discovered in this way are highly biased, which means the superinstruction found in a application could be totally useless in another one. On the other hand, static profiling only examines the static code to discover the frequently recurring sequences. Casey *et al.* [CGE05] found that static profiling usually finds a better set of superinstructions and hundreds of superinstruction are required to gain good performance improvement. For most bytecode VMs, the size of opcodes is only one byte, which allows for 256 opcodes, most of them already used by VM instructions. So superinstruction optimization works best with direct-threaded code because opcodes in VM instructions are encoded with address pointers (4 bytes on 32-bit platform and 8 byte on 64-bit platform) and allow more than 256 opcodes (instructions).

3.3.6 Inline-Threaded Dispatch

After the interpreter core is compiled into native code on a hardware platform, each VM instruction's associated native implementation in the interpreter core consists of

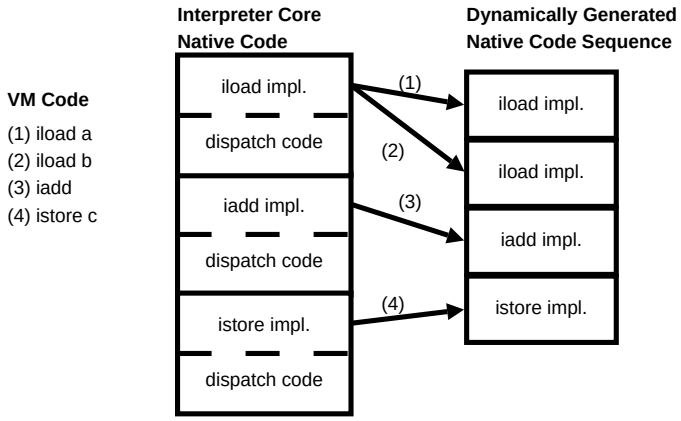

Figure 3.9: Inline-threaded dispatch. The VM instruction sequence to implement the arithmetic assignment: `c = a + b` is on the left. The interpreter core native VM instruction implementation is copied to form the corresponding native code sequence to carry out `c = a + b`

the native code to carry out the function of that VM instruction and the native code required to dispatch to the next VM instruction (Figure 3.9). For a straight line basic block of VM instructions, inline-threaded dispatch [PR98] dynamically copies the native executable code of the interpreter code segments for VM instructions in the sequence without the dispatch native code and concatenates the copied executable code together to form a straight line of native code. Then the new dynamically created native code for the straight line of VM instruction will be executed whenever the same sequence of VM instructions needs to be executed. In this way, all the instruction dispatch code is eliminated for the sequence. Figure 3.9 shows the VM instruction sequence to do `c = a + b`: `iload a, iload b, iadd, istore c`. Generally speaking, one block of native code will usually be created for each basic block in VM code.

Inline-threaded dispatch is probably the fastest dispatch mechanism so far. However, it sacrifices portability and code size to gain performance benefits for the interpreter.

3.3.7 Context-Threaded Dispatch

Context-threaded dispatch [BVZB05] tries to align the virtual PC of the VM code with the hardware program counter to leverage the hardware prediction resources. A

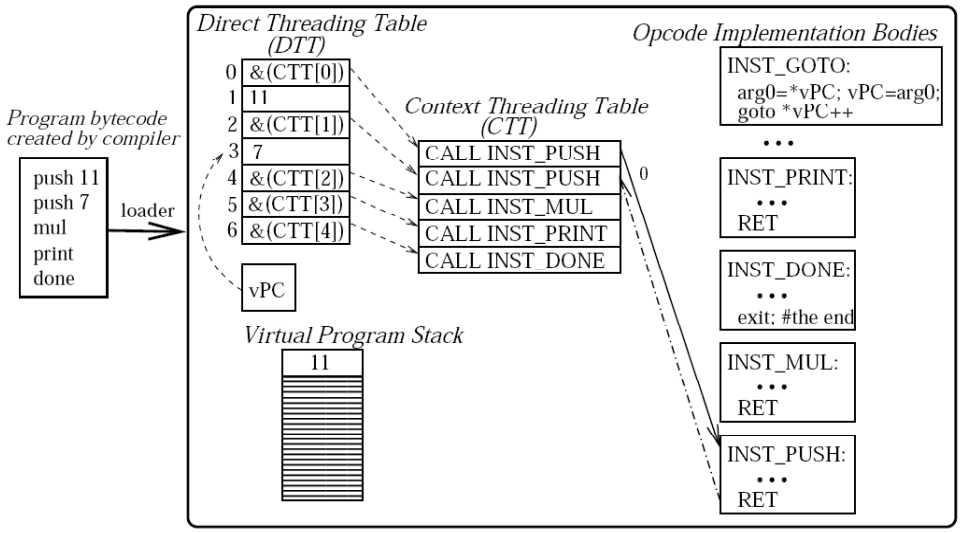

Figure 3.10: Context-Threaded VM interpreter Sequential Code. Source: [BVZB05]

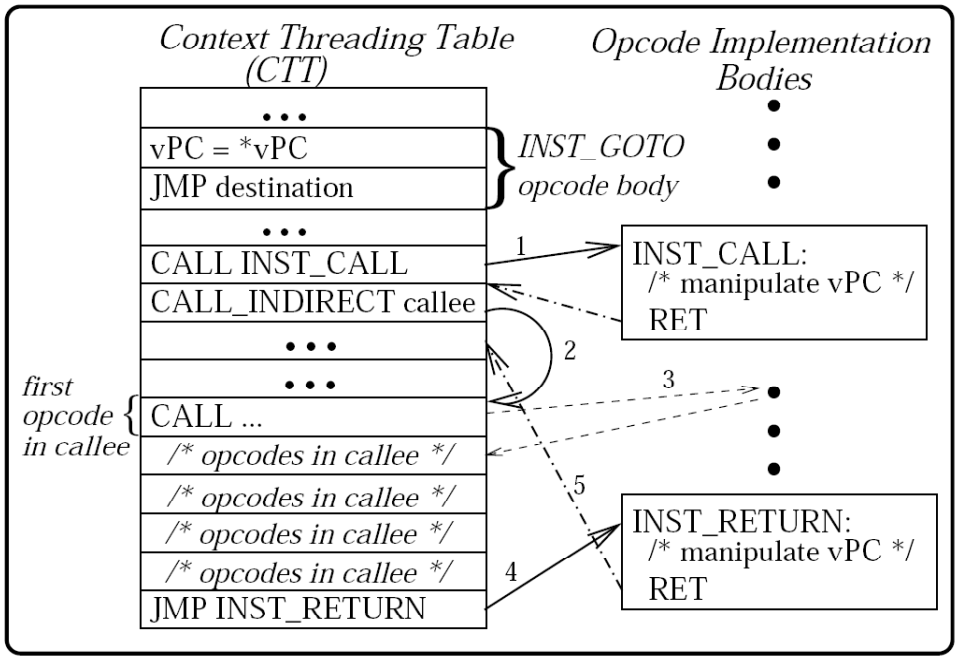

Figure 3.11: Context-Threaded VM interpreter - VM branch instruction handling. Source: [BVZB05]

straight line of VM code is converted into a sequence of function calls (Figure 3.10). Now, the costly indirect branches are replaced with function calls and returns, which take advantage of the hardware return-address stack to make very accurate prediction. Inlining is required to handle the VM branch instructions. VM call and return instruction are handled specially to align the virtual PC and hardware PC (Figure 3.11). Native code generation is required during runtime to generate the native functional calls, handle VM branches and VM calls/returns.

3.3.8 Vmgen Interpreter Generator

Interpreters for different dispatch mechanisms share a lot of common code templates. Vmgen [EGKP02] is an interpreter generator which can be used to produce an interpreter with `switch`, token-threaded, and direct-threaded dispatches. The generator uses an instruction specification file, either in stack-format or register format, as an input. It mainly supports stack-based architectures, and includes stack-caching and superinstruction.

3.3.9 Summary

Figure 3.12 shows all the dispatch cost reduction techniques discussed in this section in terms of efficiency and complexity. Generally speaking, the more complex a dispatch method is, the less portable it is. The *switch* statement is the least complex and most portable dispatch method for building an interpreter. When ANSI C is the only available programming environment, *switch* is the only option. On the other end of the spectrum, inline-threaded and context-threaded are the most complex and efficient dispatch mechanisms. It requires more effort to port inline-threaded and context-threaded interpreters and specific hardware platform knowledge is required. Superinstructions usually require direct-threaded dispatch to allow more than the 256 opcodes available in bytecodes to achieve the benefits of dispatch reduction.

3.4 Interpreter Stack Caching

Interpretation of a VM instruction consists of three parts:

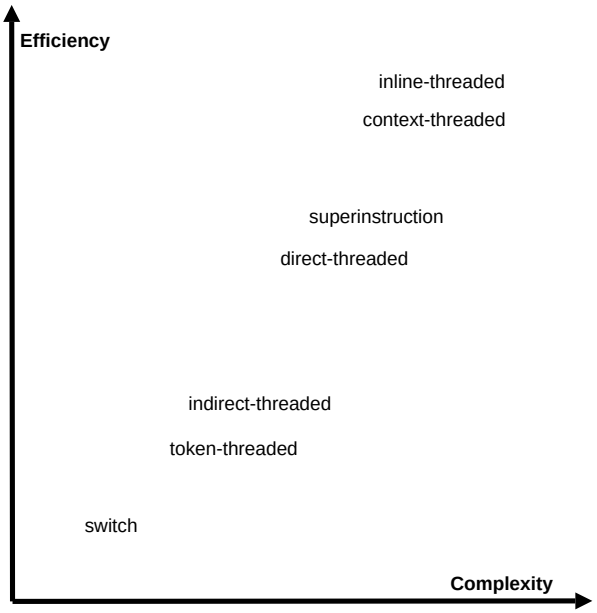

Figure 3.12: Comparison of dispatch reduction techniques

- Accessing operands of the VM instruction
- Performing the function of the VM instruction
- Dispatching to the next instruction

Most virtual machines (Pascal P-Code VM [PD82], Smalltalk [GR83], Java [LY99]) and Microsoft.NET CLR [ECM02]) use stack-based VM instruction architecture. VM instructions implicitly access the operands from the top of an operand stack using a stack pointer (SP) and/or the local variables. The operand stack and the local variables are typically an array of memory locations.

For a stack-based VM, all computation is done through the operand stack. For example, the arithmetic expression `c = a + b` will produce the Java bytecode instructions: `iload a, iload b, iadd, istore c`. The first two instructions push the numbers `a` and `b` from local variables onto the operand stack. These two `iload` instructions will first load the local variables (memory locations) into the physical registers of a real processor and save the values back to the operand stack (memory location). The `iadd` instruction pops the values of `a` and `b` from the operand stack (memory locations $\rightarrow$ physical registers), adds them, and pushes the results back onto the operand stack (move from a physical register into a memory location). The `istore c` instruction will load the value from the operand stack (memory location) into a register and then save the value into a VM local variable (memory location). There is a lot of data traffic between the physical registers and memory locations (the operand stack and the local variables).

One important characteristic of stack-based VMs is that the top of an operand stack is consumed very quickly by the following instructions. Instead of moving the data between physical processor registers and the operand stack (main memory), stack-caching [Ert94, Gri01, PWL04] for a stack-based VM uses physical registers in a real processor as an extension to the operand stack to keep the top `N` elements of the operand stack. For example, the results of the `iadd` do not have to be saved on the operand stack and later to be loaded again to move the result to the local variable `c`. Stack caching has the potential to cut down the traffic between the physical registers and the operand stack (memory locations).

Given `N` registers used for the stack caching, there are `N + 1` cache states. There could be between 0 and `N` top elements of the operand stack stored in the physical

registers. The register cache can overflow and underflow. Overflow happens when there are already `N` values stored in the register and a new value is pushed onto the operand stack. Underflow happens when an instruction needs to access some operands which are not in the register cache. There are two ways to maintain the register cache. The first way [PWL04] is to keep the top of the operand stack in a fixed register, such as the register R_n, as shown in Figure 3.13. When a new value is pushed onto the operand stack, the existing values are shifted down: $R_n \rightarrow R_{n-1}$, ..., $R_2 \rightarrow R_1$, $R_2 \rightarrow (SP + 1)$. The number of shifts depends on the cache state. If the cache is full, the shift will go all the way and one item will be moved to the operand stack. On the other hand, the values from the operand stack will be loaded into the register cache if it is not full after the execution of an instruction. The second way [Ert94] is just to regard the register cache as an integral part of the operand stack. R_1 is the next item above the item pointed by the stack pointer (SP), as shown in Figure 3.13. New items pushed will be loaded into R_1, R_2, ..., and R_n. If the register cache is full, the shift operation will still happen. There will be no prefetching to fill the register cache from the operand stack when the register cache is not full.

In different cache states, the techniques used to access the operands of instructions can be different. There are two schemes [Ert94] to implement stack caching in an interpreter: static stack caching and dynamic stack caching. Static stack caching uses the compiler to analyze the cache states and produce the necessary VM code to maintain the cache state and dispatch to the next instructions. Dynamic stack caching needs one interpreter loop for each register cache state. The interpreter will keep track of the cache state and dispatch to the appropriate interpreter loop. One problem with this approach is code explosion because of the replication of the same interpreter for different cache state. Code sharing [PWL04] can can help to solve this problem.

An interesting variation of stack caching [Gri01] uses the data types of the item in the register cache as the cache state. Only one item is kept in the register. However, the item will be in different registers when the data type is different.

On hardware platforms with limited physical registers, such as Intel X86 processors, it is extremely difficult to dedicate even one physical register to stack caching.

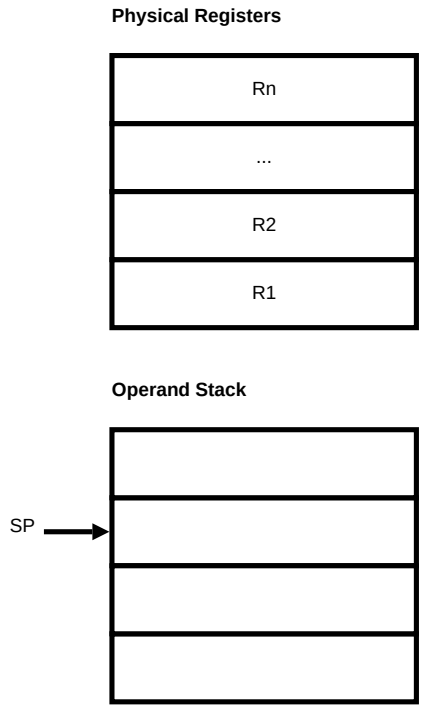

Figure 3.13: Stack caching

3.5 Register Machines

Current computer processors (Intel, AMD, PowerPC) use register-based architectures. On the other hand, virtual machine (VM) implementations have been predominately stack-based (Smalltalk [GR83], Java [LY99], and Microsoft.Net CLR [ECM02]) since Pascal's P-machine [PD82].

3.5.1 Stack vs. Register Instruction Sets

Real Stack Computers

Stacks are widely used in computer science. An evaluation stack is used to compute the value of arithmetic expressions. In a processor, a call stack saves the traces of subroutine calls and returns.

A stack computer with a stack-based instruction set [Koo89] uses a stack to store the operands (temporaries) for instructions (computation). In a stack computer, most of the instructions have implicit operands on the top of the operand stack. Any result

produced by an instruction will be pushed onto the operand stack. There are two important instructions `load` and `store`. The `load` instruction pushes a value from an arbitrary RAM location onto the top of the computational stack and the `store` instruction saves a value from the top of the computational stack into a memory location. The main advantages of a stack-based instruction set are:

1. Very high code density compared to other form of instruction sets (such as register-based instruction sets)
2. Simplicity of the instruction set
3. Simple compiler implementation to generate stack-based code from source programming language.

Real Register Computers

A register machine uses the registers to store the operands (temporaries/result) of instructions (computation). In a register machine, the operands (register/memory location) must be encoded as part of an instruction. Most compilers for register architectures will use registers as much as possible because accesses to the registers are faster and a limited number of registers allow for shorter encoding of the instructions. Generating code for a register machine is more complex and a sophisticated register allocator is often needed to make best use of a limited number of registers to maximize the performance of a source program.

Comparison

There have been many arguments between stack and register-oriented instruction set architectures. Glenford J. Myers [Mye77] compared register architectures, stack architectures, and storage to storage architectures. He used assignments and simple arithmetic expressions as examples to draw the conclusion that stack architectures are not superior and that storage to storage architectures (more close to VM register architecture) are more desirable. In reply to Myers [Mye77], Schulthess and Mumprecht [SM77] argue that Myers's examples only represent a subset of programming language usage cases. It is more difficult to draw any definitive conclusions.

3.5.2 Register-Based Virtual Machines

There have been several virtual machines (Dis [WP97], Perl 6 [Fag05] , Lua 5.0 [IdFC05], Mamba [PA02], and Rain VM [Bro06]) implemented with register instruction set architectures.

Dis

Dis [WP97] is a virtual machine with a register architecture created at Bell Labs to support application portability. The extra memory traffic of a stack-based VM was given as the reason for using the register-based VM. From their experiences of implementing a stack computer in the AT&T Crisp microprocessor, Winterbottom et al. [WP97] believed that a stack architecture is inherently slower that a register-based machine. Moreover, it was argued that closer resemblance of VM instructions and real processor instruction allows easier JIT compilation. Dis uses a three-address instruction encoding. The first source and the last destination operand can be memory addresses or arbitrary constants while the second operand is limited to smaller constants and stack offsets to reduce code size. Each operand specifies an address either in the stack frame of the executing procedure or in the global data of its module.

The Parrot Virtual Machine

Perl 6 [Fag05] moves away from its earlier stack-based versions to a new register architecture. It is intended to support multiple languages including Perl itself. It has many higher-level features such as objects, thread synchronization support and garbage collection. The designers [Fou07] of Perl 6 give some of the following reasons for moving to the register architecture:

1. Fewer register-based VM instructions are required than those of a stack VM.
2. More research in optimization for register-based hardware to take advantage of.
3. Break away from the tradition of stack VM implementation to innovate.

Parrot originally used a scheme similar to a real processor. It had four groups (integers, floating-point numbers, strings and PMCs) of 32 registers. In the later

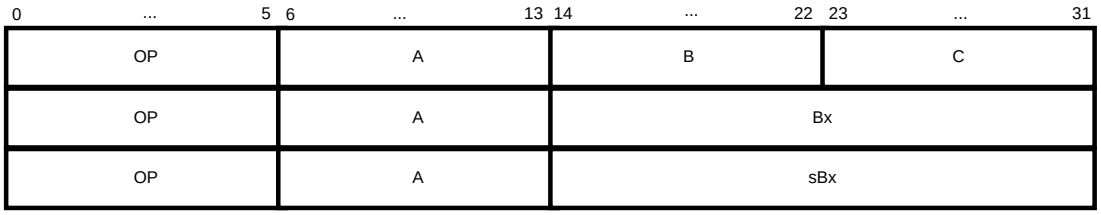

Figure 3.14: Lua 5.0 instruction format. Source: [IdFC05]

evolution, the number of registers became unlimited to eliminate register spills. The virtual registers of the Parrot VM are stored in a register frame. These frames can be pushed and popped onto a virtual register stack.

Lua 5.0

Lua is a scripting language widely used in game industry. Lua 5.0 [IdFC05] moved to register-based architecture partly because of earlier work on register machines in our group. There are 35 instructions in Lua's virtual machine. Virtual registers are kept in the run-time stack, which is implemented with an array. Constants and upvalues are also stored in arrays. Lua 5.0 uses 32 bits instruction encoding, as shown in Figure 3.14. The first 6-bits are the opcode. The next 8 bits are the first operand (A) and always present. The second (B) and third (C) operands are 9 bits. These second and third operands can be combined into one larger operand. Performance comparisons between Lua 5.0 and 4.0 show around a 20% improvement.

Mamba

Mamba [PA02] is a new VM for Python. The new register instruction set enables the number of VM instructions to be reduced to 18 from 103 stack-based VM instructions. The reduced instruction set size is achieved by moving the functionality from the instructions into the objects that the instructions act upon. There is a maximum of 4096 registers in the Mamba virtual machine. All the registers are the same; they can be used with any instructions. The first 64 registers are global; their contents are visible across function calls. The remaining registers (4032) are local; their values are saved before a function call and restored after the return from the call. All instructions (except `move`) use one byte (8-bits) to encode the operand. The `move` instruction uses

a 12-bit operand encoding, enabling accesses to all of the registers.

Rain Virtual Machine

The Rain VM [Bro06] is implemented with a register architecture for concurrency. The decision was made based on research done for this dissertation [SGBE05] to demonstrate that a register interpreter-based VM has better performance than a stack VM. In the Rain VM, the instruction set uses 32-bits to encode two-address instructions. The first byte is the opcode and the second one is a sub-opcode. The third and fourth bytes are source and destination operands. The registers are addressed by one byte (256 registers). The Rain VM initially allocates 8 or 16 registers. The VM can increase the number of registers as needed. It uses context pointers to quickly switch between threads. Part of the context information is related to register blocks. It is not clear from the paper whether Rain VM creates a new register block or saves/restores the register block for each method/function call/return.

3.5.3 Virtual Register Organization

There are various ways to organize the virtual registers in a virtual machine. Some VMs (such as earlier versions of the Parrot VM [Fag05] and Mamba [PA02]) have a fixed number of general purpose registers or even a fixed number of registers for different data types, like a real processor. The state of the registers has to be saved/restored for function calls and returns. Another problem with a fixed number of registers is that a register allocator is needed and virtual register spilling can happen. This can cause a lot of memory copy operations. Other VMs (such as the current version of the Parrot VM [Fag05]) create a new set of registers on a stack for each method call. Usually the number of required registers can be determined when compiling the source code. The number of addressable registers is limited by the size of operands (typically one byte). 256 registers are usually more than enough for modern object-oriented programming languages which encourage small methods. A register allocator is not needed, although one can be used to minimize the number of registers to save some space. Furthermore, all the VM registers are not physical registers in a real processor. They are typically represented using an array and indexed by an integer (register number).

3.5.4 Java Virtual Machine Related Research

Davis et. al. [DBC+03, GBC+05] began the first large-scale quantitative study of stack vs. register instruction set architectures on the Java virtual machine. They translated the stack-based Java bytecodes into register ones. All the stack push/pop instructions were translated into `move` instructions. A simple copy propagation algorithm was applied to eliminate the redundant `move` instruction in the register architecture. Of the resulting register code, the number of executed VM instructions could be reduced by 35% while the bytecode loads increased by 45%. There were no timing results.

3.6 Indirect Branch Prediction

Branch instructions in a program are used to transfer the execution of a program from one part to the other. There are mainly two types of branch instructions: conditional branch instructions and unconditional branch instructions. Unconditional branch instructions can be further divided into unconditional direct jump instructions and unconditional indirect jump (indirect branch) instructions. A conditional branch instruction has two directions of program control flow: the fall-through target (the next instruction) and the jump target (encoded in relative or absolute address as part of the instruction). An unconditional direct jump has only one target. An indirect branch (indirect jump) is a type of branch instructions, whose targets are determined by a computed value in a register. Indirect branch instructions in a processor can have more than two possible branch targets.

On modern wide-issue and deeply pipelined processors, speculative execution beyond branch instructions is necessary to better take advantage of available hardware resources and gain more benefits from instruction-level parallelism. The penalty is very high when the speculative execution goes in the wrong direction. When this happen, all the work done beyond a branch has to be flushed and restarted in the new direction of execution.

Conditional branches can be predicted very accurately (over 97%) [YP93]. Unconditional branches will always be taken and can be predicted very accurately using branch target buffer (BTB) because of their static targets. Indirect branch prediction has received relatively little attention because there are a relative low percentage of

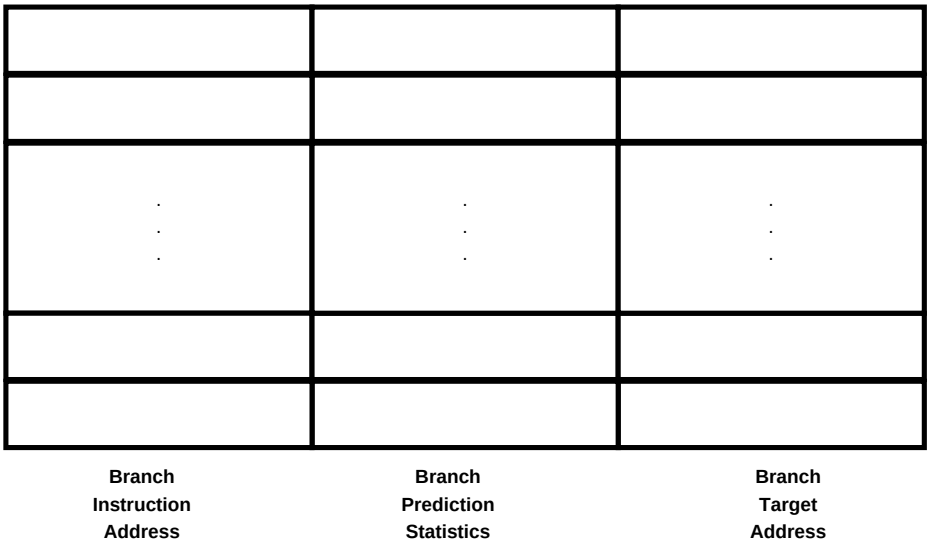

Figure 3.15: Branch target buffer organization. Source: [PS93]

them in general applications. For example, only one out of 12 benchmarks in SPEC-Cint200 has over 1% indirect branches in executed instructions [SÖG06]. The trend towards object-oriented programming languages will increase the numbers of indirect branches (2% in C++ programs) [DH98]. Interpreters as a category of applications usually have a much higher percentage (up to 13% of executed instructions) [SÖG06].

The most widely available branch target prediction mechanism in current processors is the branch target buffer (BTB). The branch target buffer is a small cache (see Section 2.4.1) that retains the addresses of recently executed branches and their targets [SL84, PS93]. The BTB can reduce the performance penalty of branch instructions by predicting those branches and caching information about their most recent targets. As shown in Figure 3.15, there will usually be three types of information stored in a BTB: a tag identifying a branch instruction (usually the branch instruction address), branch prediction statistics to help make branch predictions, and the target address of the branch instruction. Typically, a branch instruction address is used to index into a associative set in the BTB, which contains multiple entries. Then the branch instruction address is compared. If a matching entry is not found in the set, no prediction is made. After the branch is resolved, the information about the branch instruction is added to the relevant set. If the branch instruction can be found in the set, a prediction can be made. After the branch is resolved, information about the branch is used to update the

corresponding entry. For an indirect branch instruction, the direction of the branch is always taken. However, the target address of corresponding entry will be updated if the predicted target address is incorrect.

The default BTB misprediction update policy is not very effective (51.8% prediction rate for SPECint 95 [CHP97]). It will always predict the previous target address as the target address of the same indirect branch instruction when it is executed again.

3.6.1 BTB with 2-bit Counters

Galder and Grunwald proposed a BTB with 2-bit counters (BTB-2bc) to improve the indirect branch prediction for C++ programs [CG94]. BTB-2bc stores a 2-bit counter for each indirect branch in the BTB. The branch target will only be updated after two consecutive mispredictions of the branch target. The scheme reduces the misprediction rate from an average of 28.1% for the standard BTB to 24.9%. The BTBs with 2-bit counters perform well when a few targets dominate and there is only an occasional target change. Polymorphic branches occasionally switch their target, a situation observed in object-oriented programs [AH96]. However, Chang et al. [CHP97] found that 2-bit strategy is not very successful in predicting the targets of indirect branches in C programs such as the SPECint95.

3.6.2 2-Level Prediction of Indirect Branches

A 2-level conditional branch predictor can reach the prediction rate of 95-97% [YP93]. The success of 2-level conditional branch prediction inspired an interest in indirect branch prediction research [CHP97, DH98].

