Caching and the Java Virtual Machine

by
M. Arthur Munson

A Thesis
Submitted in partial fulfillment of
the requirements for the Degree of Bachelor of Arts with Honors
in Computer Science

WILLIAMS COLLEGE
Williamstown, Massachusetts
May 14, 2001

Abstract

Most of the research done to improve the performance of Java Virtual Machines (JVM's) has focused on software implementations of the JVM specification. Very little consideration has been given to how Java programs interact with hardware resources and how hardware components can be used to improve Java performance. We generated and analyzed opcode and memory access traces for eight different Java benchmarks, finding that each Java opcode typically causes multiple memory accesses. We next investigated the effectiveness of unified and split caches at caching Java programs, using the performance of traditional compiled programs for a basis of comparison.

Motivated by the subdivision of memory by the JVM specification into heap memory, constant pool data, and operand stacks, we examined the possible benefits of adding small specialized caches to hold constant pool data and stack data. We found that constant pool accesses have very low locality of reference and that the addition of a constant cache is detrimental to cache performance for large main caches. The 24% of memory accesses made to operand stacks were found to be efficiently cached in a 64 byte cache. We simulated the cache performance of Java programs in the presence of registers by adding a register cache to unified and split caches. We found that given fair hardware resources, Java programs cached at least as well as traditional programs.

Acknowledgements

Roughly a year ago I received a call from Duane Bailey, Professor of Computer Science at Williams College. The conversation was relatively short:

Duane: I noticed that you didn't sign up for doing an honors thesis next year.

Me: Yeah, I decided that I didn't have a topic I liked enough to spend a year researching.

Duane: Why don't you come talk to me so I can try to talk you into doing an honors thesis?

Well, Duane, it worked. Thank you.

I would like to thank those who helped me in so many indispensable ways during this year. First, Sean Sandys for ploughing through my drafts and providing extremely useful feedback on clarity, quality of scholarship, and the abilities of his introductory programming students. Thanks also for continually lifting my spirits.

Thanks to Mary Bailey, the department's system administrator, for helping me not destroy my thesis despite my best efforts. Thanks to all the computer science faculty at Williams College for giving me the background necessary for independent research and for answering my questions at all times.

Thank you to Nate Foster '01 for keeping me company in the lab and suffering through a thesis like myself. Without him prodding me along by his example and helping me with the small details of putting a thesis together, I would never have stayed on track.

Thanks to Shimon Rura '03 and Chris Kelley '03 for answering my various LaTeX 2ε questions. Thanks to my friends and family for continuing to talk to me despite my many hours of isolation in a lab.

And once again, thanks to Duane Bailey: for roping me into this project, for helping me think about the issues at hand, for commenting on scholarship and writing styles, and for not abusing his root access to turn off my work station.

Contents

1 Introduction **1**

2 Background **3**

2.1 Overview of the Java Virtual Machine Architecture 3
2.1.1 Method Area 4
2.1.2 Runtime Memory Sections Specific to Each Thread 5
2.1.3 The Heap 6
2.1.4 Design Rationales of JVM 8
2.2 Cache Overview 8
2.2.1 Theory Behind Caching 8
2.2.2 Defining Characteristics of Caches 10
2.2.3 Reading from Cache 11
2.2.4 Replacing a Line of the Cache 12
2.2.5 Writing to Cache 12
2.2.6 Cache Misses 14
2.2.7 Advanced Caching 14
2.3 Related Java Work 16
2.3.1 JVM Execution Models 16
2.3.2 Java Bottlenecks 17
2.3.3 Profiling Java Programs and the JVM 19
2.3.4 Hardware JVM Implementations 19
2.4 Previous Cache Research 20
2.4.1 Benchmarks 20
2.4.2 Caches and OS 21
2.4.3 Hybrid and Adaptive Caches 21
2.5 Other Related Research 22
2.5.1 Garbage Collection 22
2.5.2 Exception Handling 22
2.5.3 Stack Caching 22

2.6 Conclusion 23

3 JVM Personality 24

3.1 Small Bytecode 24
3.2 Java Constants 27
3.3 Stack Architecture 27
3.3.1 The Operand Stack 28
3.3.2 Lack of Registers 29
3.4 Object Oriented Paradigm 29
3.5 Conclusions 30

4 Monitoring and Profiling the JVM 31

4.1 Interpretation vs. JIT 31
4.2 Instrumentation 33
4.2.1 Virtual Addresses 34
4.2.2 Maximum Trace Size 34
4.2.3 Bytecode Distributions 34
4.3 Benchmarks 37
4.3.1 Linpack 37
4.3.2 SPEC JVM98 37
4.4 Distributions of Memory Accesses 39
4.4.1 Memory Access Distribution by Opcode Classifications 39
4.4.2 Memory Access Distribution by Access Type 42
4.5 Conclusions 45

5 Cache Simulation 46

5.1 Simulation Setup 46
5.1.1 Data Free 46
5.1.2 Policies 48
5.1.3 Monitoring 48
5.1.4 Experiment Parameters 49
5.2 Results 49
5.2.1 Associativity 50
5.2.2 Unified Cache 55
5.2.3 Harvard Cache 57
5.2.4 Comparing Unified and Harvard Caches 60
5.3 Conclusions 66

6 New Caches for Increased Performance **67**
6.1 Register Cache . . . 69
6.1.1 Isolated Register Cache . . . 69
6.1.2 Unified Cache and Register Buffer (UNIREG) . . . 70
6.1.3 Harvard Cache and Register Buffer (HARREG) . . . 73
6.1.4 Summary . . . 75
6.2 Stack Cache . . . 76
6.2.1 Isolated Stack Cache . . . 76
6.2.2 Unified Cache and Stack Cache (STACKUNI) . . . 78
6.2.3 Harvard Cache and Stack Cache (STACKHARVARD) . . . 80
6.2.4 Summary . . . 82
6.3 Constant Cache . . . 82
6.3.1 Isolated Constant Cache . . . 84
6.3.2 Unified Cache and Constant Cache (CONSTUNI) . . . 85
6.3.3 Harvard Cache and Constant Cache (CONSTHARVARD) . . . 87
6.3.4 Summary . . . 87
6.4 Mix and Match Caches: An Evaluation . . . 89
6.4.1 Unified, Stack, and Register Caches (STACKUNIREG) . . . 89
6.4.2 Harvard, Stack, and Register Caches (STACKHARREG) . . . 92
6.4.3 Unified, Constant, and Register Caches (CONSTUNIREG) . . . 92
6.4.4 Harvard, Constant, and Register Caches (CONSTHARREG) . . . 95
6.4.5 Unified, Constant, and Stack Caches (CONSTSTACKUNI) . . . 95
6.4.6 Harvard, Constant, and Stack Caches (CONSTSTACKHARVARD) . . . 95
6.4.7 Harvard, Constant, Stack, and Register Caches (CONSTSTACKHARREG) . . . 99
6.4.8 The Bottom Line . . . 99
6.5 Conclusions . . . 101

7 Conclusions **103**
7.1 Summary . . . 103
7.2 Difficulties of Adding Specialized Caches . . . 104
7.3 Applicability Outside of Java . . . 105
7.4 Future Work . . . 105

A Rules of Thumb **107**

B Benchmark Distributions **108**

C Unified and Harvard Cache Results **115**

D Hybrid Cache Results 120

List of Figures

2.1 JVM Memory Organization 4
2.2 Java Call Frame Structure 7
2.3 Memory Hierarchy 9
2.4 Organization of a line of cache. 11
2.5 Cache Write and Write Miss Policies 13

3.1 Foo Source Code 25
3.2 Foo Java Bytecode and 68000 Assembly 26

4.1 Average Opcode and Memory Access Distributions 42

5.1 Simulator Architecture 47
5.2 Java Unified Cache Miss Rates by Associativity 51
5.3 Traditional Unified Cache Miss Rates by Associativity 52
5.4 General Method for Calculating Access Time 53
5.5 Method for Calculating Unified Cache Access Time 54
5.6 Speeds for Different Memory Layers 54
5.7 Java Unified Cache Performance 56
5.8 Java Harvard Cache Performance 58
5.9 Method for Calculating Harvard Cache Access Time 60
5.10 Java Unified and Harvard Cache Access Times 62
5.11 Time Spent Accessing Memory w/ Harvard Cache (Java) 64
5.12 Time Spent Accessing Memory w/ Harvard Cache (Traditional) 65

6.1 Icon Explanation 68
6.2 Method for Calculating Access Time for UNIREG Caches 72
6.3 Method for Calculating Access Time for HARREG Caches 73
6.4 Comparison of UNIREG/HARREG with UNIFIED/HARVARD 75
6.5 Method for Calculating Access Time for STACKUNI Caches 80
6.6 Method for Calculating Access Time for STACKHARVARD Caches 80
6.7 Comparison of STACKUNI and STACKHARVARD 83

6.8 Method for Calculating Access Time for CONSTUNI Caches 85
6.9 Method for Calculating Access Time for CONSTHARVARD Caches 87
6.10 Comparison of CONSTUNI and CONSTHARVARD 89
6.11 Method for Calculating Access Time for Hybrid Caches 90
6.12 Hybrid Unified Caches Average Access Time 99
6.13 Hybrid Harvard Caches Average Access Time 101

List of Tables

2.1 Summary of Cache Parameters 15

4.1 Average Opcode Distributions 35
4.2 Java Opcode Distributions #1 35
4.3 Java Opcode Distributions #2 36
4.4 Java Opcode Distributions #3 36
4.5 Distributions for mtrt 40
4.6 Average Memory Access Distributions, by Opcodes 41
4.7 Memory Accesses by Type 43
4.8 Data Memory Accesses by Subtype 44

5.1 Experiment Parameters for Unified and Harvard Caches 50
5.2 Memory Access Distribution for Javac (JDK1.2) 53
5.3 Data Memory Access Distribution for Javac (JDK1.2) 53
5.4 Relative Access Time Ratios for Associativity 55
5.5 Average Cycles per Memory Access by Associativity (Java) 57
5.6 Average Cycles per Memory Access by Associativity (Traditional) 59
5.7 Average Cycles per Memory Access for Unified and Harvard Caches 61

6.1 Average Register Cache Miss Rates 70
6.2 UNIREG Cache Miss Rates and Access Times 71
6.3 HARREG Cache Miss Rates and Access Times 74
6.4 Average Stack Cache Miss Rates and Access Times 77
6.5 Maximum Stack Lines Used, by Benchmark 78
6.6 STACKUNI Cache Miss Rates and Access Times 79
6.7 STACKHARVARD Cache Miss Rates and Access Times 81
6.8 Average Constant Cache Miss Rates and Access Times 84
6.9 CONSTUNI Cache Miss Rates and Access Times 86
6.10 CONSTHARVARD Cache Miss Rates and Access Times 88
6.11 STACKUNIREG Cache Miss Rates and Access Times 91

6.12 STACKHARREG Cache Miss Rates and Access Times 93
6.13 CONSTUNIREG Cache Miss Rates and Access Times 94
6.14 CONSTHARREG Cache Miss Rates and Access Times 96
6.15 CONSTSTACKUNI Cache Miss Rates and Access Times 97
6.16 CONSTSTACKHARVARD Cache Miss Rates and Access Times 98
6.17 CONSTSTACKHARVARD Cache Miss Rates and Access Times 100

B.1 Total Memory Access Counts 108
B.2 Java Opcode Classifications 109
B.3 Distributions for compress 110
B.4 Distributions for db 110
B.5 Distributions for `Empty` 111
B.6 Distributions for jack 111
B.7 Distributions for javac 112
B.8 Distributions for jess 112
B.9 Distributions for linpack 113
B.10 Distributions for mpegaudio 113
B.11 Distributions for mtrt 114

C.1 Unified Cache Miss Rates 115
C.2 Harvard Cache Miss Rates #1 116
C.3 Harvard Cache Miss Rates #2 117
C.4 Harvard Cache Miss Rates #3 118
C.5 Harvard Cache Miss Rates #4 119

D.1 Register Cache Miss Rates 120
D.2 Stack Cache Miss Rates #1 120
D.3 Stack Cache Miss Rates #2 121
D.4 Constant Cache Miss Rates 121
D.5 compress Hybrid Cache Results 122
D.6 db Hybrid Cache Results 129
D.7 jack Hybrid Cache Results 136
D.8 javac Hybrid Cache Results 143
D.9 jess Hybrid Cache Results 150
D.10 linpack Hybrid Cache Results 157
D.11 mpegaudio Hybrid Cache Results 164
D.12 mtrt Hybrid Cache Results 171

Chapter 1

Introduction

"Our research department forecasts that it is very likely that Java will replace C++ over time. And I strongly agree. Once performance issues are worked out, C++ will not necessarily be better than Java."
—Edgar Saadi, April 1997[1]

The Java programming language has grown in popularity to become one of the more important development tools on the market today. Running on top of a virtual machine, Java bytecode can be written once and run everywhere a virtual machine is implemented, saving developers hours of time. Although hardware implementations of the Java Virtual Machine (JVM) specification exist (see PicoJava for example [22, 25]), the majority of JVM's are implemented in software. These JVM's translate compiled Java bytecode to the native instructions of the underlying platform.

When the JVM is implemented in software it is essentially a program that manipulates values. Some of these values are the operation codes (opcodes) of the Java program that is running in the JVM—instructions that have a high locality and that will never be written to in memory. Unfortunately, the computer on which the JVM is running is unaware of these opcodes and sees all of the values that the JVM manipulates as data values. This could be sidestepped if the Java language included special instructions for directly manipulating registers and cache—ways through which to interact with the hardware. This, however, would force the language to favor a particular computer architecture, thereby losing the platform independence that has made it so powerful in today's market. Indeed, Java treats all of memory like a black box that answers requests for data and instructions; the JVM specification makes no assumptions about the internal organization of memory.

At a lower level, current processors are much faster than the speed of off chip memory.

[1] Quoted from an online letter column found at: http://www.unixinsider.com/unixinsideronline/swol-04-1997/swol-04-career_p.html

Computers built today are designed with a memory cache to narrow the gap between the speed of the processor and the latency of off chip memory. Almost as a rule, a contemporary cache is divided into two halves—one to hold instructions and one to hold data. There are two main motivations for this division. First, instructions are never modified during the execution of a well behaved program; therefore, when an instruction needs to be replaced in the cache, there is no chance that the instruction needs to be written back to memory. The design of the instruction cache can be simpler than the design of the data cache. More importantly, instruction memory accesses have a much higher locality than data memory accesses and are often very sequential in nature. In a unified cache, data can knock instructions out of the cache, causing problems when the replaced instruction is soon to be executed or when the instruction is part of a currently executing loop. A divided cache further enhances the benefits of a unified cache by guaranteeing that instructions will be in cache well over 90% of the time.

The Java instruction set can be divided into several classifications, particularly with respect to how they might manipulate memory. Java opcodes are very specific about the operands (data) they manipulate. Certain instructions must access certain kinds of memory in certain ways. For example, specific instructions exist for loading constant values from a constant pool. In this case, the data is read only, as there is no chance that the value will be changed. Similarly, all computations are performed using an operand stack for scratch space. The locality of accesses to the top of this stack will be very high since all computations are constantly accessing the memory values there.

This work examines the nature of Java bytecode with respect to how it interacts with memory. As implied above, memory accesses made by an executing Java program can be classified into categories. Such a grouping suggests the addition of special purpose caches to improve Java application performance. Different classes of memory accesses would be segregated in different caches just as instruction accesses and data accesses are separated under a divided cache architecture. These additions would represent non-trivial computer architecture modifications and cannot currently be supported on existing machines. We take the position of a hypothetical inquiry: what gains could be made if such support existed?

Chapter 2 presents a background fundamental to this work. The organization of the Java Virtual Machine and the basics of caching are discussed. Related research surrounding the JVM and caching is summarized. Chapter 3 looks at the differences between the JVM and a traditional platform architecture. The impact of these differences on memory accesses is discussed. Chapter 4 explains the procedure used to obtain memory access and executed opcode traces. The memory profile revealed by these traces is compared to the memory profile for traditional compiled programs. Chapter 5 compares the cache performance of Java programs with the cache performance of traditional compiled programs for unified and split cache configurations. Chapter 6 presents the results of experiments in which specialized caches were added to the memory hierarchy. Chapter 7 addresses the obstacles to adding specialized caches and briefly discusses further work that remains to be done.

Chapter 2

Background

"[T]he performance of numerically intensive Java programs . . . can be as low as 1 percent of the performance of FORTRAN programs . . . a 100-fold performance slowdown is unacceptable."
—Moreira *et al.* [23]

Before presenting our results we need to provide a background for the investigation. The two primary topics of interest are the Java Virtual Machine specification and how caching offers significant performance improvements for executing programs. Before pressing forward in Chapters 5 and 6 to look at how caches interact with current Java virtual machine implementations and then how other cache configurations might interact with future virtual machines, it is important to first understand the basics of each component separately. Sections 2.3 through 2.5 cover related previous work concerning Java and its virtual machine implementations, past methods for examining the effectiveness of caches, experimental cache designs, and performance boosting optimizations for exception handling, garbage collection, and virtual stack machines.

2.1 Overview of the Java Virtual Machine Architecture

The Java Virtual Machine (JVM) specification is abstract by design. This is to allow implementors a great deal of freedom in how they implement the JVM to serve specialized purposes. It also makes the implementation of JVM's for different individual platforms feasible. Nonetheless, the specification requires certain information to be available to the JVM. This work is interested in the organization of the runtime memory area. This area contains three subsections [30]:

1. A method area containing the loaded types.

2. Runtime memory sections specific to each thread:
   (a) All of the program counter (PC) registers for the threads running in the program,
   (b) A Java call frame stack for each thread in the program that stores the state of method invocations in the thread, and
   (c) A native method stack for each thread in the program.
3. A heap which provides all dynamic memory needed by the running program.

Figure 2.1 depicts this organization. Sections 2.1.1 through 2.1.3 treat these areas in some detail.

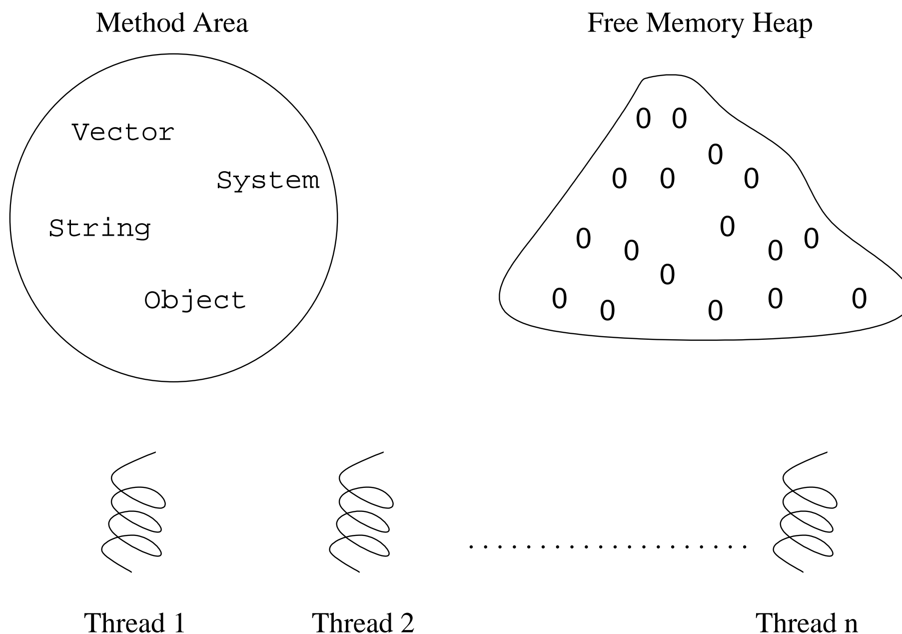


Figure 2.1: Overview of the JVM specification's organization of memory. The *method area* contains Java classes (*types*) loaded thus far by the JVM. A *free memory heap* holds zeroed out memory available for allocation. Threads of execution are separate from each other.

2.1.1 Method Area

All of the non-primitive *types* used by the JVM and the executing program must be loaded by the JVM before they can be used. We will drop the non-primitive qualifier and use the term *type* to refer to classes and interfaces. Type information is loaded into the method area, where it is accessed by all threads in the JVM. As a result, the method area's data structures

needs to be thread safe. Although these data structures exclusively contain read only data, types can be loaded dynamically in Java. If the method area data structures were not thread safe, two threads could try to simultaneously load the same (previously unloaded) type. Each type in the method area has to have the following information:

1. The fully qualified name for the type.
2. The fully qualified name for the immediate superclass of the type.
3. Whether or not the type is an interface or a class.
4. The modifiers for the type (public, abstract, final).
5. An ordered list of the fully qualified names of the direct superinterfaces of the type.

A *fully qualified name* is the complete path to the class, interface, method, or instance variable. It consists of the package followed by the class or interface name (such as `java.lang.Object`). For methods and instance variables the class/interface name is succeeded by the method or variable name. A superinterface is an interface implemented by the type.

Additionally each type has its own constant pool containing both literal constants and symbolic references to types, fields, and methods. The constant pool is accessed whenever an instance variable or a method is resolved. Class variables that are non-final are stored within the method area but not within the constant pool. The method area also contains information on all the fields in the type, specifying their names, types, modifiers, and the order in which they were declared. Information about methods specifying their names, return types (if any), parameter counts, and modifiers is also required. Non-abstract, non-native methods additionally keep an exception table, the sizes of the operand stack and of the local variable section (for their stack frames – see Section 2.1.2), and their bytecodes in the method area. Finally, all types include a reference to the `ClassLoader` that loaded them and a reference to the `Class` object used to construct the type [30].

2.1.2 Runtime Memory Sections Specific to Each Thread

Each thread in the JVM has its own PC register, Java call frame stack, and native method stack. The PC register points to the next instruction to be executed for the thread. The native method stack is entirely platform dependent. When a native method call is made, the method invocation frame is placed on the call frame stack; however, the frame is a place holder that points to the native method stack. The native code execution occurs on the native method stack.

The call frame stack is the heart of the JVM, containing stack frames for each method invocation made in the thread. The JVM itself can only push and pop the stack frames. All

other manipulation of the Java stack is done by the executing thread [30]. For example, the pushing and popping of values onto the operand stack contained within a call frame (see below) is handled by the thread.

To allocate the memory for the call frame the JVM needs to know how much space will be required for the operand stack and the local variable section (hence the storing of the sizes of these spaces within the method area). With respect to the operand stack this requires knowing the maximum height attained by the stack during the execution of its method.

There are two ways that the call frames can be allocated. In the first scheme they are allocated from the global free memory heap. In the second, the call frames are allocated from a contiguous stack. Under this scheme the size of the operand stack does not need to be known since the operand stack is always located at the top of the call frame; the currently active method frame will consequently have its operand stack at the top of the call frame stack where it can grow freely.

While a stack frame is executing the JVM keeps track of the current class (the class containing the executing method) and the current constant pool. Each stack frame is comprised of three sections:

1. Frame data that maintains a reference to the constant pool and contains information needed for both normal and abnormal method completion,
2. A local variable section which stores both the local variables for the method and the parameters for the method, and
3. An operand stack.

Figure 2.2 shows the organization for an example call frame.

Since the Java instruction set is built around a stack machine, all computations are performed on the operand stack. Most instructions either load data onto the stack, store data popped from the top of the stack, or perform a computation using stack-based operands [30].

2.1.3 The Heap

Like the method area, the heap is used by all of the threads in the JVM. Although not explicitly required by the JVM specification, most JVM implementations have a garbage collector that maintains the heap by freeing memory no longer referenced by any objects [30].

The most straightforward approach to garbage collection is the mark/sweep algorithm. The garbage collector looks through all of the program threads' call frame stacks. Each live object reference is placed into a mark stack for later marking. Each object in the mark stack is taken off and marked as live. Additionally, every object referenced by the marked object is placed into the mark stack for later processing. When all of the run-time stacks have been

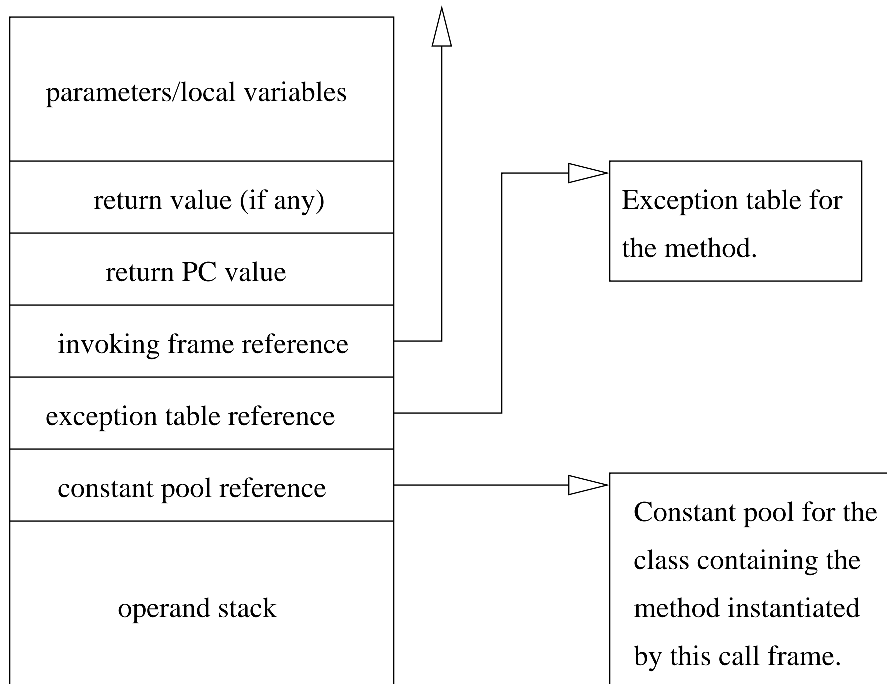


Figure 2.2: Java call frame structure. The call frame stack is assumed to be growing down in this picture, placing the operand stack at the top of the stack when the call frame is active.

traversed and the mark stack is empty, all of the live objects have been marked. Then the garbage collector sweeps through the heap freeing unmarked objects and returning them to the free store. Finally, if the heap is seriously fragmented the garbage collector compacts the heap, moving all of the free space to one end of the heap [8].

Since the mark and sweep collector traverses the thread run-time stacks and potentially moves items around the heap, many implementations of the algorithm suspend thread executions during garbage collection. For large programs, this results in significant delays. Two improvements have been made to this design. The first is to perform garbage collection concurrently with JVM worker threads that are unrelated to the execution of the program, allowing the JVM to move forward during garbage collection [7]. The second makes use of the observation that more recently allocated objects are more likely to be unreferenced sooner, since many of these are temporary variables. “Studies show that, in the time it takes to allocate 32 Kbytes of memory off the heap, over 90 percent of the newly created objects will already be dead [22].” Thus, it is worthwhile for the collector to frequently sweep through the most recent generation of objects (those most recently allocated) and to only

occasionally collect the entire heap [7].

2.1.4 Design Rationales of JVM

Centering the JVM around stack operations reduces performance since additional operations are needed to push and pop values to and from the operand stack. Register machines do not require the extra operations used to manipulate the stack since register machine instructions refer to registers. The decision to use a stack architecture was made to make the JVM more platform independent. By not accessing any registers, the JVM can easily be implemented as on platforms that have few registers as those that have dozens of registers. Another motivation for a stack-centered instruction set was that many compilers produce stack-based intermediate files. By using a stack-centered instruction set, compiled Java bytecode is very similar in structure to object files generated by compilers of other languages. Thus, the JVM is like a linker, and can benefit from known linker optimizations to enhance bytecode execution. Finally, the Java language was designed with the network in mind. To this end, Java class files are small and compact and can be easily transmitted across the network. Almost all of the opcodes specify what types they expect, making bytecode verification feasible and network transmission safe [30].

2.2 Cache Overview

For the past several years increases in computer processor speeds have outpaced increases in memory speeds. Today, CPUs execute instructions far more quickly than memory can provide the required data. To narrow this gap in speeds, computer architectures now include several level of caches—small, expensive, high speed memory that can better fill the requests of the processor for data. These caches sit between the processor and main memory (see Figure 2.3), usually residing on the processor chip itself (at least for the first level cache—see Section 2.2.7). Since the cache is part of the chip, the time for signal propagation is much shorter than for accessing main memory [26]. In addition, the cache is implemented in a more expensive technology than main memory; using this technology for main memory would be prohibitively expensive. Furthermore, the access speed for a cache decreases as the size of the cache increases.