Chang et al. [CHP97] proposed a two-level indirect branch target prediction with a target cache (similar to a BTB). The branch instruction address and branch history/path history are hashed to index the target cache, as shown in Figure 3.16 and 3.17. This makes it possible for the same indirect branch with different histories (same/different indirect branch target) to coexist in the target cache. A tagless (direct-mapped, see Section 2.4.1) target cache, as shown in Figure 3.16, uses an indirect branch instruction's address and history information to index to a BTB entry. Two indirect branch instructions may be mapped to the same BTB entry. Thus it will create interferences (conflict) between different branch instructions. A tagged

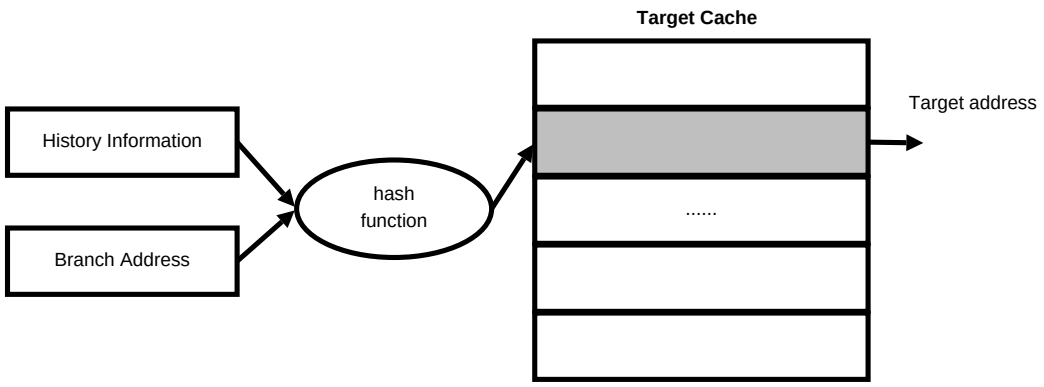

Figure 3.16: Structure of a Tagless Target Cache. Source: [CHP97]

(set-associative, see Section 2.4.1) target cache , as shown in Figure 3.17, solves the interference problem by first indexing into a set in a set-associative target cache. Then a tag, usually the branch address, is used to select the right indirect branch instructions. The pattern history uses the target address of previous branches. The pattern history using the indirect branch target address is particularly effective for SPECint95 *Perl*, which is an interpreter. Change et al. [CHP97] showed that a 16-way set associative tagged target cache with pattern history can reduce the execution time of gcc and Perl by 12.66% and 4.74% respectively.

Driesen et al. [DH98] also used a two-level indirect branch predictor. The branch predictor has a global (shared) history pattern which consists of past indirect branch target addresses. They compared different schemes for two-level indirect branch prediction and found that a global history pattern and per-address table resulted in the lowest misprediction rate of 6.0% [DH97]. The global history pattern together with the indirect branch address is hashed to index into a history table (similar to BTB with 2-bit counter update policy) to make the indirect branch target prediction. In their study, only two consecutive mispredictions of the indirect branch target will cause the target to update in the BTB. They started from unlimited resources (no hardware constraints on predictor size or organization) to discover the best strategy for making the indirect branch prediction. The best unconstrained predictor achieved a misprediction rate of 5.8% while a fully-associative branch target buffer (BTB) only achieved a best-case misprediction rate of 24.9%. For a 4-way associative table, the misprediction rate of the best hybrid predictor improved to 8.98% for 1K entries and 5.95% for 8K

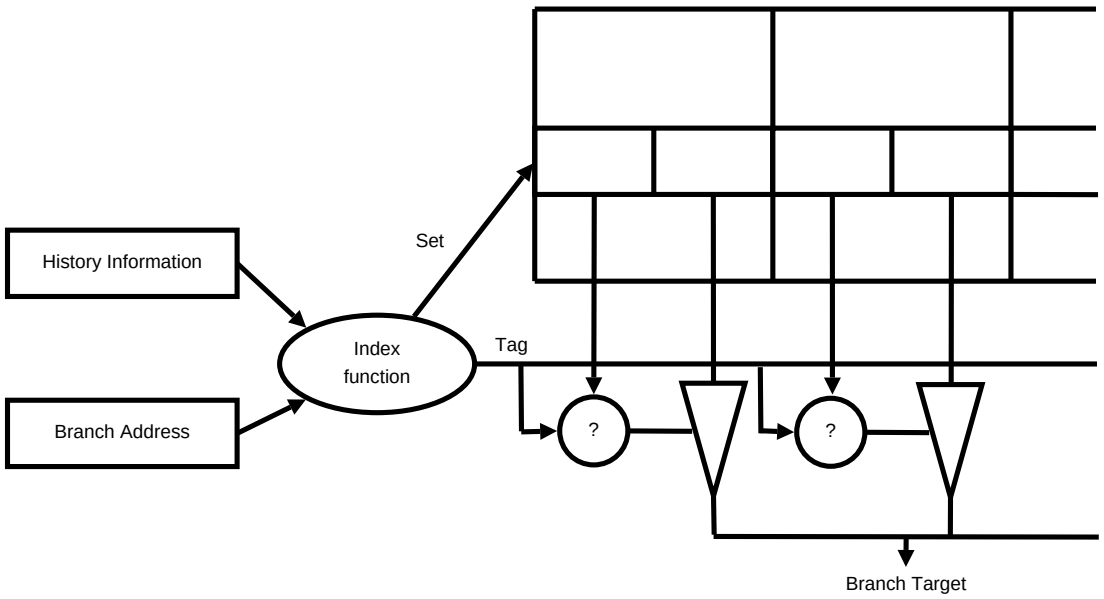

Figure 3.17: Structure of a Tagged Target Cache. Source: [CHP97]

entries.

It is important to note that the past path/pattern history provides context for improving indirect branch prediction. The global path history consisting of previous indirect branch target addresses performs better than other history schemes [DH98, CHP97]. For an interpreter, the global path history helps to align the virtual code sequence to the predictor's history state.

3.7 Trace Cache

The pipeline of a superscalar processor can be divided into instruction fetch/decode and instruction execution, as shown in Figure 3.19. The instruction issue buffers are the interface between instruction fetch mechanism (producer) and instruction execution mechanism (consumer). Instruction fetches are usually done sequentially by incrementing the instruction pointer. Branch instructions redirect the instruction fetching to new locations. Thus branch instructions act as feedback to the instruction fetch mechanism from the instruction execution mechanism.

As the issue width of a superscalar processor increases, more and more instructions

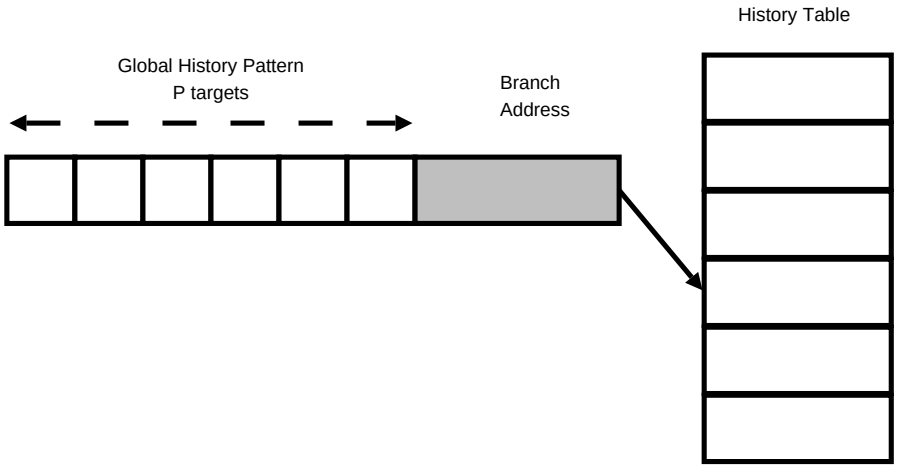

Figure 3.18: Two level indirect branch prediction. Source: [DH98]

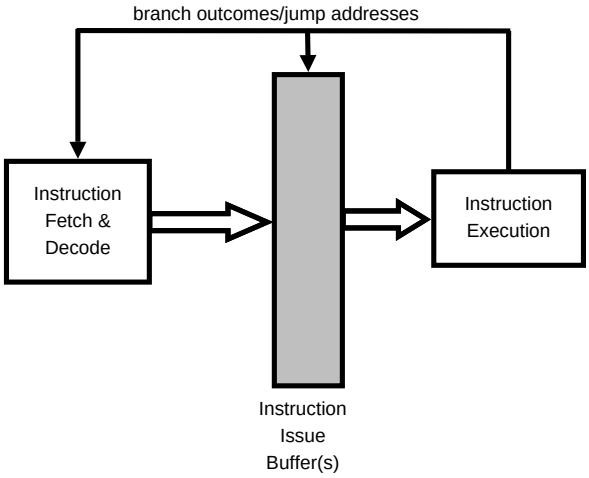

Figure 3.19: Instruction fetch and execute mechanisms, separated by instruction buffers. Source: [RBS96]

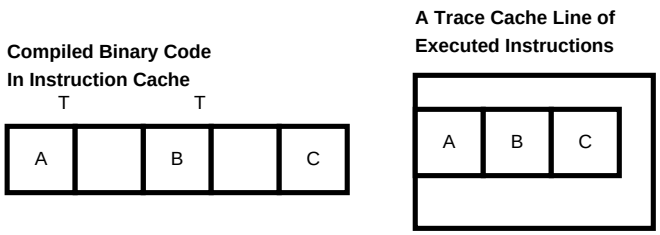

Figure 3.20: Non-contiguous compiled code in the instruction cache is shown on the left. The same code in a contiguous trace cache line is shown on the right. The execution order is basic block A, B and C in non-contiguous location in their compiled form. The trace will be stored sequentially in a trace cache line after it is executed for the first time.

needs to be fetched per-cycle to satisfy the instruction demands of the instruction execution engine. There are only 4 to 5 instructions in a basic block in most integer code. For a superscalar processor, an issue rate of over 4 instructions per cycle means fetching instructions across basic blocks per cycle to keep the pipeline full. A single branch predictor can predict the target of one branch instruction. Instruction fetch can begin from the target address until the next branch instruction. With a single branch predictor, only one basic block can be fetched from the instruction cache and/or memory per cycle because the target of the next branch is still unknown. More than 60% of branch instructions are taken branches. Fetching instructions across basic blocks means accessing non-contiguous code in their compiled form, as shown in left part of Figure 3.20. Even though the instruction cache can be enhanced to fetch multiple basic blocks per cycle from the instruction cache with a multi-branch predictor [CMMP95, SJSM96], it will create complexity and add latency at the critical path of the instruction fetch mechanism.

The trace cache [RBS96, PFP97] was proposed as a scheme to improve instruction fetch bandwidth in order to fully exploit instruction level-parallelism with low latency and complexity. The basic idea is to capture snapshots of the executed instruction stream and to store them in a special cache (trace cache), as shown in the right part of Figure 3.20. Later on, the same execution sequences stored in a trace cache line is predicted to be in the next execution path; the cached and contiguous code in the trace cache line consisting of multiple basic blocks can be issued into the instruction issue buffers in one cycle.

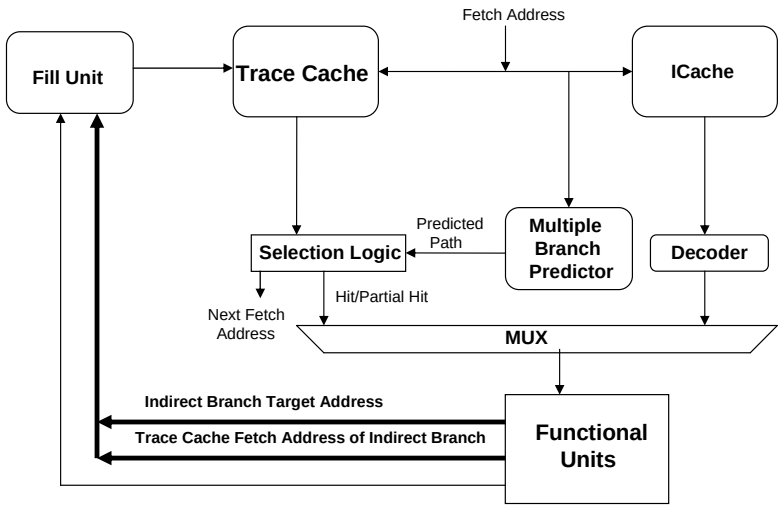

Figure 3.21: Trace cache microarchitecture

A trace cache consists of multiple trace cache lines. Each trace cache line has `n` slots for storing dynamic instruction sequences with no more than `m` basic blocks (branches). Each trace cache line has the following information [RBS96]:

1. *valid bit*: indicates this is a valid trace.
2. *tag*: the tag field identifies the starting address of the trace.
3. *branch flags*: a single bit for each branch within the trace to indicate the path followed after the branch (taken/not take).
4. *branch mask*: indicate (1) the number of branches in the trace and (2) whether or not the trace ends in a branch.
5. *trace fall-through address.*
6. *trace target address.*
7. The alternative target addresses for the conditional branches before the last branch instruction which ends the trace cache line. These target addresses are needed in Patel et al's trace cache performance improvement technique - partial matching [PFP97].

The trace cache works together with the existing instruction cache to supply instructions to the execution engine of a processor, as shown in Figure 3.21. When an instruction is needed, the fetch address of the instruction is sent to both instruction cache and trace cache. The multiple branch predictor will supply the direction of the branches in the following instruction stream given the fetch address. The fetch address is used to index into the trace cache to see whether a trace is available at the fetch address. If one is found, its branch flag (execution path) is compared with those made by the multi-branch predictor. The trace cache line predicted to be in the right execution path will be delivered to the instruction issue buffer. Otherwise, the instruction cache will supply the needed instructions. A fill unit will collect instruction traces retired from the functional units. Those traces can then be added to the trace cache or can be used to update the existing traces.

Since its introduction into mainstream microarchitecture research in 1996, the trace cache was studied mainly as a high instruction delivery mechanism for superscalar processors.

Patel et al. [PFP97, FPP97, PEP98, PFP99, Pat99] focused on approaches to improve the fetch rates of the trace cache. They examined issues and techniques related to the partial-matching of trace cache lines, new indexing functions to create path associativity, inactive issue with a partial-matching trace cache line, issuing dual path trace segments, branch promotion of highly predictable trace, and trace packing to allow more replication of execution traces. The set-associativity and updating policies of trace caches were also studied. The fill unit was studied to find out whether it is better to collect traces when instructions are issued to, or retired from, the execution unit.

Rotenberg et al. [RBS96] first demonstrated that the trace cache has better performance and lower latency than existing instruction cache enhancement mechanisms for improving the instruction fetch rate per cycle. They went further to propose a new microarchitecture - the trace processor [RJSS97], which is trace-centric. In the proposed trace processor, a next trace predictor will make the predictions instead of a multi-branch predictor. The control, register and memory dependencies are handled at the trace level using value prediction.

One important part of using a trace cache is to make multiple branch predictions. One way to do this is to extend existing two level single branch predictors

to predict multiple branches in one cycle without losing too much prediction accuracy [RBS96, PFP97]. The next stream predictor [SFR+02] is able to predict a sequence of executed instructions starting from the target of one taken branch to the next taken branch. Coupled with an optimized code layout, the 40KB stream predictor is 22% better in mispredictions per instruction than the best evaluated *gskewed* predictor. In a next trace predictor [JRS97], a trace is treated as basic units. The next trace predictor predicts explicitly sequences of traces instead of individual branches inside, which are predicted implicitly. From the results of six chosen benchmarks, the average misprediction rate of next trace predictor is shown to be 26% lower than the most aggressive previously proposed multiple-branch predictor.

3.8 Conclusion

In this chapter, we first reviewed the techniques for reducing the dispatch cost of an interpreter. Then we looked at stack-caching - a way to reduce the cost of operand accesses in a stack-based interpreter. Unlike real processors, register-based VM is not the predominate choice for VM implementation. We compared the difference between the two architectures and studied some existing register VM implementations. Finally, we reviewed the trace cache and some variations.

In the next chapter, the research on the effect of the trace cache on indirect branch prediction is presented.

Chapter 4

The Trace Cache and Indirect Branch Prediction

4.1 Introduction

Instruction-level parallel (ILP) processors, whether superscalar or VLIW, require a large number of functional units to extract higher ILP from applications. A wider execution engine demands a wider instruction fetch unit that must potentially match the width of the functional units in order to fully utilize them. This means that the instruction fetch unit must fetch a very large number of instructions from the instruction cache across several basic blocks at every cycle. In general, these basic blocks are placed in non-contiguous locations in the instruction cache. More than one cache access may have to be performed in order to fetch all the required basic blocks. This non-contiguous fetching places limits on the number of instructions that can be fetched in a single cycle (of reasonable length). Hence, the trace cache has been proposed as an alternative approach to fetching multiple basic blocks in a single cycle.

The trace cache is a microarchitectural technique for increasing instruction fetch bandwidth of a superscalar processor. The processor collects snapshots of the executed instruction stream and stores them as trace cache lines along with branch target addresses. This way, instructions from non-contiguous execution path are placed into contiguous locations in the trace cache. When a trace cache line is predicted to be the next execution path, the entire trace cache line can be fetched in a single cache access.

In general, most general purpose C and Fortran programs have low numbers of indirect branches, and conditional branches and function returns are the most important types of branches to predict. However, the frequency of using indirect branches will be much higher in future applications [DH99, SFF+02], although conditional branches will continue to outnumber indirect branches in most applications. For example, object-oriented programs use larger numbers of indirect branches to implement virtual function calls. Dynamically linked libraries are also called using indirect branches. Finally, an increasing number of program languages are being implemented using virtual machines that are implemented partially (such as Sun's standard client JVM) or wholly using interpretation, a technique that involves very large numbers of indirect branches. Thus, higher indirect branch prediction accuracy rates will be sought in future ILP processors.

We observe that in an ILP processor with a trace cache, it is possible to improve the indirect branch prediction accuracy of programs using the trace cache to make indirect branch predictions instead of using a branch target buffer. Furthermore, using this scheme involves in very simple modifications to the trace cache hardware structure without using any extra hardware tables.

In this chapter, we show how the trace cache can capture some of the context information used by two-level indirect branch predictors [CI01][DH98][KK98]. Although the improvement in accuracy is much lower than that achieved by a two-level indirect branch predictor, the cost is also much lower. If one has already decided to implement a trace cache, then it can be used to significantly improve indirect branch prediction over a BTB at little or no additional cost. We are not attempting to compete with two-level indirect branch predictors or next-trace/next-stream predictors [JRS97] [SFR+02]. We merely show that some fraction of the benefits of two-level prediction can be captured by the trace cache, a result that is not widely known.

The main contribution of this chapter is an exploration of the observation that the trace cache captures context information about the control-flow of the program. This context information can have an impact on the accuracy of other predictors, in this case the indirect branch predictor. Capturing this sort of context information is, to our knowledge, a mostly unintended side effect of the trace cache. Although others have noticed the same effect in other contexts (see Section 4.8), the results are under-reported and not well-known. Our results are interesting primarily because

of the practical effect on prediction accuracy. However it is also interesting because it demonstrates how different microarchitectural features can interact in unexpected ways.

Based on our initial observation, we have a number of smaller contributions, which come from exploring variations of the trace cache. 1) We propose to update the indirect branch target address in the trace cache if a trace cache line ends with an indirect branch instruction. We will motivate the use of the update policy for the rest of the chapter. 2) We show that a 2-bit saturating update counter associated with an indirect branch target address at the end of each cache line can improve prediction accuracy in much the same way that such counters are used in other predictors. 3) We measure the impact of each individual trace cache configuration or strategy on the prediction of indirect branches such as using trace packing or varying cache size, line size and associativity.

4.2 Background

Most trace cache research [Pat99][RLPN+99][RBS99][RBS96] has focused on how different configurations of the trace cache can help improve the instruction fetch bandwidth of superscalar processors. *Jacobson et al.* [JRS97] predicts indirect branch target addresses using the next-trace predictor along with all other branch types in trace caches. *Santana et al.* in [SFR+02] and [SRLPV04] uses the next-stream predictor, which is quite similar to the next-trace predictor, to predict indirect branch target addresses. Both the next-trace and next-stream predictors use large tables to predict all types of branches in a single branch prediction framework. On the other hand, we propose, in this chapter, a simple updating mechanism in the trace cache that can moderately improve the indirect branch prediction rate without any extension in the BTB mechanism.

4.2.1 Trace Cache

The trace cache is a microarchitectural technique for increasing instruction fetch bandwidth of a superscalar processor. The model of the trace cache in this chapter is based on the description in Patel's *PhD dissertation* [Pat99]. A trace cache system consists

of an instruction cache, the trace cache, a multiple-branch predictor and a fill unit. The trace cache contains 2^x trace cache line entries ($x > 0$) and supplies the functional units with stored trace cache lines. Each trace cache line (Figure 4.1) stores multiple basic blocks for an execution path, namely it can contain up to n instructions with no more than m conditional branches. The line is accessed by only a fetch address of the first instruction in the first basic block. It is possible to make the trace cache path-associative, which allow the storage of multiple trace cache line starting with the same address. But for simplicity, we don't implement that feature because the the set-associative trace cache can have the same effect. Each trace cache line contains the starting address of a trace, the path information (the number and directions of branches), the target address of the basic blocks to generate the next fetch address even for the partial match of a trace. In the case of a trace cache miss, the instruction cache provides instructions to the functional units.

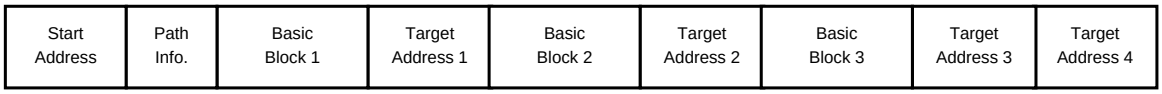

Figure 4.1: A trace cache line of 3 basic blocks

The multiple-branch predictor, which is based on a two-level branch predictor, is used to predict m branches simultaneously for a trace cache line [Pat99, P3]. The branch target buffer (BTB) provides target addresses for the predicted-taken branches in case of trace cache miss. The BTB saves only one target for each *taken* branch. A target address in the BTB is updated when the target address changes. For return instructions, a return address stack (RAS) is used. Each time a function call is executed, its return address is pushed onto the RAS. The branch predictor uses the RAS to get the return address for a return instruction.

The fill unit forms execution traces and places them into the trace cache as instructions retire from the functional units. A trace is terminated when it contains n instructions or m conditional branches or an indirect branch, return or a trap instruction. If there is a trace cache line starting at the same address, the new trace cache line replaces the existing trace cache line only when it is longer or follows a different execution path (i.e. *keep-longest* write policy).

4.2.2 Indirect Branch Prediction

Indirect branch instructions are control instructions whose target addresses are loaded into registers. Thus, they can have multiple branch targets and this makes them very hard to predict [CG94][CHP97][CI01][KK98]. The most commonly used predictor for indirect branches in current processors is the BTB, which caches only one target address for each indirect branch instruction. The predicted target address is updated when the actual target changes.

Two-level indirect branch predictors combine the address of the branch with a *history* register of recent branch targets. They are effective because there is often a correlation between the outcome of different indirect branches. The history register stores context information on the outcome of these other branches, allowing indirect branch prediction rates of more than 90% [CI01][DH98][KK98]. To achieve the very best prediction accuracy, a two-level indirect branch predictor should be used. However, such predictors can be large and complicated.

4.3 Indirect Branch Prediction using Trace Cache

The trace cache can store multiple indirect branch targets in different trace cache lines ending with the same indirect branch instruction. Each trace cache line stores two pieces of context information that are useful for indirect branch prediction. First, every trace cache line has a starting address, which provides some context information for any indirect branch in the line. Secondly, if there are conditional branches before an indirect branch in a trace cache line, the directions of these conditional branches provide a context or path history for the indirect branch target. Although this context information is not as complete as the information in a two-level indirect branch predictor, we show in this chapter that it can be used to provide better predictions than a BTB.

The inherent property of storing multiple indirect branch targets in different trace cache lines ending with the same indirect branch in the trace cache provides us with a storage unit capable of storing multiple target addresses for an indirect branch. Thus, we propose to explore this inherent property of trace cache to attain higher indirect branch prediction rates by incrementally adding new functionalities such as updating

indirect branch target address in the trace cache lines, adding a 2-bit saturating update counter, associated with an indirect branch target address, to each trace cache line, using trace packing and tuning the trace cache parameters such as cache size, cache associativity and cache line size.

The trace cache described by Patel [Pat99] does not update indirect branch addresses stored at the end of each trace cache line after the indirect branch instruction is executed. If the indirect branch target is mispredicted, it is not updated in the trace cache line unless the old trace cache line is replaced by the new one if the trace cache write policy is *always overwrite* [Pat99].

```
for (i = 16; i >= 0; i--) {
    offset = i & 7;
    switch( intArray[offset] ) {
        case 0: x = y + z; break;
        case 1: x = y * z; break;
        case 2: x = x + y + z; break;
        case 3: x = y - z; break;
    }
}
```

Figure 4.2: Sample loop for case study

An example of a *switch* statement inside a loop in **Figure 4.2** is presented to demonstrate the effects of the trace cache on the indirect branch target prediction. The targets of the *switch* statement are decided by an array holding 8 integer numbers. The loop simply iterates through the same integer array elements (targets) 16 times.

Figure 4.3 shows the basic blocks and program flow chart constructed from machine code. Here; A, B, C, X0, X1, X2 and X3 denote the basic blocks in the program. The conditional branch at A and indirect jump at B are the *switch*- related instructions. The direct jump is the jump instruction for *break* statements in *case*s. The conditional branch at C is the loop test instruction. Finally, X#(0-3) is the basic block associated with each *case* statement.

First, let us assume that we use only a BTB to predict branches. If the array contents are (0, 1, 2, 3, 0, 1, 2, 3), the indirect branch prediction accuracy becomes *0%* because the indirect branch target is updated in the BTB after each iteration. If the

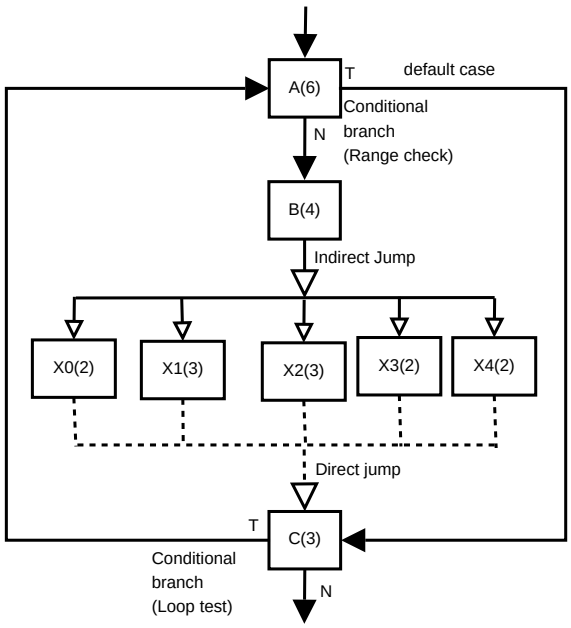

Figure 4.3: Basic Block Program Flow Diagram

array contents are (0, 1, 2, 3, 3, 2, 1, 0), then the prediction accuracy rises to *18.75%*.

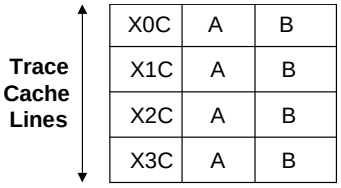

Figure 4.4: Trace cache line layout for the case study example

When the trace cache is used as shown in **Figure 4.4**, basic blocks A and B become replicated a number of times. Trace cache lines must always end at an indirect branch, so each of the successors of the indirect branch in block B (i.e. blocks X0..X3) becomes the start of a trace cache line. Each of these trace cache lines include their own copies of basic blocks A and B. Therefore, the indirect branch at the end of block B is split into four different instances, each with context information about the most recent outcome of the same indirect branch. In this case, the trace cache is equivalent to a two-level indirect branch predictor with a history length of one. The trace cache lines and their indirect branch targets are perfectly correlated and the indirect branch target

prediction rate becomes *75%* with or without the updating policy for the branch target sequence of (0, 1, 2, 3, 0, 1, 2, 3). However, the prediction accuracy drops to *25%* with no updating and *0%* with the updating policy if the branch target sequence is (0, 1, 2, 3, 3, 2, 1, 0).

Updating or not updating indirect branch targets can have significant impact on the prediction of indirect branches depending on the target address access pattern. The update policy can outperform the non-update policy if one target is repeatedly accessed. For instance, the prediction accuracy is only *18.75%* for the non-update but *68.75%* for the update policy if the target address pattern is (0, 1, 2, 1, 1, 1, 1, 1).