2.2.1 Theory Behind Caching

Despite having only a fraction of the capacity of main memory, *caches are often able to fill over 90% of the processor's requests for data.* If memory addresses were accessed with a uniform distribution, the cache hit rate (the percentage of the time that the requested address is resident in the cache) could not be better than the ratio of the cache size to the size of main memory. However, programs exhibit a *principle of locality of reference*:

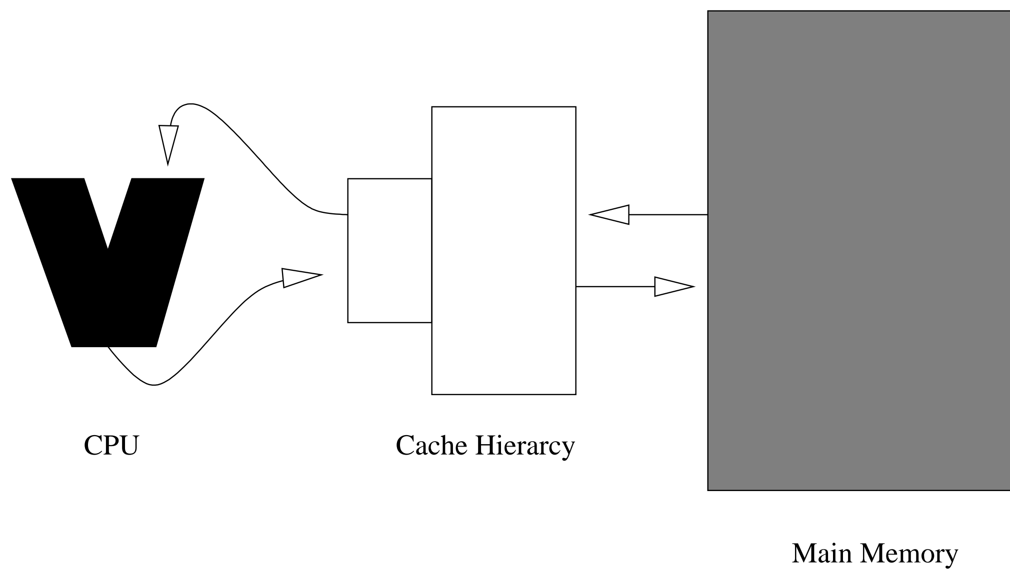


Figure 2.3: Simplified memory hierarchy. The CPU writes values to the cache hierarchy, which in turn writes those values to main memory. When retrieving a value from memory, the CPU looks in the cache hierarchy for the value. The hierarchy will then read the value from main memory if needed.

> Most programs spend 90% of their execution time in 10% of their source code [26].

In other words, memory that has been accessed recently is more likely to be used again soon. This behavior allows caches to cover the great majority of memory accesses when fetching the next instruction to execute.

To be effective, a cache does not need to store all of the memory used by the program. Only the most recently used portion of the *working set* needs to be held in the cache.

> The *working set* of a program is the set of memory values frequently accessed by the program during its execution. It does not include the memory values used solely during initialization and/or shut down.

When the address is not in the cache, it is loaded from main memory both to the cache and to the processor. The principle of locality of reference tells us that this address is likely to be used again in the near future; the cost of the cache miss and of fetching the address is offset by the likelihood of shorter fetching times later in the program's execution [26].

Memory accesses of data do not show as much locality as the accessing of instructions. Since data tends not to be accessed in the linear fashion of instructions, the retrieval of data addresses is more distributed throughout memory. Nonetheless, data accesses do have both temporal and spatial locality [26]. Temporal locality is fairly intuitive once the connection is made that accessing a data address is caused by accessing a variable in a program. Often variables in a program are repeatedly accessed during a short period of time, for example during a loop or within a function.

Spatial locality is created both by how programs are structured and by how compilers layout out programs in memory. A well-structured program subdivides its tasks into multiple functions with variables local to the functions. As a function executes, it will be primarily accessing the variables local to it, all of which are close to each other in memory. When a compiler lays out a program in memory, it places all of the variables next to each other. Thus, accesses to a variable in a program are restricted to certain range of memory. The ramifications of spatial locality on caching success are less intuitive than those of temporal locality. We will consider this topic later.

2.2.2 Defining Characteristics of Caches

As an extension of memory, a cache needs to be able to handle both reads and writes to memory addresses. In addition to these tasks, a cache has several characteristics that distinguish it from main memory and from other cache implementations:

1. How the cache determines if a given memory address resides in cache. This varies from cache to cache depending on the *associativity* of the cache. Associativity is the number of places in a cache where an address can reside.

2. The replacement strategy for replacing a *line* of the cache when an address must be fetched into an already full cache. The line of the cache is the indexable block of the cache.

3. When writing to an address, a cache can either update the value in main memory (*write through*) or wait until the line is replaced before writing to main memory (*write back*).

We will now look at each of these characteristics in more detail.

2.2.3 Reading from Cache

The location of an address in the cache is found using some mapping. Usually this function mods the memory block address (the address truncated to cut off the block offset) by the number of lines in the cache. This yields an index into the cache, mapping multiple memory blocks to the same *line*. Each line of cache holds a memory block's worth of addresses, indexable by the address offset (see Figure 2.4). To determine if the cache line contains the

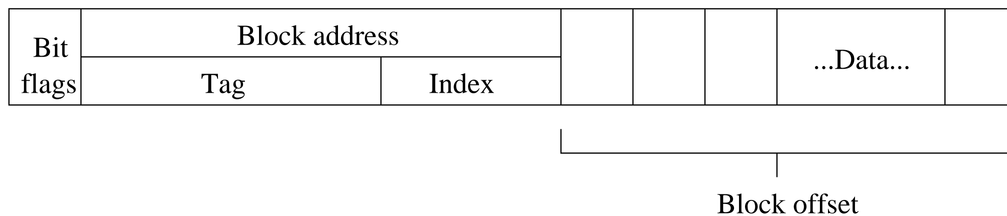


Figure 2.4: Organization of a line of cache.

correct memory block for a given memory address, the cache line also stores the address tag against which the cache compares the address tag of the memory address in question. In addition, there is a *valid bit* which, if set, says that the line is a valid block of memory. When the computer first starts up all of the cache lines have their valid bits set to 0. On a match with a valid bit, the offset of the memory address indexes into the cache line to retrieve the appropriate word of memory [26]. If the address is not in the cache (if either the tag does not match or the valid bit is not set), the address is fetched from main memory and placed in the cache (see below for the most common ways of determining which line of the cache to replace), setting the valid bit for the line to 1.

Since multiple memory blocks map to the same location in the cache, conflicts arise. To avoid some of these conflicts some caches have the address lookup function map to sets of cache lines. These caches are called *set-associative caches*, or just *associative caches*. Typical associative caches are two-way, four-way, and eight-way associative caches; the

number indicates how many lines comprise each set. Once a set of lines is indexed, the cache matches all of the address tags in parallel to determine if the memory address is resident in any of the lines. In general, the higher the associativity of the cache, the fewer *conflict misses* (see Section 2.2.6). However, the gains of each additional level of associativity decrease. Thus, an eight-way associative cache, with a memory block mapping to eight different cache lines, performs nearly as well as a fully associative cache. Still, direct mapped caches (or one-way set-associative caches) are often used due to their simplicity and speed. With a direct mapped cache, the memory address value can be read while the address tag match is performed. If the match succeeds the read has been performed more quickly; if not, the value read is ignored and there is no loss in time [26]. In addition, the higher the associativity of the cache the more complex the hardware used, the more expensive the cache, and the slower the access to the cache [13].

2.2.4 Replacing a Line of the Cache

Replacing a line of a direct mapped cache is straight forward, as there is only one choice for which line to replace. With a set-associative cache, there are either two, four, or eight choices for which line to replace with the incoming memory block. If a given set of lines is at capacity, that is, if all of the lines in the set have the valid bit set, one of the lines needs to be selected for replacement. The two most commonly used algorithms for deciding which line to replace are Least Recently Used (LRU) and Random [26]. LRU chooses the cache line that was used the least recently for replacement, following the principle of temporal locality (that which was accessed most recently is more likely to be accessed again). In practice, LRU is expensive to implement and is only approximated. A random selection algorithm chooses a cache line at random to replace. Either way, the incoming memory block is read into the selected line and the line's valid bit is set.

2.2.5 Writing to Cache

The two policies for writing to an address in the cache are write through and write back. Each policy has its advantages. With a write through policy, the value at an address is updated in the cache and in the lower level of memory. In a write back policy, the value at the address is updated in the cache and not in the next lower level of memory. Instead, a dirty bit is set for the cache line to indicate that the line does not match the line in main memory. A read miss in the future that causes the dirty line to be replaced will need to write the line out to main memory before replacing the line [26]. Figure 2.5 illustrates this difference.

Write through boasts a simpler scheme: a read miss never needs to write back to memory before replacing the line. The next lower level of memory is always consistent with the cache. Write back, on the other hand, allows a faster write to the cache (since the write back

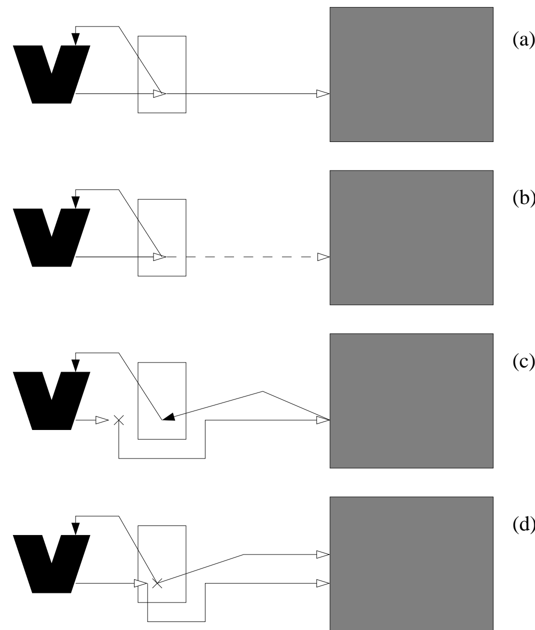


Figure 2.5: Different cache write and write miss policies. White arrows show the path taken by a write from the CPU (the black **V**) through or around the cache (white box) to main memory (grey box). The path taken by a subsequent memory read of the updated value is shown by a black arrow. Picture (a) shows a write hit on a cache using write through. The memory value is updated in the cache and in main memory. Picture (b) shows a write hit on a write back cache; the memory value is only updated in the cache and the modified cache line is marked as *dirty*. When that line is later replaced, it is written to main memory (dashed arrow). Picture (c) shows a write miss for a cache using a write around write miss policy. A subsequent read of that value must fetch the value into the cache. Picture (d) shows a write miss for a cache using a write fetch write miss policy. The miss replaces a line (causing it to be written back if a write back policy is used and the replaced line is dirty), reads the correct line from memory to the cache (not shown), and updates the value in both the cache and main memory. A subsequent read will find the value in the cache.

to the next lower level is put off). Since data accesses follow the principal of spatial locality, addresses close together are likely to be updated over a short period of time. This means that multiple words in a line of cache could be dirty or that any given word could have been changed multiple times. Waiting to write back a dirty line until absolutely necessary avoids frequent writes to individual addresses. The whole line can be written back to memory in one block, reducing the band width taken up by memory writes [26].

Similarly, there are two policies followed when a write to an address results in a cache miss: Fetch on Write, or Write Around. With fetch on miss, the new value is loaded into the cache, replacing a line of the cache. Write around updates the value in the lower level of memory but does not load the address into the cache. Typically fetch on write is used with write back caches and write around is used with write through caches [26].

2.2.6 Cache Misses

As indicated above, the address requested by the CPU is not always resident in the cache, resulting in a cache miss. These misses fall into three categories:

1. *Compulsory* misses that are inevitable the first time a given memory address is accessed.

2. *Capacity* misses resulting from a full cache. These indicate that the currently running program does not fit into the cache.

3. *Conflict* misses caused by multiple address blocks mapping to the same line of cache.

The strategy to reducing compulsory misses is to make larger memory blocks, thereby increasing the likelihood that an address has already been loaded as the side effect of loading another address in the same block and taking advantage of spatial locality. At the same time this forces the cache to be larger (increasing it's cost, physical size on the chip, and the time needed to locate an address in the cache) or causes an increase in conflict misses. Capacity misses can be reduced by increasing the size of the cache, with the corresponding draw backs listed above. As mentioned in Section 2.2.3, conflict misses are reduced by caches with higher associativity. A fully associative cache suffers no conflict misses [26]. The effects of changing cache parameters are summarized in Table 2.1.

Determining the type of a cache miss is sometimes difficult. To distinguish compulsory misses, a history of which memory addresses have been accessed is required. Separating capacity from conflict misses is trickier. Chapter 5 briefly discusses a way to classify them.

2.2.7 Advanced Caching

One of the limitations of a small, fast cache is that it has a very limited capacity that results in a relatively high number of conflict and capacity misses. The cost of these misses is also

Summary of Cache Parameters

	compulsory misses	**capacity misses**	**conflict misses**	**access time**	**space used**	**cache cost**
↑ **cache size**	NC	↓	NC	↑	↑	↑
↑ **associativity**	NC	NC	↓	↑	NC	↑
↑ **line size**	↓	NC	↑	NC	NC	NC

Table 2.1: Summary of cache parameters and how varying them changes cache behavior. NC indicates no change occurs.

high, since the address requested must be fetched from the relatively slow main memory that resides off-chip. A solution is to add a second larger cache between the level one cache and main memory that acts as a storage back up for the level one cache. This second level cache is much faster than main memory, but slower than the level one cache. The increased capacity allows the level one cache to first look in the level two cache for a requested address, limiting the cost of a level one cache miss. This memory hierarchy eliminates some conflict misses and most capacity misses [26]. Liptay reports that the System/360 Model 85 with caching has over a 95% chance of finding a requested address in the cache. Compared to the ideal system, where main memory is as fast as the cache, the cached System/360 is 81% efficient [21].

Another observation made from empirical studies is that about 75% of memory references are to retrieve instructions. Furthermore, instructions have a much greater locality of reference than data. Combine this with the fact that load and store instructions require both an instruction from memory and a data access and quickly a single, unified cache becomes a bottleneck for accessing memory. Data can knock out instructions that will be executed in the near future. A *split cache* hierarchy, or a *Harvard cache*, can be employed in which instructions and data are cached in separate caches. This allows an instruction and a data word to be loaded from the cache simultaneously, doubling the bandwidth into the CPU. Instruction accesses have a lower miss rate because data accesses no longer collide with instruction lines. Finally, optimizations can be made for each individual cache. For example, the instruction cache contains read only values; there will never be any writes made to the instruction cache and the cache can be optimized for quick reading [26].

Clearly a cache hierarchy can take many different forms. There can be a single unified level one cache, a split level one cache, or a level one cache backed up by one or more larger off-chip caches. Each of these caches can have different degrees of associativity. This variety is hidden from executing programs by the CPU; *the cache hierarchy is transparent*. Writing a program to handle the numerous types of caching configurations would be time consuming and error prone; furthermore, future cache hierarchies would require mainte-

nance of this complicated code. For these reasons caching protocols are implemented on the chip.

From an executing program's point of view, the CPU retrieves values from memory. This makes caching hierarchies *substitutable* and *malleable* (from the program point of view).

2.3 Related Java Work

A lot of research has been conducted concerning the JVM. IBM, for example, published an entire journal worth of Java research in January, 2000 [15]. The majority of research has focused on improving the execution speed using various forms of compilation instead of interpretation. A good deal of work has also looked at what bottlenecks exist in the JVM and how to reduce or eliminate their impact. Kazi *et al.* present an excellent survey of current Java technology and efforts to improve JVM performance [17].

2.3.1 JVM Execution Models

Initially all JVM's were implemented as simple to implement interpreters. These JVM's translated Java bytecode into the underlying machine code for the platform. Because this translation needs to occur on every single instruction, interpreted Java programs run much more slowly than more traditional compiled programs. While interpreters continue to serve as JVM's in small devices, Just-in-Time (JIT) compilers have become the predominant JVM implementation available for desktop and server machines. Since Java programs cannot be compiled ahead of time (if they were, then they would lose their platform independence and network mobility), a JVM needs to compile the bytecode to native code when the program is run. Unfortunately, compiling is usually a long, processor intensive procedure. A JIT compiler only compiles bytecodes as they are executed, amortizing the costs of compilation over the execution of the program. This can be done in the background while an interpreter executes the program, removing the initial wait for compiled code to arrive [28].

Still, there can be quite a lag while the interpreter runs and the compiler produces native code, especially since the majority of a program's code is rarely executed. Following the principle of locality, most programs spend 90% of their execution time in 10% of the code. Some newer JVM's take advantage of this by running in interpreted mode and noting which code is executed frequently. These hot spots are then heavily compiled and optimized, making the best use of the compiler's time. Other JVM's compile everything as quickly as possible, sacrificing the optimizations that could be made in favor of compiling quickly. Then they locate the execution hot spots and adaptively recompile those sections with heavy optimizations [6].

Where portability is not a concern Java bytecode can be directly compiled to native

machine code ahead of time. *Bytecode-to-source* translating JVM's convert Java bytecode to a high-level intermediate language (such as C). An existing compiler for the intermediate language then generates machine code [17].

2.3.2 Java Bottlenecks

A lot of work has been done in locating the Java performance bottlenecks both for client and server JVM's. Several studies cited above describe this sort of work. The San Francisco project [5], started to give businesses a library framework for large Java applications, found that Java programs suffered from long code paths, immutable strings, excessive and expensive exception use, and unnecessary synchronization. Often the long code paths were hidden in the standard Java libraries. As an example of unnecessary synchronization and inefficient string implementation, Christ *et al.* report that string concatenation causes an implicit synchronized `StringBuffer` to be created by the Java compiler. Since this buffer is temporary and will never be shared between multiple threads, there is no need for the synchronization. Similarly, all of the data structures in the Java libraries are synchronized since there exists the possibility of multiple threads accessing them. When these structures are used within synchronized code, multiple, unnecessary levels of synchronization restrict application performance.

Similar results were found by Gu in the process of improving one of IBM's JVM's [12]. Other bottleneck areas include monitor contention, memory allocation, run time type resolution, and the AWT graphics. All objects, by specification, have a monitor associated with them to provide for the possibility of needing to synchronize a given object. If this required monitor is naïvely implemented, the performance cost can be impressive. Dimpsey *et al.* found in their work to build a viable Java server that synchronization can account for 19% of application running time [8]. When allocating memory the JVM has to locate a large enough piece of memory to contain the new object. A naïve memory manager searches through the heap looking for an available chunk large enough to hold the requested object. Gu proposes a better approach in which some free chunks of memory are stored in bins of specific sizes, ranging from only a few bytes to 512 bytes. When an object of a specific size is requested, the manager can go directly to the correctly sized bin to retrieve the needed memory. The upper size of 512 bytes was chosen since most object allocations are less than this size. For larger objects the manager must search through the entire heap for a large enough block of memory.

Another bottleneck with memory allocation that is related to monitor contention is reported by Dimpsey. Each instance of a JVM has a single heap from which all memory for the executing program is allocated, even in a multi-threaded program. As such, a heap lock is necessary to ensure that the same piece of memory is not allocated simultaneously to two different threads. Java's object oriented design encourages frequent object allocation; thus, the heap lock is one of the most contended locks in the JVM. This situation is helped by the

free lists mentioned by Gu [12], since the heap lock needs to be held for a shorter period of time. Dimpsey also proposes adding Thread Local Heaps (TLH). Despite the name, a TLH is simply a 1024 byte block of free memory to which a thread has exclusive access. If a thread cannot allocate the needed memory from its TLH then a new TLH is given to the thread, provided it is not simply a problem with allocating a very large object.

Gu reports that performance bottlenecks lie hidden within the Java class libraries. He gives the example of `String` class methods being called very frequently, and being inefficiently implemented. Since these methods are called not by the user program (instead being called internally to the libraries), the programmer has no control over avoiding costly methods or avoiding unnecessary repetitions of calls.

Other bottlenecks lie within the JVM itself. Run time type resolution is often needlessly performed twice. Consider the following rather common code:

```
if(someObject instanceof SomeClass){
    SomeClass a = (SomeClass)someObject;
}
```

This style of code can nearly always be found in overridden `equals()` methods. When the code is compiled to Java bytecode, an implicit type check is made when the cast occurs so that the JVM can determine whether or not to throw a `ClassCastException`. This check is clearly extraneous since the programmer has already explicitly forced the JVM to look up the type of `someObject`. Unless the bytecode compiler is intelligent this information is then discarded and then immediately looked up again.

Another study from IBM on using Java to do high-performance numerical computations found other bottlenecks caused by the Java language specification [23]. Specifically, every array access, both read and write, requires an array bounds check to determine whether or not the access causes an `ArrayIndexOutOfBoundsException` to be thrown. Furthermore, for arrays that contain objects accesses also force a check for a `NullPointerException`. These constraints particularly hamper scientific numerical computations which rely heavily on multi-dimensional arrays. Since each dimension of the array can have variable length arrays, the bounds checking cannot be compiled out into a single range check. Furthermore, accesses cannot be computed as a simple offset as they can be in FORTRAN 90 (where a given dimension has a set length); instead, each dimension must be dereferenced in turn, with the appropriate bounds and null pointer checks. Other obstacles to Java numerical computing cited by the authors include the lack of lightweight classes and objects that can be treated like the primitive types, the non-existent support for efficient complex arithmetic, and the inability of a JVM to access specialized floating point hardware of the underlying machine. Taken together, numerical benchmarks written in Java showed a 100-fold performance slowdown when measured against FORTRAN 90. Most of these problems, however, are not insurmountable. The authors developed a new Array package that provides

FORTRAN-like arrays and modified their JVM to eliminate extraneous bounds checking. They also developed a complex number package and made further JVM modifications to support complex arithmetic efficiently.

2.3.3 Profiling Java Programs and the JVM

A large amount of work has looked at measuring the performance of Java programs. Viswanathan and Liang present an addition to the built in JVM profiler to provide focused profiling. The profiling centers around a group of call back functions that enable and disable specific profiling for different sections of executing code. Functionality for the profiler includes stack traces, heap dumps, and the checking of thread status. An interactive front end can be added that does not impact the executing program since it runs in a different process from the JVM [31]. Kazi *et al.* present another visual profiling tool aimed at client/server profiling that can trace remote method calls [18].

Another important aspect to measuring performance is the use of suitable benchmarks for testing. Baylor *et al.* address the problem of a lack of informative benchmarks for Java servers. Existing Java microbenchmarks like SPEC JVM98 are not suited to the evaluation of large scale server applications. They present their own microbenchmark suite aimed at the operating system services provided by the JVM, including measuring hardware events such as cache misses and instruction counts [4].

The ability to compare performance evaluations is equally important as measuring performance. This desire led to the use of benchmarks that can be used across different platforms. Following in the same path, another project from IBM presents a unified arcflow model for performance analysis in the Java environment. The standardization of performance metric representations and methodology is designed to allow easier comparisons between different systems, as well as facilitating switches from one kind of analysis to another (such as memory analysis to delay analysis) [1]. The Jalapeño virtual machine has incorporated a good deal of these research efforts [2].

2.3.4 Hardware JVM Implementations

Although this study is interested in the interaction between a software implementation of the JVM and the underlying hardware, it is nevertheless instructive to look at how the JVM can be implemented in hardware. The benefits of a hardware implementation are readily apparent; without the cost of translating or compiling to native machine code, the Java bytecode can be quickly and efficiently executed. The JVM can also benefit from any hardware optimizations such as caching and floating point hardware units. Since the Java bytecode is denser than traditional compiled binary code, the JVM's resources can be smaller and still achieve the performance marks of software JVM's running on machines with large amounts of resources [22].

Inside the PicoJava JVM implementation simple instructions are executed directly in hardware, while moderately complex instructions (like floating point operations) are done in microcode. Complex instructions like the creation of a new object are implemented in software. Three caches are used for storing instructions, data, and the top of a Java call frame stack. This last cache is a rolling circular cache that allows values to dribble into and out of the cache from and to memory as call frames are pushed and popped onto the stack. Since the call frame stack now has direct access far deeper than most JVM implementation Java stacks do, there is no real need for an operand stack for scratch space. Similarly, the operations that load values into the operand stack and store values from the operand stack can be eliminated by directly accessing the appropriate location on the call frame stack cache. This effectively folds instructions together during the instruction decode step. To handle references to objects that are not resident in the stack cache, instructions were added to the instruction set to reference objects in memory [22, 25].

O'Connor and Tremblay report that loads account for more than 50% of bytecode instructions executed, with stores accounting for roughly 11% of bytecode instructions. The instruction folding performed by PicoJava eliminates approximately 15% of the dynamic instruction count [25]. Chapter 4 presents the results of our distribution experiments.

2.4 Previous Cache Research

Although thousands of papers have been written concerning caching, this investigation is particularly interested in cache benchmarks that have been used in the past and with how these measurements have been interpreted. Given that we look at specialized caches in Chapter 6, two experimental cache designs are cited to give an idea of the design optimizations that can be made when caches serve special purposes.

2.4.1 Benchmarks

In a study to experimentally prove the speed gains of cache memories, Ripps and Mushinsky look at the impact of internal and external cache on the 68020 family of processors. They begin by providing a methodology for testing cache speed, including how to account for compulsory cache misses. As expected, the best performance is achieved when both the on chip cache (instructions) and the off chip cache are used [27]. Gee *et al.* demonstrate the need to choose cache benchmarks carefully by showing that the SPEC92 Benchmark Suite miss rate is below the cache miss rate for typical multiprogram work loads. They suggest that the reason the SPEC92 suite has a lower than typical miss ratio is due to the lack of operating system components in the suite's tests. Operating system (OS) code tends not to loop and is not called frequently enough to stay resident in cache, effectively causing OS code to nearly always suffer the effect of cold start misses [11].

2.4.2 Caches and OS

Torrellas, Xia, and Daigle look at why OS code fails to reap the full benefits of caching and how OS code can be rearranged to benefit more from caching. Even though most common workloads make frequent calls to the underlying OS, these system calls are often written with long pieces of code that can cause self-interference within the cache. The problem is compounded by the fact that system calls are not made often enough to keep the relevant code in the cache. Thus, calls to the OS cause a large portion of cache misses. One example of self-conflicting OS calls is the conflict between performing a context switch between the system and the user, and the routines for starting system calls. Both are used frequently, yet the starting of system calls causes a context switch to occur. Fortunately, the most frequent system calls are tightly clustered and fairly deterministic, containing few variations in branching. This allows compiler optimizations in the layout of OS code in memory by allowing common sequences to be grouped together so that they can benefit from memory locality [29].

2.4.3 Hybrid and Adaptive Caches

A good deal of research has been done investigating more exotic and flexible cache designs. Fowler and Veenstra consider the impact of hybrid cache coherency methods in a multiprocessor system. A hybrid cache has the capability of having some blocks use a write update (write through) method while other blocks follow write invalidate (write back). The paper compares different granularities of hybridness: all lines coming from the same page in memory share the same write policy, each line of cache can have a separate write policy, and the traditional single write policy for the whole cache. While each of these are decided statically at compile time, the authors also consider the possibility of a cache that can dynamically select which write policy to use for which line. Note that this is only a theoretical consideration of the possible gains from these different methods; they are not tested in practice [10].

Hsu *et. al.* address the problem of a data cache not effectively containing all of the most recently used blocks of memory. They claim that lines not containing most recently used memory have only about 1% chance of being reused before they are replaced. They present a group-associative cache that tracks which lines in the cache have been the most recently used. The tracking is done using a supplementary table; cache lines not listed in the table are considered less-recently used. When a conflict miss could displace a most recently used line, the hit is displaced into one of the lines that has been less frequently used. These displaced lines are tracked in another supplementary table. Finally, the less recently used lines can also be used, to some extent, as a prefetch buffer. A performance comparison is made with conventional caching, column associative caching, and victim buffer caching [14].

2.5 Other Related Research

Some research dating before the release of Java in 1995 directly addresses performance problems facing JVM's. As of this writing, none of this research has yet been applied to speeding up Java programs. Once the language matures some more and improvements focused on JVM implementation become less dramatic, the Java community will likely consider incorporating some of this research.

2.5.1 Garbage Collection

Nilsen and Schmidt investigate the advantages and disadvantages of adding a special cache onto the memory bus to handle automatic allocation and freeing of memory. The focus of the study was whether or not garbage collection could be tightly bound so that it could be done in real time. The garbage collection unit was separated from the processor so that the hardware was not overly specialized (and thus, not too expensive). The garbage collection cache constantly maintains a free pool of open space. When that pool becomes too small, the non-free pool is garbage collected [24].

2.5.2 Exception Handling

The introduction of user defined exceptions has led to a new programming style in which exceptions are used to control the flow of an executing program. As Levy and Thekkath explain, however, the exception handling mechanism provided by operating systems doubles as the interrupt handler. Exception and interrupt handlers were not designed to be called frequently; handling them involves entering the kernel and executing system code. The consequent context switch is a huge performance hit. Levy and Thekkath explore the option of hardware and software support for passing the responsibility of certain user exceptions from the OS to the user program, thereby avoiding entering system code [19].

Java user exceptions are never passed to the OS kernel since the JVM acts as the OS for Java programs. As such, the penalty associated with handling exceptions is not as severe as it could be. Nonetheless, the ability for the JVM to automatically forward an exception to the correct handler could speed up exception handling.

2.5.3 Stack Caching

Ertl recalls that stack machines are frequently used to implement interpreters for special purpose languages. He considers the benefits from caching the top of the stack in a stack machine in registers. Specifically, two methods of caching are investigated. Statically the compiler generating the virtual machine can map the state of the stack machine to the registers. Transitions are defined over a finite set of states; these transitions modify the

cache, leaving it in a ready state for the next step of the interpreter. Alternatively, the interpreter can dynamically map its states to the registers at run time [9].

2.6 Conclusion

Java and the virtual machine that support the language are still relatively young. A lot of work has been done to improve the performance of JVM's. Very little of this research, however, has considered the benefits of hardware support for the Java runtime environment. If Java continues to be popular as a development tool and as an internet tool, hardware components designed to support Java might be justifiable within processors. Particularly enticing are components to support garbage collection in hardware and to handle exceptions. Both of these topics have been explored in previous research, but have yet to be applied to JVM's [19, 24].

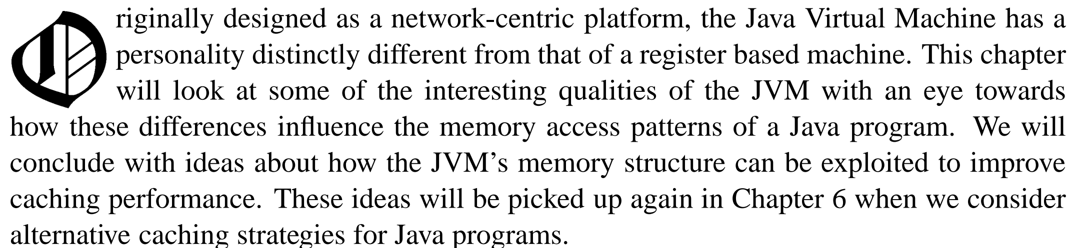
As mentioned above, thousands of papers have been published on the topic of caching and memory hierarchies. Due to its design, Java treats memory like a black box, allowing JVM implementations a great deal of freedom. In exchange for this platform independence, Java programs are unable to benefit from available cache support and the advances made in cache performance. A JVM, however, could have access to those caches and might be able to take advantage of different cache formations. We will be considering what caching strategies could be beneficial to Java performance.

Chapter 3

JVM Personality

"Sun Microsystems Inc.'s Java technology has undergone a remarkable transformation in a mere four years – from promising and sexy to useful and boring."

—Miguel Helft, June 1999[1]



Originally designed as a network-centric platform, the Java Virtual Machine has a personality distinctly different from that of a register based machine. This chapter will look at some of the interesting qualities of the JVM with an eye towards how these differences influence the memory access patterns of a Java program. We will conclude with ideas about how the JVM's memory structure can be exploited to improve caching performance. These ideas will be picked up again in Chapter 6 when we consider alternative caching strategies for Java programs.

3.1 Small Bytecode

Java class files are very small and compact compared to traditional compiled programs, allowing them to be more easily transmitted across networks to run on client machines. A `HelloWorld` Java class file takes up 462 bytes of space, compared with 3280 bytes for the same program written in C and compiled to machine code.[2] Since the two programs do the same work, they must require the same amount of data. The reduced size of Java class files is the result of smaller instruction size, individual instructions doing more work, and the implicit understanding that libraries will be dynamically loaded. For `HelloWorld`, the code that performs the actual printing is not contained within the class file; it lies within

[1] Quoted from online news article: http://www0.mercurycenter.com/svtech/news/indepth/docs/java061499.htm

[2] Compiled using gcc-2.95.1 for FreeBSD 4.

`System`, a class file which is assumed to be installed along with the Java Virtual Machine implementation.

The small average instruction size of Java programs is well documented by McGhan and O'Connor. As they point out, Java bytecodes are followed by a variable number of bytes of data needed to execute the instruction. Frequently, no extra bytes follow the bytecode. On average, the length of an executed instruction is only 1.8 bytes long [22].

Java instructions operate at a higher level of abstraction than traditional RISC[3] and CISC[4] instructions. For example, the Java instruction set includes instructions to allocate memory for objects and arrays, invoke virtual and static methods, and check casts at run-time. This level of abstraction means that the execution of each instruction requires more work to be done than for RISC or CISC instructions. Figure 3.1 shows the source code for `Foo`, a simple Java program. Figure 3.2 gives the disassembled Java bytecode for `Foo`, as well as a rough 68000 assembly language implementation.

```
public class Foo{
  /**
   * Pre:  last > 0
   * Post: returns sum of positive integers <= last
   */
  public static int sum(int last){
    int total = 0;
    for(int i = 1; i <= last; i++){
      total += i;
    }
    return total;
  }

  public static void main(String [] args){
    int sum = Foo.sum(1000);
  }
}
```

Figure 3.1: Source code for `Foo` example.

The 'i' prefix for Java opcodes indicates that the operation involves integer values; such opcode typing makes bytecode verification possible. The numbers at the beginning of each line are the offsets into the method's instruction bytecode stream. For example, `IADD` in *int*

[3]Reduced instruction set architecture.
[4]Complex instruction set architecture.

Java Bytecode

```
 int sum(int)
  0  iconst_0 ;; push 0 onto op. stack
  1  istore_1 ;; store 0 from stack to total
  2  iconst_1 ;; push 1 onto op. stack
  3  istore_2 ;; 1 from op. stack into i
  4  goto 14
  7  iload_1 ;; push total to stack
  8  iload_2 ;; push i to stack
  9  iadd
 10  istore_1 ;; store val. on stack to total
 11  iinc 2 1 ;; i++
 14  iload_2 ;; push val. in i to op. stack
 15  iload_0 ;; push val. in last to op. stack
 16  if_icmple 7 ;; if i ¡= last goto instr. byte 7
 19  iload_1 ;; push val. in total to op. stack
 20  ireturn

 void main(java.lang.String[ ])
  0  sipush 1000 ;; push 1000 onto op. stack
  3  invokestatic #4 <Method int sum(int)>
  6  pop
  7  return
```

68000 Assembly

```
sum:      link #-8,fp
          clr.l #-4(fp)
          move.l #1,-8(fp)
for:      cmp.l -8(fp),8(fp)
          blt end_for
          add.l #-8(fp),-4(fp)
          add.l #1,i
          bra for
end_for:  move.l #8(fp),d0
          unlk fp
          rts

main:     link #-4,fp
          move.l #1000,-(sp)
          bsr sum
          add.l #4,sp
          move.l d0,-4(fp)
          exit
```

Figure 3.2: Java bytecode for `Foo` and a possible 68000 assembly language translation. The numbers at the beginning of each Java bytecode line are the offsets into the method's instruction bytecode stream.

sum(int) can be found at offset 9. Each opcode takes up one byte in the instruction stream and is usually followed by zero to three bytes of operands. Many of the Java opcodes shown do not require arguments. The `ICONST_*` opcodes, for example, push a constant value onto the operand stack where the constant value is the value of the `*`. The integer add (`IADD`) similarly takes the top two values from the operand stack, adds them, and pushes them back onto the stack; no offsets or registers need be specified as source and destination. Also note the high level features of Java directly mapped into opcodes. The invocation of a static method is accomplished by `INVOKESTATIC #4`.

The 68000 assembly version includes almost exclusively instructions requiring arguments. The `ADD` instruction takes the location of the two operands as arguments, with the latter operand also serving as the store destination. Each of these instructions is specified by two bytes with the potential for an additional two bytes of supplementary operand data.

Ultimately, both the smaller average instruction size and the high level abstraction level of instructions skew the the traditional 75% instruction, 25% data split of memory accesses. As we'll see in Chapter 4, Java programs access memory far more frequently to fetch data than traditional programs do. Keeping in mind that instruction accesses have a higher locality rate than data accesses, this poses a potential problem to high cache hit rates.

3.2 Java Constants

All Java constants for a given class are stored in the class' constant pool. In Chapter 2 we saw that the resolution of instance variables and methods involves looking up the offset within a constant pool. Consequently, improving constant pool memory accesses could make a significant difference in performance. Moving constant memory accesses to a separate cache has the potential to reduce cache pollution and to make constant accesses consistently fast. Since this data is only written to once (when it is initialized), a special write once cache could be used to cache constant pool accesses. Finally, a relatively small cache could store most (if not all) of a class' constant pool.

3.3 Stack Architecture

The stack architecture of the JVM specification removes any reliance on the underlying machine's register configuration. This has two important ramifications with respect to memory accesses made by Java programs. First, nearly all data manipulated by a Java program passes into and out of operand stacks. Second, the typical scratch space provided by registers to an executing program is unavailable to Java.

3.3.1 The Operand Stack

Instead of having operands ready waiting in registers before performing an ALU operation, the CPU needs to take the necessary operands off of an operand stack. A simple example serves to illustrate how pervasive an impact this difference makes. To perform a simple integer addition using a RISC instruction set requires the following steps:

1. Fetching the instruction from memory. The instruction contains the source and destination registers.

2. Reading the operands from the specified source registers.

3. Performing the addition and storing the sum in the destination register.

It is straightforward to see that there is a single memory access involved in executing such an addition operation. On current RISC machines, reading the instruction from memory would require reading 4 bytes (or possibly 8 bytes). For the JVM to execute the same integer addition requires the following steps:

1. Fetching the instruction from memory. After decoding the instruction as an addition, the JVM knows that there is no need to read any arguments from the instruction stream.

2. Popping the needed operands from the top of the stack, causing two memory accesses.

3. Performing the addition and pushing the sum onto the stack, causing another memory access.

In executing a single addition operation, the JVM accesses memory once for the instruction (1 byte read) and three times for data (4 bytes each time). A total of 13 bytes must be read from memory, as compared to 4 bytes.[5] It is not immediately evident, however, how large an impact this increase in data accesses has on caching performance. Since the top of the stack is frequently the address being accessed in memory, the data accesses should have a higher degree of locality than in a traditional program. In Chapter 6 we consider whether the increased locality is worth the increased number of data accesses.

[5] Attentive readers will note that even though RISC add instructions always operate on registers, RISC processors still need to load the data operands into registers and store the sum in memory (at some point). At first glance this might seem to reduce the relative memory liability of Java's stack architecture. In reality, the loads performed by a RISC machine best correspond to the memory accesses the JVM makes to get values from local variables and parameters onto the stack. Similarly, saving the sum from the top of the stack to a local variable or parameter mirrors a RISC store.

3.3.2 Lack of Registers

By now the lack of register access presents an obvious performance problem: *the fast data storage provided by registers is not only unavailable, but replaced by memory accesses to the top of the stack*. Now both scratch space and memory accesses need to be cached, placing a greater responsibility on the cache. As long as the cache can cover most of the memory accesses, thereby hiding memory latency, a reasonable degree of performance should still be attainable. Chapter 6 considers the masking effects of registers and how they act like a level 0 cache.

3.4 Object Oriented Paradigm

Well-structured Java programs feature multitudes of small, layered methods. Such a programming style leads to many more method invocations. Since resolving a method invocation requires reading data from a constant pool, a high number of method invocations contributes to a high number of data accesses.

Looking at how a static method invocation executes is enlightening. First, the INVOKESTATIC opcode is read from the instruction stream; once it is decoded, two arguments are read from the instruction stream to serve as an index into the constant pool of the current class (three memory accesses). The indexed entry in the pool must have the tag CONSTANT_Methodref (one memory access to check). Assuming this is not the first invocation, the entry has a pointer to the location of the method bytecode (one memory access).[6]

In an effort to hide implementation details and to provide security, objects in Java can only be manipulated through references and not through pointers. Objects are passed by reference in parameters, their methods are accessed through references, and so on. This paradigm introduces a level of indirection when accessing object fields and methods, since an object cannot be treated as a value (like in C).

Finally, the treatment of arrays as full-fledged objects introduces further memory accesses. As we mentioned in Chapter 2, the language specification requires an `ArrayOutOfBoundsException` to be thrown when an illegal array access is made; therefore, the length of the array needs to be checked on every access. This adds at least one memory hit to every array access.

[6]If this is the first invocation, then the class containing the method needs to be resolved. If that class has not yet been loaded, the JVM tries to load it.

3.5 Conclusions

The combination of small bytecodes, a more powerful instruction set, and support for an object oriented programming paradigm leads to a much higher ratio of data accesses to memory accesses in Java programs than in tradition compiled programs. Since data accesses typically have a much lower locality than instruction accesses, this has the potential to reduce the effectiveness of caching. In Chapter 5 we will examine the performance of both unified and Harvard caches that are caching Java memory accesses.

Since class fields, methods, and constants are looked up in a class' constant pool, that pool is accessed frequently enough to consider caching constant pool data in a separate cache. Similarly, caching the top of the operand stack, an area with extremely high access locality, could help compensate for the higher rate of data accesses in Java programs. Finally, the lack of registers to act as scratch space could be partially hidden by using a very small, fast cache to hold the most recently used data. While this cache would not have the benefits of register mapping performed by a compiler, if the cache had a high enough degree of associativity it could still hold a small working set of recent accesses. In Chapter 6 we will consider the possible benefits of adding a constant cache, a stack cache, and a register cache, and compare their performance with more traditional cache configurations.

Chapter 4

Monitoring and Profiling the JVM

Here we will examine the process undertaken to monitor the memory access patterns of Japhar, an interpreted JVM. After first discussing why Japhar was selected as the JVM to monitor, we will look at the nature of the instrumentation inserted into Japhar to trace memory accesses. Then we will briefly describe the benchmarks that were traced. The distributions of these traces will be presented in Section 4.4. We will conclude by touching on the importance of the distribution of memory accesses with respect to caching.

4.1 Interpretation vs. JIT

Japhar is the Hungry Programmers' interpreted JVM. The version used for our experiments is Japhar 0.09, which supports JDK 1.1.5. Although some features remain to be implemented (such as support for the AWT libraries), most Java programs we've run on Japhar have not exhibited unexpected behavior. The project aims to provide an open source JVM to be embedded in proprietary/commercial applications. The open source nature of Japhar made it an easy candidate for our work, since we needed to modify a JVM to acquire memory traces of executing Java programs.

Although the first JVM's were interpreters, today most JVM's are just-in-time compilers (JIT's). In the domain of portable devices such as cellular phones and personal digital assistants (PDA's), however, computing resources are still very limited. The small memory footprint of an interpreter is an indispensable feature.[1] Nonetheless, the largest and possibly most important classification of JVM's today is the JIT JVM. This fact is underscored by the numerous research projects concerned with improving JIT compilation performance.

[1] At least, that is, until hardware implementations of the Java Virtual Machine specification become widespread enough to be a small add-on chip.

Given that JIT JVM's dominate the market today, it is important to address why the instrumentation and examination of an interpreter is worthwhile. The feature of an interpreter that makes it an unviable Java platform in most domains, its simplicity, makes it an attractive JVM to work with when investigating the behavior of Java programs. An interpreter is nothing more than a loop that fetches the next Java bytecode, decodes it, and executes it. The execution of the Java program is isolated, without the complications of compiling Java bytecode to native machine code in the background. The difficulty of determining if compiled native code exists and how to execute it instead of the bytecode is a non-issue with interpreters. Since this is the first investigation into the memory accessing character of Java programs, it is reasonable to limit the scope and complexity of the problem by working with an interpreter.

Limiting the investigation to the performance of Java programs makes the results applicable to Java native platforms. Hardware JVM's do not incur any of the overhead associated with simulating a program translator. As the PDA market expands there will likely be a demand for hardware JVM's.

Ultimately it is important to bear in mind that the memory profile of Java programs is largely blind to the implementation of the underlying JVM. Whether the virtual machine uses JIT compilation, adaptive compilation, interpretation, or is a hardware implementation, an executing Java program will still be designed to run on a stack-based architecture. The program will still need to make all of the memory accesses associated with the retrieval of data.

One could argue that a non-interpreted, non-hardware JVM has no need to fetch Java bytecodes from memory; hence, the inclusion of instruction accesses to memory separates our results from the behavior of JIT JVM's and adaptive compilation JVM's. This is at least partially true; however, it is not clear that JIT and adaptive JVM's are wholly exempt from the process of fetching instructions that are to be "translated." A JIT JVM *interprets* Java bytecode while it compiles that same bytecode in the background. Once the bytecode has been compiled, the JVM still has the overhead of locating the compiled code where it has been saved in memory. The quantity of memory accessed to fetch these instructions may differ from that accessed to fetch the Java bytecode; nonetheless, the JVM still needs to lookup the location of the compiled instructions in much the same way that it looked up the bytecodes. While the compiled code is fetched by the underlying CPU as instructions and the bytecode is fetched as if it was data by the CPU, both modes of operation require memory accesses to retrieve instructions. A similar argument can be made for JVM's that use adaptive compilation.

Gaining an understanding of how interpreted Java programs use memory is important. By limiting our scope to the accesses made on behalf of the Java program, and not the supporting JVM, we lose some of the memory profile of the whole JVM/Java program system. In exchange, the results are applicable to Java programs running on any JVM. In particular, they are immediately relevant to hardware JVM's natively running Java bytecode.

4.2 Instrumentation

In instrumenting Japhar we located the interpretation loop and followed the various paths of execution taken to translate the different Java bytecodes. When the bytecode was first fetched from memory the opcode was logged to an instruction log, tracing the order of instructions translated. The distribution of these instructions is discussed in Section 4.2.3.

It should be noted that not all of the memory accesses made by Japhar were logged. The intent was to capture the memory accesses made on behalf of an executing Java program as if the program were running on a CPU natively executing Java bytecode. The memory accesses made by Japhar as part of the JVM framework were ignored. Most notably, the pushing and popping of method frames off of a thread's method invocation stack were not logged.

Each time memory was accessed as part of the execution of an opcode, the access was logged to a trace file that served as input to a cache simulator (see Chapter 5). For each access, the address in memory hit was recorded along with a bit flag sequence classifying the access. Bits were used to mark an access as a read or write, and as an instruction or data access. Data accesses were also marked as being made into the operand stack or into the constant pool where appropriate.

Additionally, the opcode that caused the access was logged. The high level nature of Java opcodes (see Chapter 3) results in multiple memory accesses during the opcode execution. For example, the integer add opcode (`IADD`) generates four memory accesses:

1. The initial access to the bytecode stream to fetch the `IADD` opcode,
2. An access to the top of the (current) operand stack to retrieve the first operand for the instruction,
3. A second access to the top of the operand stack to retrieve the second operand, and
4. An access to the top of the operand stack to save the sum.

Memory accesses were occasionally made in instrumented code before the execution of the Java program started. These JVM initialization accesses might result from the loading of compiled class files and should be included in the memory profile of Java program. Since no opcode can be assigned to these accesses, they were assigned to NOP (which was never actually used as an opcode during benchmarks' executions).

It is worth noting that the values used in these memory accesses were not logged. Our cache simulator is only concerned with how successful a given cache structure is at capturing the current working set of memory addresses—it does not attempt to recreate the execution of a Java program. Ignoring the data values, we reduce the size of our trace files without losing any information that influences cache performance.

4.2.1 Virtual Addresses

The addresses being traced are the virtual memory addresses of Japhar. In reality, most caches deal with physical memory addresses that have already been translated from virtual addresses by the operating system. This does not have an impact on cache performance evaluation as long as all of the addresses sent to the cache are virtual addresses from the same virtual address space. By ignoring physical memory addresses and issues of page faults, we are freed from the task of invalidating lines of the cache that context switches would necessitate. The end result is a simpler cache simulator that still accomplishes the monitoring tasks we desire. The cache simulator will be discussed in greater detail in Chapter 5.

4.2.2 Maximum Trace Size

Most of the benchmarks accessed memory millions of times. After 116,508,000 memory accesses (just under one gigabyte of data), our instrumentation code stops logging accesses. The opcode logging terminates at the same time so that the number of opcodes executed can be compared with the number of memory accesses made during their execution. The number of unlogged memory accesses is recorded so that an idea of how much of the benchmark program executed before the access cap was reached (see Table B.1 for these numbers). The trace files are sufficiently large to provide a good idea of the memory character of each benchmark, despite not recording all of the accesses made during execution. They are long enough to ensure that cache performance is tested against sequences of accesses that do not cause mostly compulsory misses. If the sequences were too short, simulated caches would have performances suffering from disproportionately high numbers of compulsory misses.

4.2.3 Bytecode Distributions

In our overview of recent research surrounding Java Virtual Machines in Chapter 2 we included work by O'Connor and Tremblay on the PicoJava JVM. They give distributions of opcodes recorded during execution of javac and a raytracing program [25]. The opcode trace files we recorded were tabulated into the same classifications as those used by O'Connor and Tremblay, to verify the results they reported.[2] The complete breakdown of opcode classifications is given in Appendix B.

The numbers shown in Table 4.1 differ slightly from the averages given by O'Connor and Tremblay. Their numbers are based on the javac and raytrace benchmarks, while ours are drawn from monitoring the SPEC JVM98 benchmark suite (excluding check and checkit) and the linpack benchmark. While we did not trace the same raytrace benchmark,

[2] Since O'Connor and Tremblay do not list the break down of their categories, we made an educated guess as to what opcodes fell into each category.

Average Opcode Distributions

Type	Average
LVAR_LOAD	32.98%
LVAR_STORE	6.58%
MEM_LOAD	15.59%
MEM_STORE	4.79%
COMPUTE	10.67%
BRANCH	9.06%
CALL_RET	6.08%
PUSH_CONST	9.51%
MISC_STACK	4.09%
NEW_OBJ	0.25%
OTHER	0.42%

Type	Average
LVAR_LOAD	34.5%
LVAR_STORE	7.0%
MEM_LOAD	20.2%
MEM_STORE	4.0%
COMPUTE	9.2%
BRANCH	7.9%
CALL_RET	7.3%
PUSH_CONST	6.8%
MISC_STACK	2.1%
NEW_OBJ	0.4%
OTHER	0.6%

Table 4.1: The table on the left shows the average of opcode distributions for compress, db, jack, javac, jess, mpegaudio, mtrt, and linpack. The table on the right reprint the results reported by O'Connor and Tremblay in [25] obtained by running javac and raytrace.

Java Opcode Distributions #1

Type	`Empty`		**compress**		**db**	
LVAR_LOAD	104,888	35.71%	5,210,769	31.38%	8,716,464	42.55%
LVAR_STORE	32,439	11.04%	1,483,183	8.93%	1,357,670	6.63%
MEM_LOAD	41,849	14.25%	3,019,219	18.18%	2,067,214	10.10%
MEM_STORE	10,914	3.72%	769,032	4.63%	746,393	3.64%
COMPUTE	23,527	8.01%	2,015,318	12.14%	1,658,068	8.10%
BRANCH	31,261	10.64%	1,056,936	6.37%	3,056,178	14.92%
CALL_RET	19,272	6.56%	741,399	4.47%	477,526	2.33%
PUSH_CONST	25,371	8.64%	1,370,877	8.27%	1,955,483	9.55%
MISC_STACK	1,509	0.51%	931,089	5.61%	197,172	0.96%
NEW_OBJ	1,351	0.46%	3,239	0.02%	133,399	0.65%
OTHER	1,334	0.45%	3,373	0.02%	120,070	0.59%
TOTAL	293,715		16,604,434		20,485,637	

Table 4.2: Java opcode distributions.

Java Opcode Distributions #2

Type	jess		mpegaudio		jack	
LVAR_LOAD	5,619,818	37.85%	5,945,503	33.27%	3,763,667	26.25%
LVAR_STORE	1,054,238	7.10%	1,488,636	8.33%	306,664	2.14%
MEM_LOAD	2,703,206	18.21%	3,532,706	19.77%	2,996,309	20.89%
MEM_STORE	247,678	1.67%	594,547	3.33%	1,429,723	9.97%
COMPUTE	341,615	2.30%	2,684,887	15.03%	790,945	5.52%
BRANCH	1,947,029	13.11%	734,579	4.11%	1,634,844	11.40%
CALL_RET	1,671,152	11.26%	351,649	1.97%	409,709	2.86%
PUSH_CONST	936,114	6.31%	2,306,763	12.91%	915,590	6.38%
MISC_STACK	93,490	0.63%	206,959	1.16%	2,008,609	14.01%
NEW_OBJ	45,581	0.31%	5,269	0.03%	31,339	0.22%
OTHER	186,544	1.26%	16,422	0.10%	52,473	0.37%
TOTAL	14,846,465		17,867,920		14,339,872	

Table 4.3: Java opcode distributions *(continued)*.

Java Opcode Distributions #3

Type	mtrt		linpack		javac	
LVAR_LOAD	4,892,076	33.21%	4,737,260	27.12%	2,803,876	32.17%
LVAR_STORE	1,082,567	7.35%	997,029	5.71%	561,458	6.44%
MEM_LOAD	2,010,082	13.65%	1,005,675	5.76%	1,583,739	18.17%
MEM_STORE	757,661	5.14%	589,206	3.38%	569,968	6.54%
COMPUTE	1,093,320	7.42%	5,035,349	28.85%	524,843	6.02%
BRANCH	1,558,830	10.58%	417,951	2.39%	836,456	9.60%
CALL_RET	1,579,581	10.72%	1,175,859	6.74%	723,767	8.30%
PUSH_CONST	812,205	5.51%	3,495,494	20.03%	618,107	7.09%
MISC_STACK	847,844	5.76%	1,511	0.01%	396,256	4.55%
NEW_OBJ	71,690	0.49%	1,357	0.01%	25,429	0.29%
OTHER	22,690	0.15%	1,334	0.01%	72,605	0.83%
TOTAL	14,728,546		17,453,525		8,716,495	

Table 4.4: Java opcode distributions *(continued)*.

the mtrt benchmark is a ray tracing benchmark. As Tables 4.2 through 4.4 show, there is a fair deal of variation amongst the benchmarks in the frequency of the different categories. The slight differences between our results and those of O'Connor and Tremblay are the consequence of using different benchmarks.

Kazi *et al.* [17] report opcode distributions across similar categories for LinPack, CaffeineMark, Dhrystone, JMark2.0, and JavaWorld. They have separated data conversions from computation opcodes and comparisons from branch opcodes. The average of the different benchmarks roughly corresponds to the results given in Table 4.1. The numbers for our linpack differ in several places from theirs. Changing linpack to run as an application instead of an applet might have caused some of this difference.

4.3 Benchmarks

We used a number of benchmarks to generate opcode and memory trace files. Each benchmark is briefly examined below. Additionally, an empty Java program was traced to act as a yardstick for measuring the amount of overhead due to JVM startup and shutdown costs. The data for this program appears under the `Empty` heading. These numbers are not included in any averages, since `Empty` is not one of our benchmarks.

4.3.1 Linpack

Linpack is a benchmark that performs matrix operations on 500 x 500 matrices. Consequently, a huge number of memory accesses are made during the execution of the program. The version we ran was modified to run as an application from the command line instead of running as an applet in a web browser. The particular interest with this benchmark is the heavy use of arrays, both uni-dimensional and multi-dimensional, the large amount of computation that is performed, and the high concentration of constant pushes to the top of operand stacks.

4.3.2 SPEC JVM98

This is a benchmark suite developed by SPEC for comparing performance of JVM's. Our purpose in using this suite was not to be able to compare Japhar to other JVM's, but to look at the memory profile of well-known and accepted benchmarks. Since our purpose was only to gather data and not submit the results, we ran the benchmarks as applications from a command line. Several of the benchmarks did not run to completion, due to problems originating in the incomplete implementation of Japhar. However, all of the benchmarks that did not finish logged the maximum number of memory accesses before dying; therefore, the only information lost is a rough idea of what percentage of the benchmark had finished.

Each benchmark in the suite is briefly described below. For the benchmarks that failed to finish, a brief reason for the failure is forwarded where possible.

check

The benchmark verifies that the executing JVM supplies certain language features such as virtual and static method invocation, bit and arithmetic operations, access protection for fields and methods, and array indexing beyond its bounds. Japhar prematurely finished execution due to an unexpected `IndexOutOfBoundsException` that was generated within the method `System.arraycopy()`. Other than illustrating an implementation error for Japhar, the benchmark is not noteworthy (and is therefore not included in any averages we present).

compress

This is a port to Java of the 129.compress benchmark from CPU95. It runs a modified Lempel-Ziv (LZW) compression method on data read from files. Among the changes, the program no longer takes input from stdin and outputs to stdout; instead, both input and output deal with memory. This causes a large number of memory accesses to be made—8.77×10^8.

db

The benchmark loads a database from a file into memory and performs a number of additions, deletions, search, and sort operations on the database. Simulating the cache performance for this benchmark gives an idea for how well the database's organization deals with the random access of its information.

jack

Jack is a Java parser generator that takes a grammar as input and produces a parser for the grammar. The input in the benchmark is the grammar for jack itself, which is used to generate a tree multiple times.

javac

The javac benchmark tests the Java compiler from JDK 1.0.2. Since Japhar is not backwards compatible before JDK 1.1, the benchmark did not finish. As a side note, Japhar has not had problems running a JDK 1.1 or higher javac program.

jess

Jess is an expert shell system that takes a fact list and applies given if-then rules to them to solve logic puzzles. On each iteration of solving the puzzles extra non-applicable rules are added to the rule set, increasing the size of the search space. jess understands a special rules-language borrowed from Nasa's CLIPS program; as such, jess is basically an interpreter for this language. The language itself is list-oriented and is syntactically and semantically very LISP-like. Japhar executed most of the benchmark before abnormally quitting with a "fatal signal handler called, signal = 11" message.

mpegaudio

Mpegaudio decompresses ISO MPEG Layer-3 audio encoded files. About 4 MB of audio data makes up the workload. This benchmark makes nearly as many memory accesses as the compress benchmark while performing a relatively high amount of computation.

mtrt

This is a raytracer program that works on a scene containing a dinosaur. Two threads each render the scene. The benchmark features a lot of CALL_RET instructions. Japhar stalled and did not finish execution, possibly due to a problem with thread switching.