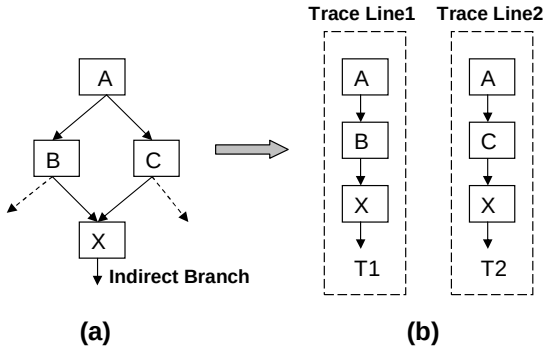

Figure 4.5: Trace cache lines starting with the same address may end with the same indirect branch instruction in a set-associative trace cache.

If the trace cache is designed as set-associative, then it can have different trace cache lines starting with the same address leading to the same indirect branch instruction. For instance, **Figure 4.5a** shows a control flow graph of a program that ends with the same indirect branch instruction. Here; A, B, C and X represent basic blocks where A is the starting and X is the ending blocks. Two traces are shown in **Figure 4.5b** in which the trace lines 1 and 2 follow ABX and ACX paths and each having the same indirect branch instruction in X but with two different target addresses (i.e. $T1$ and $T2$). If two trace cache lines paths somehow correlate to their corresponding targets, the indirect branch prediction accuracy can be improved. Now, let us assume that we choose to update indirect branch targets in the trace cache lines when they change. If the ABX and ACX paths always take the indirect branch targets $T1$ and $T2$, respectively, then the indirect branch prediction accuracy drastically improves. However, the indirect branch prediction accuracy can be very poor for the following

sequence:

$$ABX \rightarrow T2,\ ABX \rightarrow T1,\ ABX \rightarrow T2 \ldots$$
or
$$ACX \rightarrow T1,\ ACX \rightarrow T2,\ ACX \rightarrow T1 \ldots$$

4.4 Experimental Framework

The SPECint2000 benchmark suite and six virtual machine interpreter benchmarks [EG01] are used to evaluate different trace cache configurations/strategies and their influences on the indirect branch target prediction. Reduced data sets [KL02] are used as inputs for SPECint2000 benchmarks that run to completion. **Table 4.1** shows the major characteristics of benchmarks. *Num Instr.* and *% Indirect Branches* represent the total number of dynamic instructions and the percentage of dynamic indirect branch instructions (excluding return instructions) in the total number of dynamic instructions. Note that in most of these programs indirect branches account for a small proportion of total instructions.

We implemented our trace cache model in *sim-bpred* in the *SimpleScalar* 3.0d [ALE02a] simulator. The baseline model is the microarchitecture model that has a 2-level conditional branch predictor and branch target buffer (BTB) for predicting indirect branches but has no trace cache. **Table 4.2** shows the configuration of the baseline model.

The indirect branch prediction accuracy of the BTB in the baseline model is compared with the base trace cache model whose simulator parameters are shown in **Table 4.3**. The base trace cache model uses the conditional branch prediction, BTB and RAS parameters of the baseline model as given in **Table 4.2**. However, unlike the baseline model, indirect branch predictions are more complicated. Indirect branch predictions are made using the trace cache lines, provided there is a trace cache hit. In the case of a trace cache miss, the BTB is instead used to make the indirect branch prediction. Thus, the overall indirect branch prediction rate is a combination of the rate achieved using the trace cache, and the rate for the BTB where trace cache missed occur.

An important question when measuring the performance of computer systems is summarizing data in an appropriate way. The most common way to average results

Table 4.1: Benchmark statistics

SPECint2000	**Num Instr.**	**% Indirect Branches**
twolf	972,726,535	0.02%
vortex	1,153,664,377	0.03%
gap	761,346,123	0.77%
bzip2	1,819,780,259	0.000007%
vpr	1,566,703,859	0.00023%
gzip	1,361,319,057	0.00002%
perlbmk	2,061,197,349	1.47%
eon	1,070,281,136	0.61%
parser	4,527,012,522	0.0001%
crafty	834,909,899	0.25%
mcf	793,869,356	0.01%
gcc	5,117,054,875	0.34%

VM Interpreter	**Input Set**	**Num Instr.**	**% Indirect Branches**
gforth	benchgc	64,395,745	13.01%
li	boyer	183,304,695	1.13%
ocamlc	ocamllex	69,738,732	11.35%
ocamlc-switch	ocamllex	122,559,342	6.46%
perl	jumble	40,496,306	0.71%
scheme48	build	113,143,169	3.29%

Table 4.2: Baseline model

No Trace cache
2-level conditional branch predictor Global History Size = 8
2-level conditional branch predictor Pattern History Table Size = 1024-entry
BTB size = 512×4
Return Address Stack (RAS) Size = 8-entry

Table 4.3: Base trace cache model

1024-entry directed-mapped Trace cache
Trace Cache Write Policy: *keep-longest*
Trace lines of **16 instructions** with **at most 3 branches**
No updating of indirect branch targets in the trace cache

from several different benchmarks is to simply use the arithmetic mean. However, in this chapter we primarily measure prediction rates, and a more statistically meaningful average for rates is the harmonic mean [Lil00]. The harmonic mean H of the positive real numbers $a_1, ..., a_n$ is defined to be:

$$H = \frac{n}{\frac{1}{a_1} + \frac{1}{a_2} + ... + \frac{1}{a_n}} \tag{4.1}$$

One downside of the harmonic mean is that very small values tend to dominate the result. Furthermore, as most hardware designers are more familiar with the arithmetic mean, and that there are often significant differences between the two measures, we show both means in all charts. However harmonic mean is the more meaningful measure, and that is the one used in the text. For convenience, corresponding arithmetic mean values are shown in brackets after.

4.5 Initial Prediction Accuracies

4.5.1 BTB versus Trace Cache with Non-update Policy

Figure 4.6 shows the indirect branch target address prediction accuracies for a model with a BTB (i.e. *BTB*) and the trace cache model with the non-update policy (i.e. *TC- No Update*). The last two columns in the figure shows the arithmetic and harmonic means of prediction accuracies across all benchmarks. The results for *bzip2*, *gzip* and *vpr* are not shown in the following figures because the indirect branch prediction accuracy does not change at all due to the extremely small number of indirect branch instructions.

In *BTB*, the BTB stores only one branch target for each indirect branch instruction

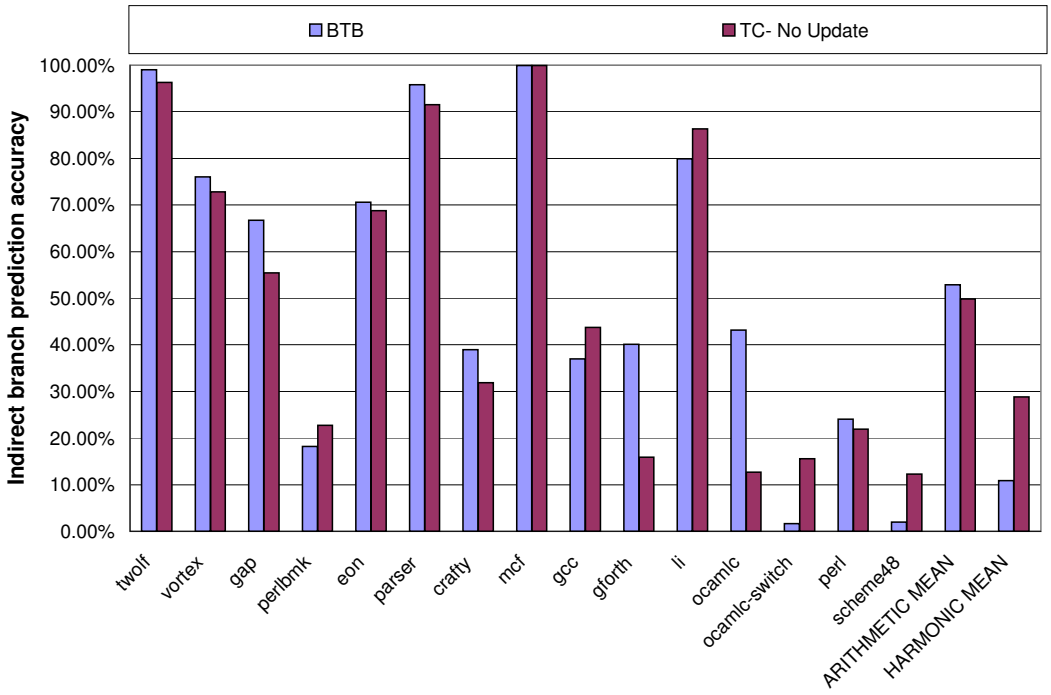

Figure 4.6: Indirect branch target prediction accuracies for the BTB and trace cache with non-update policy

and updates the branch target when it changes. *BTB* outperforms *TC- No Update* in these benchmarks: *twolf*, *vortex*, *gap*, *eon*, *parser*, *crafty*, *gforth*, *ocamlc* and *perl*. Not updating the target of the indirect branch when it mispredicts has a very negative impact on these benchmarks. This limits the trace cache's ability to adapt to changing program behavior.

On the other hand, the prediction accuracies in *TC- No Update* is higher than *BTB* for *perlmbk*, *gcc*, *li*, *ocamlc-switch* and *scheme48*. In particular for *ocamlc-switch* and *scheme48*, the advantage of some context information outweighs the significant disadvantage of not updating indirect branch targets when the behavior of the program changes. These virtual machine interpreters consist of code very similar to that in our example in **Figure 4.2**. The single indirect branch has a large number of targets, but there is a correlation between the previous outcome of this branch and the current one. Thus, prediction accuracy improves considerably.

The behavior of *perlmbk* and *li* is somewhat different. In both benchmarks, there is actually a benefit from not updating the branch target when it is incorrect. We investigated *li* and found out that there are indirect branches with a number of different targets, but where one target is more frequent than the others. If this most frequent outcome can be found, the best strategy is to stick with it, rather than update on every misprediction. Our experiments indicate that this target is finding its way into the trace cache, and the strategy of not updating is effective.

Overall, the harmonic mean (arithmetic mean in brackets) of indirect branch prediction accuracies across all benchmarks are 11% (52.88%) for *BTB* and 29% (49.85%) for *TC- No Update*. Note that the harmonic mean accuracy for the BTB is heavily dominated by the very low accuracies for *ocamlc-switch* and *scheme48*. In many cases the BTB is actually more accurate than using the trace cache. This shows that the model using the trace cache *can* predict indirect branches more accurately than the BTB in some, but not all, cases.

4.5.2 Trace Cache with Update Policy

In the base trace cache model in **Table 4.3**, we use the *keep-longest* policy for writing to the trace cache since it is shown to be best policy by *Patel* [Pat99] for maximizing fetch bandwidth. However, if the *always overwrite* policy, which always overwrites the

trace cache lines, was used, then updating the indirect branch target addresses would be handled automatically. However, *Patel* showed that the *always overwrite* policy is very ineffective and wasteful in terms of cycle times spent for overwriting.

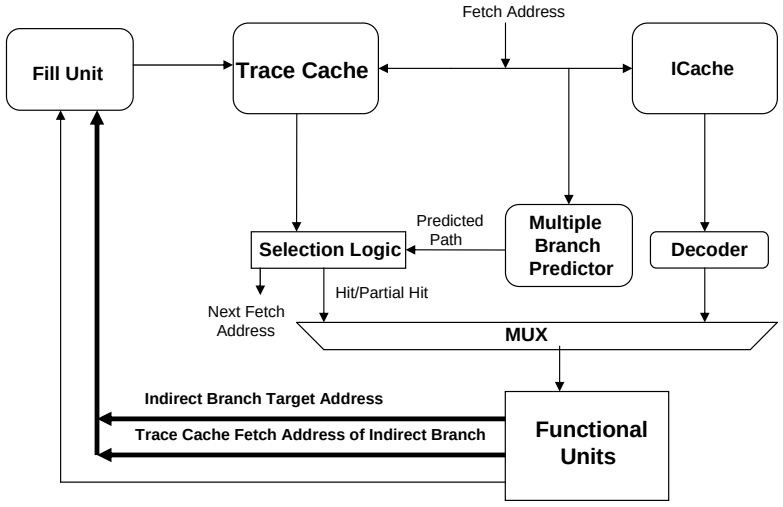

Figure 4.7: Trace cache model with the update policy

Hence, we slightly modify the trace cache microarchitecture by adding the capability of updating indirect branch targets in the trace cache lines as shown in **Figure 4.7**. When the indirect branch instruction is executed and retired, its computed target address (i.e. Indirect Branch Target Address) and the fetch address of the trace cache line to which the indirect branch instruction belongs arrive at the fill unit. The fetch address can be sent to the pipeline as a part of the indirect branch instruction or can be acquired from the reorder buffer since updating the indirect branch target addresses in the trace cache occurs at retire time. If the trace cache line is still in the cache, then the fill unit overwrites the indirect branch target address at the end of the trace cache line. No target updating is performed if the trace cache line has been thrashed from the cache.

Figure 4.8 shows the trace cache with the update policy along with the *BTB* and *TC- No Update* models. Updating indirect branch targets in the trace cache performs better than not updating them for *twolf*, *vortex*, *gap*, *eon*, *parser*, *crafty*, *gforth*, *ocamlc* and *perl*. This is expected because for these benchmarks, the *BTB* model also performs better than the *TC- No Update*.

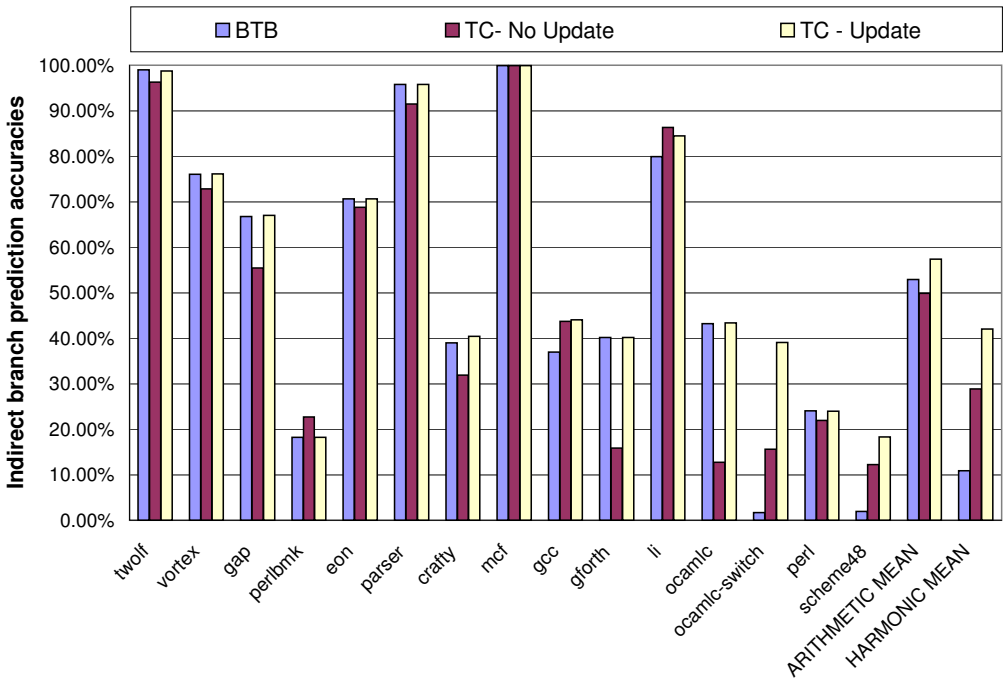

Figure 4.8: Indirect branch target prediction accuracies of the BTB and the trace cache models with the non-update and update policies

In contrast, *TC- Update* is worse than *TC- No Update* in *perlmbk* and *li*. The limited amount of context information provided by the trace cache gives little benefit to these programs, and recall from *Section 4.5.1* that these benchmarks benefit from not updating the branch targets.

On the other hand, using *TC- Update* in *gcc*, *ocamlc-switch* and *scheme48* improves the prediction accuracies. This is expected because indirect branch target addresses in these benchmarks change relatively frequently as the program runs, and the updating predictor can take account of these changes. Furthermore, all these programs benefit significantly from the context information provided by the trace cache. Thus, the chance of getting a target address hit for the same indirect branch in the *TC- Update* model is much higher than that of the *TC- No Update* model. As a consequence, *TC- Update* outperforms both *BTB* and *TC- No Update* in *gcc*, *ocamlc-switch* and *scheme48*.

Overall, the average indirect branch prediction accuracy of *TC-Update* across all benchmarks now becomes 42% (57.35%), which is 13% (7.5%)-points better than the *TC- No Update*. In fact, it is even higher than *BTB* by about 31% (5%)-points on the average. Thus, in the following sections, all the benchmarks will update indirect branch targets.

4.6 Prediction Accuracies of Various Trace Cache Configurations

The goal of the trace cache is to provide sufficient bandwidth to the execution pipeline within the constraints of not taking up too many resources, or causing the design to become so complex that it impacts on clock speed. Several enhancements to the trace cache have been proposed to improve its fetch bandwidth, many of which increase its size and complexity. Many of these enhancements affect the storage of instructions in the trace cache, and thus have the potential to have a knock-on effect on indirect branch prediction. The main purpose of these optimizations is, however, to improve fetch bandwidth, not indirect branch prediction.

In this section, the configuration of the trace cache model with indirect branch target updating is varied incrementally in order to show the effects of each configuration

on the indirect branch prediction accuracy. The variations in the configuration consist of applying trace packing, adding 2-bit saturating update counters per trace cache line, varying trace cache set associativity, cache size, cache line size, and finally the combination of all.

4.6.1 Trace Packing

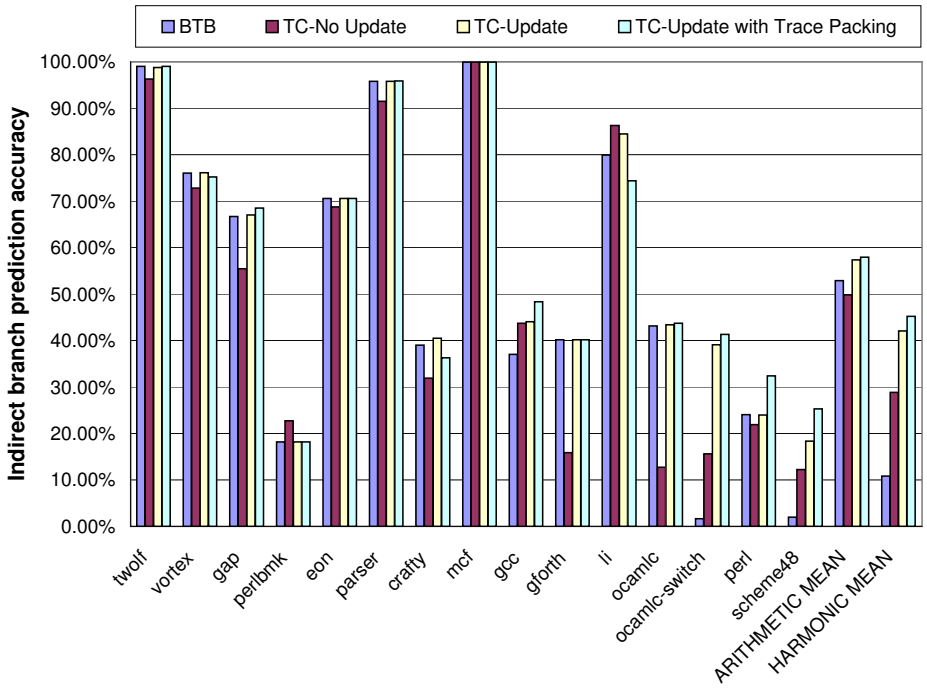

Figure 4.9: Indirect branch target prediction accuracies of the BTB and the *TC-Update* model with trace packing

Trace packing [Pat99] puts as many instructions as possible into a trace cache line by fragmenting fetch blocks. The fragmentation offers a way to form different trace cache lines, particularly in a loop. Thus, it enables the fill unit to increase the potential of creating more trace cache lines ending with a particular indirect branch instruction. However, it increases the chance of contentions in the trace cache since a large number of trace cache lines can be created.

Figure 4.9 shows the indirect branch prediction accuracies of the previous models along with *TC-Update with Trace Packing*. Most of the benchmarks enjoy improvement

in the prediction accuracies as compared to the *TC-Update* model. This is the result of indirect branches being split into more separate instances, each with more context information. The improvement is particularly high in *perl* by 8 percentage points and *scheme48* by about 7 percentage points. On the other hand, the prediction accuracies of *vortex*, *crafty* and *li* become worse than the *TC-Update* and even *BTB* models. We believe this is mostly due to trace cache capacity misses caused by the increased number of trace cache lines created by trace packing. However, we found the phenomenon difficult to study, because the breaking of trace cache lines can vary considerably during the running of the program. Overall, the average indirect branch prediction accuracy of the *TC-Update* model with trace packing across all benchmarks is now 45% (57.95%) which is 34.38% (8%), 34.38% (5%) and 3.18% (0.6%) points better than the *TC-No Update*, *BTB* and *TC-Update* models, respectively.

4.6.2 2-bit Saturating Update Counter

We also propose to extend trace cache lines with 2-bit saturating counters along with indirect branch target addresses in order to increase indirect branch prediction. When a new trace cache line ending with an indirect branch is created, it is placed into the trace cache and the 2-bit update counter at the end of the line is set to 1. The update counter is incremented if the indirect branch target is predicted correctly. Otherwise, it is decremented. When the same trace cache line is accessed again, the indirect branch target is not updated until the update counter in the trace cache line reaches zero. This technique is widely used both in conditional branch predictors and in BTBs to improve prediction accuracy. In BTBs it is particularly effective for monotonic indirect branches, that is branches that almost always jump to the same target, but occasionally jump to another target.

This method of updating indirect branch targets prevents a frequent target from being removed from the trace cache line unless it is shown to be incorrect more than once. This enables benchmarks whose indirect branch targets are biased to one or two specific targets such as *perlmbk*, *li*, *gcc*, *ocamlc-switch* and *scheme48* to keep these predicted targets in the face of mispredictions. On the other hand, it may reduce the responsiveness of the indirect branch predictor to changes in behavior, as in *vortex*. **Figure 4.10** shows that the prediction accuracies of all benchmarks improve by us-

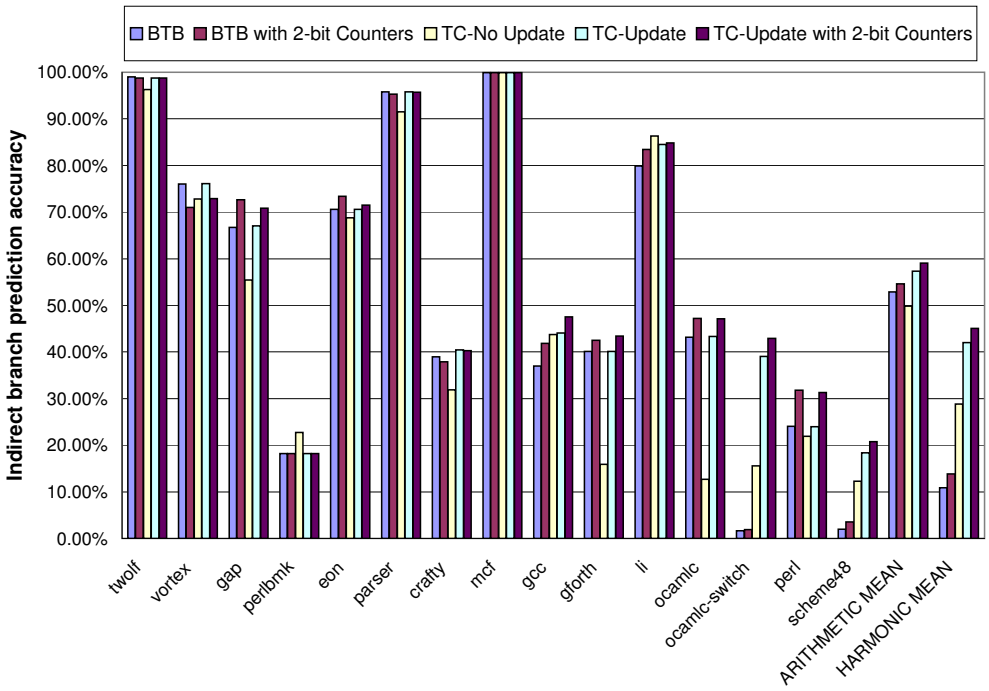

Figure 4.10: Indirect branch target prediction accuracies of the BTB, *TC-No Update*, *TC-Update* and *TC-Update* with 2-bit update counters

ing 2-bit saturating update counters with respect to the *TC-Update* model. Overall, the average indirect branch prediction accuracy of the *TC-Update with 2-bit Counters* across all benchmarks is now 45%(59.06%) which is 16%(9%), 34%(6%), 31%(2.72%) and 3%(2%) points better than the *TC-No Update*, *BTB*, *BTB with 2-bit Counters* and *TC-Update* models, respectively.

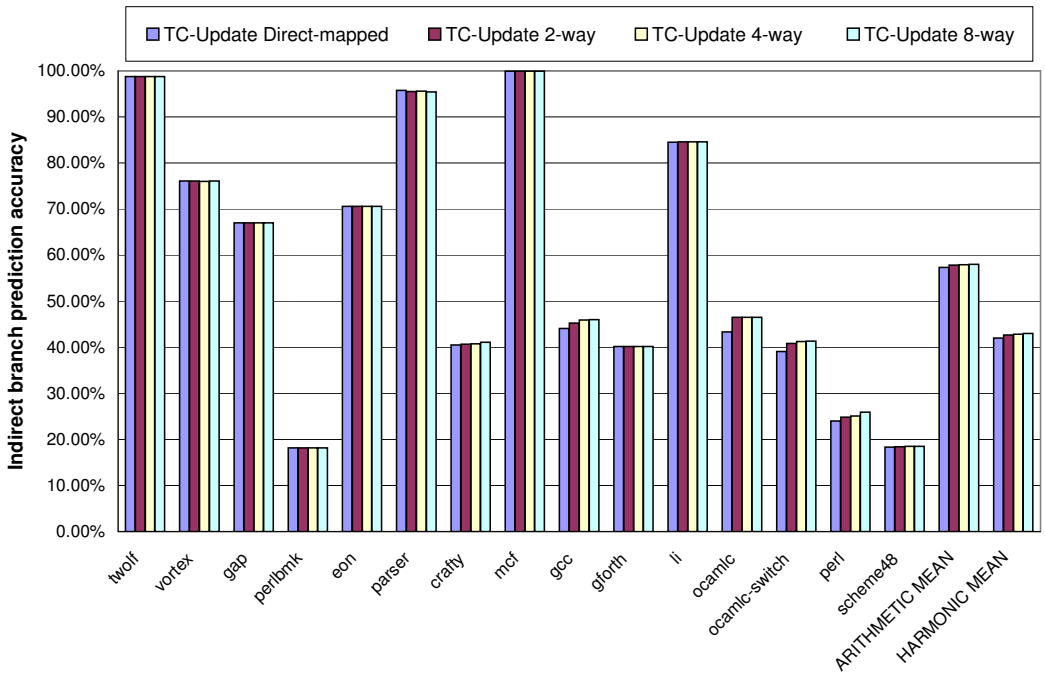

Figure 4.11: Indirect branch target prediction accuracies of the *TC-Update* model with varying set associativity

4.6.3 Trace Cache Associativity

We also vary trace cache set associativity to measure the effects of a set associative trace cache on the indirect branch prediction. In a set-associative trace cache, multiple cache lines starting at the same address can co-exist in the trace cache. Using Patel's scheme [Pat99], different trace cache lines starting with the same address are mapped to the same set. The starting address is used to identify the set, and the pattern of m conditional branches is used to identify the line within the set. This allows the trace cache with path associativity to hold multiple cache lines having the same starting

address and ending with the same indirect branch instruction because we can index a trace cache line into a set using its starting address and use its branch history to identify whether it is unique. Thus, each will be more specialized versions of the indirect branch, with more context information embedded in the trace cache line.

Figure 4.11 shows the indirect branch prediction accuracy for 2, 4 and 8-way set associative trace caches with updating. Almost all of the benchmarks show slight improvements in the prediction accuracy as associativity increases. Overall, the average indirect branch prediction accuracies of the *TC-Update* model with 2,4 and 8-way set-associative trace caches across all benchmarks are now 42.63%(57.82%), 42.80%(57.93%) and 42.98(58%), respectively.