4.4 Distributions of Memory Accesses

While the distribution and frequency of Java opcodes gives an insight into what instructions are most frequently executed, this data alone fails to completely illustrate what instructions dominate the execution time of a Java program. Accesses to memory is a well-known performance bottleneck. As such, the memory profile of Java opcodes gives a truer picture of how much time the JVM spends executing a given opcode; this time can be thought of as a critical portion of the performance of the executing Java program.

We will look at the memory accesses logged during the execution of the benchmarks described above in two lights. First, we will look at how memory accesses are distributed across the opcode categories used in Section 4.2.3. Second, we will look at memory accesses in the broader categories of data hits vs. instruction hits and stack vs. constant pool vs. random access data hits.

4.4.1 Memory Access Distribution by Opcode Classifications

Although a Java program might spend 10.67% of its time performing computation, only 7% of its memory accesses are made on behalf of COMPUTE opcodes. Certain opcodes

mtrt: Distributions

Type	Opcode Dist.			Mem. Access Dist.			Mem/
	Freq.	%	bench.	Freq.	%	bench.	Ops
LVAR_LOAD	4,892,076	33.21%	32.98%	15,831,006	13.59%	15.28%	3.25
LVAR_STORE	1,082,567	7.35%	6.58%	5,310,223	4.56%	7.52%	4.78
MEM_LOAD	2,010,082	13.65%	15.59%	32,360,418	27.78%	31.50%	16.58
MEM_STORE	757,661	5.14%	4.79%	11,808,401	10.14%	9.75%	14.86
COMPUTE	1,093,320	7.42%	10.67%	4,120,879	3.54%	7.00%	3.78
BRANCH	1,558,830	10.58%	9.06%	14,443,162	12.40%	11.26%	4.41
CALL_RET	1,579,581	10.72%	6.08%	24,935,980	21.40%	12.37%	22.19
PUSH_CONST	812,205	5.51%	9.51%	2,433,604	2.09%	4.48%	6.08
MISC_STACK	847,844	5.76%	4.09%	3,317,765	2.85%	2.17%	2.71
NEW_OBJ	71,690	0.49%	0.25%	1,743,497	1.50%	1.16%	75.89
OTHER	22,690	0.15%	0.42%	203,065	0.17%	0.51%	39.88
TOTAL	14,728,546			2,320,137			7.90

Table 4.5: Opcode and memory access distributions for the mtrt benchmark, classified by opcode types.

require more memory accesses than others. Consider, for example, the distributions for the mtrt benchmark shown in Table 4.5 (the same data for the rest of the benchmarks can be found in Appendix B).

The opcode distribution column contains the values from Section 4.2.3, with the addition of an average percent. This percent is the average of the frequency percentages for all of the benchmarks. Next, the memory access distribution column tallies the frequency of memory accesses, classified by what opcode type caused them. As before, a frequency percentage is given; here it measures the frequency of a category against the total number of memory accesses logged. The average percent, again, is the mean of the memory frequency percent for all of the benchmarks. It is immediately obvious that there is a discrepancy between how frequently a given opcode is executed and for what portion of the memory accesses it is responsible. For example, COMPUTE opcodes occur 7.42% of the time in mtrt but only account for 3.54% of the memory accesses made by mtrt. In contrast, CALL_RET opcodes comprise 10.72% of the executed Java instructions and make up 21.40% of all the memory accesses made during the logged period. Clearly, CALL_RET opcodes are far more expensive in terms of memory requirements than COMPUTE opcodes. If memory access time dominates the performance of an executing program, then CALL_RET instructions could account for as much as $1/5$ of the total running time of the program.

Ideally, this discrepancy would not exist. If the frequency of an opcode equaled the

memory access frequency, then the time spent accessing memory on the behalf of a given instruction type would be proportional to the time the JVM spent executing that type of instruction. Having each instruction spend the same amount of time accessing memory would also bring the time to fetch/decode/execute different instructions closer. Tighter bounds on instruction execution allows for easier instruction pipelining.

The last column in Table B.11 provides an easy way to compare the different categories of instructions. A category whose (memory accesses)/(opcode count) is greater than the average of the total number of memory accesses over the total number of opcodes executed makes a higher proportion of memory requests than the average per opcode. The MEM_LOAD, MEM_STORE, CALL_RET, and NEW_OBJ all make relatively high numbers of memory accesses. BRANCH and OTHER are both close to the average number of memory accesses per opcode (for this benchmark), indicating that the time spent fetching memory for them is roughly proportional to the time spent executing those kinds of opcodes. The remainder of the categories fall below the average and use relatively fewer memory accesses than the average.

Average Memory Distributions, by Opcodes

Type	Average Freqency	Average Mops	Mops w/o javac
LVAR_LOAD	15.28%	3.37	3.41
LVAR_STORE	7.52%	4.96	5.04
MEM_LOAD	31.50%	15.56	15.43
MEM_STORE	9.75%	15.89	15.78
COMPUTE	7.00%	4.19	4.23
BRANCH	11.26%	12.04	5.18
CALL_RET	12.37%	16.14	16.09
PUSH_CONST	4.48%	3.05	3.06
MISC_STACK	2.17%	3.85	3.81
NEW_OBJ	1.16%	55.52	56.04
OTHER	0.51%	14.58	15.32
Overall Average: 7.45			*Without javac: 7.01*

Table 4.6: Averages of memory access distributions by opcode category. The mops column lists the average number of memory accesses per opcode executed. Since javac has an abnormally high BRANCH mops, the average mops without javac are given in the right most column.

Table 4.6 lists the average frequency of memory accesses across all of the benchmarks, categorized by opcodes. The average memory operations per opcode are also listed. Comparing Table B.11 to Table 4.6, we see that the mtrt benchmark has more expensive BRANCH instructions than the majority of the benchmarks. We also see that the NEW_OBJ

and OTHER categories are much more expensive on average than for mtrt. These two categories, however, show a lot of variation amongst the different benchmarks. The NEW_OBJ category ranges from 24.32 mops for mtrt to 113.63 mops for linpack. In general, the more NEW_OBJ opcodes executed, the less astronomical the NEW_OBJ mops. This also holds for the OTHER category.

Figure 4.1 summarizes the discrepancy between opcode frequency distributions and memory access distributions for the average across the benchmarks.

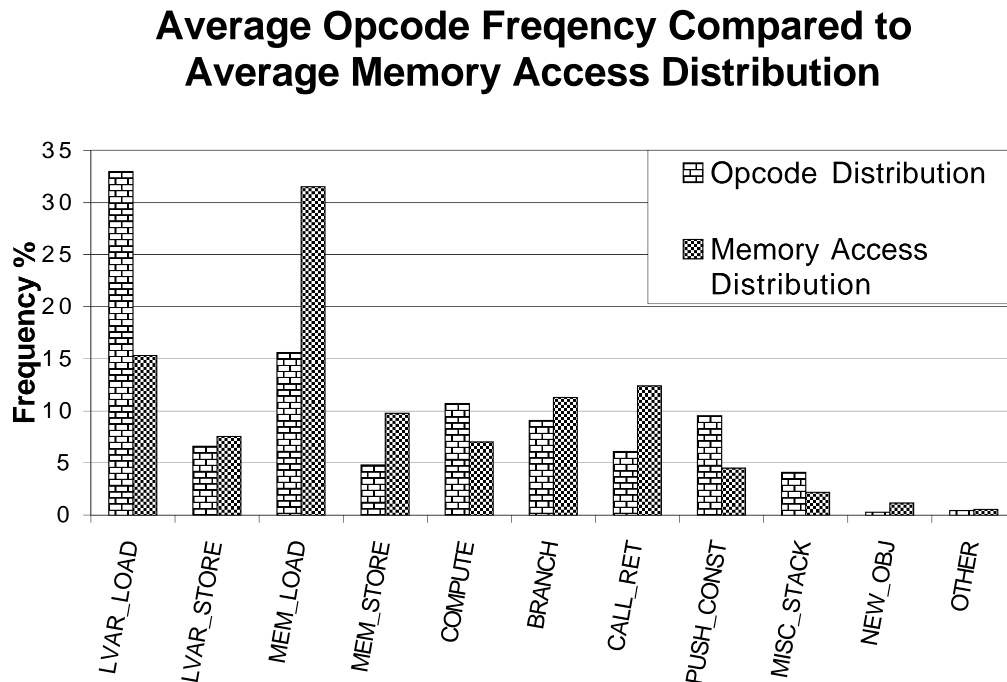


Figure 4.1: Comparison of average opcode and average memory access distributions, classified by opcode categories.

4.4.2 Memory Access Distribution by Access Type

The memory access profile of a Java program in terms of opcode categories reveals a lot of information about which opcodes are potential performance bottlenecks when accessing memory is slow. Another way to classify a memory profile is in terms of what kind of access is being made. Traditionally, instruction fetches and data retrievals were the only categories of memory accesses. Table 4.7 categorizes memory accesses for each benchmark in this way.

Memory Accesses by Type

Benchmark	Instruction Freq.		Data Freq.	
compress	28,472,120	24.44%	88,035,880	75.56%
db	38,885,226	33.38%	77,622,774	66.62%
Empty	537,003	23.15%	1,783,134	76.85%
jack	26,186,046	22.48%	90,321,954	77.52%
javac	62,744,599	53.85%	53,763,401	46.15%
jess	27,949,441	23.99%	88,558,559	76.01%
linpack	26,760,965	22.97%	89,747,035	77.03%
mpegaudio	29,151,192	25.02%	87,356,808	77.03%
mtrt	34,860,703	29.92%	81,647,297	70.08%
Average	34,376,286.5	29.51%	82,131,713.5	70.49%

Table 4.7: Classification of memory accesses by access type. The averages do not include `Empty` frequencies.

The *75/25 instruction/data* rule of thumb that describes the proportion of instruction accesses to data accesses for traditional compiled programs clearly does not hold here. Rather, the ratio can be described by the *30/70 Java Memory Rule:*

> *30/70 Java Memory Rule:* Whereas traditional compiled programs demonstrate a 75/25 instruction to data memory access ratio, Java programs follow a 30/70 instruction to data memory access ratio.

The javac benchmark is the exception to this distribution split, making 53.85% of its accesses to fetch instructions. This is particularly odd given that javac executed the fewest opcodes before reaching the memory access cap (see Table 4.4). A close inspection of the memory accesses made by javac shows that 40.63% of the accesses were caused by LOOKUPSWITCH opcodes (which comprised only 2.03% of the opcodes executed during the logged period). This opcode is followed by a variable number of bytes in the instruction stream; these form the lookup table. These bytes form match-offset pairs. Execution of a LOOKUPSWITCH opcode pops an `int` off of the operand stack and locates the match in the match-pairs list. The corresponding offset is used to branch to the next instruction to execute [20]. It would seem that the javac benchmark contains a number of large switch-case statements, causing Japhar to run through a large segment of the instruction stream looking for the correct match each time the LOOKUPSWITCH opcode appears.[3] As a

[3] As Lindholm and Yellin point out [20], a JVM does not need to perform a linear search through the bytes following a LOOKUPSWITCH opcode. A binary search, for example, could be used instead. Japhar apparently does use a linear search, however.

result, javac causes an abnormally high number of instruction fetches despite the reduced number of opcodes executed.

We can further categorize the data accesses according to whether they occurred on the stack, in the constant pool, or in generic memory. Table 4.8 lists such a break down of the memory accesses made by each benchmark. All of the benchmarks excluding javac (whose data percentages will be skewed as a result of more instruction accesses) and linpack made 19-26% of their memory accesses to the stack. This large a portion of stack accesses implies that even data accesses should have a good deal of locality.

Data Memory Accesses by Subtype

Benchmark	Stack Freq.		Constant Pool Freq.		Other Data Freq.	
compress	27,227,692	23.37%	10,656,013	9.15%	50,152,175	43.05%
db	30,109,795	25.84%	3,184,626	2.73%	44,335,353	38.05%
Empty	413,552	17.82%	225,564	9.72%	1,144,018	49.31%
jack	27,347,928	23.47%	12,257,486	10.52%	50,716,540	43.53%
javac	13,415,491	11.51%	7,620,881	6.54%	32,727,029	28.09%
jess	21,699,875	18.63%	11,190,534	9.60%	55,668,150	47.78%
linpack	45,909,465	39.40%	8,709,517	7.48%	35,128,053	30.15%
mpegaudio	30,299,903	26.01%	6,290,418	5.40%	50,766,485	43.57%
mtrt	22,410,799	19.24%	9,743,546	8.36%	49,492,952	42.48%
Average	27,301,743.5	23.43%	8,706,627.625	7.47%	46,123,342.125	39.59%

Table 4.8: Percentage of total memory accesses, by data subtype. The averages do not include Empty data points.

The linpack benchmark made more memory accesses to the top of the stack than generic memory accesses. This is probably a consequence of the heavy use of arrays in the benchmark, since the array opcodes often push or pop an object reference off of the operand stack. The ARRAYLENGTH opcode, for example, pops an array reference off of the stack, determines the length of the array (accessing generic memory), and pushes the length back onto the stack as an `int` [20]. The high concentration of PUSH_CONST opcodes also contributes to the large number of stack accesses since all of these opcodes push values onto operand stacks.

The constant pool accesses also form a sizeable portion of the memory accesses for all of the benchmarks excepting db, making up 5.4-10.52% of all memory accesses. Returning to Table 4.2, we find that the db benchmark has a very high proportion of LVAR_LOAD opcodes, with relatively few PUSH_CONST opcodes. This difference is probably a difference in programming style, showing a preference for using local variables over using constants.

It is worth noting that despite the significant portions of data accesses that are stack or constant accesses, all of the benchmarks still have large numbers of data accesses to generic memory. Locality for these data accesses will be much lower than the locality for instruction accesses, as it is for traditional programs. These accesses will also pollute any cache they share with stack and constant pool accesses.

4.5 Conclusions

When memory accesses are categorized by the type of opcode that caused them, we find executed opcodes do not have a proportionate amount of memory accesses made on their behalf. In particular, MEM_LOAD, MEM_STORE, CALL_RET, and NEW_OBJ opcodes account for more than their share of memory accesses.

In addition, far more memory accesses retrieve or write data than those that fetch instructions. This is a complete inversion of the memory profile of a traditional compiled language like C or FORTRAN, where 75% of the memory accesses fetch instructions. Given that data accesses have a lower rate of locality than instruction accesses, this distribution leads to a lower hit rate for both unified and split caches. Just how much of a difference this makes will be the topic of Chapter 5.

At the same time, the JVM has an inner subdivision in how it treats memory. Generic memory resides in the heap, while other memory accesses use an operand stack or a constant pool. Both stack and constant pool accesses form a sizeable portion of all memory accesses. Since accesses to a stack always use the top of the stack, these accesses have an extremely high locality. Similarly, the number of constants used in a program is usually small enough that taken all together they would have a high degree of locality. Unfortunately, these accesses are intermixed with data accesses to the heap, potentially reducing the effectiveness of stack and constant locality. On the other hand, data locality in general is certainly improved by their presence. This slightly improved locality helps to offset the increase in data accesses, reducing the performance hit of not making 75% of accesses to fetch instructions. The extent of these interactions will be examined in Chapter 6.

Chapter 5

Cache Simulation

At the end of Chapter 2 we touched on the fact that executing Java programs view memory as a black box. This abstraction frees Java programs from any particular computer architecture. As we also mentioned in Chapter 2, however, the presence or absence of a cache hierarchy is hidden from all user programs (Java or otherwise). Caches are *transparent*. The platform independence enjoyed by Java programs comes from the JVM Specification's ignorance of an architecture's register set.

This chapter describes the experimental setup for both unified and split cache simulation. Section 5.2 reports the results of the cache simulations and compares the results to cache performance results reported for traditional language benchmarks.

5.1 Simulation Setup

The cache simulator is a fully configurable program that takes as input a memory trace and a cache configuration and outputs the tallies of how often each cache in the configuration captured the memory accesses requested of that cache (see Figure 5.1). All caches are byte addressable; otherwise, each cache can be parameterized entirely independently of other caches in the configuration. Everything from line size and number of lines to the degree of associativity and read, write, and replacement policies can be set in the configuration file. Finally, there is a hook for adding a backup cache. This can be used to place a level two cache behind a level one cache, a level three cache behind a level two cache, and so on.

5.1.1 Data Free

As discussed in Chapter 4, the memory trace of executing benchmarks does not contain the memory values fetched. Only memory addresses were recorded, along with descriptions

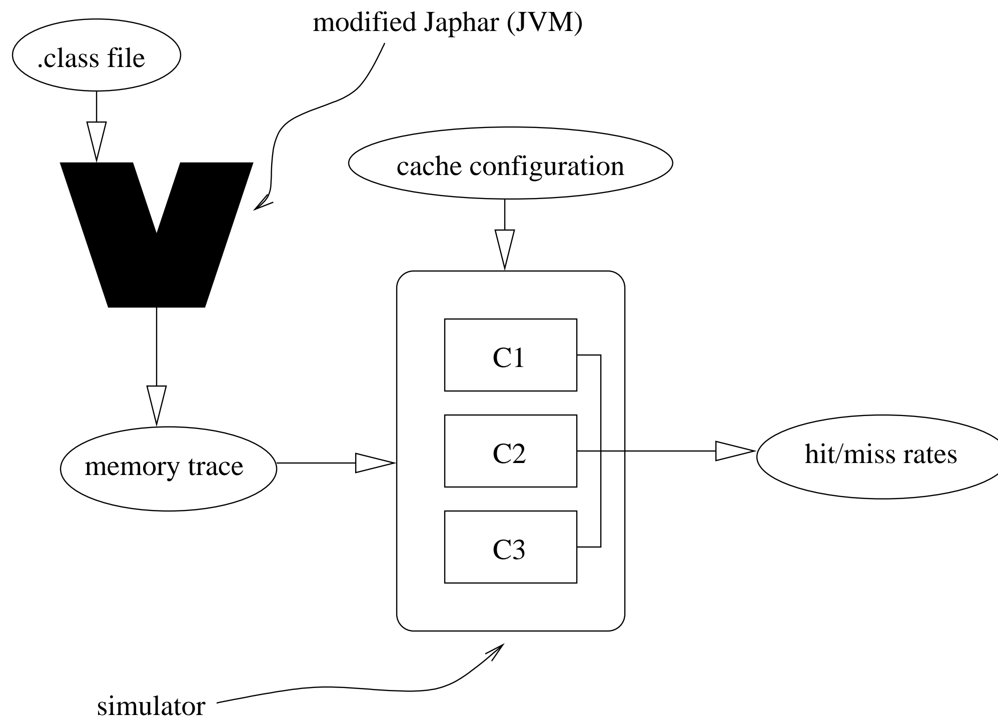


Figure 5.1: Architecture for the cache simulator. The simulator takes as input a cache configuration and a memory trace generated by the modified Japhar JVM and outputs the performance for each cache in the configuration. The diagram happens to show three simulated caches (C1, C2, and C3).

specifying the state of the JVM at the time the access was made. For our purposes of simulating the performance of a given cache configuration, this is sufficient.

Accordingly, the cache simulation program does not store the value of a memory address. Each line of the simulated cache stores the address tag (that part of the memory address that is not used to locate the line of the cache) and bits that denote the line as potentially dirty, valid, or read only. A time stamp is also present in each line to enable least recently used replacement policy simulation.

5.1.2 Policies

The simulator can be compiled with specialized read and write functions without rewriting the simulation program. This generality provides a great deal of flexibility to the simulator. By default the read function looks in a cache for an address; if it is not there and if a backup cache is present, the function then looks in the backup cache using the backup's read policy. Both random and least recently used (LRU) replacement policies are handled by the default read policy.

The default write function can handle combinations of write-through/write-back and write-fetch/write-around write and write miss policies. Like the default read policy, random and LRU replacement policies are handled by the default write policy.

Each cache has a predicate function that filters the memory accesses allowable in that cache. This can be as simple as filtering out non-instruction accesses to as complicated as only allowing accesses that fall within a certain address range. The default predicate function uses the accept and reject masks for the cache. The accept mask specifies access flags that must be set in order for an access to be potentially accepted for reading or writing from the cache. The reject mask specifies access flags that must not be true for a previously accepted memory access.

5.1.3 Monitoring

Each line of the cache tallies its hits and misses. If a missed address block has not yet been accessed during the trace for that cache, then the miss is recorded as a compulsory miss. All other misses are listed as conflict misses.

Categorizing non-compulsory cache misses as capacity or conflict misses would require post-simulation comparison and calculation. The method used by Patterson and Hennessey compares a cache's performance with the performance of a fully associative cache of equal size (using least recently used replacement) [26]. Tracking capacity misses for a fully associative cache is straightforward and can be easily done during the simulation. Fully associative caches do not experience conflict misses; therefore, any non-compulsory miss is a capacity miss. The same number of capacity misses occur in the non-fully associative cache of the same size; after simulation, a simple subtraction gives the number of capacity

and conflict misses incurred by the cache. Given that all cache misses are equal in terms of the delay they cause to the CPU, it is only worthwhile to distinguish between the miss types if the categorizations reveal something about the memory trace used as input. Compulsory misses roughly describe the memory addresses used by the program and give a general sense of the memory footprint of the program.[1] Hence, it is worth distinguishing these accesses from the others. Capacity misses give an indication of whether or not a program fits in the cache; they are inherently tied to the size of the cache. While this could be used to determine a good size for a cache, the end result can be roughly seen by the different performances for differently sized caches.

The literature on caching reports that eight-way associative caches have nearly the same miss rate as fully associative caches. Since fully associative caches do not suffer conflict misses, the number of capacity misses can be approximated by looking at the number of non-compulsory misses incurred by the eight-way associative cache experiments. Therefore, we have lumped capacity and conflict misses together.

Cache misses caused by a write instruction are handled differently depending on the write miss policy of the cache. If the cache uses a write fetch miss policy, then the miss is assigned to the line into which the address is fetched. If the cache uses a write around policy, the miss cannot be assigned to any particular line in the cache. Instead, a global count of write around misses is kept in each cache that uses that write miss policy.

5.1.4 Experiment Parameters

Both the unified and the Harvard cache (split cache) experiments use caches with 32 byte lines. Caches ranged in size from 1K to 128K. All memory traces were simulated with direct mapped caches and eight-way set-associative caches. The default predicate, read, and write functions were used. For non-direct mapped caches random replacement was used instead of the expensive LRU replacement policy. A write-through, write-around write policy was used for the unified cache and the data cache of the Harvard configuration. The instruction cache in the Harvard configuration was read only. Table 5.1 summarizes the parameters used.

5.2 Results

We simulated the caching performance of unified and Harvard caches for all of the benchmarks described in Chapter 4. Additionally, a simulation for a unified cache with a complete range of associativities was run with a shorter memory trace obtained from running

[1]As Sean Sandys wisely pointed out, a cache miss pulls a whole line into the cache. For caches with large lines, the line may contain memory that is never accessed; alternatively, the line may contain memory that is used in the near future. In the latter case a compulsory miss will not show up for the prefetched memory.

	Unified Cache	Harvard Cache	
		Data	Instruction
Size (KB)	1, 2, 4, 8, 16, 32, 64, 128	.5, 1, 2, 4, 8, 16, 32, 64	
Associativity	direct mapped and 8-way (random replacement)		
Predicate Policy	default		
Read Policy	default		
Write Policy	default	default	n/a

Table 5.1: Experiment parameters for unified and Harvard cache simulations.

the JDK1.2 javac program on `HelloWorld`. The associativity experiment was aimed at verifying that varying the degree of associativity of a cache had the same impact for caching Java programs as it did for caching natively compiled programs. The more extensive unified and Harvard cache simulations only simulate direct mapped caches and eight-way set-associative caches to reduce the computation time needed to perform the simulations. This decision was made since the absent cache miss ratios can be estimated from the eight-way miss ratio using the technique presented by Hill and Smith [13].

The results reported by Patterson and Hennessey and by Gee *et al.* are used as a basis of comparison representing cache performance for natively compiled programs. These results were obtained by running benchmark programs written in C [11, 26].

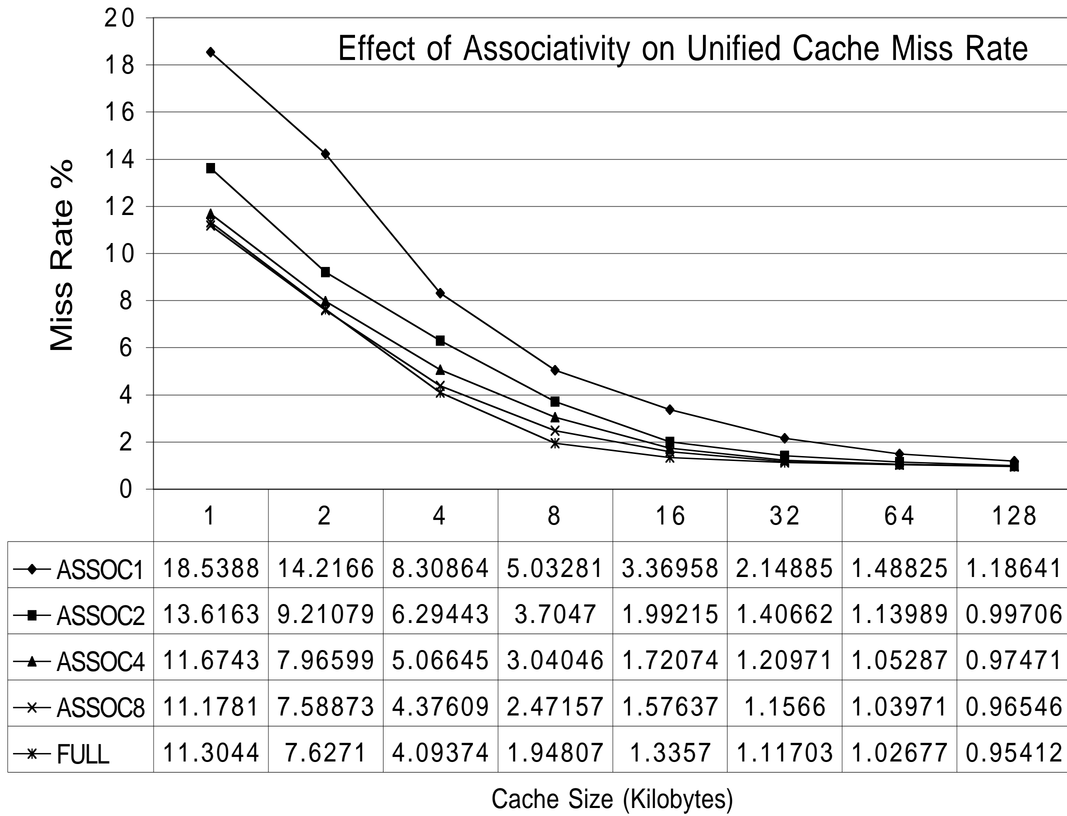
5.2.1 Associativity

The first observation that can be made by comparing Figure 5.2 with Figure 5.3 is that the Java benchmark caused a higher cache miss ratio than the C benchmark results reported by Gee *et al.* for comparable caches [11]. This is true for all cache sizes and for all degrees of associativity. Reasons for the higher cache miss ratios will be discussed in Section 5.2.2.

Otherwise, the two figures exhibit similar trends. Higher associativity results in a lower miss ratio, but with diminishing returns. Increases in cache size reduce miss ratios for all degrees of associativity. We also see that the *2:1 Cache Rule* holds for the Java cache simulations [26]:

> *2:1 Cache Rule:* The miss rate of a direct-mapped cache of size N is about the same as a two-way set-associative cache of size $N/2$.

For example, the miss ratio for a 16K direct mapped cache is roughly the same as an 8K two-way set-associative cache. The *2:1* rule of thumb applies to higher degrees of associativity, although the variation is not as tightly constrained. In our data, a 2K two-way set-associative cache performs about the same as a 1K four-way set-associative cache.



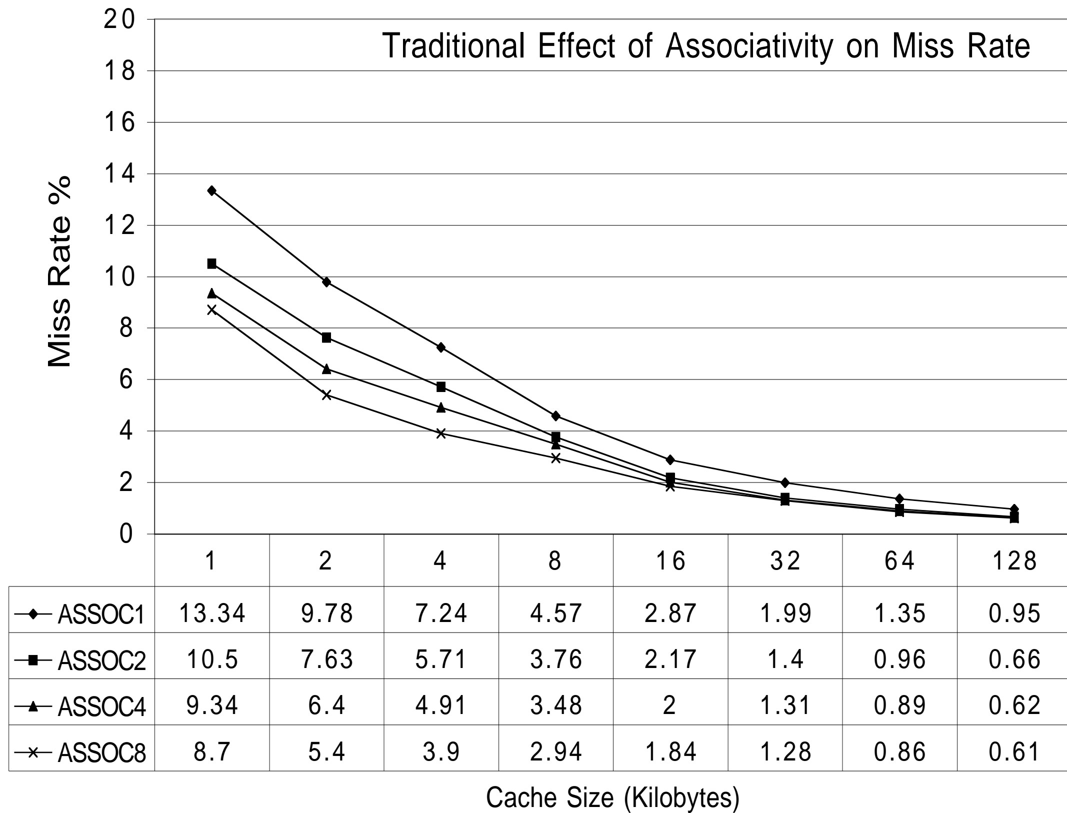
	1	2	4	8	16	32	64	128
ASSOC1	18.5388	14.2166	8.30864	5.03281	3.36958	2.14885	1.48825	1.18641
ASSOC2	13.6163	9.21079	6.29443	3.7047	1.99215	1.40662	1.13989	0.99706
ASSOC4	11.6743	7.96599	5.06645	3.04046	1.72074	1.20971	1.05287	0.97471
ASSOC8	11.1781	7.58873	4.37609	2.47157	1.57637	1.1566	1.03971	0.96546
FULL	11.3044	7.6271	4.09374	1.94807	1.3357	1.11703	1.02677	0.95412

Figure 5.2: Unified cache miss rates for different associativities for javac (JDK1.2) compiling `HelloWorld`.

Figure 5.2 also contains a data plot for a fully associative cache. Our results confirm the conventional wisdom that an eight-way set-associative cache has a miss ratio nearly identical to a fully associative cache.

Two concerns with the setup of this experiment need to be addressed before looking at the effect of associativity on average memory access time. First, the replacement policies used for the different experiments were different. Our results were obtained using a random replacement policy for set-associative and fully associative caches, while Gee *et al.* ran their experiments with the LRU replacement policy. It is interesting that despite this difference, the two figures do not reveal any noticeable impact. Random replacement performs nearly as well as LRU replacement without the overhead of implementing the LRU algorithm or storing extra bits to track the least recently used cache line in a set.

Second, the traditional results report the average miss ratios for a suite of benchmarks while the Java figure reports on only a single benchmark. Running javac to compile `HelloWorld`



	1	2	4	8	16	32	64	128
ASSOC1	13.34	9.78	7.24	4.57	2.87	1.99	1.35	0.95
ASSOC2	10.5	7.63	5.71	3.76	2.17	1.4	0.96	0.66
ASSOC4	9.34	6.4	4.91	3.48	2	1.31	0.89	0.62
ASSOC8	8.7	5.4	3.9	2.94	1.84	1.28	0.86	0.61

Figure 5.3: Unified cache miss rates for different associativities for SPEC92 benchmark suite [11, 26].

generated memory access distributions representative of the average of the Java benchmarks used (see Chapter 4). Tables 5.2 and 5.3 show the percentage of each kind of memory access. Note that the ratio of instruction to data accesses adheres to the *30/70 Java Memory Rule*. The assumption here is that memory traces with similar type distributions will have similar localities of reference. Hence, the caching performance would be similar.

Reducing the cache miss ratio by increasing the degree of associativity is only worthwhile if the overhead of the higher associativity is less than the gains from a lower miss ratio. To make this comparison we calculated the average memory access time in clock cycles using the method presented by Patterson and Hennessy [26] (see Figure 5.4). Since the caches in question are unified caches, there is a structural hazard created by fetching an instruction on the same cycle as fetching a piece of data. This leads to a modified formula as shown in Figure 5.5. The speeds for the different levels of cache/memory are taken from Patterson and Hennessy and are shown in Figure 5.6.

Memory Access Distribution for Javac (JDK1.2)

Type	Frequency	Freq. %
INSTR	8342553	29.26%
DATA	20166674	70.74%

Table 5.2: Distribution of memory accesses by type for compiling `HelloWorld` with JDK1.2 javac.

Data Memory Access Distribution for Javac (JDK1.2)

Data Type	Frequency	Freq. %
STACK	6723588	23.58%
CONST	1563774	5.49%
OTHER	11879312	41.67%

Table 5.3: Distribution of data memory accesses by type for compiling `HelloWorld` with JDK1.2 javac. Percentages are out of total memory accesses made.

General Method for Calculating Access Time

$$T_{cycles} = T_{hit} + M_{ratio} \times M_{penalty}$$

where

T_{cycles} = average memory access time in cycles

T_{hit} = time needed to access the cache on a hit, in cycles

M_{ratio} = miss ratio for the cache

$M_{penalty}$ = time needed to access the next lower level of memory on a miss, in cycles

Figure 5.4: General method for calculating average memory access time in cycles.

Method for Calculating Unified Cache Access Time

$$\begin{aligned} T_{cycles} &= I_{ratio} \times (T_{hit} + M_{ratio} \times M_{penalty}) \\ &\quad + D_{ratio} \times (2T_{hit} + M_{ratio} \times M_{penalty}) \end{aligned}$$

where

$$\begin{aligned} I_{ratio} &= \text{ratio of instruction accesses to total number of accesses} \\ D_{ratio} &= \text{ratio of data accesses to total number of accesses} \end{aligned}$$

Figure 5.5: Method for calculating average memory access time in cycles for unified caches.

Speeds for Different Memory Layers

$$\begin{aligned} \text{Register Speed} &= \text{1 clock cycle} \\ \text{Level One Cache Speed} &= \text{2 clock cycles} \\ \text{Level Two Cache Speed} &= \text{10 clock cycles} \\ \text{Main Memory Access Speed} &= \text{50 clock cycles} \end{aligned}$$

Figure 5.6: Speeds for different memory layers.

Following Patterson and Hennessey's example, the overhead for associativity can be included by increasing access speeds for cache and memory relative to a direct mapped cache hierarchy. The rationale is that looking up the correct line in a set in a cache adds time to the act of accessing the cache; this in turn forces the clock cycle to be slightly longer. For caches with associativity higher than 1, the average memory access time has been scaled by an appropriate associativity constant to increase the number of clock cycles spent *relative to a memory hierarchy using a direct mapped cache*. Table 5.4 lists the value of these associativity constants.

Tables 5.5 and 5.6 give the average memory access time in cycles for Java programs and the SPEC92 benchmark suite using a unified cache backed by main memory. The numbers were computed using the formula given in Figure 5.5 and the data values shown in Figures 5.2 and 5.3.

The values of interest are italicized in the tables. At these points the cost of a higher degree of associativity outweighs the gains gleaned from a lower miss ratio. For both the Java benchmark and the SPEC92 suite the tradeoff point comes at larger caches.

Patterson and Hennessy give their own numbers for comparing the average memory access time of different degrees of associativity. While we used their miss ratios, the average memory access times we report are higher by one to two clock cycles. The discrepancy

Relative Access Time Ratios for Associativity

Associativity	Time Ratio
direct mapped	1.00
two-way	1.10
four-way	1.12
eight-way	1.14

Table 5.4: Associativity overhead constants used by Patterson and Hennessy [26]. Note that there is a much larger difference between direct mapped and two-way associative caches than there is between two- and four-way and four- and eight-way caches. The transition from a direct mapped cache to a two-way associative cache requires the addition of new cache control logic. The transition from two- to four-way or from four- to eight-way associative caches only requires modifying the cache control logic already in place to accomodate larger sets.

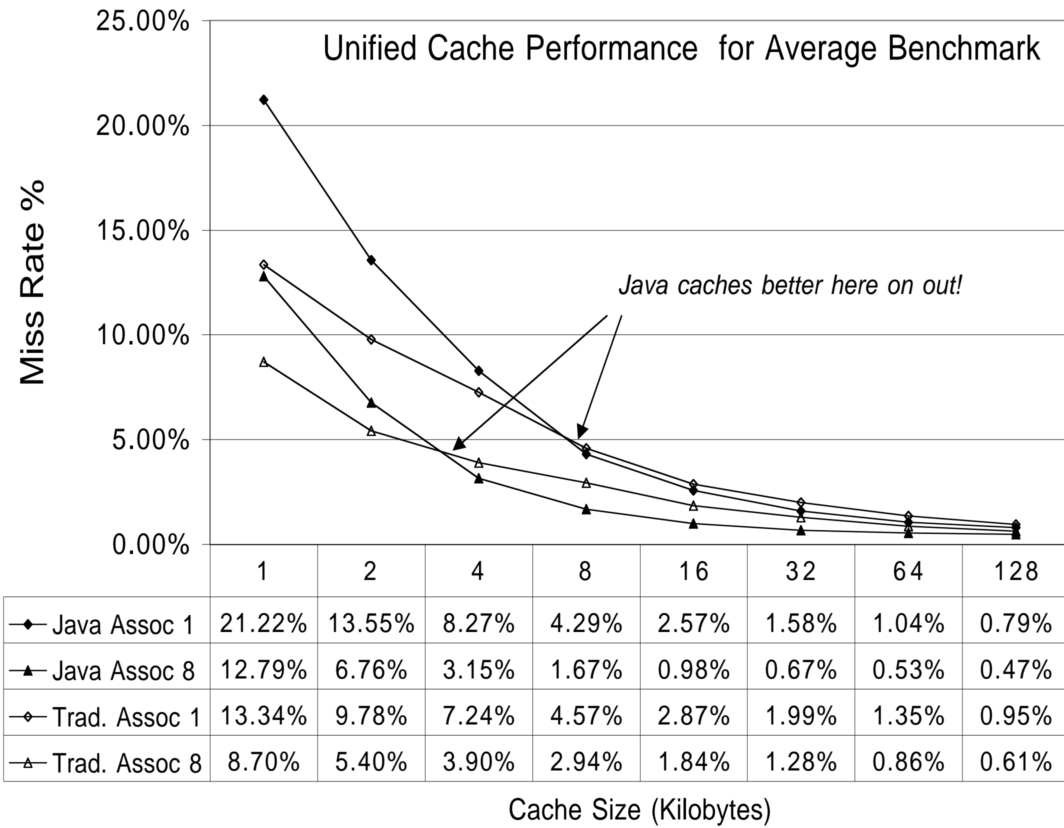
comes from using two clock cycles as the access time for a level one cache instead of the one clock cycle access time that they use. Interestingly, they use two clock cycles for the level one access time when comparing the performance of a unified cache with a Harvard cache. For consistency, we use two clock cycles as the level one access time in our calculations.

Despite the differences in our data with the numbers given by Patterson and Hennessy, the same trend appears in the data. Higher associativity comes with an overhead cost that can dominate average memory access times in larger caches. The slight improvement in cache hit rate gained by higher associativity is outweighed by the slower access time. Finally, the intuition that higher cache miss rates correspond to slower memory accesses is supported here *for unified caches*. In Section 5.2.3 we will see that higher miss ratios do not always lead to slower average access times.

5.2.2 Unified Cache

A unified cache tries to capture both instruction and data accesses made to memory. Our experiments simulated the performance of a unified cache for all of the benchmarks described in Chapter 4 except `HelloWorld`. Here we present the average results across the benchmarks. Simulation results for the individual benchmarks can be found in Appendix C.

Figure 5.7 contrasts the unified cache performance for an average of Java benchmarks against the unified cache performance for SPEC92 reported by Gee *et al.* and Patterson and Hennessy [11, 26]. For small caches (< 4K), the traditional benchmarks have a lower miss rate for both direct mapped and eight-way associative caches. This is not unexpected since Java programs make more data accesses than traditional programs. Data accesses have less locality of reference than instruction accesses, so it is easily understandable why the Java programs have a higher miss rate than the SPEC92 suite.



	1	2	4	8	16	32	64	128
Java Assoc 1	21.22%	13.55%	8.27%	4.29%	2.57%	1.58%	1.04%	0.79%
Java Assoc 8	12.79%	6.76%	3.15%	1.67%	0.98%	0.67%	0.53%	0.47%
Trad. Assoc 1	13.34%	9.78%	7.24%	4.57%	2.87%	1.99%	1.35%	0.95%
Trad. Assoc 8	8.70%	5.40%	3.90%	2.94%	1.84%	1.28%	0.86%	0.61%

Figure 5.7: Unified cache performance across an average of Java benchmarks. Data for direct mapped and eight-way associative caches is given.

Average Cycles per Memory Access for Java

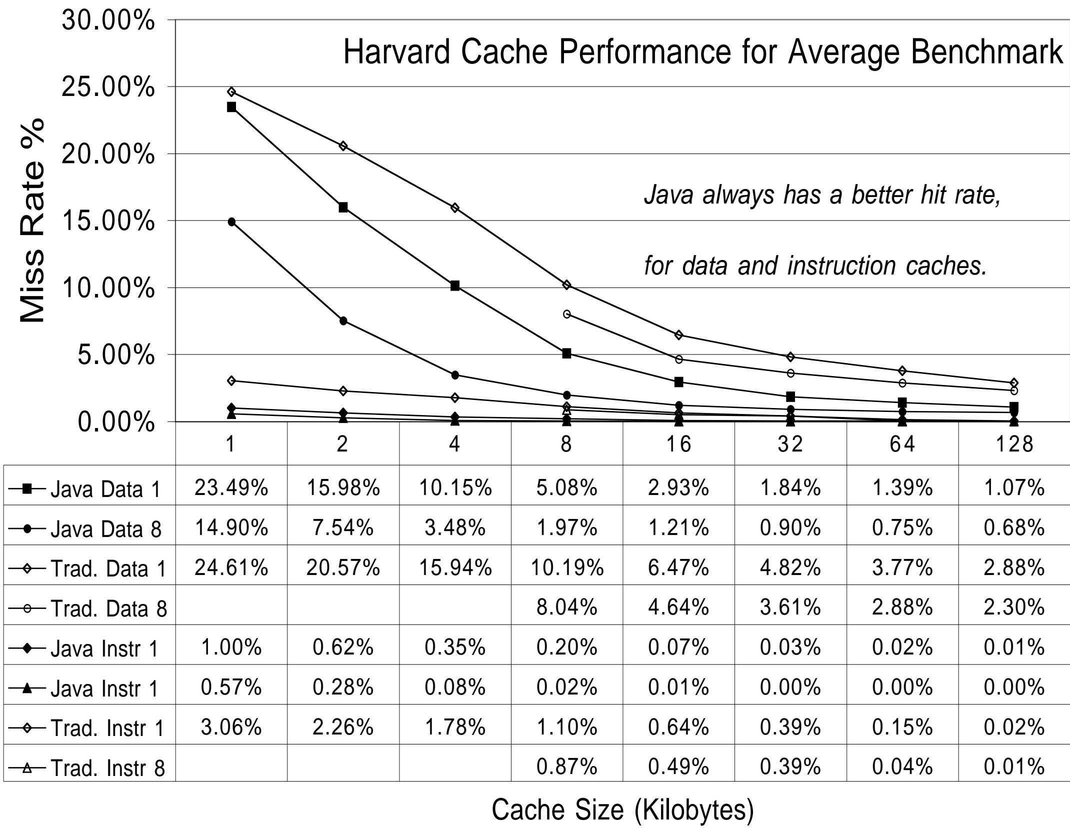
Cache Size (KB)	Associativity			
	One-way	**Two-way**	**Four-way**	**Eight-way**
1	12.68	11.25	10.36	10.26
2	10.52	8.82	8.29	8.22
4	7.57	7.23	6.66	6.39
8	5.93	5.79	5.53	5.30
16	5.10	4.85	4.79	*4.79*
32	4.49	*4.53*	4.50	*4.55*
64	4.16	*4.38*	*4.41*	*4.48*
128	4.01	*4.30*	*4.37*	*4.44*

Table 5.5: Average cycles per memory access by associativity. Calculations from miss rates for a unified cache simulating a trace of javac compiling `HelloWorld`(JDK1.2). Italicized type means that this time is not faster than the time to the left; that is, higher associativity *increases* average memory access time.

For larger caches, however, the Java benchmarks have a lower miss rate, again for both direct mapped and eight-way associative caches. The Java curves drop far more steeply than the traditional curves. This dramatic improvement in miss rate is probably caused by the Java programs nearly fitting into a 4K cache. Even with more data accesses and a resulting lower locality of reference, 8K or larger caches can capture almost all of the working set of memory addresses being accessed. Thus, the Java programs have relatively small memory footprints. Looking back at Figures 5.2 and 5.3 we see that here too the Java program has a steeper drop to its miss rate than SPEC92. Here, however, the direct mapped miss rate for the Java program never drops below the direct mapped miss ratio for the SPEC92 suite. The two-way, four-way, and eight-way associative cache miss rates do drop below the corresponding SPEC92 rates for 8K caches and larger.

5.2.3 Harvard Cache

Unlike a unified cache, a Harvard cache separates data accesses from instruction accesses. The rationale behind this segregation is discussed in Chapter 2. Like our unified cache experiments, all of the benchmarks were simulated on a Harvard cache. Here we will only present the results for an average across our Java benchmarks (excepting `HelloWorld` as always). Simulation results for the individual benchmarks can be found in Appendix C. Complete miss rates for eight-way associative data and instruction caches for the SPEC92 suite were not given by Gee *et al.* [11]; Figure 5.8 reproduces the numbers that were available.



	1	2	4	8	16	32	64	128
Java Data 1	23.49%	15.98%	10.15%	5.08%	2.93%	1.84%	1.39%	1.07%
Java Data 8	14.90%	7.54%	3.48%	1.97%	1.21%	0.90%	0.75%	0.68%
Trad. Data 1	24.61%	20.57%	15.94%	10.19%	6.47%	4.82%	3.77%	2.88%
Trad. Data 8				8.04%	4.64%	3.61%	2.88%	2.30%
Java Instr 1	1.00%	0.62%	0.35%	0.20%	0.07%	0.03%	0.02%	0.01%
Java Instr 1	0.57%	0.28%	0.08%	0.02%	0.01%	0.00%	0.00%	0.00%
Trad. Instr 1	3.06%	2.26%	1.78%	1.10%	0.64%	0.39%	0.15%	0.02%
Trad. Instr 8				0.87%	0.49%	0.39%	0.04%	0.01%

Figure 5.8: Harvard cache performance across an average of Java benchmarks. Data for direct mapped and eight-way associative caches is given.

Average Cycles per Memory Access for Traditional Programs

	Associativity			
Cache Size (KB)	**One-way**	**Two-way**	**Four-way**	**Eight-way**
1	9.17	8.53	8.03	7.81
2	7.39	6.95	6.38	5.93
4	6.12	5.89	5.55	5.07
8	4.79	*4.82*	4.75	4.53
16	3.94	*3.94*	3.92	3.90
32	3.50	*3.52*	*3.53*	*3.58*
64	3.18	*3.28*	*3.30*	*3.34*
128	2.98	*3.11*	*3.15*	*3.19*

Table 5.6: Traditional average cycles per memory access by associativity. Calculated from miss rates reported in [11, 26] for unified caches caching the SPEC92 benchmark suite. Cycle times are higher than those given in [26] on page 397 due to allocating two clock cycles to access the cache instead of one clock cycle. Italicized type means that this time is not faster than the time to the left; that is, higher associativity *increases* average memory access time.

Figure 5.8 contrasts Harvard cache performance across an average of Java benchmarks against the Harvard cache performance for SPEC92. The first thing to note here is the vast difference between the miss rates for the instruction and data caches. Increasing the size of the instruction cache gives only a slight improvement in the miss rate since the rates begin so low. Increasing the size of the data cache dramatically improves the miss rate, especially for the Java benchmarks. The direct mapped Java data cache ranges from a painful 23.49% miss rate for a 1K cache to an impressive 1.07% miss rate for a 128K cache.

There are two subtle points that can be deduced from Figure 5.8. First, the Java instruction cache always has a lower miss rate than that of the SPEC92 instruction cache. This is probably a consequence of Java programs having small bytecodes. With a small bytecode the instruction working set is naturally small, allowing the instruction cache to hold most of the working set and leading to a lower miss rate.

Second, even though small unified caches servicing Java programs have a higher miss rate than comparable caches servicing traditional programs, the miss rates for Java servicing data caches are always lower than the data cache miss rates for traditional programs *regardless of size*. By the time the Java direct mapped data cache is 8K, its miss rate is less than the miss rate for an 8K eight-way associative cache servicing the SPEC92 programs. As Section 5.2.2 explained, this indicates that the working set for the Java data accesses is not much larger than the size of the Java data cache. Removing the instructions to a separate cache frees up enough space in the data cache to allow the Java cache to outperform

the SPEC92 cache.

5.2.4 Comparing Unified and Harvard Caches

We saw in Section 5.2.1 with unified caches that a high miss rate translated into a high average memory access time. Harvard caches require a more careful analysis, since the miss rates for the data and instruction caches only affect the average access time by the percent of the time each cache is used. In other words, the miss rates for each cache are weighted by how frequently that cache is referred to. Figure 5.9 gives the formula used to calculate the average memory access time for Harvard caches. Because the expense of a cache is proportional to its size, it is important to compare a unified cache of size N with a Harvard cache of size N. This means that a 2K unified cache is compared with a Harvard cache comprised of a 1K instruction cache and a 1K data cache. All of the tables and figures in this section use this basis of comparison.

Method for Calculating Harvard Cache Access Time

$$\begin{aligned} T_{cycles} &= I_{ratio} \times (T_{hit} + M_{instr} \times M_{penalty}) \\ &\quad + D_{ratio} \times (T_{hit} + M_{data} \times M_{penalty}) \\ \text{where} & \\ M_{instr} &= \text{miss rate for instruction cache} \\ M_{data} &= \text{miss rate for data cache} \end{aligned}$$

Figure 5.9: Method for calculating average memory access time in cycles for Harvard caches.

Table 5.7 lists the average memory access times by cache type for the average of the Java benchmarks and SPEC92. The SPEC92 times were calculated assuming a 75/25 instruction/data access ratio while the Java times use the frequencies given in Chapter 4. The unified cache calculations favor the SPEC92 suite over the Java benchmarks since the structural hazard associated with a unified cache can be thought of as penalizing data accesses. With a much greater number of data accesses, the mtrt program is penalized much more frequently. While one could argue that the penalty should be assigned to the instruction accesses for Java programs, the interpreter's instructions are also stored in the unified cache. The hazard may exist on the machine; it just is not a hazard that falls within the main focus of this work. For this reason, we've left the hazard penalty adjusting the data access time for unified caches. Figure 5.10 compares the average memory access time for direct mapped caches.

Despite having better miss rates for both the instruction and the data caches, the Java benchmarks had a worse Harvard cache access time than the SPEC92 suite for caches less

Average Cycles per Memory Access for Unified and Harvard Caches

	Unified Cache		Harvard Cache	
Cache Size (KB)	**One-way**	**Eight-way**	**One-way**	**Eight-way**
Java benchmarks				
2	10.19	7.74	*10.43	*8.36
4	7.55	5.68	*7.72	5.35
8	5.56	4.84	*5.63	3.69
16	4.69	4.45	3.82	3.08
32	4.20	*4.27*	3.04	2.79
64	3.93	*4.19*	2.65	2.64
128	3.80	*4.15*	2.49	*2.58*
SPEC92 benchmark suite				
2	7.39	5.93	6.22	
4	6.12	5.07	5.42	
8	4.79	4.53	4.66	
16	3.94	3.90	3.69	*3.80*
32	3.50	*3.58*	3.05	*3.15*
64	3.18	*3.34*	2.75	*2.96*
128	2.98	*3.20*	2.53	*2.71*

Table 5.7: Average cycles per memory access for an average of Java benchmarks and SPEC92. Data is included for unified and Harvard caches. Eight-way associative entries that have a worse access time than the comparable direct mapped cache are italicized. Harvard cache entries that have a slower access time than a unified cache of the same size and associativity are marked with a '*'.

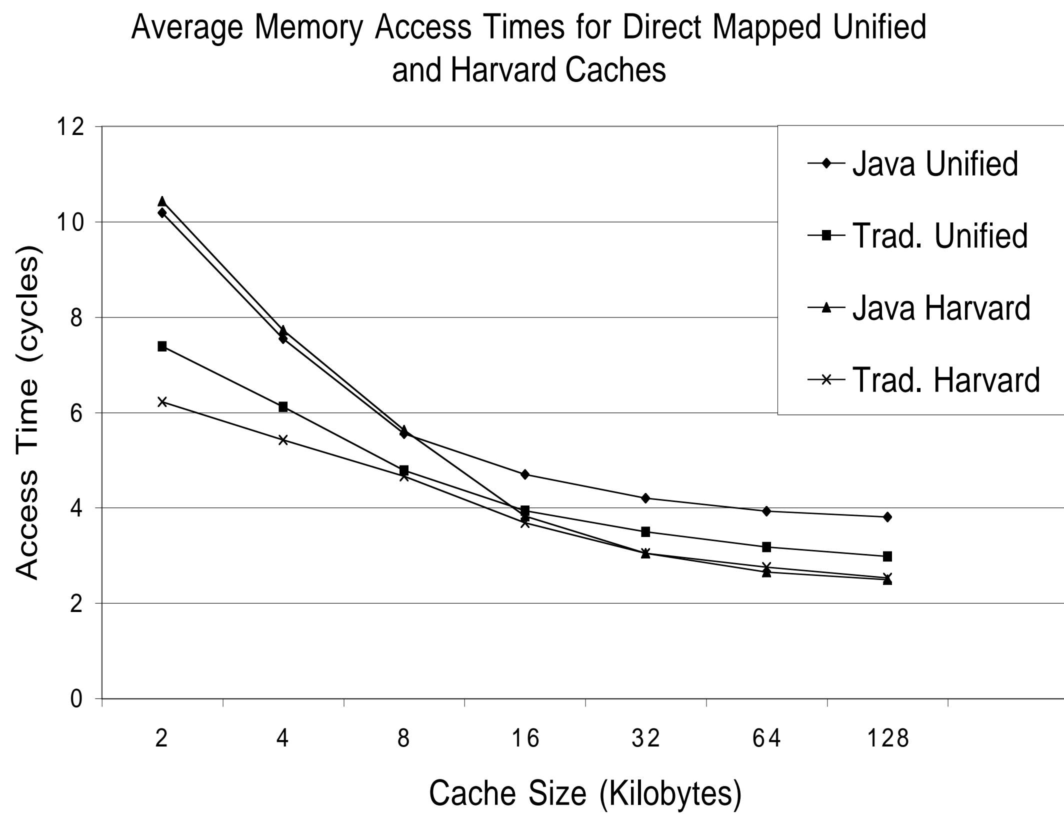


Figure 5.10: Comparison of direct mapped unified and Harvard cache average access times for Java and traditional programs.

than 32K in size. The reason lies in the *30/70 Java Memory Rule*; data accesses occur far more frequently than instruction accesses in Java programs, making the miss rate for the data cache much more important than the instruction cache miss rate. For the SPEC92 suite, the opposite is true since the instruction/data ratio is 75/25. Since the data cache holding Java accesses more strongly impacts the access time for the Harvard cache, its miss rate must be extremely low to achieve a fast time. With small data caches the miss rate is not low enough to offset the high demand placed on it by Java programs. For larger Harvard caches the Java programs have faster average access times than SPEC92. Unified caches servicing the SPEC JVM98 and linpack benchmarks never have faster access times than unified caches servicing SPEC92. Considering that for unified caches greater in 4K in size Java programs have better miss rates, the access time would be better for Java if the bus hazard was not assigned to data accesses.

A comparison between direct mapped unified and Harvard caches for the Java benchmarks reveals some surprising results. For small cache sizes the unified cache has a slightly better access time than the Harvard cache; this is not the case for the SPEC92 suite. The reason for this is the overwhelming number of data accesses made by Java and the high miss rates for small Java data caches. The performance cost of using the data cache outweighs the benefits gained from segregating instructions and data.