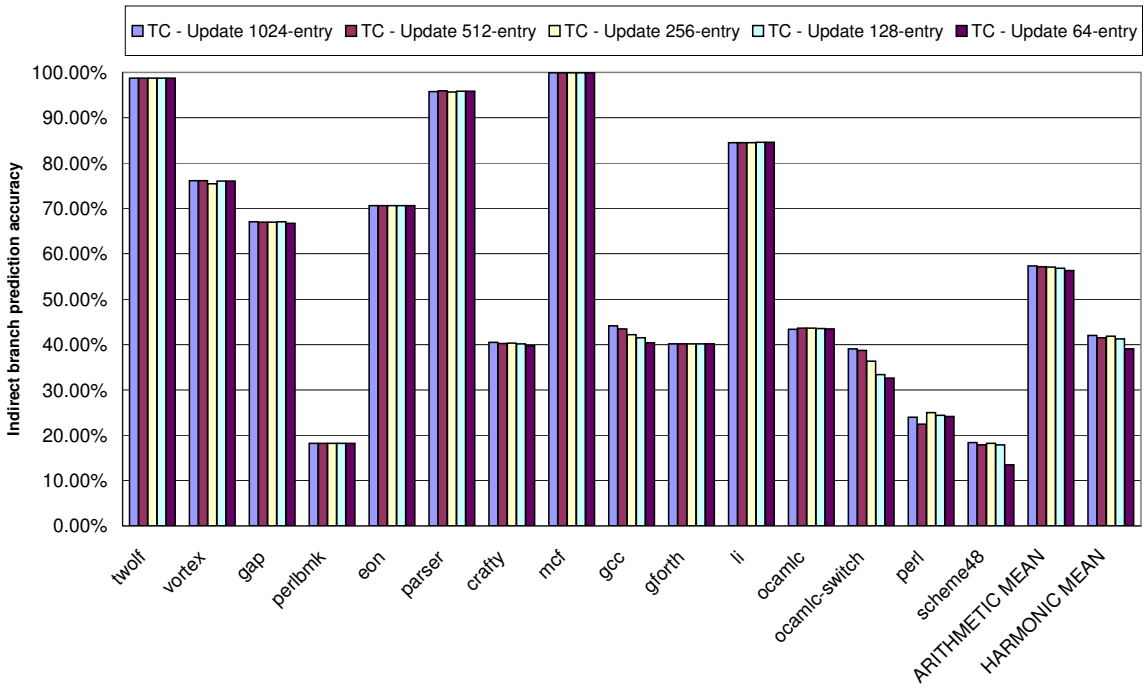

Figure 4.12: Indirect branch target prediction accuracies of the *TC-Update* model with 64, 128, 256, 512 and 1024-entry trace caches

One complication with interpreting the effects of increased associativity on indirect branch prediction is that there are two effects at work. First, associativity may allow more instances of the same branch with separate targets, as described above. Secondly, associativity simply increases the hit rate of the trace cache (see **Figure 4.17**) which

is also likely to increase indirect branch prediction accuracy. Given the very small increase in accuracy, it is difficult to separate the two effects.

4.6.4 Trace Cache Size Variance

The cache size of 1024-entry direct-mapped trace cache has been used in the prior runs. We gradually reduce its size from 1024 entries to 512, 256, 128 and 64 entries to capture the sensitivity of the indirect branch prediction to the reduction in the cache size. Clearly, the main goal of varying the trace cache size is to find the optimal balance of hit rate and hardware resources. However, varying the size also has a small impact on indirect branch prediction because it affects the trace cache hit rate. The higher the hit rate, the more indirect branches are likely to be predicted by the trace cache (and thus have the benefits of limited path context information) rather than the BTB.

Figure 4.12 shows the prediction accuracy results of cache size variance. The overall prediction accuracy percentage point difference between 1024-entry and 64-entry trace caches is slightly greater than 3% (1%). Thus, with a smaller cache size such as 256 entries, it is possible to attain nearly the same indirect branch prediction accuracy across all benchmarks as a 1024-entry direct-mapped trace cache.

4.6.5 Trace Cache Line Size Variance

So far, we have used a fixed trace cache line size of 16 instructions with 3 branches. The cache line size configuration is also varied to measure its effect on the prediction of indirect branches. We use three other cache line configurations: 20 instructions with 4 branches, 24 instructions with 5 branches and 32 instructions with 6 branches. Clearly, the longer the trace cache line, the more context information on recent conditional branch outcomes is stored into it. However, longer lines also lead to more conflict misses, assuming a fixed total size for the trace cache.

Figure 4.13 shows the comparison of the trace cache models with four different cache line sizes. Out of these four cache line configurations, the 24-instruction-5-branch model performs the best for indirect branch prediction. It is important to note, however that this configuration does not give the highest trace cache hit rates. On the contrary, **Figure 4.17** shows the hit rates for various trace cache configurations and we see that the configuration with five branches per trace cache line has the worst hit rate of all

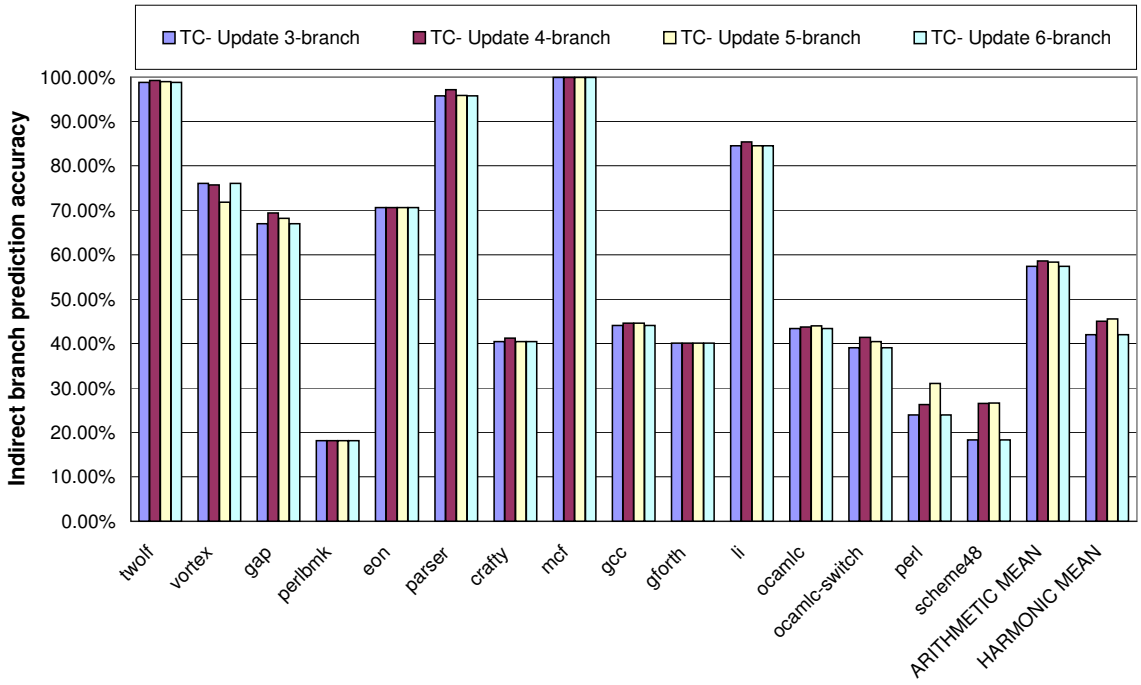

Figure 4.13: Indirect branch target prediction accuracies of the *TC-Update* model with 3-branch, 4-branch, 5-branch and 6-branch cache line sizes

the variants we examined. Thus, while such a trace cache is interesting because it gives us some indication of the limits on improvements in indirect branch prediction using our scheme, it is highly unlikely to be implemented in practice. The main goal of a trace cache is fetch bandwidth, and so it will be designed to maximize this bandwidth rather than to improve branch prediction.

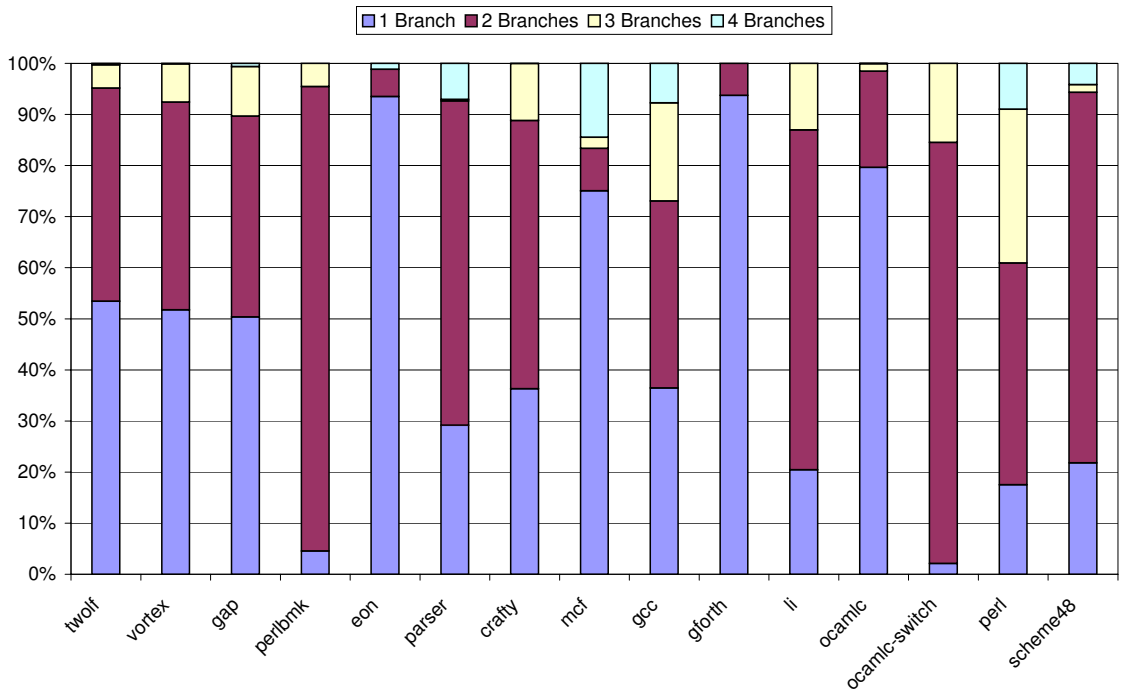

Figure 4.14: Breakdown of all used trace cache lines ending with an indirect branch based on the number of branches in a trace cache line

Improvements in branch prediction accuracy using the trace cache come from the inclusion of a limited amount of context information in the prediction. In **Figure 4.14** we show the amount of context used in trace cache predictions in a trace cache with up to four branches per line. We show the number of branches in the trace cache line for each indirect branch prediction made using the trace cache. One branch in the line means that the indirect branch is the only branch in the line. Greater numbers of branches show that more path context is being used. The figure indicates that on average around 90% of predictions are made with one or two branches per line.

Figure 4.15 show the accuracy of indirect branch predictions made with varying

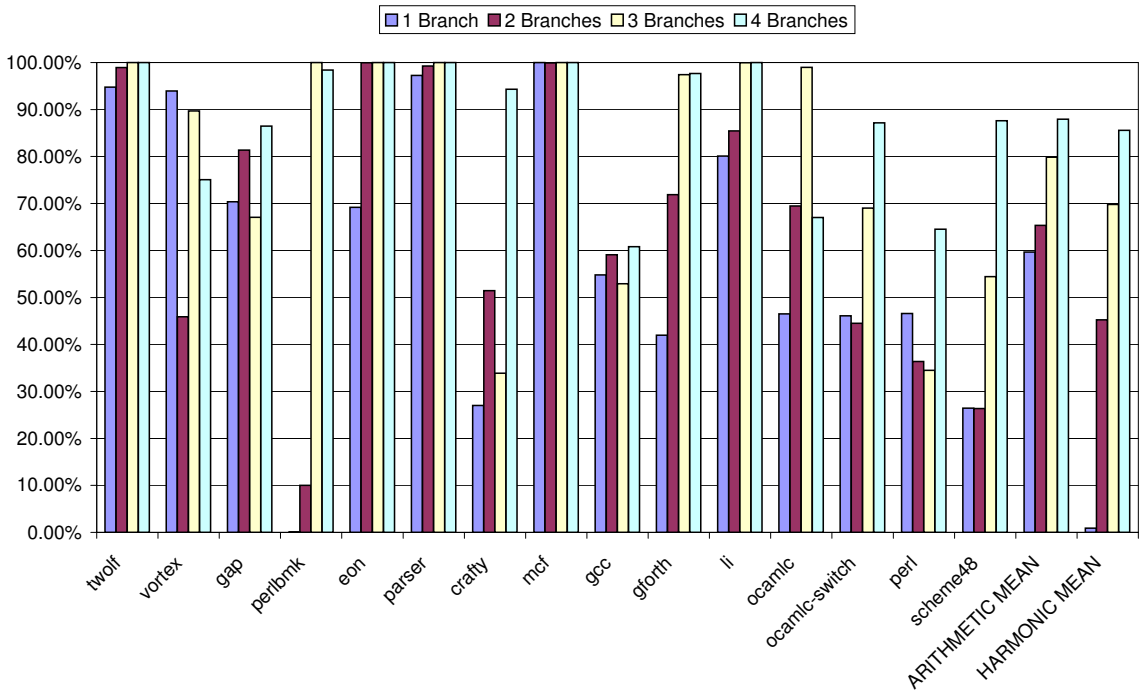

Figure 4.15: Breakdown of the prediction rates for all used trace cache lines ending with an indirect branch based on the number of branches in a trace cache line

number of branches per line. There is a clear trend showing a strong correlation between the amount of path context information captured by branches and the accuracy of the prediction. This is consistent with results in two-level branch predictors which use much greater amounts of context than we are able to capture [CI01][DH98][KK98].

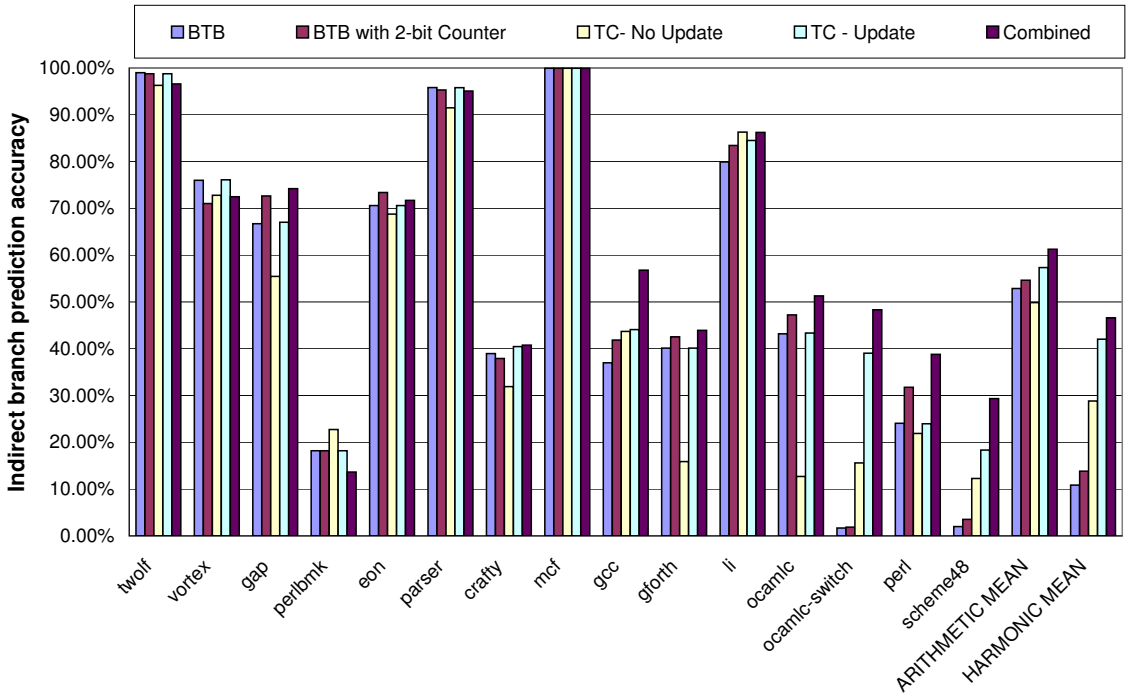

Figure 4.16: Indirect branch target prediction accuracy of the combined model of 1024-entry, 8-way, 20-instruction-4-branch line size, updating, 2-bit saturating counters and trace packing in comparison to the *BTB*, *TC-No Update* and *TC-Update* models

4.6.6 Combining Various Configurations

In this section, we present the results of a trace cache model with various trace configurations together. The combined configuration model includes branch target updating policy, 2-bit saturating update counter per trace line and trace packing. Also, we select the best performing parameter from cache size, cache associativity and cache line size. These are 1024-entry, 8-way set associativity and 20 instructions with 4 branches. This model gives the best overall prediction accuracy of the models we investigated. Thus,

it gives us some idea of the limit on improvement in branch prediction accuracy that is possible using these techniques.

Figure 4.16 shows the prediction accuracy results of the combined model in comparison with the *BTB*, *BTB with 2-bit Counters*, *TC-No Update* and *TC-Update* models. Overall, the average indirect branch prediction accuracy of the *Combined* model across all benchmarks is now 46.60% (61.27%), which is about 35.75% (8.40%), 32.76% (6.65%), 17.77% (11.43%), and 4.56% (3.92%) points better than the *BTB*, *BTB with 2-bit Counters*, *TC-No Update*, and *TC-Update* models, respectively.

Figure 4.17 shows the trace cache hit rates for several different combinations that we have tested. Similarly, **Figure 4.18** shows the percentage of instructions executed that come from the trace cache. In all cases, the size of the trace cache is 1024 entries. The figure clearly shows that the best approach for optimizing the hit rate that we have examined is to use the base trace cache (16 instructions, 3 branches per line) in an 8-way associative configuration. This configuration is not the one that maximizes prediction accuracy, but it is the one that is most likely to be used, as the main objective of a trace cache is to improve fetch bandwidth.

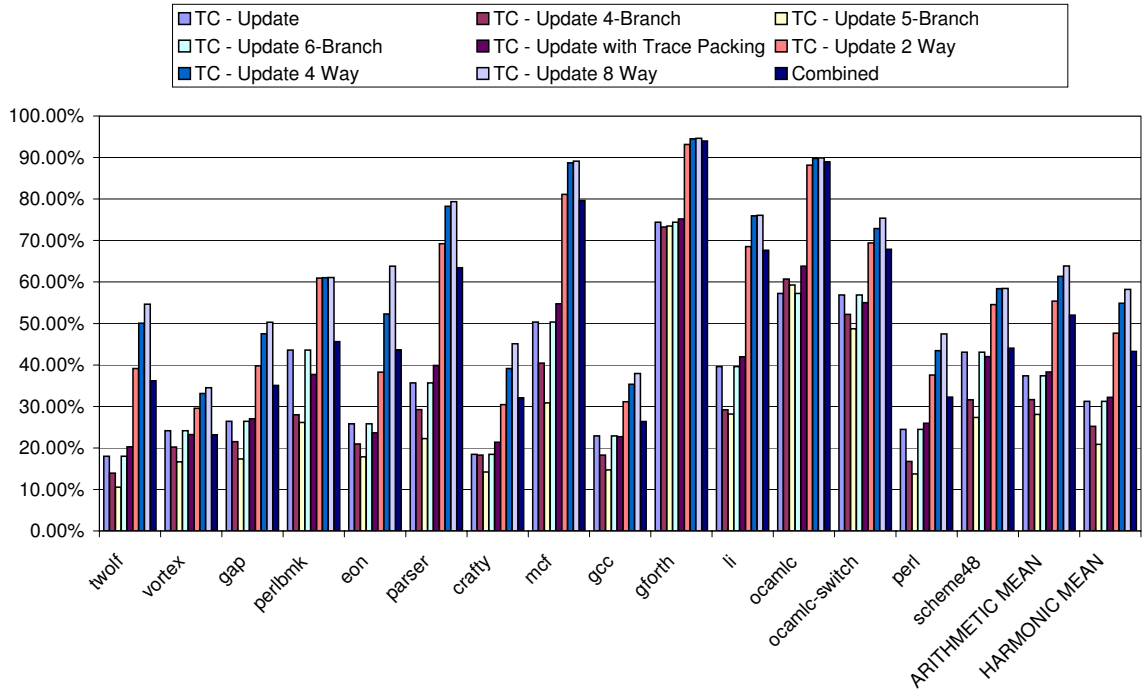

Figure 4.17: Trace cache hit rates for different configurations.

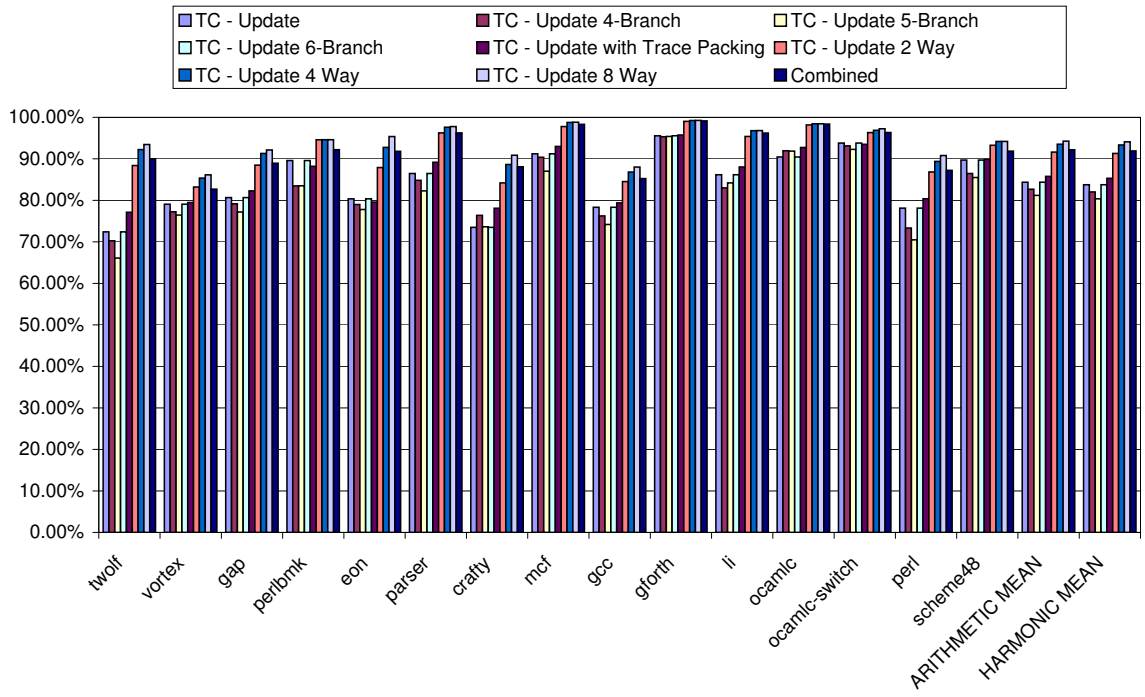

Figure 4.18: The percentage of executed instructions from the trace cache

4.7 Other Trace Cache Models

4.7.1 Real World Trace Cache

The implementation of the trace cache described in this chapter is based on Patel's *PhD dissertation* [Pat99]. However, other configurations are possible. For example, the Intel Pentium 4 processor [BBH+04] uses a trace cache and two separate BTBs. One of the two BTBs works as normal, but the other is dedicated to branches stored in the trace cache, and appears to be addressed by the location of the branch in the trace cache rather than the branch's address in main memory.

In such a model, if we assume that there is a dedicated BTB entry for each branch in the trace cache, the effect on indirect branch prediction will be the same as our TC-Update model (see section 4.5.2). Initial experiments using hardware performance counters on the Pentium 4 suggest that the indirect branch prediction accuracies on small test programs with particular indirect branch behaviour are much higher than

one would expect with a simple BTB, and that the context stored by the trace cache is reducing the number of indirect branch mispredictions.

To demonstrate this effect, we ran a variant of the program in **Figure 4.2** on a lightly loaded Pentium 4 machine. In the switch statement in the inner loop we used eight different cases rather than four, so that the compiler implemented the statement with an indirect branch, rather than a tree of conditional branches. We used an array containing values (0,1,2,3,4,5,6,7), so that the target of the next branch can be predicted perfectly, provided we know the outcome of the previous one. We also increased the number of iterations of the loop to ten million in order to make the effect more visible. We used the *perfex* utility to access hardware performance counter measures of the process.

Table 4.4 shows the range of results from running this program with the same inputs 10,000 times. The runs are divided into rows, grouped according to the number of indirect branch mispredictions that occurred in that run. In 78.54% of cases, there are fewer than 260 indirect branch mispredictions. Among more than ten million indirect branches, this is a misprediction rate of almost 0%. This is exactly the result that we would expect. The trace cache captures sufficient context information to identify the most recent outcome of the indirect branch. In this particular program, this is enough information to perfectly predict the next outcome, as we explained in section 4.3. Thus, the Pentium 4 behaves exactly as our model predicts in almost 80% of the cases of running this program.

Range of mispredictions	Percentage of total	Average % misprediction	Average cycles
0 – 260	78.54%	0.0017%	64,432,642
261 – 250,000	0.58%	1.14%	66,442,092
250,000 – 500,000	8.88%	4.05%	75,394,126
500,000 – 750,000	7.71%	6.04%	80,425,580
750,000 – 1,000,000	3.24%	8.48%	86,027,876
>1,000,000	1.06%	12.10%	94,912,349
Total	100%	1.24%	67,673,178

Table 4.4: Pentium 4 indirect branch prediction results on simple benchmark

In the remaining 21.46% of cases there are between 3,993 and 3,090,300 mispredic-

tions, representing a misprediction rate of 0.04% to 30.1%. We see that most of these cases involve 250,000 to 750,000 branch mispredictions. We suspect that the main reason why the indirect branch is less predictable on some runs is that a trace cache line in the Pentium 4 contains only six microinstructions (pre-decoded RISC translations of x86 instructions). To capture useful context, a trace cache line needs to contain both a control-flow and an indirect branch. We suspect that in some cases random effects are causing a poor choice of starting point for some of the trace cache lines, which result in less context being captured by the trace cache. This is especially likely with short trace cache lines, because the amount of context captured is already likely to be small. However, even where a relatively large number of mispredictions occur, the misprediction rate is very much lower than for a simple BTB which would mispredict almost 100% of the time on this program. **Table 4.4** also shows the average running time (in cycles) for each group in the final column, which increase sharply in line with the increase in indirect branch mispredictions.

It is important to recall, however, that Intel releases relatively little information about the microarchitecture of their processors. We know that the Pentium 4 has a separate BTB for branches in the trace cache [BBH+04]. We also know that the Pentium 4 does not have a two-level branch predictor [GRA+03]. However, it is not possible to state categorically that the processor is working as exactly as we describe, although it certainly appears to work in this way. For this reason, we do not examine more complicated programs running on the Pentium 4. There are simply too many unknown variables.

4.7.2 Trace Cache Context Study

In order to demonstrate that the trace cache context can help the branch prediction, we run the benchmarks with a PC-indexed bimodal branch predictor with no global history. If the trace cache is used, a trace cache hit overrides the branch predictor (multiple branch predictor is not used and the single branch predictor is only used for I-cache fetch path). A trace cache hit predicts branches implicitly for each embedded branch. The trace cache is configured as the direct-mapped one with 1024 entries, which means only one trace cache line for the same starting fetch address based on our trace cache implementation. We use two schemes here:

Scheme 1: we embed 2-bit saturating counters in the trace cache lines in the same way as used in the bimodal branch predictor and replace a trace cache line only if it contains one or more weak counters or the new trace cache line is longer the existing one.

Scheme 2: we don't embed 2-bit saturating counters in the trace cache line and a trace cache line is updated using the same updating policy (ie. keep-longest policy) in the earlier experimentation.