At caches larger than 8K, however, Harvard caches perform better than unified caches. Here the data cache miss rate is sufficiently low so that the cost of accessing more data than instructions does not outweigh the benefits of a separate instruction cache. This behavior is captured by the *16K Instruction Rule:*

> *16K Instruction Rule:* For the average Java program using direct mapped caches, separating instructions into a separate cache is only worthwhile when the total cache size is 16K or greater.

In truth, any JVM implemented in software does not use the instruction cache. Instead of performing like the direct mapped Harvard cache column, the Java program has an average memory access time *of a unified cache 1/2 the size*. For example, a Java program running in a software JVM on a desktop machine with a 32K level one Harvard cache (16K data, 16K instructions) does not have an average memory access time of 3.04 clock cycles. Instead, the program has an average access time equal to a 16K unified cache: 4.69 clock cycles.

Figures 5.11 and 5.12 illustrate the shifting importance of instruction and data accesses for the Java benchmarks and SPEC92 benchmarks for direct mapped Harvard caches. The areas are computed as a function of the average memory access time (for the whole cache/memory system) multiplied by the frequency of the different access types. The Java benchmarks spend most of their memory access time fetching data; the SPEC92 suite spends most of its memory access time fetching instructions. The figures also show how increasing cache size dramatically reduces the overall time spent making memory accesses.

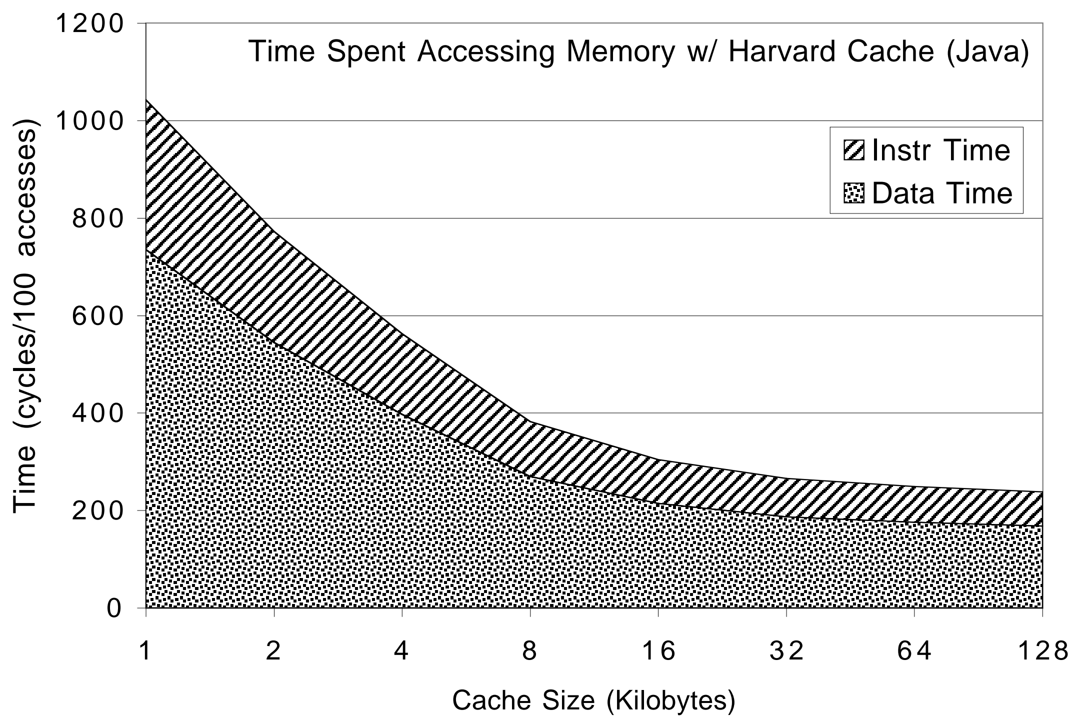


Figure 5.11: Time spent accessing memory with a direct mapped Harvard cache averaged across the Java benchmarks.

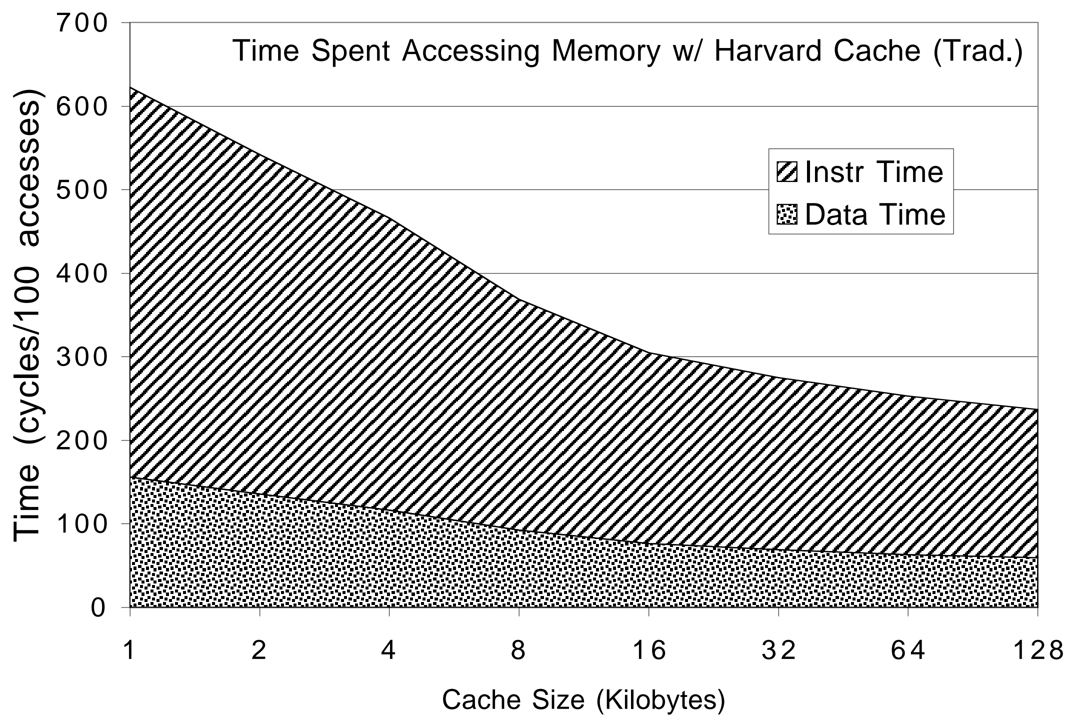


Figure 5.12: Time spent accessing memory with a direct mapped Harvard cache for traditional programs.

For the Java benchmarks the difference is a large decrease in the number of cycles spent fetching data.

5.3 Conclusions

With the exception of small Harvard caches applied to Java programs, Harvard caches perform better than unified caches. They provide a lower average memory access time than unified caches, better hiding the latency of main memory.

The Harvard cache experiment showed that the key to efficiently caching Java programs lies in efficiently caching data accesses. Chapter 6 will look at alternative cache configurations that try to exploit the structure the JVM specification imposes on memory.

Yet while Java programs spend the majority of their memory accesses retrieving data, cache performance could outperform that shown when servicing traditional natively compiled programs like SPEC92. This is due to the small memory footprints made by Java programs; these footprints are small enough to fit within the data cache. Thus, even with such the large number of data accesses and the lower locality of reference associated with data accesses, data caches are still very effective for Java programs.

Given that small instruction caches perform very well, especially when caching Java programs, the need for an instruction cache equal in size to the data cache is questionable. The chip real estate could be put to better use, possibly helping improve data access time.

Chapter 6

New Caches for Increased Performance

"Il faut cultiver notre jardin."
—Voltaire, *Candide*

In Chapter 5 we saw that Java programs have a slower average memory access time than traditional natively compiled programs for memory hierarchies containing unified or Harvard caches. Here we explore alternative caching configurations that reflect the internal memory structure imposed by the JVM specification. Specifically, three new caches will be considered:

Register Cache: A tiny, fast, eight-way associative cache that serves as a data buffer between the CPU and the main cache hierarchy (much like the register bank on a real machine).

Stack Cache: A small direct mapped cache that buffers the top of operand stacks.

Constant Cache: A small direct mapped cache that holds constant pool accesses.

These address the consequences of the lack of registers, the importance of accesses to the top of operand stacks, and the importance of constant pool accesses, respectively. All experimental results are an average across the benchmarks described in Chapter 4. The average is unweighted, considering all of the programs as equally important. Complete results for the individual benchmarks are given in Appendix D. We have not had time to thoroughly analyze this data; we include it in the appendix for completeness.

Each cache configuration we present in this chapter will be accompanied by a schematic key in the margin to aid in distinguishing the different cache topologies. Figure 6.1 is a large version of this schematic, with labels accompanying each cache type. The margin

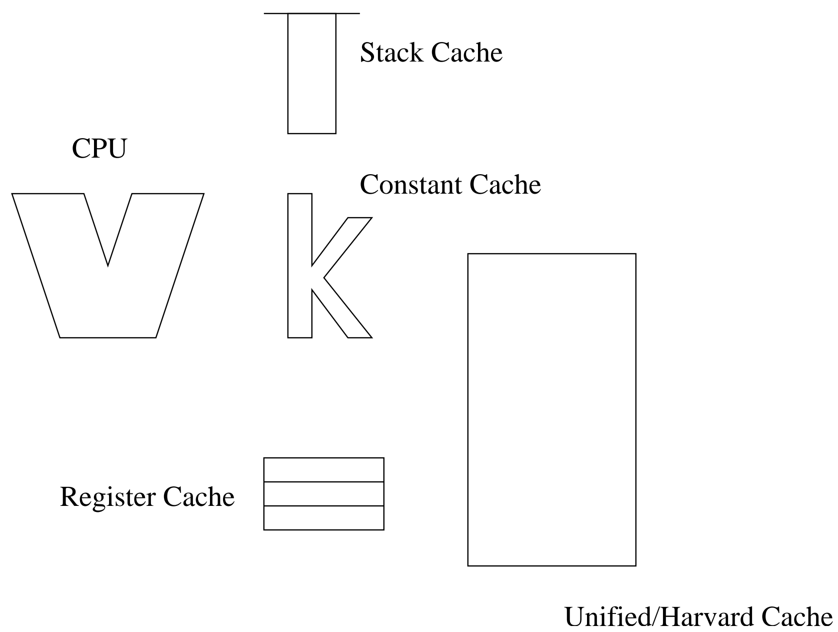


Figure 6.1: Sample cache topology schematic, without data flow arrows. When the large cache is used as a Harvard cache a line representing the data/instruction split is added. Otherwise, the large cache is a unified cache.

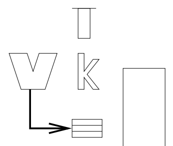
schematics use arrows indicating the flow of data from the CPU to the different caches contained in the configuration.

A brief word on terminology. While the hybrid caches are comprised of multiple caches and best described as configurations, the transparent nature of caches lets us think of a configuration as a unit. Often we refer to the whole of a cache configuration as a single cache entity. For example, a UNIREG configuration might also be referred to as a UNIREG cache. This follows the precedent of Harvard caches, which are comprised of a data cache and an instruction cache yet thought of as a single cache.

6.1 Register Cache

The JVM specification avoids referring to machine specific registers to maintain platform independent. As a result, no data is stored in registers on the physical machine. What if some data, however, were stored in registers? How does the lack of registers hurt the memory access time?

Adding a small, eight-way LRU associative cache is an attempt at approximating the resources provided by registers. While the cache will not be as efficient as the register mapping performed by compilers, making the cache eight-way associative with LRU tries to ensure that the line replaced is that which is least likely to be accessed in the near future. Since caches do not usually serve the purpose of containing register scratch space, we have little experience evaluating their effectiveness in this role. This makes comparing the caching results of stack-based architectures with register architectures difficult.



6.1.1 Isolated Register Cache

We now consider the appropriate size of a register cache, or *register buffer* (we will use these terms interchangeably). Since the register buffer simulates register resources for the JVM, the latency of the register cache is one clock cycle. This is consistent with the time needed to access registers on real machines and is appropriate given the hardware complexity of such a buffer. Because the register buffer simulates registers, only data accesses are passed to the buffer; therefore, this hardware is not involved with the instruction stream.

Register buffers from 64 to 512 bytes were tested. A line size of eight bytes was selected to be consistent with the size of registers. The isolated register buffer uses a write back, write fetch policy to simulate register behavior. When a register value changes, the new value is not immediately updated in memory; instead, memory is updated when the value is temporarily done being used. On writes the register buffer only writes back to the next level of memory when a dirty line is replaced.

We use a write fetch policy when handling write misses. This keeps the hottest memory addresses in the register cache, at the expense of occasionally swapping in data that is

not immediately reused. Incidentally, employing a write back, write fetch policy yields a slightly better miss rate than a write through, write around policy. In Section 6.1.2 we'll see that using a write through, write around policy increases the register buffer's miss rate by about 2%.

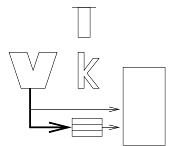
Average Register Cache Miss Rates and Access Times

Size (bytes)	Miss Rate	Access Time
64	47.93%	36.88
128	33.62%	31.13
256	24.42%	27.44
512	15.23%	23.74
Frequency: 70.22%		

Table 6.1: Miss rates and access times for an average of Java benchmarks. The access time is the average access time of a machine using just a register cache (in cycles). The frequency given is the percentage of total memory accesses passed to the cache.

Table 6.1 lists the miss rates for different sized register buffers. The effectiveness of such a small cache at capturing the working set of data accesses is impressive. A 512 byte cache has a miss rate of only one in seven accesses. Table 6.1 also lists the percentage of accesses made to the cache (the percent of memory accesses made to fetch data).

For all remaining experiments that employ a register buffer, the cache is a 256 byte cache.[1] A write through, write around policy is used instead of write back, write fetch; this policy is consistent with any downstream caches.



6.1.2 Unified Cache and Register Buffer (UNIREG)

Table 6.2 lists the miss rates for a cache configuration comprised of a 256 byte register buffer and a variable sized unified cache. Data accesses are first passed to the register buffer. If the requested data is not in the buffer, it then forwards the request to the unified cache. Since the register buffer has a write through update policy, any changes to the register cache are also made to the unified cache. The unified cache is inclusive: it contains all the data held in the register cache. Instruction accesses bypass the register buffer and go directly to the unified cache. For easy comparison the miss rates for a unified cache are also included in Table 6.2. The average memory access times are listed for a unified cache with and without a register buffer. The formula used to calculate the access times for this UNIREG configuration are given in Figure 6.2.

[1]If we did not limit the number of experiments by using a single register cache size, we'd still be gathering data. The 256 byte cache is reasonably sized in view of current register resources. For the same reason, stack and constant caches for hybrid cache configurations are a set size.

UNIREG **Cache Miss Rates and Access Times**

Unified Size (KB)	**Register + Unified**			**Unified**	
	register miss %	**unified miss %**	**Time (cycles)**	**unified miss %**	**Time (cycles)**
Direct Mapped Unified Cache					
1	26.37%	30.85%	9.46	21.22%	14.02
2	26.21%	19.77%	6.78	13.55%	10.19
4	26.21%	12.68%	5.08	8.27%	7.55
8	26.21%	6.67%	3.64	4.29%	5.56
16	26.21%	4.24%	3.05	2.57%	4.69
32	26.21%	2.62%	2.66	1.58%	4.20
64	26.21%	1.66%	2.43	1.04%	3.93
128	26.21%	1.23%	2.33	0.79%	3.80
Eight-way Associative Unified Cache					
1	26.14%	26.98%	*9.69*	12.79%	11.18
2	26.21%	15.35%	6.52	6.76%	7.74
4	26.21%	6.61%	4.13	3.15%	5.68
8	26.21%	3.21%	3.20	1.67%	4.84
16	26.21%	1.78%	2.81	0.98%	4.45
32	26.21%	1.17%	2.64	0.67%	*4.27*
64	26.21%	0.94%	*2.58*	0.53%	*4.19*
128	26.21%	0.81%	*2.54*	0.47%	*4.15*

Table 6.2: Miss rates and average access times for a unified cache augmented with a 256 byte write through, write around register buffer. Eight-way associative unified cache access times slower than the corresponding direct mapped cache times are italicized. The two rightmost columns reproduce the results from Table 5.7

Method for Calculating Memory Access Time for UNIREG Caches

$$
\begin{aligned}
T_{cycles} &= I_{ratio} \times (T_{L1} + U_{miss} \times T_{mem}) \\
&\quad + D_{ratio} \times (T_{reg} + R_{miss} \times (2T_{L1} + U_{miss} \times T_{mem}))
\end{aligned}
$$

where

$$
\begin{aligned}
T_{cycles} &= \text{average memory access time in cycles} \\
T_{L1} &= \text{time needed to access a level one cache on a hit} \\
T_{reg} &= \text{time needed to access a register} \\
T_{mem} &= \text{time needed to access main memory} \\
I_{ratio} &= \text{ratio of instruction accesses to total number of accesses} \\
D_{ratio} &= \text{ratio of data accesses to total number of accesses} \\
U_{miss} &= \text{unified cache miss rate} \\
R_{miss} &= \text{register buffer miss rate}
\end{aligned}
$$

Figure 6.2: Method for calculating average memory access time in cycles for UNIREG caches.

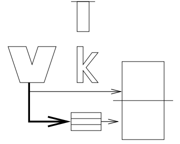
The addition of a register buffer causes the miss rate for the unified cache to dramatically increase. The register cache "steals" the highest locality data references. The write through policy grants the unified cache all of the locality of the data writes. However, only occasional data reads are requested from the unified cache. Otherwise, the register cache does not need to refer the request to the unified cache.

At the same time, the average memory access time improves with the addition of a register cache. There are two causes for this improvement. First, even though the unified cache has a higher miss rate, that miss rate only affects register cache misses (only one out of every four data accesses) or when the access is an instruction. Thus, the importance of the unified cache is greatly reduced. Missing the register cache introduces the overhead cost associated with looking in the register cache. Second, the penalty for accessing the register cache on a cache hit is only one clock cycle.

As a final point, the register cache miss rate is higher than that shown for the same size cache in Table 6.1. The cache used to obtain the results in Table 6.1 has the benefit of a fetch back write miss policy. This is especially important for the stack accesses that *always* write to the top of the stack before reading from it. We have not had time to further investigate the impact of different write policies on cache performance and average memory access time.[2]

[2]Our calculations for cache performance and average memory access time also do not consider the latency associated with always writing a value back to main memory for write through caches. The oversight is an intentional simplification; were we to account for the time needed to write back to memory for write through caches, we would need to treat issues of data hazards encountered via rapid successions of reads and writes to

6.1.3 Harvard Cache and Register Buffer (HARREG)



The impact of adding a register buffer to a Harvard configuration can be seen in Table 6.3. The size column gives the individual sizes for each of the instruction and data caches. Thus, the total cache size for the listed 1K entry is 2K: 1K data cache and 1K instruction cache. The HARREG cache configuration is just like the UNIREG configuration except that the cache behind the register cache no longer services instruction accesses. Those accesses are relegated to a dedicated instruction cache.

Deprived of instruction accesses and high locality data accesses, the data cache has a much higher miss rate than either the data cache from the Harvard configuration or the unified cache from the UNIREG configuration. Compared to a Harvard cache, the direct mapped HARREG cache has a miss rate twice as large; the eight-way associative HARREG cache has a miss rate three times as large. On the other hand, the data cache was accessed less frequently than those caches. A simple example makes this obvious. The register buffer in the HARREG cache has a miss rate of roughly 26%. In other words, 26% of the accesses it sees (70% of all memory accesses) are requested from the data cache—26% × 70% = 18.2%. Therefore, the 49% miss rate for the 1K data cache is only a performance concern 18% of the time.

The addition of a register cache amplifies the effect of increasing the size of the data cache; the miss rate improvement is greater for register buffered data caches. However, the miss rate decrease caused by an increase in data cache size *proportional to the previous size's miss rate* is roughly equal with and without a register buffer. The instruction cache is entirely unaffected by the addition of a register buffer.

As with the UNIREG configuration, the average memory access time with the inclusion of a register cache is shorter than the access time without the register cache. Figure 6.3 shows the formula used in calculating these times. Note that the only time the miss rate of the data cache is a consideration is when the register cache does not contain the requested data access.

Method for Calculating Memory Access Time for HARREG Caches

$$T_{cycles} = I_{ratio} \times (T_{L1} + I_{miss} \times T_{mem}) + D_{ratio} \times (T_{reg} + R_{miss} \times (T_{L1} + D_{miss} \times T_{mem}))$$

where

$$I_{miss} = \text{instruction cache miss rate}$$
$$D_{miss} = \text{data cache miss rate}$$

Figure 6.3: Method for calculating average memory access time in cycles for HARREG caches.

the same address.

HARREG **Cache Miss Rates and Access Times**

Instruction, Data Size (KB)	**Register + Harvard**				**Harvard**		
	register miss %	**instr. miss %**	**data miss %**	**Time (cycles)**	**instr. miss %**	**data miss %**	**Time (cycles)**
Direct Mapped Harvard Cache							
1	26.38%	1.00%	48.78%	6.35	1.00%	23.49%	10.43
2	26.21%	0.62%	33.71%	4.87	0.62%	15.98%	7.72
4	26.21%	0.35%	22.19%	3.77	0.35%	10.15%	5.63
8	26.21%	0.20%	11.00%	2.71	0.20%	5.08%	3.82
16	26.21%	0.07%	6.81%	2.30	0.07%	2.93%	3.04
32	26.21%	0.03%	4.33%	2.07	0.03%	1.84%	2.65
64	26.21%	0.02%	3.16%	1.96	0.02%	1.39%	2.49
128	26.21%	0.01%	2.40%	1.89	0.01%	1.07%	2.38
Eight-way Associative Harvard Cache							
1	26.14%	0.57%	47.23%	*6.95*	0.57%	14.90%	8.36
2	26.21%	0.28%	24.92%	4.57	0.28%	7.54%	5.35
4	26.21%	0.08%	10.33%	3.00	0.08%	3.48%	3.69
8	26.21%	0.02%	5.41%	2.47	0.02%	1.97%	3.08
16	26.21%	0.01%	3.19%	2.24	0.01%	1.21%	2.77
32	26.21%	<0.00%	2.22%	*2.13*	<0.00%	0.90%	2.64
64	26.21%	<0.00%	1.81%	*2.09*	<0.00%	0.75%	*2.58*
128	26.21%	<0.00%	1.61%	*2.07*	<0.00%	0.68%	*2.55*

Table 6.3: Miss rates and average access times for a Harvard cache augmented with a 256 byte write through, write around register buffer. The size listed is the size for the instruction and data caches individually. Eight-way associative data cache access times slower than the corresponding direct mapped cache time are italicized. The three rightmost columns reproduce the results from Table 5.7.

6.1.4 Summary

Adding a register buffer significantly improves the average memory access time of both unified and Harvard caches, particularly with small direct mapped caches. As the size of the caches increases the absolute improvement from a register buffer is reduced. Figure 6.4 shows this trend for direct mapped caches (the Harvard and HARREG plots for 2K are entries listed as 1K in Table 6.3 since the graph shows the total size for each cache configuration). Figure 6.4 also plots the average access times for unified and Harvard caches servicing traditional natively compiled programs.

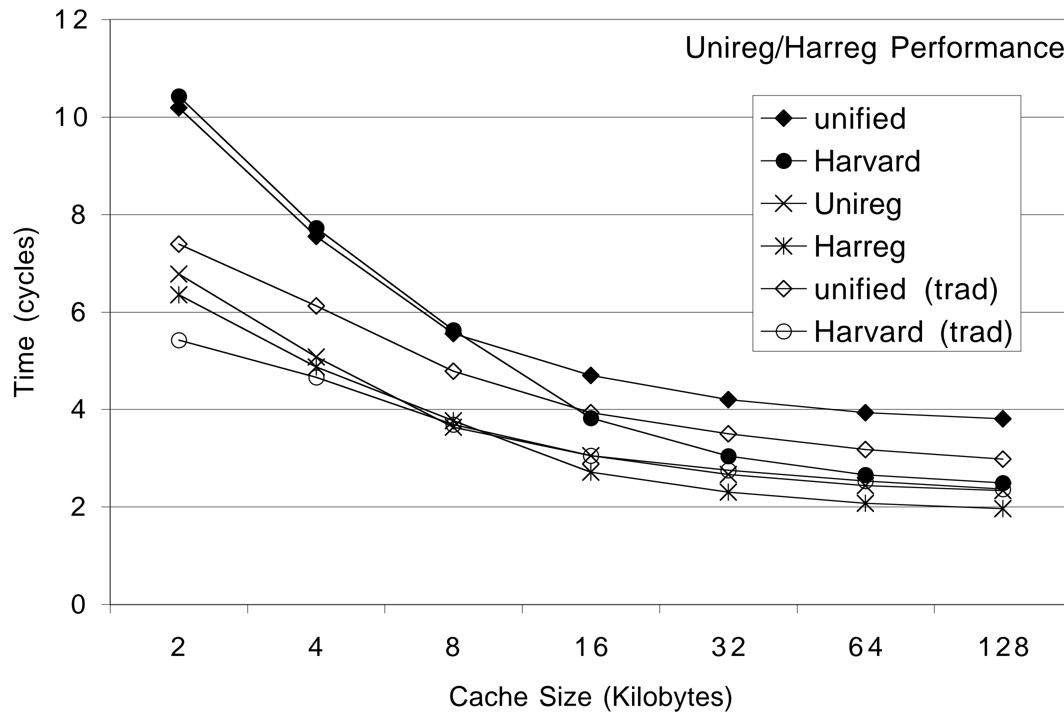


Figure 6.4: Comparison of UNIREG/HARREG with UNIFIED/HARVARD and traditional UNIFIED and HARVARD results. Times are for direct mapped caches. The filled in points are for Java benchmarks; the outlined points are for SPEC92 benchmarks.

Three facts are revealed by Figure 6.4. Adding a register buffer to a unified cache decreases the access time to the point where a UNIREG cache is faster than a Harvard cache for all the cache sizes tested. This is somewhat surprising given the decided advantage a Harvard cache has over a unified cache both for traditional programs and for Java programs. It underscores the fundamental difference between the memory profile for a Java program and the memory profile for a traditional compiled program, as summarized by the *30/70*

Java Memory Rule. A traditional program makes 75% of its memory accesses to fetch instructions. Isolating instructions in a separate cache has a large impact on performance. Java programs, on the other hand, only make 3/10 of their memory accesses to fetch instructions. The benefits from a dedicated instruction cache are less obvious. An optimization to improve the average data access time, like adding a register buffer, gives a performance boost.

Second, the HARREG configuration only performs better than the UNIREG cache for caches 16K and larger, and performs worse for caches <8K. Again, this points to the fact that data access optimizations are more important than instruction access modifications for the memory performance of Java programs. Moving high locality instruction accesses into a separate cache is not more beneficial than adding a register buffer when the fast instruction accesses do not offset a high data cache miss rate.

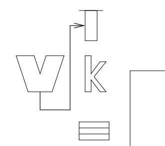
Third, the penalty incurred by not having access to fast register resources by the JVM is high. Keeping in mind that the SPEC92 results implicitly included register scratch space, we get the *Fair Hardware Rule:*

> *Fair Hardware Rule:* Under equal hardware resources Java programs have faster average memory accesses than traditional native programs.

UNIREG always has a faster average access time than the traditional unified cache. For >8K caches HARREG is faster than the traditional Harvard cache. The traditional Harvard cache is faster for caches smaller than 8K.

6.2 Stack Cache

The operand stack is central to the computation performed by the JVM—it is the only scratch space available to the execution engine. All (operand) stack accesses, by definition, hit the top of the stack. Chapter 4 reported that Java programs make 10-40% of their memory accesses to the top of the operand stack. Adding a small stack cache isolates high locality stack accesses and lets us examine the locality for the rest of the data accesses.



6.2.1 Isolated Stack Cache

As with the register cache we first look at the performance of an isolated stack cache of various sizes to determine a good size to use in the other experiments. Like the register cache, the stack cache was assigned an access latency of one clock cycle. With the small size of the cache, this is realistic. The stack cache is a write through, write around direct mapped cache with 8 bytes per line. The line size was chosen such that line replacements would not force large portions of the cache out at once.

Since stack accesses always occur at the top of operand stacks and because Japhar (the JVM used to collect our data—see Chapter 4) places all operand stacks contiguously

in memory, the stack cache behaves circularly. As space runs out in the cache (a rare occurrence as we'll see below) and the top of the operand stack grows incrementally, the line to replace wraps around to the beginning of the cache.

Average Stack Cache Miss Rates and Access Times

Size (bytes)	Miss Rate	Time (cycles)
64	0.000782%	38.52
128	0.000131%	38.52
256	0.000067%	38.52
512	0.000067%	38.52
Frequency: 23.43%		

Table 6.4: Miss rates and access times for an average of Java benchmarks. The frequency given is the percentage of total memory accesses passed to the cache.

Table 6.4 shows the miss rate and frequency for the stack cache over an average of the Java programs used. Even for the tiny 64 byte stack cache the miss rate was miniscule. Most of the misses were compulsory misses. An examination of the cache logs shows that for 256 and 512 byte caches not all of the cache lines were used. Table 6.5 lists the maximum number of stack cache lines used by each benchmark. Most of the benchmarks were efficiently cached (in terms of their operand stack accesses) in 64 byte caches. This result is summarized by the *25 < 64 Stack Rule:*

> *25 < 64 Stack Rule:* The 25% of memory accesses made by Java programs to operand stacks can be efficiently cached in 64 bytes.

The incredibly low stack cache miss rate indicates that stack accesses have a very high locality of reference. There are the obvious reasons for this locality: data on the stack is always written before being read, and any access is always happening at the current top of the stack. However, as discussed in Chapter 2 the JVM specification requires that each method have its own operand stack. It is not difficult to imagine that in the process of changing from one method invocation to another the top of the currently active operand stack is arbitrarily distant in physical memory from the last currently active operand stack. Such stack hopping would lead to extra compulsory misses since new memory addresses would need to be loaded into the cache. The number of conflict misses would also be higher since it is equally probable that the new stack top maps to a filled cache line as that the new stack top maps to an available line in the cache.

The fact that the miss rate is very low, that not all of the lines of large stack caches are used, and that the number of compulsory misses always falls short of fifty is evidence that such a problem does not arise in Japhar. As mentioned above, Japhar implements operand

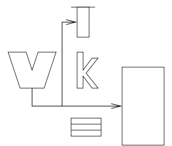
stacks on a single stack. When a method is invoked, a pointer to the bottom of the stack is moved to the address above the current top of the stack. The top of the stack then becomes the new bottom of the stack. When the method returns the pointers are reset. Therefore, stack hopping as methods are invoked and returned is simply moving across a contiguous block of memory.

Maximum Stack Lines Used, by Benchmark

Benchmark	No. of Lines Used
compress	8
db	8
jack	8
javac	25
jess	8
linpack	8
mpegaudio	8
mtrt	8

Table 6.5: Maximum number of stack cache lines used by different Java benchmarks.

Since none of the benchmarks fill the 256 byte stack cache, we chose to use a 256 byte direct mapped cache for all of the experiments that called for a stack cache. This is small enough that adding it to a chip would not be a major commitment of chip real estate.



6.2.2 Unified Cache and Stack Cache (STACKUNI)

Table 6.6 shows the miss rates for a unified cache augmented by a stack cache. The dramatic increase in the miss rate for the unified cache when a register buffer was added is not present when a stack cache is added. The miss rates for the unified cache are only slightly higher when a stack cache is added. Since stack accesses are a large portion of memory accesses this is not intuitive. Taking high locality references out of the unified cache normally would decrease the efficiency of that cache.

Two explanations are possible. First, moving the stack accesses out of the unified cache makes room for other high locality accesses such as instructions. More likely, the stack accesses only take up a small number of bytes in a cache. In the unified cache the stack accesses likely were always contained within a few cache lines. Moving them out freed up those lines to be used by other accesses. The loss of high locality accesses was offset by the addition of a few cache lines that were typically loaded.

Second, the stack cache does not shield the unified cache, nor does the unified cache backup the stack cache. Memory accesses are immediately directed towards one cache or

STACKUNI **Cache Miss Rates and Access Times**

Unified Size (KB)	**Stack + Unified**			**Unified**	
	stack miss %	**unified miss %**	**Time (cycles)**	**unified miss %**	**Time (cycles)**
Direct Mapped Unified Cache					
1	<0.00%	22.18%	11.20	21.22%	14.02
2	<0.00%	16.04%	8.85	13.55%	10.19
4	<0.00%	10.15%	6.60	8.27%	7.55
8	<0.00%	5.36%	4.76	4.29%	5.56
16	<0.00%	3.19%	3.93	2.57%	4.69
32	<0.00%	1.96%	3.46	1.58%	4.20
64	<0.00%	1.32%	3.21	1.04%	3.93
128	<0.00%	1.02%	3.10	0.79%	3.80
Eight-way Associative Unified Cache					
1	<0.00%	15.03%	9.65	12.79%	11.18
2	<0.00%	8.05%	6.60	6.76%	7.74
4	<0.00%	3.91%	4.79	3.15%	5.68
8	<0.00%	2.09%	4.00	1.67%	4.84
16	<0.00%	1.25%	3.63	0.98%	4.45
32	<0.00%	0.86%	*3.46*	0.67%	*4.27*
64	<0.00%	0.69%	*3.39*	0.53%	*4.19*
128	<0.00%	0.60%	*3.35*	0.47%	*4.15*

Table 6.6: Miss rates and average access times for a unified cache augmented with a 256 byte direct mapped write through, write around stack cache. Eight-way associative unified cache access times slower than the corresponding direct mapped cache times are italicized. The two rightmost columns reproduce the results from Table 5.7.

the other. Whereas the register cache filtered all of the high locality data references from the unified and Harvard caches it fronted, the stack cache allows any non-stack high locality data references through.

The average memory access times were calculated using the formula shown in Figure 6.5. Adding the stack cache improves the access time, particularly for the smaller unified caches. Even though the improvement in number of cycles is less for eight-way associative STACKUNI caches, as a percentage of the access time for a unified cache, STACKUNI caches have a 10% faster access time.

Method for Calculating Memory Access Time for STACKUNI Caches

$$\begin{aligned} T_{cycles} &= I_{ratio} \times (T_{L1} + U_{miss} \times T_{mem}) + \\ &\quad S_{ratio} \times (T_{stack} + S_{miss} \times T_{mem}) + \\ &\quad (D_{ratio} - S_{ratio}) \times (T_{L1} + U_{miss} \times T_{mem}) \end{aligned}$$

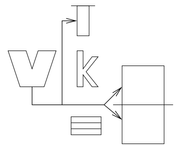
where

S_{ratio} = ratio of stack accesses to total number of accesses

S_{miss} = stack cache miss rate

T_{stack} = time to access the stack cache

Figure 6.5: Method for calculating average memory access time in cycles for STACKUNI caches.



6.2.3 Harvard Cache and Stack Cache (STACKHARVARD)

Adding a stack cache to a Harvard cache increased the miss rate for the data cache (see Table 6.7), much more than the corresponding unified cache miss rate increase. The removal of stack accesses once instruction accesses have already been moved leaves the data cache with less local memory accesses.

Method for Calculating Memory Access Time for STACKHARVARD Caches

$$\begin{aligned} T_{cycles} &= I_{ratio} \times (T_{L1} + I_{miss} \times T_{mem}) + \\ &\quad S_{ratio} \times (T_{stack} + S_{miss} \times T_{mem}) + \\ &\quad (D_{ratio} - S_{ratio}) \times (T_{L1} + D_{miss} \times T_{mem}) \end{aligned}$$

Figure 6.6: Method for calculating average memory access time in cycles for STACKHARVARD caches.

Figure 6.6 shows the formula used to calculate the average memory access times. De-

STACKHARVARD **Cache Miss Rates and Access Times**

Instruction, Data Size (KB)	**Stack + Harvard**				**Harvard**		
	stack miss %	**instr. miss %**	**data miss %**	**Time (cycles)**	**instr. miss %**	**data miss %**	**Time (cycles)**
Direct Mapped Harvard Cache							
1	<0.00%	1.00%	29.74%	8.91	1.00%	23.49%	10.43
2	<0.00%	0.62%	21.12%	6.83	0.62%	15.98%	7.72
4	<0.00%	0.35%	13.95%	5.10	0.35%	10.15%	5.63
8	<0.00%	0.20%	7.16%	3.48	0.20%	5.08%	3.82
16	<0.00%	0.07%	4.13%	2.75	0.07%	2.93%	3.04
32	<0.00%	0.03%	2.62%	2.39	0.03%	1.84%	2.65
64	<0.00%	0.02%	2.01%	2.24	0.02%	1.39%	2.49
128	<0.00%	0.01%	1.60%	2.14	0.01%	1.07%	2.38
Eight-way Associative Harvard Cache							
1	<0.00%	0.57%	19.75%	7.41	0.57%	14.90%	8.36
2	<0.00%	0.28%	10.14%	4.78	0.28%	7.54%	5.35
4	<0.00%	0.08%	4.87%	3.33	0.08%	3.48%	3.69
8	<0.00%	0.02%	2.81%	2.77	0.02%	1.97%	3.08
16	<0.00%	0.01%	1.78%	2.49	0.01%	1.21%	2.77
32	<0.00%	<0.00%	1.34%	2.37	<0.00%	0.90%	2.64
64	<0.00%	<0.00%	1.14%	*2.32*	<0.00%	0.75%	*2.58*
128	<0.00%	<0.00%	1.02%	*2.29*	<0.00%	0.68%	*2.55*

Table 6.7: Miss rates and average access times for a Harvard cache augmented with a 256 byte direct mapped write through, write around stack cache. The size listed is the size for the instruction and data caches individually. Eight-way associative data cache access times slower than the corresponding direct mapped cache time are italicized. The three rightmost columns reproduce the results from Table 5.7.

spite the higher data cache miss rate, the addition of a stack cache reduces the access time for all cache sizes.

6.2.4 Summary

The inclusion of a stack cache improves the average memory access time for both unified and Harvard caches. Despite having the same access latency as the register buffer, the addition of a stack cache does not result in as large a performance boost. The benefits of the stack cache, with its extremely low miss rate, are tempered by stack accesses comprising only 25% of total memory accesses. In contrast, the register buffer sees 70% of all memory accesses; its one cycle access time is used much more frequently. Although the register buffer has a much higher miss rate, it is backed up by a level one cache (unified or data). The penalty for missing the register buffer on a memory access is much smaller than the penalty for missing the stack cache.

Figure 6.7 compares the performance of unified and Harvard caches with each other when a stack cache is used. As with unified and Harvard caches, STACKUNI has a faster access time than STACKHARVARD when the cache size is less than 16K. For 16K and larger caches the separation of instructions into a dedicated cache is worthwhile—supporting the *16K Instruction Rule* presented in Chapter 5.

The STACKUNI cache outperforms the Harvard cache for caches smaller than 16K. As with the register buffer, an optimization for data accesses is more valuable than an optimization for instruction accesses. Once the Harvard cache's data cache is larger than 8K (16K total Harvard cache size) it outperforms the STACKHARVARD cache since it then has the space to hold the stack accesses as well as the other data accesses without too much conflict.

6.3 Constant Cache

Anywhere from 2-11% of memory accesses hit the JVM's constant pools. In addition to storing constant data values, these pools contain the lookup tables for methods and instance variable offsets. A constant pool in a class may not be all that large for most programs, and might be efficiently buffered in a small dedicated direct mapped constant cache. This could create a relative locality of reference amongst constant pool accesses since most of the constant data would be in the cache.

First we tested a constant cache in isolation to get an idea of the locality of constant accesses. A constant cache was then added to both unified and Harvard caches to determine if any performance boost was added by isolating constant values. The latency of accessing the constant cache was set to two clock cycles since the cache sizes are larger than the register buffer and stack cache sizes. Only constant pool data accesses were requested from

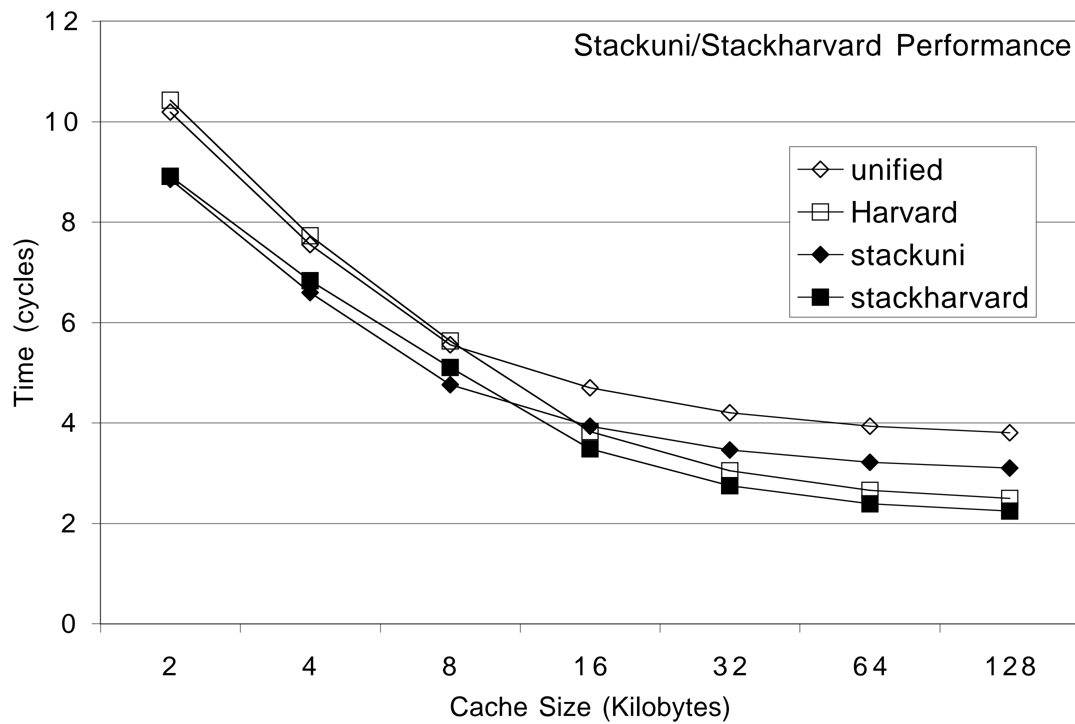
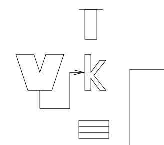


Figure 6.7: Comparison of direct mapped STACKUNI and STACKHARVARD.

the constant cache.



6.3.1 Isolated Constant Cache

After some preliminary experiments we found that extremely small constant caches were not performing well. Therefore, we extended the upper size limit on constant caches to 2K. The caches were direct mapped with 16 byte cache lines. For the larger caches 8 bytes was absurdly short; for the smaller caches 32 bytes would have been equally ridiculous. The write policy used was write back, write fetch, a decision that deserves an explanation.

Constant pool data is only written to once, when it is initialized. Therefore, the choice of whether or not to use a write through or write back policy is inconsequential: writes will be infrequent. Write fetch is normally accompanied by write back, so write back was chosen over write through.

The write fetch write miss policy fetches the address in memory when it is written to. Since all of the accesses directed to the constant cache are constant pool accesses, any write, miss or otherwise, is an initialization write. Initialization frequently precedes usage. For example, when the JVM during the translation of a program finds an unloaded class referenced, it goes and loads the missing class. Once it is loaded execution proceeds with using that class. Similarly, the first time a method is executed it needs to be resolved, that is, initialized in the constant pool. Following on a write miss there is a reasonable chance that whatever was just initialized is going to be used immediately. As such, fetching the memory value into the constant cache prepares for that possibility.

Average Constant Cache Miss Rates and Access Times

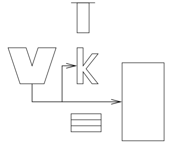
Size (bytes)	Miss Rate	Time (cycles)
64	26.93%	47.42
128	20.33%	47.17
256	15.70%	47.00
512	10.76%	46.81
1024	5.88%	46.63
2048	4.38%	46.58
Frequency: 7.47%		

Table 6.8: Miss rates and access times for an average of Java benchmarks. The frequency given is the percentage of total memory accesses passed to the cache.

Increasing the cache size from 64 bytes to 1024 bytes makes a big improvement in the miss rate of the cache (see Table 6.8). Moving from 1K to 2K does not yield such a large improvement. The experiments that use a constant cache use a 512 byte cache identical to

that used in the isolated constant cache experiment. The size was chosen so that it would always be smaller than the unified and Harvard caches being augmented.

The impact of the constant cache on average memory access time is limited by the number of constant pool accesses made. On average, only 7% of memory accesses occur within a constant pool.



6.3.2 Unified Cache and Constant Cache (CONSTUNI)

The miss rates for a unified cache augmented by a constant cache are shown in Table 6.9. Like the addition of a stack cache, the miss rates for the unified cache are virtually unchanged. This is hardly surprising given that constant accesses only make up 7% of all memory accesses. However, moving constant pool accesses to a separate cache *improves*, that is, *reduces* the miss rate for the unified cache. This indicates that constant accesses have particularly low locality of reference, leading us to the *Constant Locality Rule:*

> *Constant Locality Rule:* Accessing constant pool data has extremely low locality of reference due to infrequent use of constants and the division of constant data by `class` (each `class` has its own constant pool).

Removing constant accesses from the general unified cache reduced cache pollution.

Figure 6.8 gives the formula used to calculate the average memory access times shown. The average memory access times for the CONSTUNI caches are initially better than the unified cache access times. However, as the unified cache increases in size and its miss rate decreases, the relatively high miss rate of the constant cache becomes a handicap rather than an advantage. When the unified cache is 16K in size (8K for the eight-way associative unified cache), the access time is nearly identical with or without the constant cache. For larger caches, the unified caches without the constant cache have a slightly better access time. Table 6.9 marks these cases with a '*'.

Method for Calculating Memory Access Time for CONSTUNI Caches

$$\begin{aligned} T_{cycles} &= I_{ratio} \times (T_{L1} + U_{miss} \times T_{mem}) + \\ &\quad C_{ratio} \times (T_{L1} + C_{miss} \times T_{mem}) + \\ &\quad (D_{ratio} - C_{ratio}) \times (T_{L1} + U_{miss} \times T_{mem}) \end{aligned}$$

where

C_{ratio} = ratio of constant pool accesses to total number of accesses

C_{miss} = constant cache miss rate

Figure 6.8: Method for calculating average memory access time in cycles for CONSTUNI caches.

CONSTUNI **Cache Miss Rates and Access Times**

Unified Size (KB)	**Constant + Unified**			**Unified**	
	constant miss %	**unified miss %**	**Time (cycles)**	**unified miss %**	**Time (cycles)**
Direct Mapped Unified Cache					
1	10.76%	20.34%	13.07	21.22%	14.02
2	10.76%	12.22%	9.32	13.55%	10.19
4	10.76%	7.05%	6.92	8.27%	7.55
8	10.76%	3.73%	5.39	4.29%	5.56
16	10.76%	2.23%	*4.69	2.57%	4.69
32	10.76%	1.45%	*4.34	1.58%	4.20
64	10.76%	0.95%	*4.10	1.04%	3.93
128	10.76%	0.76%	*4.01	0.79%	3.80
Eight-way Associative Unified Cache					
1	10.76%	11.54%	10.26	12.79%	11.18
2	10.76%	5.92%	7.30	6.76%	7.74
4	10.76%	2.71%	5.60	3.15%	5.68
8	10.76%	1.47%	*4.95	1.67%	4.84
16	10.76%	0.88%	*4.64	0.98%	4.45
32	10.76%	0.62%	**4.50*	0.67%	*4.27*
64	10.76%	0.52%	**4.45*	0.53%	*4.19*
128	10.76%	0.46%	**4.42*	0.47%	*4.15*

Table 6.9: Miss rates and average access times for a unified cache augmented with a 512 byte direct mapped write back, write fetch constant cache. Eight-way associative unified cache access times slower than the corresponding direct mapped cache times are italicized. The two rightmost columns reproduce the results from Table 5.7. Cache sizes for which the addition of a constant cache results in a slower access time are marked by '*'.

6.3.3 Harvard Cache and Constant Cache (CONSTHARVARD)

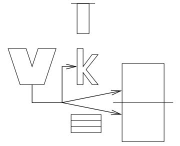


Table 6.10 lists the miss rates for a Harvard cache with and without a constant cache. Like the CONSTUNI cache, the data cache augmented by a constant cache has a slightly better miss rate than the data cache without the constant cache. The instruction cache is unchanged since it only receives instruction accesses. The formula used to calculate the average access time is given in Figure 6.9.

Method for Calculating Memory Access Time for CONSTHARVARD Caches

$$\begin{aligned} T_{cycles} &= I_{ratio} \times (T_{L1} + I_{miss} \times T_{mem}) + \\ &\quad C_{ratio} \times (T_{L1} + C_{miss} \times T_{mem}) + \\ &\quad (D_{ratio} - C_{ratio}) \times (T_{L1} + D_{miss} \times T_{mem}) \end{aligned}$$

Figure 6.9: Method for calculating average memory access time in cycles for CONSTHARVARD caches.

The average memory access times also behave similarly as they did with the CONSTUNI cache. Initially, they are better for the CONSTHARVARD cache. This is expected, given that the constant cache filters some cache pollution out of the critical data cache and that the CONSTHARVARD cache has a 512 byte size advantage over the Harvard cache. Larger CONSTHARVARD caches are penalized by the high miss rate of the constant cache, leading to an access time slightly worse than that of the Harvard cache.

6.3.4 Summary

Figure 6.10 pictorially supports the observations made above. An initial advantage given by the constant cache is eventually a disadvantage. While the cache size is less than 8K, CONSTUNI and CONSTHARVARD are faster than unified and Harvard caches. At sizes larger than 8K unified and Harvard caches without constant caches are faster. The usefulness of separating out constant pool data into a dedicated cache is questionable, unless the cache is larger than that used in these experiments. However, allocating a lot of precious hardware resources to a cache that is only used 7% of the time is not worthwhile according to *Amdahl's Law:*

> The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used [26].

This principle was first expressed by Amdahl in a paper addressing the limits of parallel computation [3].

CONSTUNI and CONSTHARVARD caches do not quite follow the *16K Instruction Rule*: the size at which separating instructions becomes valuable is 8K, not 16K. This is due to the

CONSTHARVARD **Cache Miss Rates and Access Times**

Instruction, Data Size (KB)	**Constant + Harvard**				**Harvard**		
	constant miss %	**instr. miss %**	**data miss %**	**Time (cycles)**	**instr. miss %**	**data miss %**	**Time (cycles)**
Direct Mapped Harvard Cache							
1	10.76%	1.00%	22.48%	9.63	1.00%	23.49%	10.43
2	10.76%	0.62%	14.53%	7.07	0.62%	15.98%	7.72
4	10.76%	0.35%	8.74%	5.21	0.35%	10.15%	5.63
8	10.76%	0.20%	4.41%	*3.82	0.20%	5.08%	3.82
16	10.76%	0.07%	2.53%	*3.21	0.07%	2.93%	3.04
32	10.76%	0.03%	1.72%	*2.95	0.03%	1.84%	2.65
64	10.76%	0.02%	1.31%	*2.82	0.02%	1.39%	2.49
128	10.76%	0.01%	1.06%	*2.74	0.01%	1.07%	2.38
Eight-way Associative Harvard Cache							
1	10.76%	0.57%	13.48%	7.68	0.57%	14.90%	8.36
2	10.76%	0.28%	6.62%	5.16	0.28%	7.54%	5.35
4	10.76%	0.08%	3.01%	*3.83	0.08%	3.48%	3.69
8	10.76%	0.02%	1.77%	*3.38	0.02%	1.97%	3.08
16	10.76%	0.01%	1.15%	*3.15	0.01%	1.21%	2.77
32	10.76%	<0.00%	0.90%	_*3.06_	<0.00%	0.90%	2.64
64	10.76%	<0.00%	0.76%	_*3.01_	<0.00%	0.75%	_2.58_
128	10.76%	<0.00%	0.70%	_*2.99_	<0.00%	0.68%	_2.55_

Table 6.10: Miss rates and average access times for a Harvard cache augmented with a 512 byte direct mapped write back, write fetch constant cache. The size listed is the size for the instruction and data caches individually. Eight-way associative data cache access times slower than the corresponding direct mapped cache time are italicized. The three rightmost columns reproduce the results from Table 5.7. Cache sizes for which the addition of a constant cache results in a slower access time are marked by '*'.

improved data cache miss rate (thanks to the removal of constant pool accesses and cache pollution).

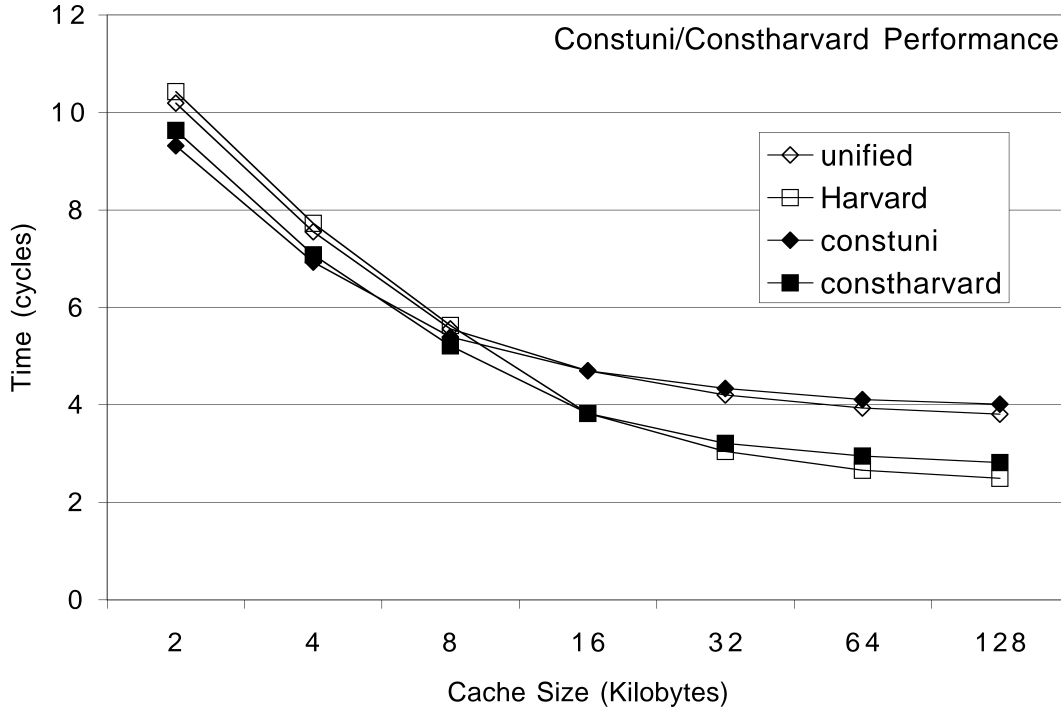
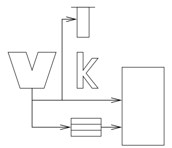


Figure 6.10: Comparison of CONSTUNI and CONSTHARVARD with UNIFIED and HARVARD (direct mapped unified and Harvard caches).

6.4 Mix and Match Caches: An Evaluation

This section looks at the various combinations of alternative caching configurations. Although Section 6.3 showed that the constant cache is not an improvement for medium to large caches, it is included as a possibility in these results for completeness. The formulas used to calculate the mean memory access times for the various configurations follow the principles used earlier in the chapter and are summarized in Figure 6.11. Stack and register caches have a one cycle latency; all other caches are assigned a two cycle access latency.



6.4.1 Unified, Stack, and Register Caches (STACKUNIREG)

Table 6.11 lists the miss rates for the caches comprising the STACKUNIREG configuration. As expected, the miss rates for the register buffer and for the unified cache are higher with

Formulas for Calculating Memory Access Time for Hybrid Caches

CONSTHARREG

$$\begin{aligned} T_{cycles} &= I_{ratio} \times (T_{L1} + I_{miss} \times T_{mem}) + \\ &\quad C_{ratio} \times (T_{L1} + C_{miss} \times T_{mem}) + \\ &\quad (D_{ratio} - C_{ratio}) \times (T_{reg} + R_{miss} \times (T_{L1} + D_{miss} \times T_{mem})) \end{aligned}$$

CONSTSTACKHARREG

$$\begin{aligned} T_{cycles} &= I_{ratio} \times (T_{L1} + I_{miss} \times T_{mem}) + \\ &\quad C_{ratio} \times (T_{L1} + C_{miss} \times T_{mem}) + \\ &\quad S_{ratio} \times (T_{stack} + S_{miss} \times T_{mem}) + \\ &\quad (D_{ratio} - C_{ratio} - S_{ratio}) \times (T_{reg} + R_{miss} \times (T_{L1} + D_{miss} \times T_{mem})) \end{aligned}$$

CONSTSTACKHARVARD

$$\begin{aligned} T_{cycles} &= I_{ratio} \times (T_{L1} + I_{miss} \times T_{mem}) + \\ &\quad C_{ratio} \times (T_{L1} + C_{miss} \times T_{mem}) + \\ &\quad S_{ratio} \times (T_{stack} + S_{miss} \times T_{mem}) + \\ &\quad (D_{ratio} - C_{ratio} - S_{ratio}) \times (T_{L1} + D_{miss} \times T_{mem}) \end{aligned}$$

CONSTSTACKUNI

$$\begin{aligned} T_{cycles} &= I_{ratio} \times (T_{L1} + U_{miss} \times T_{mem}) + \\ &\quad C_{ratio} \times (T_{L1} + C_{miss} \times T_{mem}) + \\ &\quad S_{ratio} \times (T_{stack} + S_{miss} \times T_{mem}) + \\ &\quad (D_{ratio} - C_{ratio} - S_{ratio}) \times (T_{L1} + U_{miss} \times T_{mem}) \end{aligned}$$

CONSTUNIREG

$$\begin{aligned} T_{cycles} &\quad I_{ratio} \times (T_{L1} + U_{miss} \times T_{mem}) + \\ &\quad C_{ratio} \times (T_{L1} + C_{miss} \times T_{mem}) + \\ &\quad (D_{ratio} - C_{ratio}) \times (T_{reg} + R_{miss} \times (2T_{L1} + U_{miss} \times T_{mem})) \end{aligned}$$

STACKHARREG

$$\begin{aligned} T_{cycles} &= I_{ratio} \times (T_{L1} + I_{miss} \times T_{mem}) + \\ &\quad S_{ratio} \times (T_{stack} + S_{miss} \times T_{mem}) + \\ &\quad (D_{ratio} - S_{ratio}) \times (T_{reg} + R_{miss} \times (T_{L1} + D_{miss} \times T_{mem})) \end{aligned}$$

STACKUNIREG

$$\begin{aligned} T_{cycles} &= I_{ratio} \times (T_{L1} + U_{miss} \times T_{mem}) + \\ &\quad S_{ratio} \times (T_{stack} + S_{miss} \times T_{mem}) + \\ &\quad (D_{ratio} - S_{ratio}) \times (T_{reg} + R_{miss} \times (T_{L1} + U_{miss} \times T_{mem})) \end{aligned}$$

Figure 6.11: Formulas for calculating average memory access time in cycles for hybrid caches.

the addition of a stack cache. The highly local stack accesses significantly contributed to the hit rate of the data caches and register caches. For purposes of comparison, the access times for STACKUNI and UNIREG caches are included in Table 6.11.