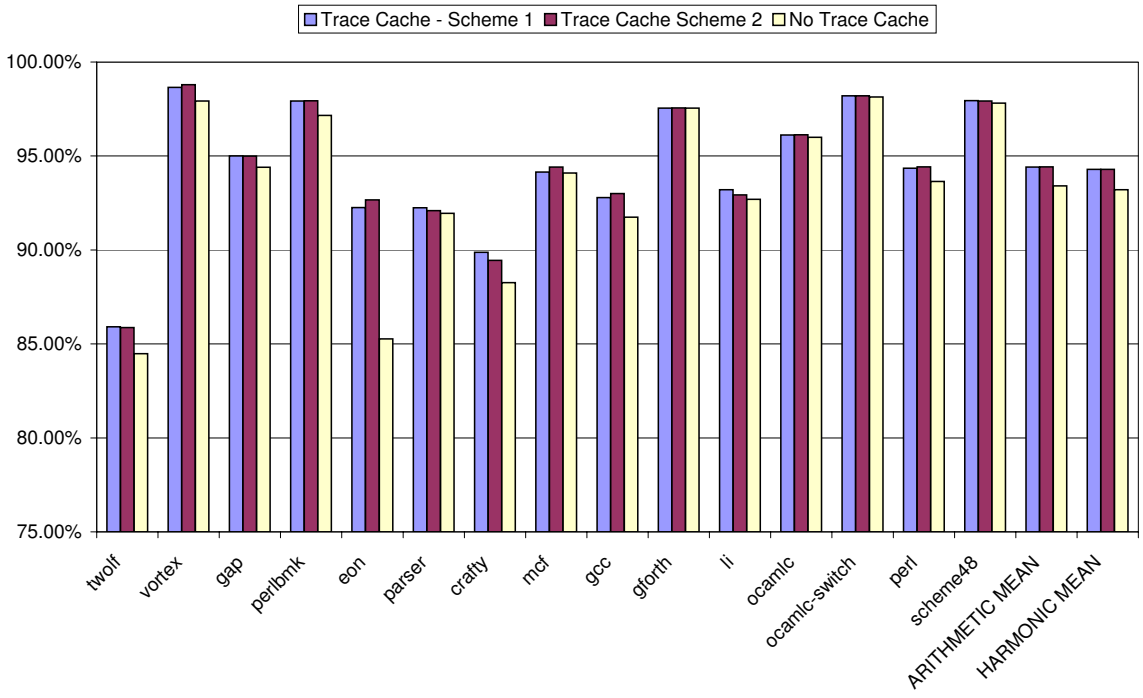

Figure 4.19: Bimodal direction predication rates with/without a trace cache (1024 direct-mapped with 3 branches)

Figure 4.19 shows that the direction prediction rates with a trace cache (scheme 1 and 2) do indeed show slight improvement (around 1%) over those without a trace cache, probably because of limited context information. There aren't two much difference between the two schemes with trace cache.

4.7.3 Other Predictors

The other main branch prediction model for the trace cache is to use a next-trace/next-stream predictor [JRS97] [SFR+02], rather than predict individual branches. These give much better prediction accuracies for indirect branches than any predictor described in this chapter, at the cost of a significant increase in complexity. In comparison, the effect we measure gives a much smaller improvement in indirect branch prediction accuracy, although at very low cost beyond that required for the trace cache. However, it is important to note that the main contribution of this chapter is not to advocate using the trace cache to get better indirect branch prediction, rather than some other approach. Our main contribution is to explore the observation that the trace cache captures context information about the control-flow of the program, and this can have a significant impact on other predictors that do not use such context.

An interesting piece of future work would be to measure the effect of our scheme on the overall performance of the processor in instructions per cycle (IPC). We believe that the impact on IPC is likely to be small because (1) indirect branches make up a relatively small number of overall executed instructions and (2) the improvement in prediction accuracy is modest. However, if the trace cache is used for prediction, it raises the possibility of trace cache misses and branch mispredictions tending to occur simultaneously. The overall effect on IPC is not clear.

Although this chapter has examined the effect of the trace cache on indirect branch prediction, there may be a similar impact on other forms of prediction. A wide variety of predictors have been proposed in the literature, including load address predictors, load value predictors and data dependence predictors. In each of these predictors there is typically a table of recent values, which is indexed by some function of the address of a particular instruction in main memory.

In a processor with a trace cache the option arises of accessing these tables with the location in the trace cache rather than the address in memory. For example, it may be more efficient to use the location in the trace cache, as this would require no further work to map it back to the original location in memory. (Mapping back to this original address would be necessary to maintain the classical behaviour of such a predictor). It seems likely that this is the reason that Intel chose to have a separate BTB for branches in the trace cache in their Pentium 4 architecture.

However, as we have shown in this chapter, if there are separate entries in a predictor for each copy of an instruction in the trace cache, there can be a significant impact on predictor accuracy. In the case of indirect branches, the impact is mostly positive, because the trace cache captures context about the recent control flow of the program. In fact, this may even have been the motivation for the separate BTBs in the Pentium 4 architecture. Furthermore, choosing to index a table by trace cache location rather than location in memory creates a further dependence: the prediction accuracy or the use of any such predictor will depend on the trace cache hit rate. Since the trace cache will be sized and organized for fetch bandwidth, the overall effect on prediction accuracy may be worse than could be achieved with other design goals. Being aware of and measuring such effects is important for any processor designer considering design alternatives surrounding the trace cache.

4.8 Related Work

In a broader discussion of trace cache design, Rotenberg [Rot05] notes that the sequence of branch outcomes stored in a trace cache line implicitly captures some execution path context. Thus, we are not the first to remark on this effect. However, there is no in-depth investigation of this phenomenon.

Peleg and Weiser [PW95] embed one-bit or two-bit counters inside traces, so that the trace cache does branch prediction merely by the act of supplying a trace. Accuracy may be slightly better than a pure bimodal predictor, because the same branch may be in multiple traces. Thus, the counters also become replicated and some measure of the context-sensitivity of a two-level predictor may be captured by their scheme.

Branch promotion [Pat99] move the predictions of some very predictable branches to the trace cache. This has the effect of reducing pressure in the external predictor. Another effect is that branches are replicated in the trace cache because the same branch can appear in multiple lines, potentially increasing accuracy because each specialized version is biased independently.

Lee et al [LWY00] investigated value prediction using a trace processor where value prediction is decoupled from the instruction fetch stage. Part of their scheme is to use a trace cache in which the instructions that will use value prediction are pre-identified. In such a scheme it is possible for multiple instances of the same instruction to appear

in different trace cache lines, with implications for prediction accuracy.

4.9 Conclusion

In this chapter, we have discussed the effects of using the trace cache on the indirect branch prediction in ILP processors. If the target addresses in the trace cache lines are used to predict indirect branches, the accuracy can be significantly different from that that achieved with a simple branch target buffer. The main reason for this is that the trace cache captures context about the recent control flow of the program. To our knowledge, this effect was not foreseen in the original design of the trace cache, and has remained underreported since then.

With a simple mechanism of updating indirect branch target addresses in the trace cache lines, we have shown that indirect branch prediction accuracy can be moderately improved with very little or no cost at all. We have analyzed various trace cache configurations and strategies such as applying trace packing, adding 2-bit update counters per trace cache line, varying trace cache set associativity, cache size and cache line size and tune them to measure the positive/negative effects of each configuration/strategy on indirect branch prediction accuracy. Finally, we have constructed a model combined with the best performer from each configuration/strategy.

Our experimental results have shown that the harmonic mean indirect branch prediction accuracy, across several benchmarks, using a trace cache model with trace cache parameters tuned for the highest prediction accuracy and *updating* indirect branch target addresses can be up to *35.75%* better than the BTB, and up to *17.77%* better than that of a trace cache with *no updating* indirect branch target addresses.

Although this chapter has examined the effect of the trace cache on indirect branch prediction, there may be a similar impact on other forms of prediction. For example, a load value predictor might be affected if there were separate entries for each copy of a given load in the trace cache. Measuring this effect would be important for any processor designer who wishes to address their predictor using locations in the trace cache rather than the address of the original instruction in main memory.

In the next chapter, the comparison of stack and register virtual machine is presented.

Chapter 5

Stack Architecture versus Register Architecture

5.1 Introduction

Whereas the previous chapter dealt with indirect branch prediction in general, this chapter returns to the core research problem of this dissertation, which is the relative performance of interpreter-based implementations of register and stack VMs. Inevitably, indirect branch prediction plays a key role in the trade-offs between the two architectures, because as we showed in section 3.3, interpreters use indirect branches to implement VM instruction dispatch, so they execute very large numbers of indirect branches.

To briefly recap from section 3.5, a long-running question in the design of VMs is whether a stack architecture or a register architecture can be implemented more efficiently with an interpreter. On the one hand, stack architectures allow smaller VM code so less code must be fetched per VM instruction executed. On the other hand, stack machines require more VM instructions for a given computation, each of which requires an expensive (usually unpredictable) indirect branch for VM instruction dispatch. Several authors have discussed the issue [Mye77, SM77, MB99, WP97] and presented small examples where each architecture performs better, but no general conclusions can be drawn without a larger study.

The first large-scale quantitative results on this question were presented by Davis

et al. [DBC+03, GBC+05] who translated JVM stack code to a corresponding register machine code. A straightforward translation strategy was used with simple compiler optimizations to eliminate instructions that become unnecessary in register format. Of the resulting register code, around 35% fewer VM instructions were needed to perform the same computation than the stack code. However, the resulting register VM code was around 45% larger than the original stack code and resulted in a similar increase in bytecodes fetched. Given the high cost of unpredictable indirect branches, these results strongly suggest that register VMs can be implemented more efficiently than stack VMs with an interpreter. However, this work did not include an implementation of the virtual register architecture, so no real running times were presented.

This dissertation extends the work of Davis et al. in two respects. First, our translation from stack code to register code and subsequent optimization are much more sophisticated. We use a more aggressive copy propagation approach to eliminate almost all of the stack load and store VM instructions. We also optimize redundant constant load and other common subexpressions and move loop invariants out of loops. The result is that an average of more than 46% of executed VM instructions are eliminated. The resulting register VM code is roughly 26% larger than the original stack code, compared with the 45% for Davis et al. We find that the increased cost of fetching more VM code requires an average of only 1 extra CPU load per executed VM instruction eliminated. Given that VM dispatches are much more expensive than CPU loads, this indicates strongly that register VM code is likely to be much more time-efficient when implemented with an interpreter. The cost of this gain is the slightly increased VM code size.

A second contribution, which appears in Chapter 6, is measurements of running times and code behaviour for a fully functional, interpreter-based implementation of a register JVM. We present comparative experimental results for four different VM instruction dispatch mechanisms on twelve different benchmark programs from the SPECjvm98 and Java Grande benchmark suites. Our results include measurements from hardware performance counters that allow us to investigate the effect of using a register rather than a stack VM on the microarchitectural behaviour of the interpreter.

While we present experimental results on interpreter running times and code size for stack and register VMs, there are other factors to consider in the choice of code format. Compiling source code to stack-based bytecode is usually simpler than compiling to

register code, one of the reasons being that there is no need for a register allocator. If the compilation has been simple, stack code is also usually relatively simple to decompile. Similarly, stack-based bytecode may be better suited than register code as a source language for JIT compilation, at least partly because there is no assumption about the number of available registers. Apart from execution speed and suitability for JIT compilation, there are other issues in the choice of code format:

Code Size One of the attractions of a stack VM is that the code is quite compact, due to the absence of explicit register arguments. In the next chapter, we present work that shows that the bytecode for a register VM is only 26% larger than stack bytecode. In the case of Java, however, the bytecode only accounts for about 18% of a class file [AP98]. Nonetheless, various techniques such as those employed by JAX [TSL+02] can be employed to reduce the constant-pool size. As a result bytecode can occupy as much as 75% of the memory footprint in some embedded systems [CSCM00].

There are other options for the code format. For example, compressed syntax-tree based representations [KF99] are around twice as compact as stack-based bytecode, and are often considered a better source language for JIT compilation because they retain most of the high-level information from the source code. However, such tree based encodings are difficult to interpret efficiently, so they are most suitable when the VM will be implemented using only a JIT compiler.

Compressed Code Size It is also important to recall that code size is not only important because of the memory consumed, but also because programs may need to be sent over networks, so smaller code may arrive more quickly and use less bandwidth. In the case of Java, although the constant pool is usually large, the contents tend to be easily compressed, repeating text, highly suitable for the JAR file format commonly used for class file transport. Typically class files are compressed to about 50% of their original size and schemes have been proposed that compress class files even further, up to 10% to 25% of their original size [Pug99]. It is worth noting that, in the case of Java, as its role has changed since its inception as a language for dynamic content in web-browsers, this feature has become less important. It may be a more significant requirement in other VMs, depending on their role.

Preparation Time Much work has been done in the JVM in the area of bytecode verification, a task which is greatly simplified by the simpler stack IR. One area which a VM designer may wish to consider, but which we do not examine in this dissertation, is the issue of how much more difficult bytecode verification becomes when dealing with a register IR.

Portability We do not envisage a huge difference between a stack-based IR and a register-based one, as long as neither make assumptions about the underlying hardware.

Complexity of Implementation As we note elsewhere, a stack IR can be an easier compilation target (the complexity of the compiler being the issue here). From a VM interpreter point of view, a stack IR and register IR in terms of complexity of implementation seem, from our experience, to be roughly equivalent. It is a different issue if one is choosing an IR with a view to the complexity of the JIT in a VM (if present). For a naive inlining JIT, a register IR is clearly preferable while for a more sophisticated JIT, a stack IR may be preferred.

The choice of IR for the work presented in this dissertation was driven primarily by a different concern to those discussed above. In order to perform a meaningful comparison between a stack VM and register VM, it was decided to keep the instruction sets as similar as possible. In an environment where experimental issues are not the driving force, the choice of IR is likely to be made on the basis of a combination of these issues.

5.2 Stack versus Register

The cost of executing a VM instruction in an interpreter consists of three components: dispatching the instruction, accessing the operands and performing the computation. In this section we consider the influence of these three components on the running time of VM interpreters.

5.2.1 Dispatching the Instruction

In instruction dispatch, the interpreter fetches the next VM instruction from memory and jumps to the corresponding segment of its code that implements the fetched instruction. A given task can often be expressed using fewer register machine instructions than stack ones. For example, the local variable assignment `a = b + c` might be translated to stack JVM code as `iload c, iload b, iadd, istore a`. In a virtual register machine, the same code would be a single instruction `iadd a, b, c`. Thus, virtual register machines have the potential to significantly reduce the number of instruction dispatches.

Instruction dispatch is typically implemented in C with a large `switch` statement, with one case for each opcode in the VM instruction set. Switch dispatch is simple to implement, but is rather inefficient. Most compilers produce a range check and an additional unconditional branch in the generated code for the `switch`. In processors using a branch target buffer (BTB) for indirect branch prediction, there is only one entry in BTB for all indirect branch targets. Thus, the indirect branch generated by most compilers is highly (around 95% [EG03]) unpredictable on architectures using BTB for indirect branch prediction. The main advantages of switch dispatch are that the bytecode executed by the VM is compact, and it can be implemented using any ANSI C compiler.

An alternative to the `switch` statement is *token threaded dispatch*. Threaded dispatch takes advantage of languages with labels as first class values (such as GNU C and assembly language) to optimize the dispatch process, at the expense of the portability of the interpreter source code. Token-threaded dispatch uses the opcodes to lookup the target address of their implementation in a dispatch target address table. This enables the range check and additional unconditional branches to be eliminated, and permits the code to be restructured to improve the predictability of the indirect branch dispatch (to around 45% [EG03]). On architectures with BTBs for indirect branch prediction, each instruction implementation has its own indirect branch instruction and thus, multiple entries of indirect branch targets can exist in the BTB.

Another alternative is *direct threaded dispatch*. Direct-threaded code directly encodes the jump addresses as the opcodes of instructions and thus further reduces the cost of dispatch. The code to be interpreted is translated from bytecode into threaded

code. In threaded code, VM opcodes are no longer bytes, but are instead addresses of the executable native code within the interpreter that performs the computation that corresponds to the original VM opcode. Thus the table lookup from token threaded code can be eliminated, further reducing the cost of dispatch. Direct threaded dispatch requires first class labels, a translation step, and the VM code size increases by up to a factor of four on a 32 bit machine or eight on a 64 bit machine.

An even more sophisticated approach is *inline threaded dispatch* [PR98] which copies executable machine code from the interpreter and relocates it to remove the dispatch code entirely. This requires an even more complicated translation from bytecode, much greater memory requirements, and is even less portable than the other forms of threaded dispatch. It is, however, the fastest VM instruction dispatch mechanism, and we present results for it in this dissertation.

Another alternative is context threading [BVZB05] uses subroutine threading to change indirect branches to call/returns, which better exploits the hardware return-address stack to reduce the cost of dispatches. However, this approach requires some mechanism to generate native executable machine code at run time. We have not implemented this dispatch mechanism, although we believe that it is slightly less efficient than inline threading, which eliminates indirect branches entirely.

As the cost of dispatches falls, any benefit from using a register VM instead of a stack VM falls. However, `switch` and token threaded dispatch are the most commonly used interpreter techniques because two of the main motivations for using an interpreter are to avoid additional translation steps, and to maintain the small size of bytecode. If ANSI C must be used (as is the case in the interpreters for many scripting languages) then `switch` is the only efficient alternative.

5.2.2 Accessing the Operands

The location of the operands must appear explicitly in register code, whereas in stack code, most operands[1] are found relative to the stack pointer. Thus, the average register instruction is longer than the corresponding stack instruction, register code is larger than stack code, and register code requires more memory fetches to execute. Small code size and small numbers of memory bytecode fetches are the main reasons why

[1]Not all stack VM instruction operands are on the stack, eg. immediate operands and local variables

stack architectures are so popular for VMs.

From the viewpoint of a VM interpreter, a stack VM must keep track of the bytecode instruction pointer (IP), the stack pointer (SP), and the frame pointer (FP) while a register VM only needs the IP and FP. Thus, when the register VM is implemented using an interpreter on a real processor, one variable fewer is required in the inner loop of the interpreter than for the stack VM. This reduces real machine register pressure, and may result in less spilling and reloading of variables. On platforms with small numbers of architected registers, such as Intel x86 processors which have only eight general purpose registers, this reduction in register pressure may impact performance. Moreover, a stack VM must update SP as values are pushed or popped.

5.2.3 Performing the Computation

Given that most VM instructions perform a simple computation, such as adding or loading, this is usually the smallest part of the cost. The basic computation has to be performed regardless of the instruction format. However, eliminating loop invariant and redundant loads (common subexpressions) is only possible on a register VM[2]. In Section 5.3.3, we exploit this property to eliminate repeated loads of identical values in a register VM.

5.3 Translation and Optimization

In this section we describe a system of translating JVM stack code to virtual register code and its optimization in a just-in-time manner. However, it is important to note that we do not advocate JIT translation (or any particular run-time translation) from stack format to register format as the best or only way to use virtual register machines. It is a possibility, maybe even an attractive one, but our main intention in doing this work is to evaluate free-standing virtual register machines. Run-time translation is simply a mechanism we use to compare stack and register versions of the JVM easily.

[2]In theory, the stack VM can benefit from eliminating complex common subexpressions by storing the computational results in local variables and reloading those values onto the operand stack when needed. In practice, we don't find any such complex common subexpressions, which may be due to the optimization already done by Java compiler. The stack VM won't benefit from simple redundant loads because the value will be loaded onto the stack anyway.

In a realistic system, we would use only the register machine, and compile for that directly. It is also important to note that we use standard, well-known JIT compiler techniques for this translation. We are interested in the results of the translation, not the translation itself.

5.3.1 Translation from Stack to Register

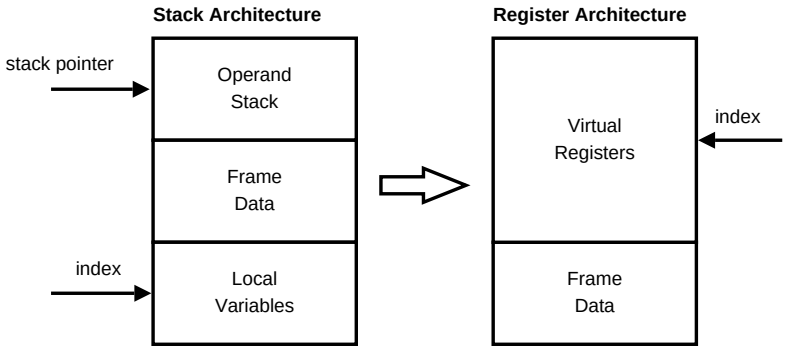

Figure 5.1: The structure of a Java frame

Our implementation of the JVM pushes a new Java frame onto a run-time stack for each method call. The Java frame for a stack architecture contains local variables, frame data, and the operand stack for the method (see Figure 5.1). In the stack JVM, a local variable is accessed using an index, and the operand stack is accessed via the stack pointer. In the register JVM, both the local variables and operand stack can be considered as virtual registers for the method. There is a simple mapping from stack locations to register numbers, because the height and contents of the JVM operand stack are known at any point in a program [Gos95]. In practice, the number of virtual registers (local variables and stack slots) in a method will only be limited by the size of the operand to specify the register number. However, it is desirable to minimize the size so that the Java frame will be small.

In the stack JVM, most operands of an instruction are implicit; they are found on the top of the operand stack. Most of the stack JVM instructions are translated into corresponding register JVM instructions, with implicit operands translated to explicit operand registers.

Figure 5.2 shows a simple example of bytecode translation. The bytecode adds two

integers from two local variables and stores the result back into another local variable.

Stack bytecode	Register bytecode
`iload_1`	`move r1 -> r10`
`iload_2`	`move r2 -> r11`
`iadd`	`iadd r10 r11 -> r10`
`istore_3`	`move r10 -> r3`

Figure 5.2: Stack bytecode to register bytecode translation. Assumption: current stack pointer before the code shown above is 10. The registers after `->` are destination registers

There are a few exceptions to the above one-to-one translation rule:

1. `pop` and `pop2` can be eliminated immediately because they are not needed in the virtual register machine code. For example, a lot of invoke instructions (method calls) push onto the operand stack a return value that is not used by the following instruction and `pop/pop2` instruction are needed in stack JVM to maintain consistency of the operand stack.

2. Instructions that load a local variable onto the operand stack or store a value from the operand stack in a local variable are translated into `move` instructions.

3. Stack manipulation instructions (e.g. `dup`, `dup2` ...) are translated into appropriate sequences of `move` instructions by tracking the state of the operand stack.

4. The `iinc` instruction in the stack JVM is used to increment a local variable by a constant value. `iinc` is an interesting VM instruction in stack JVMs because the computation is done without the operand stack. The computation should push an operand and a constant, add, and store the result to a local variable. It can be regarded as a type of register VM instruction that is available in the stack JVM. We translate an `iadd` or `isub` into an `iinc` VM instruction if one of its operands is a small integer constant (i.e. it is preceded by a VM instruction that pushes a small integer constant onto the stack).

5.3.2 Method Invocation

JVM methods invocation instructions, such as `invoke_virtual` are unusual in that they take a variable number of operands from the stack. As with other instructions, we include the locations of these operands in the register version of the instruction. The result is that method invocation instructions are variable length in the register VM. The number of bytes in the instruction depends on the number of items that the original method call takes from the stack when the method call is made.

In a stack JVM, operands (parameters) always come from the top of the stack, and become the first local variables of the called method. A common way to implement a stack JVM is to overlap the current Java frame's operand stack (which contains a method call's parameters) and a new Java frame's local variables.

In the register JVM, we don't overlap the Java frames to pass method parameters. Instead, we copy all the parameters from the virtual registers in the calling method's Java frame into the virtual registers in the Java frame for the new (called) method. We considered a similar mechanism in our virtual register machine as in a stack JVM. We would place the parameters for a method invocation in consecutive registers, in the highest numbered registers for the method. Instead of copying the values of these registers into the stack frame of the called method, we could simply move the frame pointer to point to the first of these parameters. Although this would provide an efficient parameter passing mechanism, it prevents us from copy propagating into the source registers (parameters) of a method call. Even though the operands of method invocation VM instructions are contiguous after initial translation, once we have performed copy propagation and other optimizations this ordering is lost. However, the benefits of our optimizations are much greater than the small loss in efficiency of parameter passing.

5.3.3 Optimization

In the stack architecture, computation is done through the operand stack. The operands of an instruction are pushed onto the operand stack before they can be used, and results are stored from the operand stack to local variables to save the value. In the register architecture, most of the operand stack load and store instructions are redundant. The main objective of optimization is to take advantage of the opportunities provided by

a virtual register machine architecture. There are two main categories of redundant loads.

- Loads and stores between operand stack and local variables are translated into `move` instruction in register code. On average, more than 42% (see Figure 5.3) of executed VM instructions in the SPECjvm98 and Java Grande benchmark suites (including library code) consist of loads and stores between local variables and the operand stack.

- Redundant loads of constant values and other arithmetic common subexpressions: In the stack architecture, constants are loaded onto the operand stack each time when needed for computation. The same constant could be loaded multiple times in a method, which is required on a stack-based architecture. Before optimization, an average of 6% (see Figure 5.3) executed instructions are constant load instructions.

In order to make a fair comparison, we try:

- **Not** to perform optimizations that do anything other than take advantage of the register architecture. Such optimizations would give the register VM an unfair advantage over stack code.

- to keep the instruction set and their implementation in the interpreter the same except for the adaptation to the new instruction format and those differences mentioned in the Section 5.3.1.

An important question is whether the resulting comparison is fair. If we applied the same optimizations to the stack code, would it also be improved? In fact, the Soot optimization framework [VRHS$^+$99] was used to translate stack JVM code to three-address code. They applied more aggressive optimizations than we use, and translated the resulting code back to stack JVM code. In order to achieve any improvements in running time, they needed interprocedural optimizations (which we do not perform). They concluded that intraprocedural optimizations generally have very little effect on Java bytecode, on the basis that these intra-procedural optimizations can only work on scalar optimizations. This strongly suggests that the differences in performance we

measure are the result of inherent differences between stack and register code, rather than the result of applying optimizations to one and not the other.

A similar question could be asked about the quality of the register code. If we were to design a register machine from scratch and generate code for it from source, we might produce a more efficient VM implementation. However, it is essential to our comparison that there are as few differences as possible between the stack and register VM. Otherwise, our results might be affected by other implementation issues.

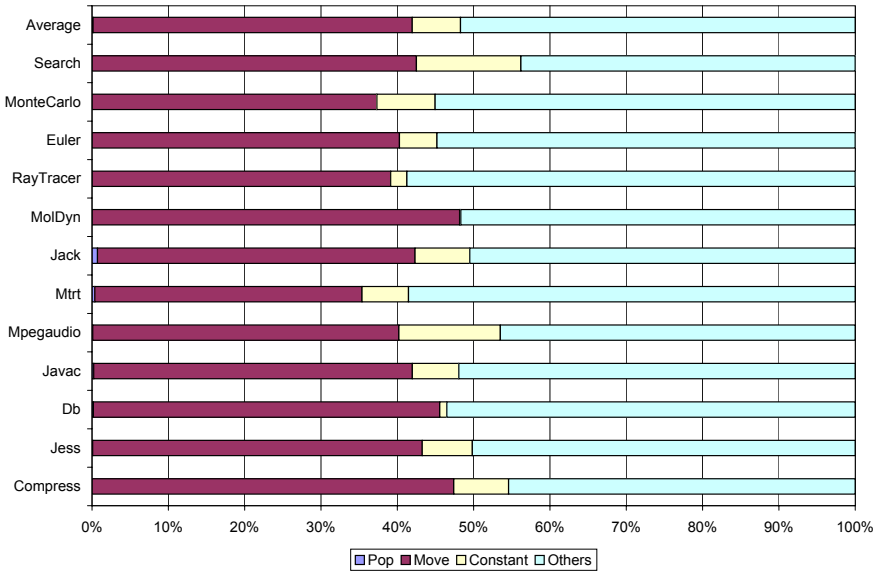

Figure 5.3: Different categories of dynamically executed instructions after translation without optimization

- Copy propagation: Copy propagation [Muc97] is applied to eliminate `move` instructions in basic blocks. The stack pointer is used to find out whether an operand on the stack is alive or dead. Forward copy propagation is used to eliminate operand stack loads and backward copy propagation is used to eliminate operand stack stores.

- Global redundant load elimination: An immediate dominator tree is used to discover and eliminate redundant constant load instructions and other common subexpressions globally.

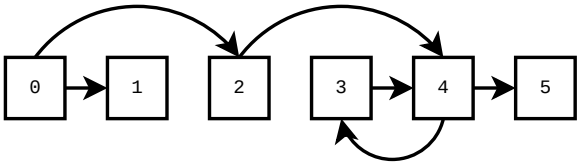

Figure 5.4: The control flow of the example

- Loop invariant motion: An immediate dominator tree and loop information are used to discover and move constant load instructions and other loop-invariant instruction out of loops.

5.3.4 Putting it all together

The runtime process for translating stack bytecode and optimizing the resulting register instructions for a Java method are as follows:

1. Translate original bytecode into virtual register intermediate representation and build a factored control flow graph [CGHS99]
2. Apply local copy propagation on basic blocks [Muc97]
3. Build a dominator tree [LT79] and enhance the intermediate representation with SSA form [CFR+91]
4. Remove dead code [CFR+91]
5. Apply global copy propagation
6. Apply global redundant load elimination
7. Apply loop invariant code motion [Muc97]
8. Virtual register allocation [Mös00, BCT94] and remove SSA ϕ functions
9. Write the optimized register code into virtual register bytecode in memory.

In order to better demonstrate the effect of the optimizations, we present the following example (see Figure 5.5 for Java source code and Figure 5.6 for corresponding bytecodes) with 6 basic blocks and one loop (see Figure 5.4 for its control flow graph). In Figure 5.6:

```
public int hashCode()
{
    if (cachedHashCode != 0)
      return cachedHashCode;

    int hashCode = 0;
    int limit = count + offset;
    for (int i = offset; i < limit; i++)
      hashCode = hashCode * 31 + value[i];
    return cachedHashCode = hashCode;
}
```

Figure 5.5: Source code for the `hashCode()` method in the `java.lang.String`(GNU Classpath 0.90) class.

- The VM instruction operands with # are immediate operands.
- Virtual register numbers are indicated with an initial `r`.
- Field identifiers are shown using the names of the fields.
- In each instruction, the register number after `->` is the destination register.
- The stack VM instructions are numbered 1 to 37.
- The instruction numbers in the register code show the stack instruction from which each register instruction originated.

All the local load and store VM instructions have been eliminated by the translation to register code. Constant load instruction 20 is loop invariant and has been moved out of the loop to its preheader. A total of 37 VM instructions has been reduced to just 19. Most importantly, the number of VM instructions in the loop (basic blocks 3 and 4) has been reduced from 13 to 6.

5.4 Conclusion

In this chapter, we first discuss the motivation to our research in the interpreter-based virtual register machine. Then the stack instruction architecture and the register instruction architecture is contrasted from the point of view of an interpreter. Next, the runtime translation of Java stack-based bytecodes into register-based bytecodes is presented and their instruction format differences are described. The optimizations

```
Stack VM Code                  Register VM Code
Basic block(0):                Basic block(0):
01. ALOAD_0                    02. GETFIELD r0.cachedHashCode -> r1
02. GETFIELD cachedHashCode    03. IFEQ r1 basic_block_2
03. IFEQ basic_block_2

Basic block(1):                Basic block(1):
04. ALOAD_0                    05. GETFIELD r0.cachedHashCode -> r1
05. GETFIELD cachedHashCode    06. IRETURN r1
06. IRETURN

Basic block(2):                Basic block(2):
07. ICONST_0                   20. ICONST #31 -> r1
08. ISTORE_1                   07. ICONST_0 -> r6
09. ALOAD_0                    10. GETFIELD r0.count -> r2
10. GETFIELD count             12. GETFIELD r0.offset -> r3
11. ALOAD_0                    13. IADD r2 r3 -> r2
12. GETFIELD offset            16. GETFIELD r0.offset -> r7
13. IADD                       18. GOTO basic_block_4
14. ISTORE_2
15. ALOAD_0
16. GETFIELD offset
17. ISTORE_3
18. GOTO basic_block_4

Basic block(3)                 Basic block(3)
19. ILOAD_1                    21. IMUL r6 r1 -> r3
20. BIPUSH #31                 23. GETFIELD r0.value -> r5
21. IMUL                       26. CALOAD r5 r7 -> r5
22. ALOAD_0                    27. IADD r3 r5 -> r6
23. GETFIELD value             29. IINC r7 #1 -> r7
24. ILOAD_3
26. CALOAD
27. IADD
28. ISTORE_1
29. IINC 3, #1

Basic block(4)                 Basic block(4)
30. ILOAD_3                    32. IF_ICMPLT r7 r2 basic_block_3
31. ILOAD_2
32. IF_ICMPLT basic_block_3

Basic block(5)                 Basic block(5)
33. ALOAD_0                    36. PUTFIELD r6 -> r0.cachedHashCode
34. ILOAD_1                    37. IRETURN r6
35. DUP_X1
36. PUTFIELD cachedHashCode
37. IRETURN
```

Figure 5.6: Original stack VM code and corresponding register VM code for the `hashCode()` method in the `java.lang.String`(GNU Classpath 0.90) class

made possible by the register instruction architecture are presented. Next, the overall process of translation and optimization are described. Finally, the `hashcode()` method in `java.lang.String` class from the Java class library is presented as an example to demonstrate the changes from stack-based bytecodes to register-based bytecodes.

In the next chapter, we are going to present our experimental results and analysis.

Chapter 6

Experimental Evaluation of Stack/Register Virtual Machines

6.1 Introduction

In the last chapter, we analyzed the difference between virtual stack VM and virtual register VM. In this chapter, we experimentally compare the interpreter-based stack and register virtual machine.

6.2 Setup

For the present work, we used the Cacao 0.95 (interpreter only with JIT disabled) as a base VM to implement the virtual register machine[1]. Cacao, released under the GPL, uses GNU Classpath as its class library and has a Boehm-Demers-Weiser garbage collector. Additionally, since version 0.93, Cacao has included a vmgen [EGKP02] interpreter generator, used to define the virtual register machine instruction set and generate the interpreter. Both the virtual register and virtual stack interpreters supports inline-threaded [PR98], direct-threaded, token-threaded, and switch dispatches.

We use the SPECjvm98 client benchmarks [SPE98] (size 100 inputs) and Java

[1]Cacao changes different types of constant instructions (such as `iconst_0` and `iconst_1`) into one generic one (such as `iconst #immediate`). In order to make a fair comparison between stack and register implementations, we retain all those forms of constant instructions, forgoing this default Cacao transformation.

Processor	OS	Compiler
AMD Athlon(tm) 64 X2 Dual Core Processor 4400+	Linux 2.6.14	GCC 4.0.3
Intel(R) Pentium(R) 4 CPU 2.26GHz	Linux 2.6.13	GCC 2.95
DEC Alpha 800MHz SimulateLCA4	Linux 2.6.8	GCC 3.3.5
IBM PowerPC 1066MHz	Linux 2.6.18	GCC 4.0.2
Intel(R) Core(TM)2 CPU 2.13GHz	Linux 2.6.18	GCC 3.2.3

Table 6.1: Hardware and software configuration

Grande [BSW+00] (Section 3, data set size A). Methods are translated to register code the first time they are executed; thus all measurements in the following analysis include only methods that are executed at least once. The measurements include both the benchmark program code and the Java library code (GNU Classpath 0.90) executed by the VMs.

Table 6.1 shows the hardware and software configuration for the experiments. In the rest of the chapter, we refer to these different processor architectures as AMD64, Intel Pentium 4, Alpha, PPC, and Intel Core 2 Duo.

6.3 Static Instruction Analysis of Register Code

Figure 6.1 shows the breakdown of statically appearing VM instructions after converting to register code (translating and optimizing). On average 1.8% of VM instructions are `pop` or `pop2` instructions. These can simply be translated to `nop` instructions in the register VM and eliminated, because they move the stack pointer, but do not move any values or perform any computation. A significant number of statically appearing `move` instructions are eliminated. Originally, `move`s account for 31% of VM instructions, but this is reduced to only 0.32% (of the original instructions) after translation. Similarly, optimization results in the elimination of constant load instructions, from an average of 28% of total statically appearing VM instructions down to 18% (of the original instructions) after translation. Eliminating other common subexpressions allows a further 2.1% of static VM instruction to be optimized away. Overall, an average of 44% of static VM instructions are eliminated.

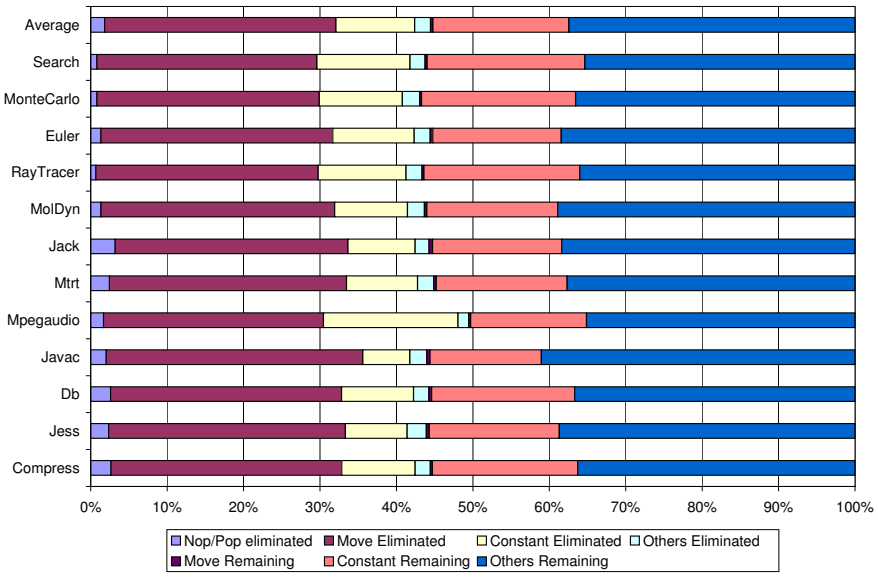

Figure 6.1: Breakdown of statically appearing VM instructions before and after optimization for all the benchmarks.

6.4 Stack Frame Space

Each method in the stack JVM has both a set of local variables and an operand stack. In order to perform computations, values must be copied from the local variables to the operand stack. Thus, within the interpreter, the stack frame for each method must contain two separate regions for local values which cannot be used interchangeably. The register VM, on the other hand, has only a single, unified set of registers which can both store local values and be used to perform operations on those values. Thus, there is potential for the register VM to require fewer slots in the stack frame than the stack VM.

As part of the translation from stack to register code we apply a simple graph-colouring register allocation to pack the values which were previously split between the locals and the evaluation stack into a smaller number of virtual registers. Table 6.2 shows the average number of stack frame slots required in a method for the locals and operand stack in the stack machine and for the virtual registers in the register machine. On average, methods for the stack VM require 5.47 slots. The corresponding number for our register VM code is 4.61. It is important to note, however, that the register VM

Benchmark	Stack	Register without redundant load	Register with redundant load
Compress	5.29	3.86	4.17
Jess	5.13	3.74	4.03
Db	5.33	3.89	4.20
Javac	6.34	4.72	5.02
Mpegaudio	5.56	4.14	5.37
Mtrt	5.38	3.97	4.28
Jack	5.19	3.83	4.26
MolDyn	5.62	4.14	4.49
RayTracer	5.60	4.07	4.32
Euler	5.57	4.09	4.44
MonteCarlo	5.31	3.90	4.13
Search	5.34	3.85	4.26
Average	5.47	4.02	4.41

Table 6.2: The comparison of required stack/local variable slots (virtual registers) between stack and register architectures

code normally has more live values. Eliminating redundant constant load instructions will make more variables alive at the same time, which means more virtual registers are required. If we do not apply these redundancy elimination optimizations, we find that the register machine needs an average only 4.02 slots. Even though a smaller stack frame size has little impact on the execution time of the VM interpreter, it may be beneficial to embedded or other small devices with tight memory constraints.

6.5 Dynamic Instruction Analysis of Register Code

In order to study the dynamic (runtime) behaviour of our register JVM code, we counted the number of VM instructions executed in the stack and register VMs. Figure 6.2 shows the breakdown of VM instructions dynamically executed before and after converting to register code.

1. The biggest category of eliminated instructions is `move`s, accounting for a much greater percentage (42%) of executed VM instructions than static ones (30%). The remaining `move`s account for only 0.28% of the original VM instructions executed.

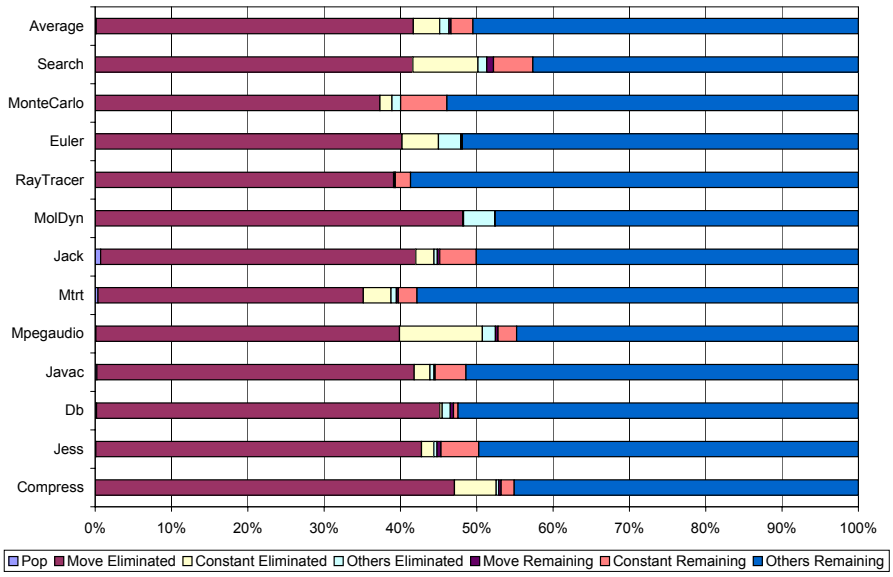

Figure 6.2: Breakdown of dynamically appearing VM instructions before and after optimization for all the benchmarks.

2. The second largest category of executed instruction elimination is constant load instructions (3.5% on average), which is much lower than the constant load instruction elimination (10% on average) in static code. The remaining dynamically executed constant VM instructions account for 2.9%. However, there are far more remaining constant load instructions (18%) in static code than those dynamically run (2.9%) in the benchmarks. We discovered that there are a large number of constant instructions in the initialization bytecode which are usually executed only once.

3. Elimination of other instructions accounts for 1.2% of VM instructions executed while the static elimination is an average of 2.1%.

4. Elimination of `pop/pop2` only contributes to a 0.14% reduction in dynamically executed instructions.

Overall, we eliminate an average of 46% of dynamically executed VM instructions. Generally speaking, copy propagation of `move` instructions produce the most effective result. Other optimizations are more dependent on the characteristics of the particular program. For example, in the benchmark *moldyn*, eliminated constant load VM

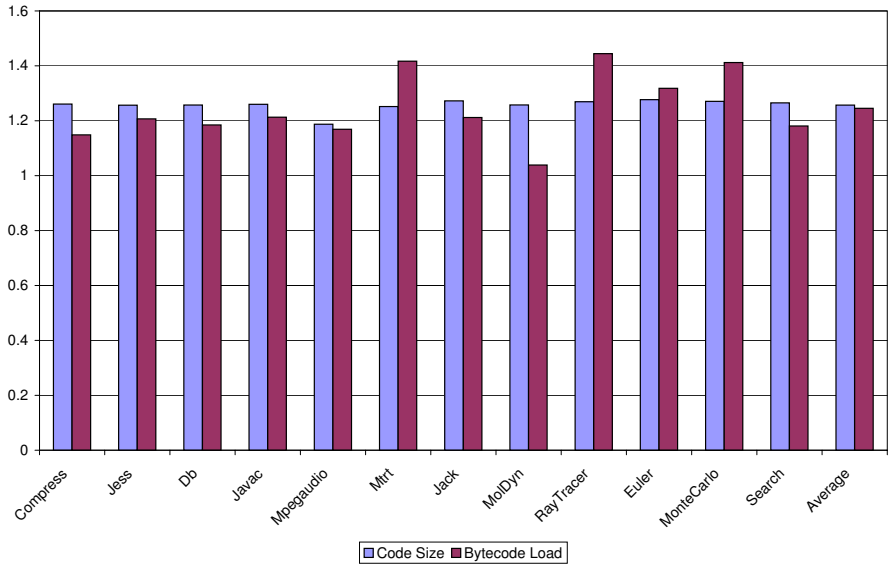

Figure 6.3: Fractional increase in total code size (static) of executed methods and bytecode loads (dynamic) for register against stack architecture

instructions account for only 0.11% of total executed instructions, although such instructions account for 10% of static instructions.

6.6 Code Size

The register VM code size is usually larger than that of stack VM. There are actually two effects in action here. First, register machine instructions are larger than stack instructions because the locations of the operands must be expressed explicitly. On the other hand, register machines need fewer VM instructions to do the same work, so there are fewer VM instructions in the code. Figure 6.3 shows the increase in code size of our register machine code compared to the original stack code. On average, the register code size is 26% larger than that of the original stack code, despite the fact that the register machine requires 44% fewer static instructions than the stack architecture. This is a significant increase in code size, but it is far lower than the 45% increase reported by Davis et al. [DBC+03].

As a result of the increased code size of the register JVM, more VM instruction bytecodes (both opcodes and operands) must be fetched, on average, from memory as

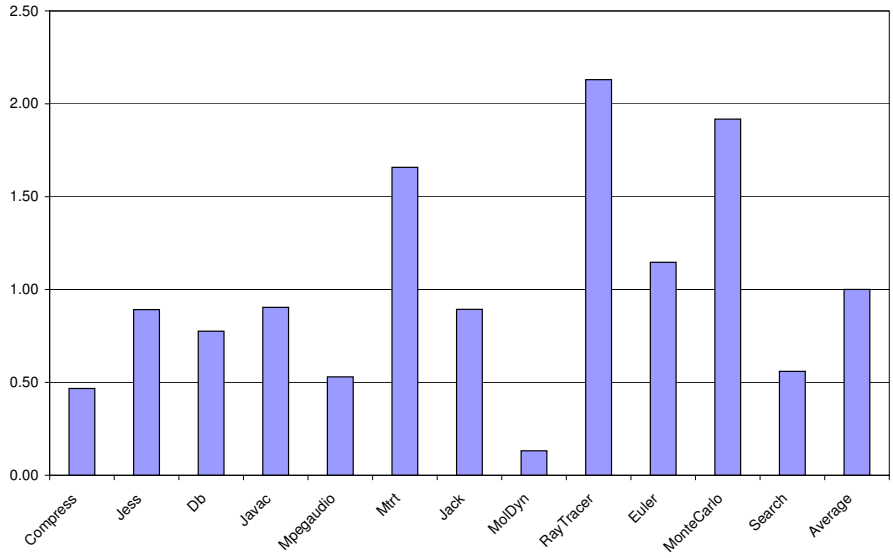

Figure 6.4: Increase in dynamically loaded bytecode instructions per VM instruction dispatch eliminated by using a register rather than stack architecture. In other words, if the register VM uses one less dispatch, how many extra bytes of bytecode must it load?

the program is interpreted. Figure 6.3 also shows the resulting increase in bytecode loads. Interestingly, the increase in overall code size is often very different from the increase in instruction bytecode loaded in the parts of the program that are executed most frequently. Nonetheless, the average increase in loads (25%) is similar to the average increase in code size (26%).

An alternative to fetching each operand location separately is to use a four-byte VM instruction containing the opcode and three register indices. This entire VM instruction could be fetched in a single load. However, it would still be necessary to extract the opcode and register numbers inside the four-byte VM instruction. This would involve shifting and masking the loaded VM instruction. Clearly the cost of such operations varies from one processor to another (for example the Pentium 4 has no barrel shifter, so large shifts are expensive). In general if a piece of code loads four successive bytes and does something with them, most compilers generate separate byte loads, rather than a single word load and using shifts and masks to extract the bytes. This strongly suggests that the latter approach is unlikely to be more efficient than single byte loads in a bytecode interpreter.

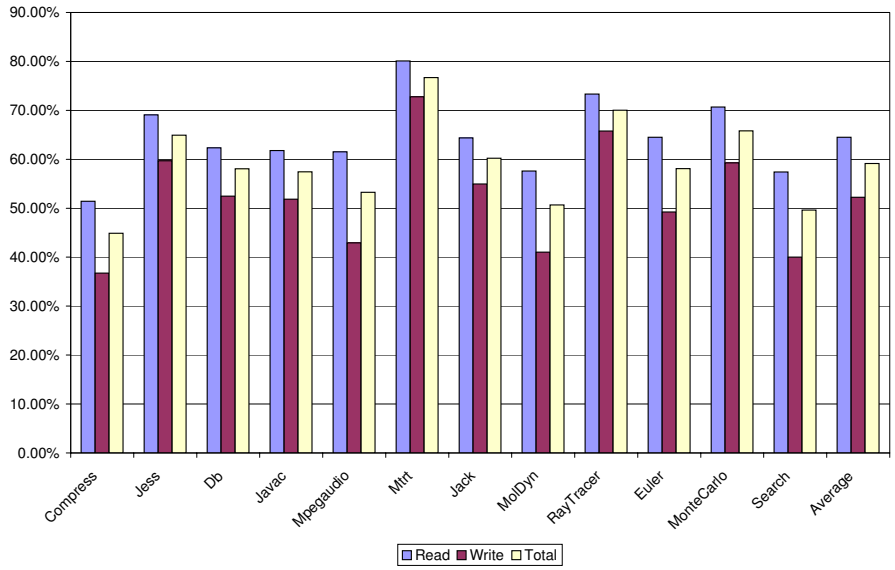

Figure 6.5: Dynamic number of CPU loads and stores required to access virtual registers in our virtual register machine expressed as a percentage of the corresponding loads and stores to access the stack and local variables in a virtual stack machine.

The performance advantage of using a register rather than stack VM is that fewer VM instructions are needed. On the other hand, this comes at the cost of increased bytecode loads due to larger code. To measure the relative importance of these two factors, we compared the number of extra dynamic bytecode loads required by the register machine per dynamically executed VM instruction eliminated. Figure 6.4 shows that the number of additional byte loads per executed VM instruction eliminated is small at an average of only 1.00 loads. On most architectures even one CPU load costs much less to execute than an instruction dispatch, with its difficult-to-predict indirect branch. This strongly suggests that register machines can be interpreted more efficiently on most modern architectures.

6.7 CPU Loads and Stores

Apart from CPU loads of instruction bytecodes, the main source of CPU loads in a JVM interpreter comes from moving data between the local variables and the stack. In most interpreter-based JVM implementations, the stack and the local variables are

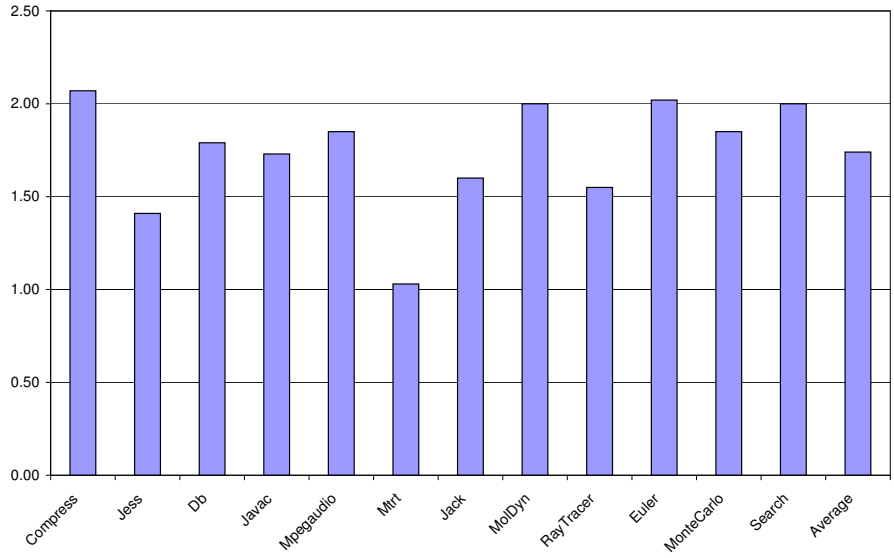

Figure 6.6: The reduction of CPU memory accesses for each executed VM instruction eliminated by using a register VM rather than a stack VM. This is analogous to the measurement in Figure 6.4

represented as arrays in memory. Thus, moving a value from a local variable to the stack (or vice versa) involves both a CPU load to read the value from one array, and a CPU store to write the value to the other array. A simple operation such as adding two numbers can involve large numbers of CPU loads and stores to implement the shuffling between the stack and registers.

In our register machine, the virtual registers are also represented as an array. However, VM instructions can access their operands in the virtual register array directly, without first moving the values to an operand stack array. Thus, the virtual register machine can actually require fewer CPU loads and stores to perform the same computation. Figure 6.5 shows (a simulated measure of) the number of the dynamic CPU loads and stores required for accessing the virtual register array, as a percentage of the corresponding loads and stores for the stack JVM to access the local variable and operand stack arrays. The virtual register machine requires only 65% as many CPU loads and 52% as many CPU writes, with an overall figure of 59%.

In order to compare these numbers with the number of additional loads required for fetching instruction bytecodes, we expressed these memory operations as a ratio

to the dynamically executed VM instructions eliminated by using the virtual register machine. Figure 6.6 shows that on average, the register VM requires 1.74 fewer CPU memory operations to access such variables per instruction dispatch eliminated. This is much larger than the number of additional loads required due to the larger size of virtual register code (1.00). Thus, the interpreter for the register VM would execute fewer loads overall.

However, these measures of memory accesses for the local variables, the operand stack and the virtual registers depend entirely on the assumption that they are implemented as arrays in memory. In practice, we have little choice but to use an array for the virtual registers, because there is no way to index CPU registers like an array on most real architectures. However, stack caching [Ert95] can be used to keep the topmost stack values in registers, and eliminate large numbers of associated CPU loads and stores. For example, around 50% of stack access CPU memory operations could be eliminated by keeping just the topmost stack item in a register [Ert95]. Thus, in many implementations the virtual register architecture is likely to need more CPU loads and stores to access these kinds of values.

6.8 Timing Results

To measure the benchmark running times of the stack and register-based implementations of the JVM, we ran both VMs on AMD64, Intel Pentium 4, Intel Core 2 Duo, Alpha and PowerPC systems (See Table 6.1). The stack JVM simply interprets standard JVM bytecode. The running time for the register JVMs includes the time necessary to translate and optimize each method the first time it is executed. However, our translation routines are fast. In the version of the virtual register machine that uses token threaded dispatch, the process of translation and optimization accounts for an average of only 0.8% of total execution time. As a result, we believe the comparison is fair. In our performance benchmarking, we run SPECjvm98 with a heap size of 70MB and Java Grande with a heap size of 160MB. Each benchmark is run independently.

We compare the performance of stack JVM interpreter and register JVM interpreter with four different dispatch mechanism: (1) switch dispatch, (2) token-threaded dispatch, (3) direct-threaded dispatch and (4) inline-threaded dispatch [PR98] (see Section 5.2). For fairness, we always compare the performance of stack and register

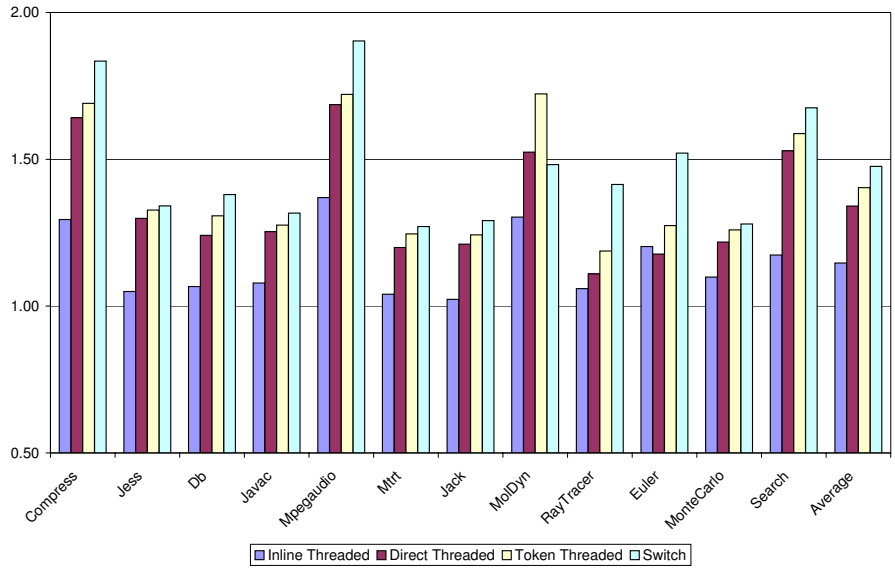

Figure 6.7: AMD64: register VM speedups against stack VM of same dispatch (based on average real running time of two runs)

interpreter implementations which use the same dispatch mechanism.

Figure 6.7 shows the speedup in running time of our implementation of the virtual register machine compared to the virtual stack machine on the AMD64 machine using the various dispatch mechanisms. With `switch` dispatch, the register VM has the highest average speedup (1.48) because `switch` dispatch is most expensive. Even with the efficient inline threaded dispatch, the register VM still has an average speedup of 1.15.

Figure 6.8 shows the same figures for a Pentium 4 machine, whose processor utilizes a trace cache. With inline threaded dispatch, the register VM has an average speedup of 1.00, and some benchmarks are very close to or worse than stack VM. The `switch` register VM has highest speedup (1.46). The `mtrt` benchmark performs very poorly for various dispatches, which may be due to high cost of threading using GCC 2.95 compiler.

Figure 6.9 shows the speedup of register VMs against stack VMs on the Intel Core 2 Duo processor. The average speedups of register over stack-based VMs are 1.15 (inline threaded), 1.32 (direct threaded), 1.29 (token threaded), and 1.65 (switch).

Figure 6.10 shows the speedups of register VMs against stack VMs on IBM PowerPC

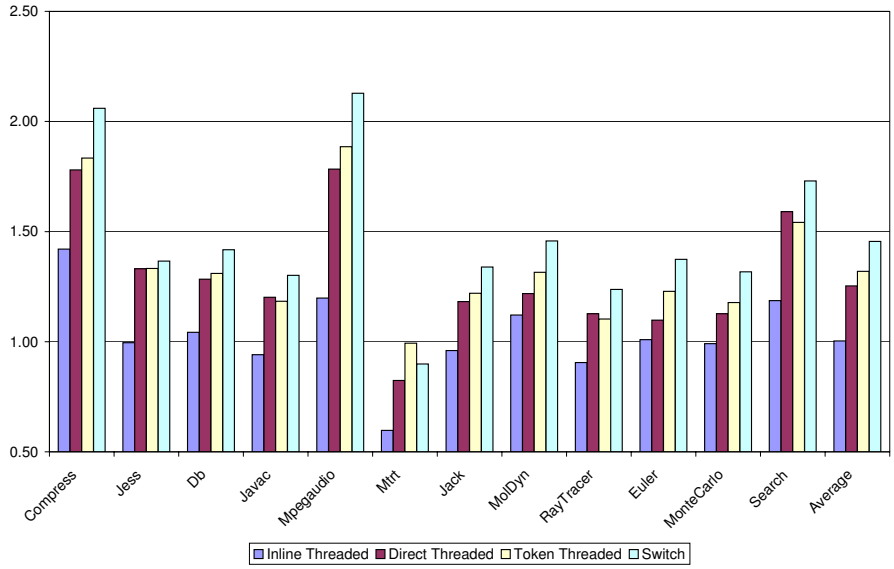

Figure 6.8: Intel Pentium 4: register VM speedups against stack VM of same dispatch (based on average real running time of five runs)

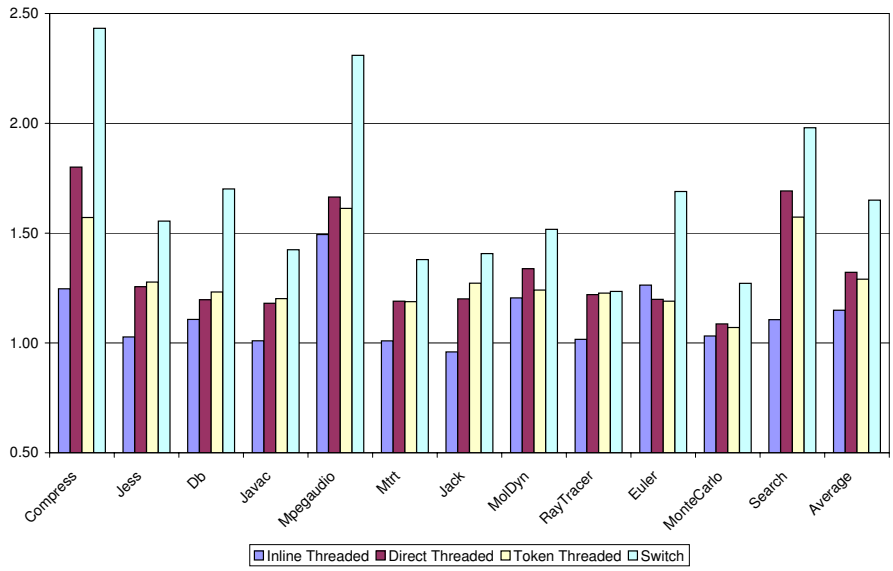

Figure 6.9: Intel Core 2 Duo: register VM speedups against stack VM of same dispatch (based on average real running time of three runs)

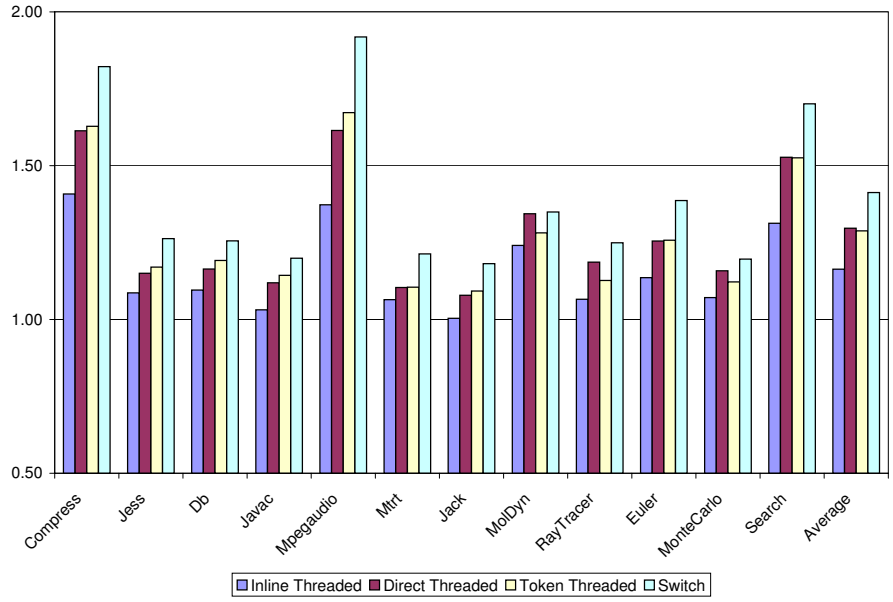

Figure 6.10: IBM PowerPC: register VM speedups against stack VM of same dispatch (based on average real running time of three runs)

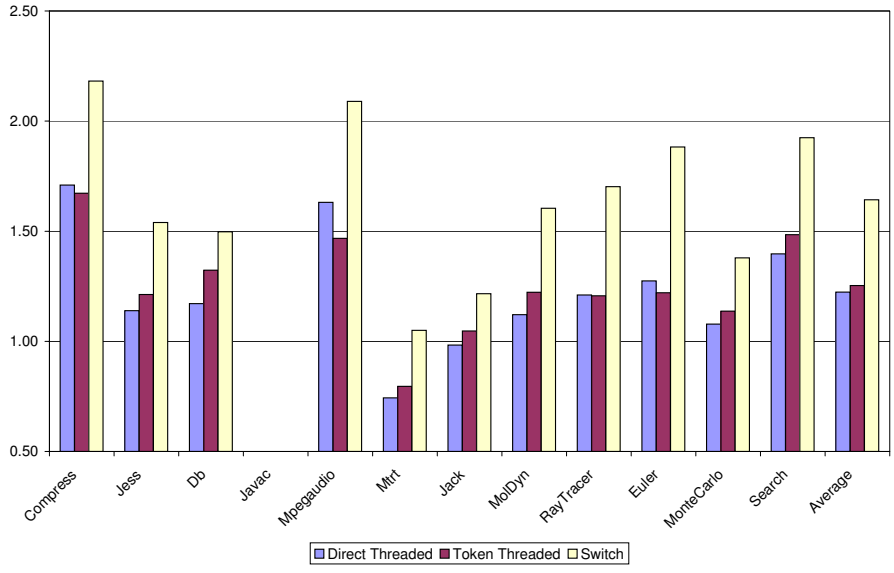

Figure 6.11: Alpha: register VM speedups against stack VM of same dispatch (based on average real running time of five runs)

processor. The average speedups for the four dispatch mechanisms (inline-threaded, direct-threaded, token threaded and switch) are 1.16, 1.30, 1.29, and 1.41 respectively.

Figure 6.11 shows the speedup of register VMs against stack VMs on the Alpha processor. The inline threaded dispatch is not working for Alpha and there are still bugs which prevent `javac` (which is thus excluded from the benchmark results) from running properly. The average speedups are 1.22 (direct threaded), 1.25 (token threaded), and 1.64 (switch).

6.9 Performance Counter Results

To more deeply explore the reasons for the relative performance, we use AMD64 hardware performance counters to measure various processor events during the execution of the programs. Figures 6.12 and 6.13 show performance counter results for the SPECjvm98 benchmarks *Compress* and *Jack* . We measure the data cache accesses, data cache misses, instruction cache fetches, instruction cache misses, retired taken branches (which include indirect branches; unfortunately there is no way to measure indirect branches alone by using AMD64's performance counters), retired taken branches mispredicted (indirect branches are the main source of misprediction), and retired instructions[2].

Figure 6.12 shows the measured performance counters for inline threaded, direct threaded and switch dispatches for the `compress` benchmark. From Figure 6.2, we know that, for the `compress` benchmark, 54% of executed VM instructions are eliminated compared to the stack architecture. As the dispatch method becomes more efficient, the difference between corresponding performance counters for register VM and stack VMs becomes smaller. For inline threaded dispatch, retired taken branches are almost the same for register and stack VMs. The main source of advantage is fewer retired instructions, which gives the register VM a speedup of 1.15 against the stack VM for inline threaded dispatch.

For the `compress` benchmark, the register version of the machine always executes fewer real machine instructions. As we saw in Figure 6.4 translation to register format

[2]On out-of-order processors, a retired instruction is one that has been executed and completed. Retired instructions indicate the number of instructions executed which contribute to the real-work of a program.

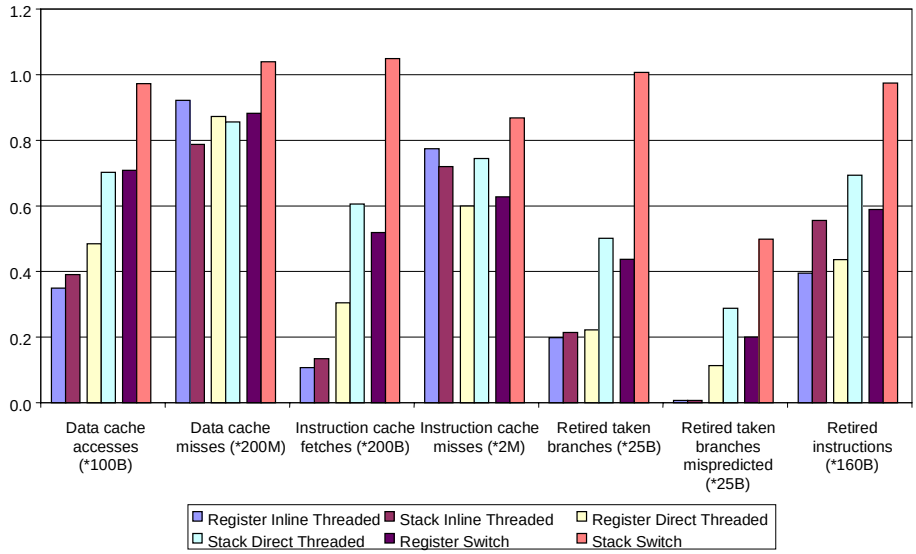

Figure 6.12: Compress: AMD64 performance counters

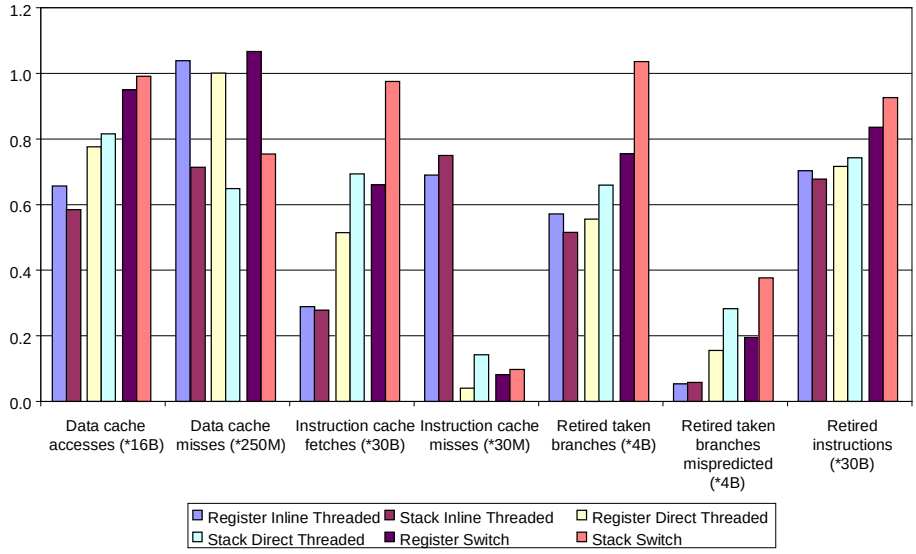

Figure 6.13: Jack: AMD64 performance counters

actually results in less than 0.5 extra bytecode loads per VM instructions eliminated. However, `compress` is the benchmark with the greatest reduction in real machine memory operations for manipulating local values (see Figure 6.5). This accounts for the much lower number of retired real machine instructions.

Figures 6.13 show the measured performance counters of the `jack` benchmark for inline threaded, direct threaded and switch dispatches. From Figure 6.2, we know that 44% of executed instructions are eliminated from `Jack` in the register VM. The data cache miss ratio and instruction cache miss ratio are much higher than those of the `compress` benchmark. For inline threaded dispatch, the register VM shows more data cache accesses, data cache misses, retired taken branches, and retired instructions than those of the stack VM. On the other hand instruction cache misses and retired taken branches mispredicted are lower. The inline threaded dispatch speedup of register VM against stack VM for the `Jack` benchmark is only 1.02. For inline threaded dispatch, both stack and register VMs show very high numbers of instruction cache misses when compared with other dispatch mechanisms because of binary executable code replication.

6.10 Dispatch Comparison

All the comparisons to this point have been the same dispatch-mechanism comparisons between stack and register architecture. For example, we have shown performance of the register VM interpreter using token-threaded dispatch as a speedup over the performance of the corresponding stack VM. In this section we compare differences between the dispatch mechanisms. The performance of the stack VM interpreter using switch dispatch is the baseline value (speedup=1.0) and all other variants are shown in comparison to that (see Figure 6.14). Sun's JDK 1.6.0 (interpreter mode only) gives a indicator of the speed of Cacao stack and register VMs and should be treated with caution because of different implementation.

We see that the more complex, less portable dispatch mechanisms give the greatest speedups. But we also observe that, at least for the benchmark results presented, the register machine has a significant edge. For example, if one has to choose between using direct-threaded dispatch on a stack VM and switch dispatch on a register VM, it should be noted that there is little difference in execution speed between the two

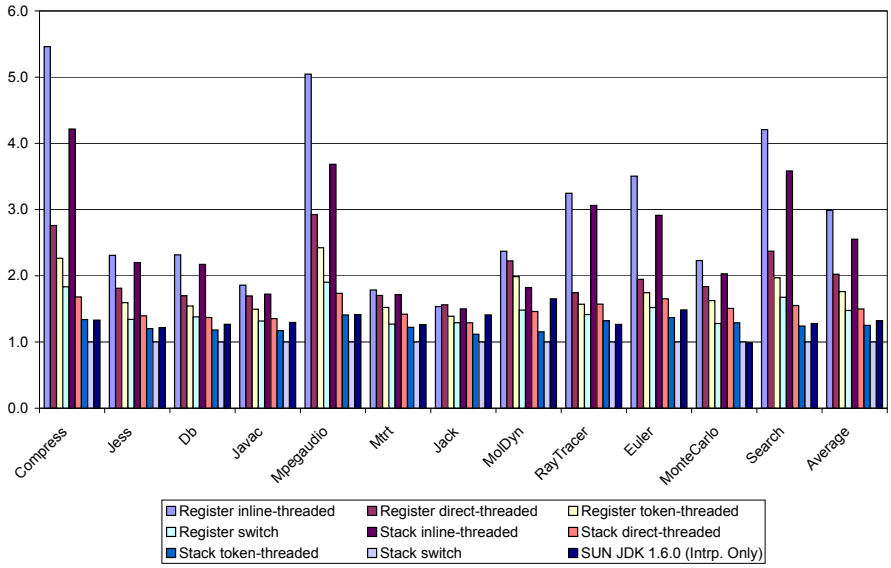

Figure 6.14: AMD64: speedups against the stack switch interpreter

implementations. However, switch dispatch is simpler to implement and much more portable. Furthermore interpreted bytecode is a fraction (typically 25%–50%) of the size of threaded code, so there is also a significant space saving. Therefore, the register VM interpreter with switch dispatch is preferable.

6.11 Discussion

Although our implementation of the register JVM translates the stack bytecode into register bytecode at the runtime, we do not envision this in a real-life implementation. The purpose of our implementation is to evaluate a virtual register JVM against an equivalent stack based one. Our register JVM implementation came directly from a modification of a stack-based JVM implementation, thus giving us two VMs identical in every other regard. Apart from the necessary adaptation of the interpreter loop along with some garbage-collection and exception handling modifications, there are very few changes to the original code segments responsible for interpreting bytecode instructions. The objective of doing so is to provide a fair comparison between the stack-based JVM and the register-based JVM.

Given a computation task, a register VM inherently needs far fewer instructions

than a stack VM does. For example, our register JVM implementation can reduce the static number of bytecode instructions by 44% and the dynamic number of executed bytecode instructions by 46% when compared to those of the stack JVM. The reduction of executed bytecode instructions leads to fewer real machine instructions for the benchmarks and significantly smaller number of indirect branches. It is these indirect branches which are very costly when mispredictions of indirect branches happen. Moreover, the elimination of large numbers of stack load and store (`move`) instructions reduces the number of loads and stores in a real processor. In terms of running time, the benchmark results show that our register JVM still outperforms an equivalent stack JVM even when both are implemented using the most efficient dispatch mechanism. This is a very strong indication that the register architecture can be implemented to be faster than the stack architecture.

An important question is whether we would generate better register code if we were to compile directly from Java source code rather than translating from register code. The *javac* compiler generates optimized stack code, but the optimizations may not suit register code. Furthermore, eliminating (partially) redundant expressions in stack code is rarely worthwhile, because the common expression must be stored and later recovered, which is often more expensive than recomputing the expression. Although eliminating simple redundant computations in stack code is easy, we might find it easier to eliminate more redundancy if we were working from source code. In particular, eliminating some kinds of redundant expressions, such as those described in the next section, depends on pointer analysis to ensure that the transformation is safe. Pointer analysis may be easier on source code than register code.

6.12 More Optimizations

6.12.1 Redundant Heap Load Elimination

As we saw in Section 5.3.3, register machines can take advantage of redundant computations much more easily than stack machines. This is because (unlike stack VMs) register VMs do not destroy operands to VM instructions as they use them. The results presented in the section 6.8 were for register machine code where redundant loads of constants and some simple common subexpressions involving local variables were

eliminated.

There is another category of redundant loads — the loads from class or object fields and array elements, and there has been some work on eliminating these redundant loads in compilers [FKS00]. However, it is very important to note that eliminating such loads from heap data structures requires sophisticated pointer alias analysis to ensure that the object or array element is not modified between apparently redundant loads [DMM98]. In particular, we need to know whether a reference to an object has escaped into another thread, which may modify the object. Alias analysis is complex and slow; we have not implemented it in our translation.

However, in order to examine the potential of the register machine to allow even more redundant loads to be eliminated, we performed some preliminary experiments without sophisticated alias analysis. Our very simple analysis is not safe — in particular it does not check for references escaping to another thread, but it allows us to get *some* idea of the potential benefit from register machines exploiting this sort of redundancy.

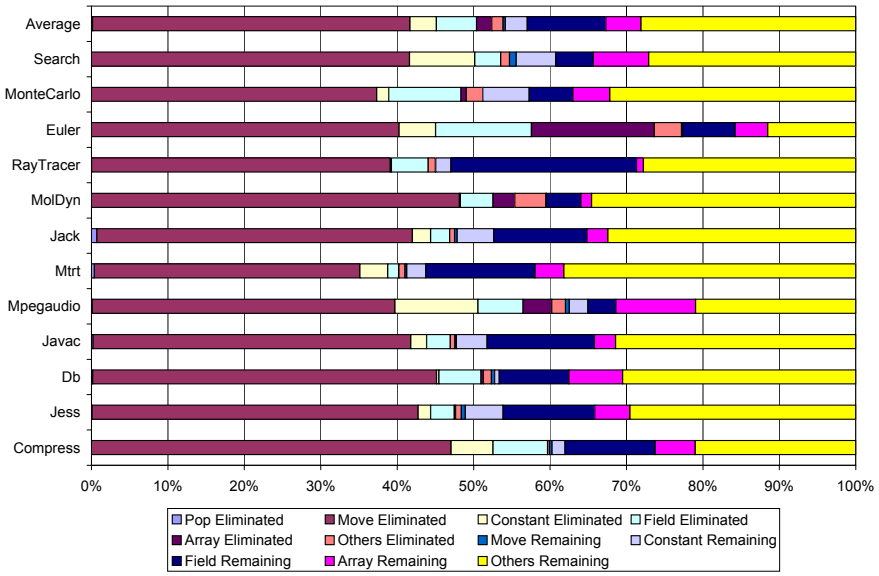

Figure 6.15: Breakdown of dynamically appearing VM instructions before and after additional redundant heap load elimination for all the benchmarks. These results are indicative only, because our translator makes unsafe assumptions about aliasing.

Figure 6.15 shows that an average of 5% of original executed VM instructions can be eliminated by removing redundant `getfield` VM instructions. The corresponding

figure for array loads is 2%. All benchmarks benefit from redundant `getfield` elimination, while only a few benchmarks benefit from redundant array load elimination. In the Euler benchmark, eliminated redundant array loads account for 13% of original executed VM instructions. After all optimizations the register machine requires only 23% of the original stack machine instructions.

Figure 6.16 shows the same dispatch speedup results for AMD64. The average speedup for inline threaded goes from 1.15 to 1.29 and that of switch dispatch from 1.48 to 1.74 as this optimization is added.

Figure 6.17 summarizes the average speedups of register VM against stack VM using the same dispatches with/without redundant heap load elimination. The register VM could potentially benefit significantly from eliminating these loads, but a real implementation of this optimization would require very sophisticated alias and escape analysis.

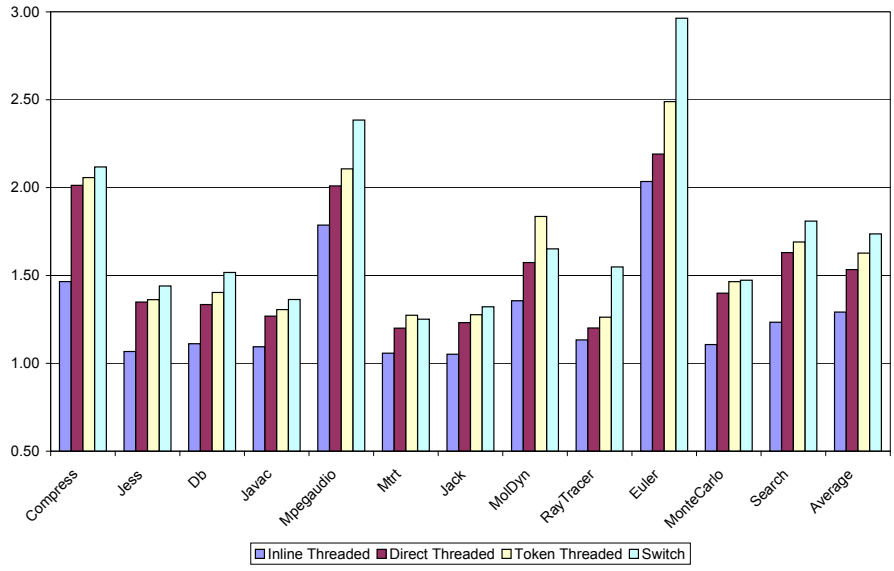

Figure 6.16: AMD64: Register VM speedups with additional redundant heap load elimination (based on average real running time of two runs). These results are indicative only, because our translator makes unsafe assumptions about aliasing.

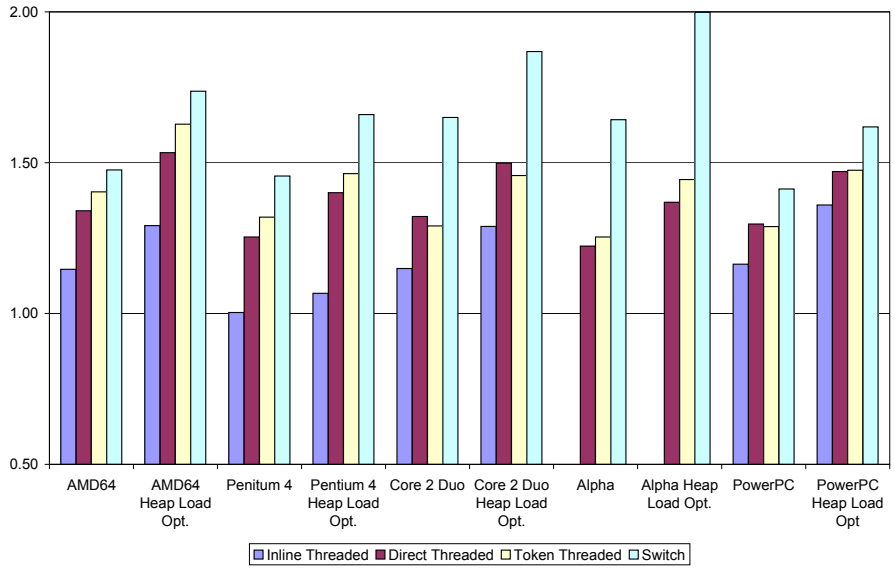

Figure 6.17: The average speedups of register VM against stack VM using the same dispatch for different processors. The results which include heap load optimization are indicative only, because our translator makes unsafe assumptions about aliasing.

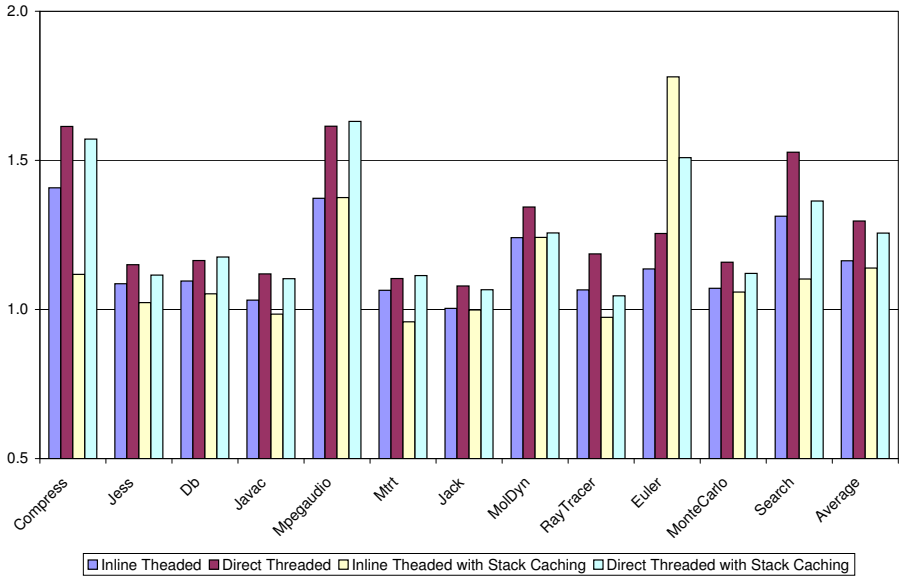

Figure 6.18: PowerPC: register VM speedups against stack VM (with and without stack caching) of same dispatch (based on average real running time of two runs)

6.12.2 Stack Caching for Stack VM

Stack caching [Ert95] can be used to keep the topmost stack values in registers, and eliminate large numbers of associated CPU loads and stores. For example, around 50% of stack access real machine memory operations could be eliminated by keeping just the topmost stack item in a register [Ert95]. Figure 6.18 shows the speedup against the stack VM with/without stack caching[3] using the same dispatch mechanisms. Stack caching did show improvements for the stack VMs. This improvement results in the speedups of the register VM going from 1.16 and 1.30 (against stack VM with no caching) down to 1.14 and 1.26 (against stack VM with caching) for inline-threaded and direct-threaded dispatch respectively.

6.12.3 Static Superinstructions

One way to reduce the number of VM interpreter dispatches is to add static *super − instructions* to the instruction set of the VM. These are new VM instructions that behave in the same way as a sequence of regular VM instructions. For example, if one found that `aload` VM instructions are often followed directly by a `getfield` VM instruction, one might introduce an `aload-getfield` superinstruction. Wherever this sequence appears in the program, it can be replaced by the superinstruction, reducing the number of dispatches. It has been argued that superinstructions can achieve the same effect as translating to a register machine, without the damaging increases in VM code size. In fact, this is not achievable in practice.

The main problem with superinstructions is choosing appropriate sequences. The superinstructions must be hardwired into the interpreter, at a time when the program to be run is usually unknown. Perhaps the best strategy for selecting sequences is to look at a large variety of programs and identify the most important sequences of VM instructions in those programs. Eller [Ell05] investigated using SPECjvm98 benchmarks to select superinstructions using a wide variety of selection strategies. He found that superinstructions could be added to a stack-based VM which would reduce the number of dispatches by up to 40%. However, to achieve that reduction, 1000 superinstructions were needed. This means that it is no longer possible to encode the

[3]We can only present the results of caching the topmost stack item for inline-threaded and direct-threaded dispatches

instruction opcode in a single byte. Furthermore, the interpreter code to implement the superinstructions becomes significantly larger than that of the original interpreter. In contrast, the register machine does not require any additional VM instructions.

6.12.4 Two-Address Instructions

Our register JVM uses a three-address instruction format for arithmetic instructions. An obvious way to reduce code size would be to use a two-address instruction format for these instructions instead, where one of the source registers would also be the target register of the instruction. Such a change would reduce the size of these instructions from four bytes (one opcode and three register indices) to only three bytes. We investigated this possibility, but found that arithmetic instructions account for only an average of only 6.3% of statically appearing register VM instructions in the SPECjvm98 benchmarks. Thus, the overall reduction in code size from two-address instructions is likely to be small. Furthermore, there are some disadvantages with two-address instructions. They make sharing of common subexpressions more difficult, because one of the input values is overwritten by the output of the instruction. Additional `move` instructions must be introduced (or retained) to prevent values from being destroyed, which would both increase code size and reduce the efficiency of the VM. A more complicated allocation of variables to registers would also be needed to minimize the number of `move` operations introduced. Given that the potential reduction in code size was small anyway, we decided that this optimization was not worthwhile.

6.13 Applicability of Results to Related Questions

Although our experiments in this work have been limited to the JVM, we believe that the results will extend to other VMs which employ an interpreter. Already, the conversion of the Lua VM from stack machine to register machine (the upgrade from version 4.0 to version 5.0) has yielded a substantial improvement in performance. Ierusalimschy *et al* [IdFC05] have compared the stack machine implementation of version 4.0 to an equivalent register machine implementation (with no additional optimizations). Across their selected benchmarks, the register machine was an average of 1.30 times faster than the stack machine. More significantly, on the benchmark they specifically

selected to test the execution engine, a speedup of 2.28 was reported.

The main benefit of the transition from stack VM to register VM is the reduction in the number of VM instruction dispatches, and consequently a reduction in branch mispredictions. There are other benefits which we have observed such as a reduction in real machine instructions at the CPU level. For coarse-grained VMs with higher-level instruction sets which have a significantly lower number of instruction dispatches to begin with, the transition to a register VM will not yield the same speedups. For example, Vitale and Abdelrahman [VA04] found that inline threading had little benefit for a Tcl virtual machine, because each VM instruction performed a lot of work so dispatch accounted for only a small proportion of running time. It is likely that we would see similar results for a comparison of a stack and register VM for Tcl. Where the cost of dispatch is a small proportion of total time, there will be little benefit in any optimization to reduce dispatches.

Another interesting question is whether a stack or register VM is more suitable as a source language for JIT compilation. Winterbottom and Pike [WP97] suggest that a register IR may be easier to compile to native code because it is closer to the register architecture used by real processors. Others argue that a stack machine is better, because stack code does not make assumptions about the number of available registers.

Unfortunately, our results apply only to interpreters, and tell us little about JIT compilers. We believe that JIT compiling from well-behaved stack architectures like the JVM is probably a little easier than from register architectures because stack code is similar to the tree representations of expressions often used in real compilers. On the other hand, a register architecture allows more optimizations to be expressed, because common subexpressions can be eliminated in the register code, rather than relying on the JIT compiler to perform these kinds of optimizations.

However, an optimizing JIT compiler is typically a complex piece of software, and translating the VM bytecode to a format more useful to the compiler is likely to be only a small part of compilation regardless of whether a stack or register VM is used. On the other hand, our results indicate that for a simple code-inlining JIT, there are some significant gains to be made in choosing a register IR rather than a stack IR. For a more aggressive JIT, the choice is not so clear. Finally, for mixed mode JIT compilers such as Sun's Hotspot VM [SM01], interpretation speed is still important,

and therefore a register VM may be used to improve the performance of interpretation.

6.14 Conclusions

A long standing question has been whether virtual stack or virtual register VMs can be executed more efficiently using an interpreter. Virtual register machines can be an attractive alternative to stack architectures because they enable the number of executed VM instructions to be substantially reduced. In this dissertation we have built on the previous work of Davis et al. [DBC+03, GBC+05], which counted the number of instructions for the two architectures using a simple translation scheme. We have presented a much more sophisticated translation and optimization scheme for translating stack VM code to register VM code, which we believe gives a more accurate measure of the potential of virtual register machine architectures. We have also presented experimental results for a fully-featured register JVM.

We found that a register architecture requires an average of 46% fewer executed VM instructions. The resulting register code is 26% larger than the corresponding stack code. The increased cost of fetching more VM code due to larger code size involves only around one extra CPU load per VM instruction eliminated. On a AMD64 machine, the register machine has an average speedup of 1.48 if dispatch is performed using a C switch statement. Even if the more efficient inline-threaded dispatch is available, the average speedup over a corresponding stack JVM is still 1.15 for the register architecture on an AMD64.

In the next chapter, we conclude the dissertation.

Chapter 7

Final Thoughts

In this thesis, we have investigated the performance of virtual machine interpreters on modern architectures, with a particular focus on stack and register architectures. In the process of doing this virtual machine work, the unexpected interaction between indirect branch prediction and the trace cache was also identified and explained. In this chapter, some thoughts on the overall results and future work will be presented.

7.1 Experimentation and Systems Research

Chapter 4 of this dissertation presents experiments which study the interaction of branch prediction and the trace cache, particularly when executing programs with a lot of indirect branches, such as interpreters. Although branch target buffers and the trace cache are both reasonably well understood, to our knowledge the effect of combining the two was entirely unexpected. In most cases, indirect branches are sufficiently rare that the effect is unlikely to have a significant impact on the running times of programs. Nonetheless, the effect is interesting, and if another type of predictor were combined with the trace cache this effect might have a significant impact.

A particular reason why this result is interesting is that, although we commonly measure the performance of computers, we often have little understanding of the detailed interactions of components within the machine. In computer science research, there is a great emphasis on novelty, in the sense of proposing new features and techniques for solving problems. However, in other branches of science, novel experimental

work often involves studying what is already there rather than trying to always create something new. In the opinion of the author, studying the behaviour and interactions of existing hardware and software systems is an important part of understanding how we can build better systems in the future[1].

7.2 Stack versus Register Virtual Machines

In the hardware arena, the stack versus register war is over. Register based processors, such as Intel X86, ARM, and PowerPC, dominate. However, in the virtual machine arena, stack-based VMs still dominate. Historically, the decisions to choose a stack-based VM instructions instead of a register one may have been influenced by the popular Pascal P-code virtual machine [PD82]. Both Glenford J. Myers [Mye77] and Schulthess et al. [SM77] felt that part of the acceptance of stack architectures is unduly based on aesthetic delectation of designers and implementers when working with stack concepts. Schulthess et al. [SM77] went further to state that a computer architecture should be judged mainly on the efficiency and quality of the systems that grow on top of it.

Over the years, there has been very little research into the comparison of the real-world implementation of the stack-based virtual machine against the register-based virtual machine. Although there has been a heated debate over many decades, Davis et. al. [DBC+03, GBC+05] were the first to attempt any sort of large-scale quantitative study, but they only counted features of the programs; they didn't design and implement a full virtual machine, nor did they attempt to generate high-quality register machine code that would be comparable with the sort of optimized stack VM code that is generated by an optimizing Java compiler such as *javac*. To our knowledge, the work in this dissertation is the first to carry out such large scale comparisons with a full VM implementation[2].

[1] Interestingly, when we presented our first register machine paper at VEE 2005, those who were most interested in the results were people from industry. They said that there are many papers that provided new techniques, but ours was one of the few papers that provided them with enough data and analysis to understand which techniques should be used.

[2] In the last stages of writing this dissertation, the author found a rather obscure paper [WC99] in which Wongsiriprasert et al. implemented simple stack and register VMs for a small language. However, their results are only for small programs, and there is little analysis. Nonetheless, their work is the earliest that we are aware of that compares corresponding stack and register implementations of the same VM.

The work in this dissertation has shown that an interpreter for register-based JVM code can be much faster than an interpreter for a corresponding stack machine. We believe that similar results can be found for other virtual machines where execution time is dominated by the interpretation of fine-grained VM instructions. In fact, during the course of carrying out the research for this dissertation, a number of VMs have switched to using register format, and the decision for the Rain VM was partly based on the work in this dissertation.

Although the work in this dissertation allows VM designers to make informed choices between stack and register VMs for interpreted execution, an open question is the best VM format for JIT compilation. The results in this dissertation shed little light on this question, except to show that stack formats allow more compact code. In fact, we are not convinced that any sort of virtual machine is the best format as a source language for a JIT compiler. The advantage of VMs is that it is easy to interpret VM code. For JIT compilation, a higher level format, such as compressed trees [KF99] which are much more compact than stack VM code and maintain more information from the original source might be more suitable.

7.3 Future Work

In the course of preparing a PhD dissertation, one inevitably encounters interesting ideas that cannot be followed up due to insufficient time. This section presents some of these.

7.3.1 Compiling Source Directly Into Register-Based Code

In our implementation of register-based VM, the stack-code is dynamically translated into the register code during runtime. The overhead of translation and subsequent optimization will have small influence on the overall performance of a register-based VM. However, the overhead is small (1-2% of running time). Thus the result should be better if the source code were compiled directly into register-based bytecode. In addition, more optimization might be possible in an ahead-of-time Java compiler because the full information contained in the source code would be available. This would allow the register code to be optimized as thoroughly as the *javac* compiler optimizes

stack code. A problem with this approach for the purposes of comparison is that, in addition to needing a new compiler and interpreter, one would also need a new class verifier, loader, compression utility (for Java *jar* files) and so forth. This would make the comparison more difficult, because differences in performance might be the result of changes in several different components of the VM rather than just a couple. Nonetheless, a system intended to be used in practice, rather than for the purposes of comparison, would certainly compile directly from source code to register VM format.

7.3.2 Object Field Access Optimization

In section 6.12.1, object field access optimization was tentatively carried out. The result shows the potential of such optimization. However, the optimization was not very safe because more rigorous alias analysis was not done. If the source code were compiled directly into bytecode, it would be much easier to do more complex and thorough alias analysis.

7.3.3 Register Instruction Architecture

In order to make a fair comparison of stack and register virtual machine, there is one-to-one mapping between stack VM instructions and register VM instructions with just a few exceptions (see section 5.3.1). There are few changes to the implementation of each VM instruction between the stack VM and register VM except for the necessary adaption of explicit operands in the register-based VM. If a new register VM were designed from scratch, there would be more options to design the instruction set, which may result in more compact code and a more efficient interpreter.

7.3.4 Bytecode Verification

Another area of future work might be to investigate bytecode verification for a register-based virtual machine. Bytecode verification is important for code security. In a stack VM, the verification is simplified because the height of the operand stack is known at any point of a program. In register-based bytecode, it will be more complex to verify the bytecode because of the usage of registers is more complex than the case of a stack VM.

7.4 Conclusion

The performance of virtual machine interpreters is closely related to the host computer's ability to predict indirect branches. This dissertation examines the behaviour of virtual machine interpreters, and as part of the process shows that the interaction of the branch predictor and a trace cache in a processor can have a significant impact on branch prediction accuracy, particularly for programs with many indirect branches, such as VM interpreters.

The main work of this dissertation examines the long standing question of whether virtual stack or virtual register VMs can be executed more efficiently using an interpreter. Virtual register machines can be an attractive alternative to stack architectures because they enable the number of executed VM instructions to be substantially reduced. In this dissertation, we have built on the previous work of Davis et al. [DBC+03, GBC+05], which counted the number of instructions for the two architectures using a simple translation scheme. We have presented a much more sophisticated translation and optimization scheme for translating stack VM code to register VM code, which we believe gives a more accurate measure of the potential of virtual register machine architectures. We have also presented experimental results for a fully-featured register JVM.

We found that a register architecture requires an average of 46% fewer executed VM instructions. The resulting register code is 26% larger than the corresponding stack code. The increased cost of fetching more VM code due to larger code size involves only 1 extra real machine loads per VM instruction eliminated. On a AMD64 machine, the register machine has an average speedup of 1.48 if dispatch is performed using a C switch statement. Much of this saving comes from avoiding expensive indirect branch mispredictions caused by switch dispatch. However, even if the more efficient inline-threaded dispatch is available, which allows most indirect branches to be eliminated from the VM interpreter, the average speedup over a corresponding stack JVM is still 1.15 for the register architecture on AMD64. This strongly suggests that where speed is more important than code size, a register VM is preferable for efficient interpretation.

Bibliography

[AH96] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In P. Cointe, editor, *Proceedings ECOOP '96*, volume 1098 of *LNCS*, pages 142–166, Linz, Austria, July 1996. Springer-Verlag.

[ALE02a] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer, IEEE Computer Society*, pages 59–67, February 2002.

[ALE02b] Todd M. Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[AP98] Denis N. Antonioli and Markus Pilz. Analysis of the Java class file format. Technical report, 1998.

[Bad95] Wil Baden. Pinhole optimization. *Forth Dimensions*, 17(2):29–35, 1995.

[BBH+04] Darrell Boggs, Aravindh Bathka, Jason Hawkins, Deborah Marr, J. Alan Miller, Patrice Roussel, Ronak Singhal, Brett Toll, and K. S. Venkatraman. The microarchitecture of the Intel Pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8(1), February 2004.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[Bro06] Neil C. Brown. Rain VM: Portable Concurrency through Managing Code. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 253–267, September 2006.

[BSW+00] Mark Bull, Lorna Smith, Martin Westhead, David Henty, and Robert Davey. Benchmarking Java Grande applications. In *Second International Conference and Exhibtion on the Practical Application of Java*. Manchester, UK, April 2000.

[BVZB05] Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *2005 International Symposium on Code Generation and Optimization*, pages 15–26, March 2005.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[CG94] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *21st Symposium on Principles of Programming Languages*, pages 397–408, January 1994.

[CGE05] Kevin Casey, David Gregg, and M. Anton Ertl. Optimizations for a Java interpreter using instruction set enhancement. Technical Report TCD-CS-2005-61, University of Dublin, Trinity College, 2005.

[CGEN03] Kevin Casey, David Gregg, M. Anton Ertl, and Andrew Nisbet. Towards superinstructions for Java interpeters. In *Proceedings of the 7th International Workshoop on Software and Compilers for Embedded Systems (SCOPES 03)*, pages 329–343, September 2003.

[CGHS99] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *PASTE '99: Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop*

on Program analysis for software tools and engineering, pages 21–31, New York, NY, USA, 1999. ACM Press.

[CHP97] Po-Yung Chang, Eric Hao, and Yale N. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*, June 1997.

[CI01] Yul Chu and M. R. Ito. An efficient indirect branch predictor. In *Proceedings of the 7th European Conference on Parallel Computing (Euro-Par 2001)*, 2001.

[CMMP95] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 333–344, New York, June 22–24 1995. ACM Press.

[CSCM00] Lars Ræder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22(3):471–489, 2000.

[DBC+03] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The case for virtual register machines. In *Interpreters, Virtual Machines and Emulators (IVME '03)*, pages 41–49, 2003.

[Dew75] Robert B.K. Dewar. Indirect threaded code. *Communications of the ACM*, 18(6):330–331, June 1975.

[DH97] Karel Driesen and Urs Hlzle. Limits of indirect branch prediction, December 11 1997.

[DH98] K. Driesen and U. Hölzle. Accurate indirect branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 167–178, New York, June 27–July 1 1998. ACM Press.

[DH99] Karel Driesen and Urs Holzle. Multi-stage cascaded prediction. In *Euro-Par'99*, volume LNCS 1685, pages 1312–1321, 1999.

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 106–117, 1998.

[ECM02] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA:adr, second edition, December 2002.

[EG01] M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *Proceedings of the 7th European Conference on Parallel Computing (Euro-Par 2001)*, volume LNCS 2150, pages 403–412, 2001.

[EG03] M. Anton Ertl and David Gregg. The structure and performance of *Efficient* interpreters. *The Journal of Instruction-Level Parallelism*, 5, November 2003. http://www.jilp.org/vol5/.

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen — A generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

[Ell05] Helmut Eller. Optimizing interpreters with superinstructions. Master's thesis, Institut für Computersprachen, Technische Universität Wien, February 2005. http://www.complang.tuwien.ac.at/Dimplomarbeiten/eller05.ps.gz.

[Ert93] M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Láznè (Marienbad), 1993.

[Ert94] M. Anton Ertl. Stack caching for interpreters. In *EuroForth '94 Conference Proceedings*, pages 3–12, Winchester, UK, 1994.

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

[Fag05] Fabian Fagerholm. Perl6 and the Parrot virtual machine, April 1 2005. http://www.cs.helsinki.fi/u/pohjalai/k05/okk/seminar/Fagerholm-Parrot.pdf.

[FKS00] Stephen Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. *Lecture Notes in Computer Science*, (Volume 1824):155 – 174, Feb 2000.

[Fou07] The Perl Foundation. Parrot faq, April 10 2007. http://www.parrotcode.org/faq/.

[FPP97] D. H. Friendly, S. J. Patel, and Y. N. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 24–33, Los Alamitos, December 1–3 1997. IEEE Computer Society.

[GBC+05] David Gregg, Andrew Beatty, Kevin Casey, Brian Davis, and Andy Nisbet. The case for virtual register machines. *Science of Computer Programming, Special Issue on Interpreters Virtual Machines and Emulators*, Vol. 57:pp. 319–338, 2005.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Gos95] J. Gosling. Java Intermediate Bytecodes. In *Proc. ACM SIGPLAN Workshop on Intermediate Representations*, volume 30:3 of *ACM Sigplan Notices*, pages 111–118, San Francisco, CA, January 1995.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[GRA+03] Simcha Gochman, Ronny Ronen, Ittai Anati, Ariel Berkovits, Tsvika Kurts, Alon Naveh, Ali Saeed, Zeev Sperber, and Robert Valentine. The Intel Pentium M processor: microarchitecture and performance. *Intel Technology Journal*, 7(2):20–36, May 2003.

[Gri01] Robert Griesemer. Interpreter generation and implementation utilizing interpreter states and register caching. Patent 6192516 B1, US, 2001.

[Hug82] R. J. M. Hughes. Super-combinators. In *Conference Record of the 1980 LISP Conference, Stanford, CA*, pages 1–11, New York, 1982. ACM.

[IdFC05] R. Ierusalimschy, L.H. de Figueiredo, and W. Celes. The implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, 2005.

[JRS97] Quinn Jacobson, Eric Rotenberg, and Jim Smith. Path-based next trace prediction. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[Kay93] Alan C. Kay. The early history of smalltalk. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 69–95, New York, NY, USA, 1993. ACM Press.

[KF99] Thomas Kistler and Michael Franz. A tree-based alternative to Java byte-codes. *International Journal of Parallel Program*, 27(1):21–33, 1999.

[KK98] John Kalamatianos and David Kaeli. Predicting indirect branches via data compression. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, November 1998.

[KL02] AJ KleinOsowski and David J. Lilja. Minnespec: A New SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, Vol.1, June 2002.

[Kli81] Paul Klint. Interpretation techniques. *Software—Practice and Experience*, 11:963–973, 1981.

[Koo89] Philip J. Koopman, Jr. *Stack Computers: the new wave*. Ellis Horwood Limited, 1989.

[Kra83] G. Krasner. *Smalltalk 80: Bits of History, Words of Advice*. Addison-Wesley, Reading, MA, 1983.

[Lil00] David J. Lilja. *Measuring Computer Performance - A Practitioner's Guide*. The Press Syndicate of the University of Cambridge, The Pit Building, Trumpington Street, Cambridge, United Kingdom, first edition, 2000.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[LWY00] Sang-Jeong Lee, Yuan Wang, and Pen-Chung Yew. Decoupled value prediction on trace processors. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture*, pages 231–240, Toulouse, France, January 2000.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., second edition, April 1999.

[MB99] Blair McGlashan and Andy Bower. The interpreter is dead (slow). Isn't it? In *OOPSLA'99 Workshop: Simplicity, Performance and Portability in Virtual Machine design.*, 1999.

[Mös00] Hanspeter Mössenböck. Adding static single assignment form and a graph coloring register allocator to the Java Hotspot client compiler. Technical Report TR-15, Johannes Kepler University Linz Institute for Practical Computer Science, Altenbergerstrae 69, A-4040 Linz, 2000.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[Mye77] Glenford J. Myers. The case against stack-oriented instruction sets. *Computer Architecture News*, 6(3):7–10, August 1977.

[PA02] David Pereira and John Aycock. Instruction set architecture of Mamba, a new virtual machine for Python, September 27 2002.

[Pat99] Sanjay Jeram Patel. *Trace Cache Design for Wide-Issue Superscalar Processor*. PhD thesis, Computer Science and Engineering in The University of Michigan, 1999.

[PD82] Steven Pemberton and Martin Daniels. *Pascal Implementation: The P4 Compiler and Interpreter*. Ellis Horwood, 1982.

[PEP98] Sanjay J. Patel, Marius Evers, and Yale N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. In *ISCA*, pages 262–271, 1998.

[PFP97] Sanjay Jeram Patel, Daniel Holmes Friendly, and Yale N. Patt. Critical issues regarding the trace cache fetch mechanism. Technical Report CSE-TR-335-97, Computer Science and Engineering, University of Michigan, May 7 1997. Thu, 15 Oct 1998 15:18:24 GMT.

[PFP99] Sanjay J. Patel, Daniel H. Friendly, and Yale N. Patt. Evaluation of design options for the trace cache fetch mechanism. *IEEE Trans. Computers*, 48(2):193–204, 1999.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.

[PS93] Chris H. Perleberg and Alan Jay Smith. Branch target buffer design and optimization. *IEEE Trans. Computers*, 42(4):396–412, 1993.

[Pug99] William Pugh. Compressing Java class files. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 247–258, New York, NY, USA, 1999. ACM Press.

[PW95] Alexander Peleg and Uri Weiser. Dyamic flow instruction cache memory organized around trace segments independent of virtual address line. United States Patent 5,381,533, January 1995.

[PWL04] Jinzhan Peng, Gansha Wu, and Guei-Yuan Lueh. Code sharing among states for stack-caching interpreter. In *IVME '04 Proceedings*, pages 15–22, 2004.

[RBS96] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *International Symposium on Microarchitecture*, pages 24–35, 1996.

[RBS99] Eric Rotenberg, Steve Bennett, and James E. Smith. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers*, 1999.

[RJSS97] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and James E. Smith. Trace processors. In *MICRO*, pages 138–148, 1997.

[RLPN+99] Alex Ramirez, Josep-L. Larriba-Pey, Carlos Navarro, Josep Torrellas, and Mateo Valero. Software trace cache. In *Proceedings of the 13th international conference on Supercomputing*, pages 119–126, 1999.

[RLV+96] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 150–159, 1996.

[Rot05] Eric Rotenberg. *Sepculative Execution in High Performace Computer Architectures*, chapter 4. CRC Press, 2005. (Eds) D. Kaeli and P.-C. Yew.

[RVJS00] Ramesh Radhakrishnan, Narayanan Vijaykrishnan, Lizy Kurian John, and Anand Sivasubramaniam. Architectural issues in Java runtime systems. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 387–398, Toulouse, France, 2000.

[SA97] Emin Gun Sirer and Prof Andrew. Measuring limits of fine-grained parallelism, January 24 1997.

[SFF+02] Oliverio J. Santana, Ayose Falcon, Enrique Fernandez, Pedro Medina, Alex Ramirez, and Mateo Valero. A comprehensive analysis of indirect branch prediction. In *Proceedings of the 4th International Symposium on High Performance Computing (ISHPC-IV)*, May 2002.

[SFR+02] Oliverio J. Santana, Ayose Falcon, Alex Ramirez, Josep L. Larriba-Pey, and Mateo Valero. Next stream prediction. Technical Report UPC-DAC-2002-15, Technical Report, April 2002.

[SGBE05] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: stack versus registers. In *ACM/SIGPLAN Conference on Virtual Execution Environments*, pages 153–163, Chicago, Illinois, June 2005. ACM Press.

[SJSM96] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud. Multiple-block ahead branch predictors. In *ASPLOS*, pages 116–127, 1996.

[SL84] Smith, A. J. and Lee, J. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, 1984.

[SM77] Peter Schulthess and Eduard Mumprecht. Reply to the case against stack-oriented instruction sets. *Computer Architecture News*, 6(5):24–27, December 1977.

[SM01] Sun-Microsystems. The Java Hotspot virtual machine. Technical report, Sun Microsystems Inc., 2001.

[SN05] James E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005.

[SÖG06] Yunhe Shi, Emre Özer, and David Gregg. Analyzing effects of trace cache configurations on the prediction of indirect branches. *The Journal of Instruction-Level Parallelism*, Volume 8, 2006.

[SPE98] SPEC. SPEC releases SPEC JVM98, first industry-standard benchmark for measuring Java virtual machine performance. Press Release, August 19 1998. http://www.spec.org/jvm98/press.html.

[SRLPV04] Oliverio J. Santana, Alex Ramirez, Josep L. Larriba-Pey, and Mateo Valero. A low-complexity fetch architecture for high-performance superscalar processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 1(2):220–245, June 2004.

[TSL+02] Frank Tip, Peter F. Sweeney, Chris Laffra, Aldo Eisma, and David Streeter. Practical extraction techniques for Java. *ACM Trans. Program. Lang. Syst.*, 24(6):625–666, 2002.

[USS97] Augustus K. Uht, Vijay Sindagi, and Sajee Somanathan. Branch effect reduction techniques. *IEEE Computer*, 30(5):71–81, 1997.

[VA04] Benjamin Vitale and Tarek S. Abdelrahman. Catenation and specialization for tcl virtual machine performance. In *IVME '04: Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators*, pages 42–50, New York, NY, USA, 2004. ACM Press.

[Ven99] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, pub-MCGRAW-HILL:adr, second edition, 1999.

[VRHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[WC99] C. Wongsiriprasert and P. Chongstitvatana. Performance comparison between two virtual machine interpreters : stack-based vs. register-based. In *Proc. of 3rd Annual National Symposium on Computational Science and Engineering*, pages 401–406, Bangkok, Thailand, 1999.

[WP97] Phil Winterbottom and Rob Pike. The design of the Inferno virtual machine. In *IEEE Compcon 97 Proceedings*, pages 241–244, San Jose, California, 1997.

[YP93] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *ISCA*, pages 257–266, 1993.