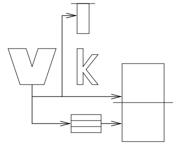
STACKUNIREG **Cache Miss Rates and Access Times**

	Stack + Register + Unified				**Stack + Unified**	**Register + Unified**
Unified Size (KB)	**stack miss %**	**register miss %**	**unified miss %**	**Time (cycles)**		
Direct Mapped Unified Cache						
1	<0.00%	35.45%	24.65%	7.66	11.20	9.46
2	<0.00%	35.34%	18.75%	6.29	8.85	6.78
4	<0.00%	35.34%	12.15%	4.76	6.60	5.08
8	<0.00%	35.34%	7.28%	3.64	4.76	3.64
16	<0.00%	35.34%	4.68%	3.04	3.93	3.05
32	<0.00%	35.34%	2.83%	2.61	3.46	2.66
64	<0.00%	35.34%	1.85%	2.39	3.21	2.43
128	<0.00%	35.34%	1.37%	2.28	3.10	2.33
Eight-way Associative Unified Cache						
1	<0.00%	35.37%	21.79%	*7.97*	9.65	*9.69*
2	<0.00%	35.34%	12.69%	5.57	6.60	6.52
4	<0.00%	35.34%	6.40%	3.92	4.79	4.13
8	<0.00%	35.34%	3.53%	3.16	4.00	3.20
16	<0.00%	35.34%	2.10%	2.79	3.63	2.81
32	<0.00%	35.34%	1.40%	*2.60*	*3.46*	2.64
64	<0.00%	35.34%	1.14%	*2.53*	*3.39*	*2.58*
128	<0.00%	35.34%	1.00%	*2.50*	*3.35*	*2.54*

Table 6.11: Miss rates and average access times for a unified cache augmented with a register buffer and a stack cache. Eight-way associative unified cache access times slower than the corresponding direct mapped cache times are italicized. The two rightmost columns summarize the results from Tables 6.6 and 6.2.

The addition of a register cache significantly improves cache performance compared to the STACKUNIFIED configuration. This is not surprising given the performance boost added by a register buffer. Compared to the UNIREG cache the STACKUNIREG cache is faster. As the size of the caches involved increases, the difference decreases. For a 128K direct mapped UNIREG cache, adding a stack cache only shaves 0.05 clock cycles off of the average memory access time. The eight-way associative UNIREG cache experiences the

improvement slowdown at 16K.

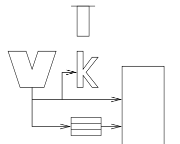


6.4.2 Harvard, Stack, and Register Caches (STACKHARREG)

The effects of adding a stack cache to a register buffered Harvard cache can be seen in Table 6.12. As with the STACKUNIREG cache, the register cache miss rate jumps to roughly 35% with the addition of a stack cache. Deprived of both stack accesses and instruction accesses, the data cache backing up the register buffer has very high miss rates, especially for small caches.

Compared to the STACKHARVARD cache, the STACKHARREG cache has a significantly faster access time. This is not the case when compared to the HARREG cache. For almost all cache sizes the HARREG cache performs better than a STACKHARREG cache (marked by '*' in Table 6.12). Given that both the stack cache and the register cache have access times of one cycle, it is surprising that the addition of a stack cache slows down the access time.

A possible explanation is that by isolating stack accesses and increasing the miss rate of the register cache and the data cache, STACKHARREG gains stack access performance at the expense of other data access speed. STACKUNIREG does not suffer this problem since the hit rate of the unified cache is helped by the inclusion of instruction accesses.



6.4.3 Unified, Constant, and Register Caches (CONSTUNIREG)

Table 6.13 lists the results obtained when a constant cache was added to a UNIREG cache. Separating the constant pool accesses from the data accesses sent to the register cache improves the miss rate for the register buffer and for the unified cache (although the improvement for the unified cache is small; see Table 6.2). This supports the *Constant Locality Rule*. Since constant pool accesses comprise 7% of all memory accesses, these accesses have low temporal locality compared to the whole pool of memory accesses. Isolating the constant accesses removes cache pollution from the register buffer.

Average memory access times are also better for the CONSTUNIREG cache than either the CONSTUNI cache or the UNIREG cache—for small unified caches. Direct mapped UNIREG caches larger than 4K perform better than similarly sized CONSTUNIREG caches. The performance of the unified and register caches in CONSTUNIREG are aided by the removal of constant pool accesses; however, the constant cache has a miss rate over 10%. When the register and unified caches are yielding a fast average access time, the asymptotic time for the whole configuration is determined by the high miss rate (and consequent high average access time) of the constant cache.

STACKHARREG **Cache Miss Rates and Access Times**

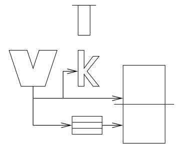
Instruction, Data Size (KB)	**Stack + Register + Harvard**					**Stack + Harvard**	**Register + Harvard**
	stack miss %	**register miss %**	**instr. miss %**	**data miss %**	**Time (cycles)**		
Direct Mapped Harvard Cache							
1	<0.00%	35.40%	1.00%	54.82%	6.34	8.91	6.35
2	<0.00%	35.34%	0.62%	41.72%	*5.19	6.83	4.87
4	<0.00%	35.34%	0.35%	27.71%	*3.98	5.10	3.77
8	<0.00%	35.34%	0.20%	16.16%	*3.00	3.48	2.71
16	<0.00%	35.34%	0.07%	10.29%	*2.49	2.75	2.30
32	<0.00%	35.34%	0.03%	6.61%	*2.18	2.39	2.07
64	<0.00%	35.34%	0.02%	4.93%	*2.04	2.24	1.96
128	<0.00%	35.34%	0.01%	3.85%	*1.95	2.14	1.89
Eight-way Associative Harvard Cache							
1	<0.00%	35.36%	0.57%	47.99%	*6.50*	7.41	*6.95*
2	<0.00%	35.34%	0.28%	27.15%	4.48	4.78	4.57
4	<0.00%	35.34%	0.08%	13.84%	*3.18	3.33	3.00
8	<0.00%	35.34%	0.02%	8.34%	*2.65	2.77	2.47
16	<0.00%	35.34%	0.01%	5.19%	*2.35	2.49	2.24
32	<0.00%	35.34%	<0.00%	3.88%	_*2.22_	2.37	*2.13*
64	<0.00%	35.34%	<0.00%	3.31%	_*2.17_	*2.32*	*2.09*
128	<0.00%	35.34%	<0.00%	2.97%	_*2.14_	*2.29*	*2.07*

Table 6.12: Miss rates and average access times for a Harvard cache augmented with a register buffer and a stack cache. Eight-way associative Harvard cache access times slower than the corresponding direct mapped cache times are italicized. The two rightmost columns summarize the results from Tables 6.7 and 6.3. Where STACKHARREG has a slower access time than HARREG the time is marked with a '*'.

CONSTUNIREG **Cache Miss Rates and Access Times**

	Constant + Register + Unified				**Constant + Unified**	**Register + Unified**
Unified Size (KB)	**constant miss %**	**register miss %**	**unified miss %**	**Time (cycles)**		
Direct Mapped Unified Cache						
1	10.76%	23.57%	30.42%	9.11	13.07	9.46
2	10.76%	23.27%	19.48%	6.66	9.32	6.78
4	10.76%	23.27%	11.99%	5.01	6.92	5.08
8	10.76%	23.27%	6.11%	*3.71	5.39	3.64
16	10.76%	23.27%	3.78%	*3.19	4.69	3.05
32	10.76%	23.27%	2.53%	*2.92	4.34	2.66
64	10.76%	23.27%	1.57%	*2.70	4.10	2.43
128	10.76%	23.27%	1.03%	*2.59	4.01	2.33
Eight-way Associative Unified Cache						
1	10.76%	23.48%	25.72%	_9.19_	10.26	_9.69_
2	10.76%	23.27%	15.41%	*6.57	7.30	6.52
4	10.76%	23.27%	6.41%	*4.30	5.60	4.13
8	10.76%	23.27%	3.56%	*3.58	4.95	3.20
16	10.76%	23.27%	1.99%	*3.19	4.64	2.81
32	10.76%	23.27%	1.32%	_*3.02_	_4.50_	2.64
64	10.76%	23.27%	0.91%	_*2.92_	_4.45_	_2.58_
128	10.76%	23.27%	0.80%	_*2.89_	_4.42_	_2.54_

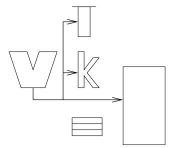
Table 6.13: Miss rates and average access times for a unified cache augmented with a register buffer and a constant cache. Eight-way associative unified cache access times slower than the corresponding direct mapped cache times are italicized. The two rightmost columns summarize the results from Tables 6.9 and 6.2. Where CONSTUNIREG has a slower access time than UNIREG the time is marked with a '*'.



6.4.4 Harvard, Constant, and Register Caches (CONSTHARREG)

Adding a constant cache to a HARREG cache had the same effect as adding a constant cache to a UNIREG cache. Table 6.14 shows that the register buffer and the data cache have lower miss rates with the addition of a constant cache. The removal of constant accesses from these caches reduces cache pollution.

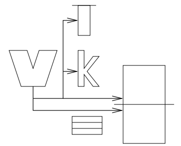
For small caches the addition of a constant cache improves the average memory access time. As the size of the data cache increases and the performance of the data cache improves, however, the average access time is more and more impacted by the high miss rate of the constant cache. For large caches the HARREG cache outperforms the CONSTHARREG cache.



6.4.5 Unified, Constant, and Stack Caches (CONSTSTACKUNI)

The CONSTSTACKUNI cache considers the effect of non-register auxiliary caches. If very simple caches are desired, the eight-way associativity of the register buffer might be undesirable. Table 6.15 shows the results for the cache compared with the CONSTUNI and STACKUNI caches. Referring back to Tables 6.6 and 6.9 we see that the CONSTSTACKUNI configuration's unified cache has a miss rate between that of the CONSTUNI and STACKUNI configurations. As with the other cache configurations containing a constant cache, the low locality constant accesses are isolated, removing cache pollution from the unified cache.

The access times for the CONSTSTACKUNI cache are faster than the access times for the CONSTUNI cache for all cache sizes. Compared to the STACKUNI cache, the access times are faster for smaller unified caches. Again we see that the high miss rate of the constant cache comes to dominate the access time for the configuration for large unified caches.



6.4.6 Harvard, Constant, and Stack Caches (CONSTSTACKHARVARD)

As with the CONSTSTACKUNI experiment the CONSTSTACKHARVARD experiment does not use a register buffer to improve data memory accesses. Comparing Tables 6.7 and 6.10 to Table 6.16 we see that the combination of a constant cache and a stack cache results in a data cache miss rate between that of a CONSTHARVARD data cache miss rate and STACKHARVARD data cache miss rate.

The CONSTSTACKHARVARD cache has a better average memory access time than the CONSTHARVARD cache for all cache sizes. For small caches the CONSTSTACKHARVARD cache has a faster access time than the STACKHARVARD cache; for larger caches the STACKHARVARD cache has a slightly faster access time.

CONSTHARREG **Cache Miss Rates and Access Times**

	Constant + Register + Harvard					**Const + Harvard**	**Register + Harvard**
Instruction, Data Size (KB)	**stack miss %**	**register miss %**	**instr. miss %**	**data miss %**	**Time (cycles)**		
Direct Mapped Harvard Cache							
1	10.76%	23.56%	1.00%	50.13%	5.94	9.63	6.35
2	10.76%	23.27%	0.62%	35.32%	4.75	7.07	4.87
4	10.76%	23.27%	0.35%	22.34%	3.75	5.21	3.77
8	10.76%	23.27%	0.20%	10.34%	*2.85	3.82	2.71
16	10.76%	23.27%	0.07%	6.18%	*2.52	3.21	2.30
32	10.76%	23.27%	0.03%	4.27%	*2.38	2.95	2.07
64	10.76%	23.27%	0.02%	3.13%	*2.30	2.82	1.96
128	10.76%	23.27%	0.01%	2.13%	*2.22	2.74	1.89
Eight-way Associative Harvard Cache							
1	10.76%	23.53%	0.57%	46.54%	_6.39_	7.68	_6.95_
2	10.76%	23.27%	0.28%	26.33%	*4.60	5.16	4.57
4	10.76%	23.27%	0.08%	10.46%	*3.24	3.83	3.00
8	10.76%	23.27%	0.02%	6.47%	_*2.90_	3.38	2.47
16	10.76%	23.27%	0.01%	3.61%	_*2.66_	3.15	2.24
32	10.76%	23.27%	<0.00%	2.72%	_*2.58_	_3.06_	_2.13_
64	10.76%	23.27%	<0.00%	2.01%	_*2.52_	_3.01_	_2.09_
128	10.76%	23.27%	<0.00%	1.71%	_*2.50_	_2.99_	_2.07_

Table 6.14: Miss rates and average access times for a Harvard cache augmented with a register buffer and a constant cache. Eight-way associative Harvard cache access times slower than the corresponding direct mapped cache times are italicized. The two rightmost columns summarize the results from Tables 6.10 and 6.3. Where CONSTHARREG has a slower access time than HARREG the time is marked with a ’*’.

CONSTSTACKUNI **Cache Miss Rates and Access Times**

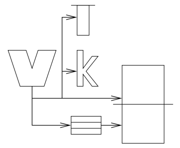
Unified Size (KB)	**Constant + Stack + Unified**				**Stack + Unified**	**Constant + Unified**
	stack miss %	**constant miss %**	**unified miss %**	**Time (cycles)**		
Direct Mapped Unified Cache						
1	<0.00%	10.76%	21.09%	10.24	11.20	13.07
2	<0.00%	10.76%	14.54%	7.98	8.85	9.32
4	<0.00%	10.76%	8.75%	5.98	6.60	6.92
8	<0.00%	10.76%	4.74%	4.60	4.76	5.39
16	<0.00%	10.76%	2.83%	*3.94	3.93	4.69
32	<0.00%	10.76%	1.84%	*3.60	3.46	4.34
64	<0.00%	10.76%	1.24%	*3.39	3.21	4.10
128	<0.00%	10.76%	1.02%	*3.31	3.10	4.01
Eight-way Associative Unified Cache						
1	<0.00%	10.76%	13.89%	8.84	9.65	10.26
2	<0.00%	10.76%	7.17%	6.20	6.60	7.30
4	<0.00%	10.76%	3.45%	4.73	4.79	5.60
8	<0.00%	10.76%	1.90%	*4.12	4.00	4.95
16	<0.00%	10.76%	1.16%	*3.83	3.63	4.64
32	<0.00%	10.76%	0.83%	**3.70*	*3.46*	*4.50*
64	<0.00%	10.76%	0.71%	**3.65*	*3.39*	*4.45*
128	<0.00%	10.76%	0.63%	**3.62*	*3.35*	*4.42*

Table 6.15: Miss rates and average access times for a unified cache augmented with a stack cache and a constant cache. Eight-way associative unified cache access times slower than the corresponding direct mapped cache times are italicized. The two rightmost columns summarize the results from Tables 6.6 and 6.9. Where CONSTSTACKUNI has a slower access time than STACKUNI the time is marked with a '*'.

CONSTSTACKHARVARD **Cache Miss Rates and Access Times**

	Constant + Stack + Harvard					**Const + Harvard**	**Stack + Harvard**
Instruction, Data Size (KB)	**stack miss %**	**constant miss %**	**instr. miss %**	**data miss %**	**Time (cycles)**		
Direct Mapped Harvard Cache							
1	<0.00%	10.76%	1.00%	29.41%	8.14	9.14	8.91
2	<0.00%	10.76%	0.62%	19.80%	6.18	7.06	6.83
4	<0.00%	10.76%	0.35%	12.47%	4.69	5.33	5.10
8	<0.00%	10.76%	0.20%	6.50%	3.48	3.71	3.48
16	<0.00%	10.76%	0.07%	3.75%	*2.92	2.98	2.75
32	<0.00%	10.76%	0.03%	2.57%	*2.68	2.62	2.39
64	<0.00%	10.76%	0.02%	1.99%	*2.56	2.47	2.24
128	<0.00%	10.76%	0.01%	1.67%	*2.50	2.38	2.14
Eight-way Associative Harvard Cache							
1	<0.00%	10.76%	0.57%	18.82%	6.81	8.50	7.41
2	<0.00%	10.76%	0.28%	9.35%	4.63	5.72	4.78
4	<0.00%	10.76%	0.08%	4.46%	*3.49	4.17	3.33
8	<0.00%	10.76%	0.02%	2.69%	*3.08	3.24	2.77
16	<0.00%	10.76%	0.01%	1.76%	*2.87	2.81	2.49
32	<0.00%	10.76%	<0.00%	1.40%	**2.79*	*2.67*	2.37
64	<0.00%	10.76%	<0.00%	1.22%	**2.75*	*2.61*	*2.32*
128	<0.00%	10.76%	<0.00%	1.19%	**2.72*	*2.56*	*2.29*

Table 6.16: Miss rates and average access times for a Harvard cache augmented with a stack cache and a constant cache. Eight-way associative Harvard cache access times slower than the corresponding direct mapped cache times are italicized. The two rightmost columns summarize the results from Tables 6.7 and 6.10. Where CONSTSTACKHARVARD has a slower access time than STACKHARVARD the time is marked with a '*'.



6.4.7 Harvard, Constant, Stack, and Register Caches (CONSTSTACKHARREG)

For completeness, Table 6.17 shows the results when simulating cache performance for a configuration with all of the experimental caches added to a Harvard cache. The closest comparable configurations are CONSTHARREG, CONSTSTACKHARVARD, and STACKHARREG. Only the fastest of the three, STACKHARREG, was included in Table 6.17 (for space reasons).

The miss rates for the register and data caches are slightly better than those listed in Table 6.12 for STACKHARREG. This is further proof of the *Constant Locality Rule*; removing constant accesses from the register and data caches improves their miss rates.

In terms of average memory access times, the CONSTSTACKHARREG cache outperforms STACKHARREG for caches smaller than 8K (4K for eight-way associative CONSTSTACKHARREG). For caches larger than 8K the CONSTSTACKHARREG cache has a slightly slower access time than the STACKHARREG cache.

6.4.8 The Bottom Line

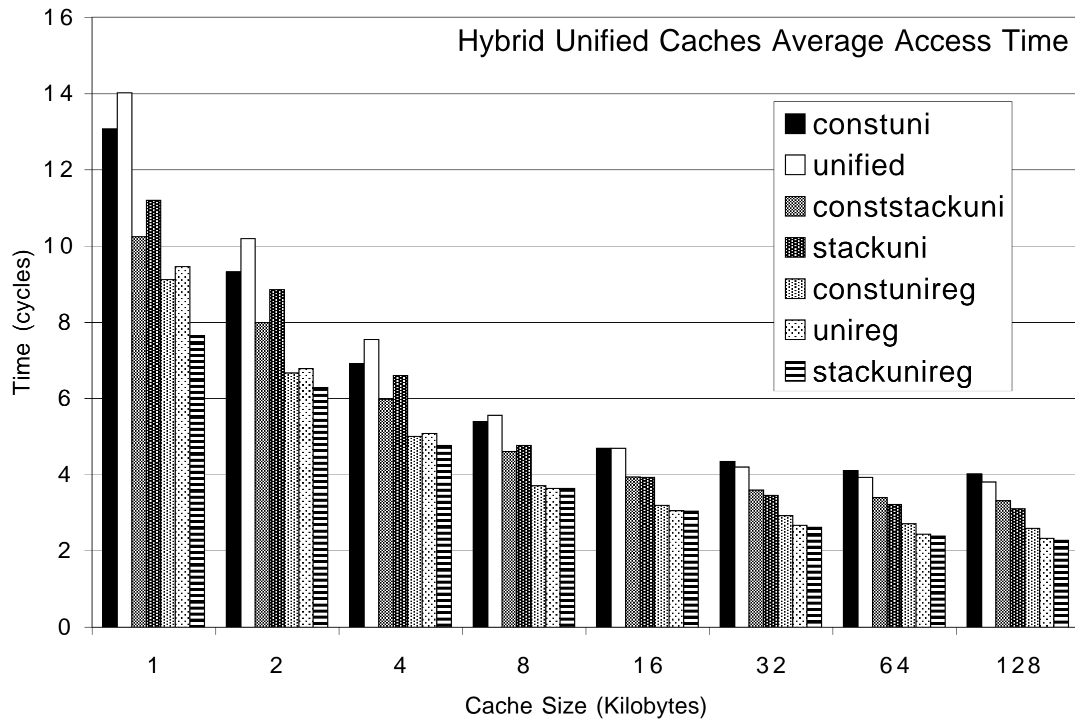


Figure 6.12: Comparison of average memory access times for unified cache variations.

ConstStackHarReg **Cache Miss Rates and Access Times**

Instruction, Data Size (KB)	**Constant + Stack + Register + Harvard**						StackHarReg
	constant miss %	**stack miss %**	**register miss %**	**instr. miss %**	**data miss %**	**Time (cycles)**	
Direct Mapped Harvard Cache							
1	10.76%	<0.00%	33.91%	1.00%	55.02%	5.88	6.34
2	10.76%	<0.00%	33.65%	0.62%	41.18%	4.87	5.19
4	10.76%	<0.00%	33.65%	0.35%	25.93%	3.82	3.98
8	10.76%	<0.00%	33.65%	0.20%	14.51%	*3.03	3.00
16	10.76%	<0.00%	33.65%	0.07%	9.44%	*2.68	2.49
32	10.76%	<0.00%	33.65%	0.03%	6.64%	*2.48	2.18
64	10.76%	<0.00%	33.65%	0.02%	5.03%	*2.38	2.04
128	10.76%	<0.00%	33.65%	0.01%	4.10%	*2.31	1.95
Eight-way Associative Harvard Cache							
1	10.76%	<0.00%	33.62%	0.57%	47.19%	_6.00_	_6.50_
2	10.76%	<0.00%	33.65%	0.28%	26.08%	4.35	4.48
4	10.76%	<0.00%	33.65%	0.08%	13.35%	*3.35	3.18
8	10.76%	<0.00%	33.65%	0.02%	8.34%	*2.96	2.65
16	10.76%	<0.00%	33.65%	0.01%	5.40%	_*2.73_	2.35
32	10.76%	<0.00%	33.65%	<0.00%	4.21%	_*2.64_	_2.22_
64	10.76%	<0.00%	33.65%	<0.00%	3.74%	_*2.61_	_2.17_
128	10.76%	<0.00%	33.65%	<0.00%	3.39%	_*2.58_	_2.14_

Table 6.17: Miss rates and average access times for a Harvard cache augmented with a stack cache, a constant cache, and a register buffer. Eight-way associative Harvard cache access times slower than the corresponding direct mapped cache times are italicized. The rightmost column summarizes the results from Tables 6.12 for StackHarReg, the best performing of the closest configurations. Where ConstStackHarReg has a slower access time than StackHarReg the time is marked with a '*'.

Figure 6.12 shows the average access times for all of the unified cache hybrid configurations. From the graph we can see that the most significant cache enhancement comes from adding a register buffer. Also, although the CONSTSTACKUNI cache is faster than the STACKUNI cache for small cache sizes, the STACKUNI cache is slightly faster for caches larger than 16K. The addition of a constant cache is only beneficial for small cache sizes.

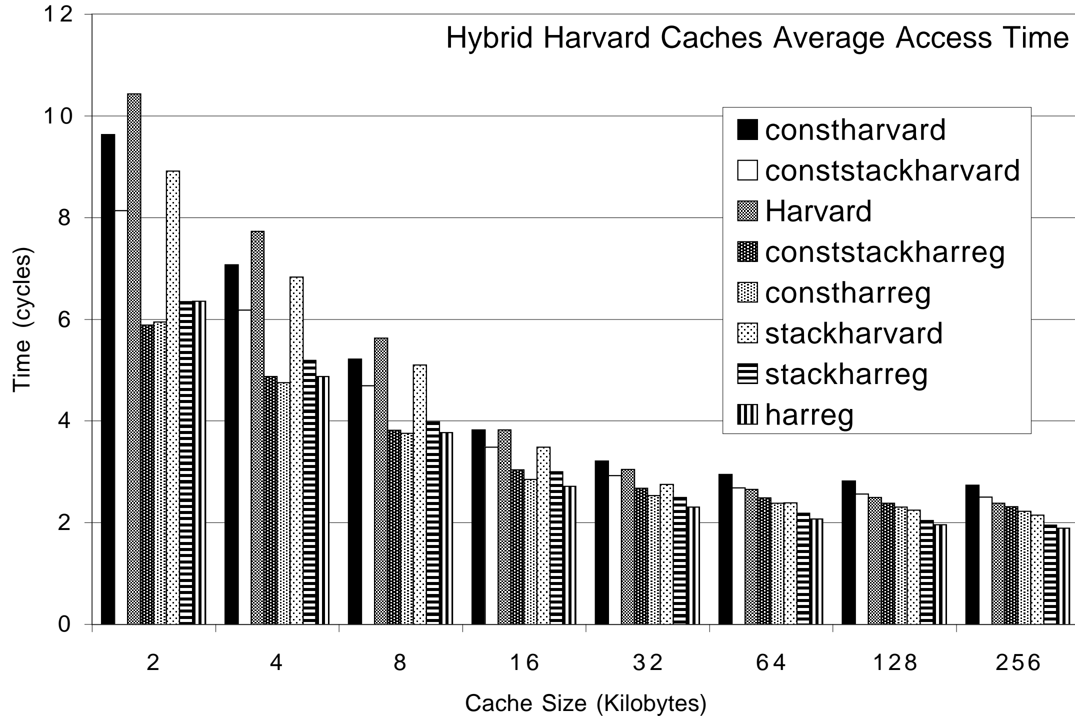


Figure 6.13: Comparison of average memory access times for Harvard cache variations.

Figure 6.13 makes an analogous comparison of access times for all of the Harvard cache hybrids. Again, the addition of a register buffer makes the most significant improvement. The CONSTSTACKHARVARD cache at large cache sizes is one of the slowest cache configurations. Interestingly, adding a register buffer (HARREG) obviates any other auxiliary cache for all Harvard caches larger than 2K. The lone addition of stack cache (STACKHARVARD) yields access times nearly as fast as the addition of just a register buffer.

6.5 Conclusions

The addition of specialized auxiliary data caches can improve the average memory access times of both unified and Harvard caches, particularly at small cache sizes. These extra

caches do not have to be very large to make a noticeable difference. A 256 byte stack cache, for example, has a miss rate less than 0.00007%.

Adding a constant cache to capture constant pool accesses helps to remove cache pollution from either a unified or a data cache. The resulting reduced miss rates for these caches leads to the *Constant Locality Rule*. Unfortunately, this separation only improves average access time when the unified and data caches are relatively small and do not have an extremely low miss rate. The high miss rate of the constant cache dominates access time when unified and data caches are large enough to have very low miss rates. The benefits gained from dedicating a 512K constant cache when cache sizes are small is unlikely to improve performance more than using the same amount of space to increase the size of the unified/data cache. Perhaps if the constant data could be prefetched into the constant cache the miss rate for the cache would decrease enough to justify adding a constant cache to configurations containing large unified or large data caches.

The stack cache with its very low miss rate improves access time for cache configurations of all sizes. Most of the programs we ran supported the *25 < 64 Stack Rule*: the 25% of memory accesses made to operand stacks can be efficiently cached in a 64 byte cache.

When both a stack cache and a register buffer are added, however, the improvement is very small. Much of the initial gain associated with adding a register buffer comes from caching stack accesses in the register buffer. Thus, a stack cache and a register cache overlap some in the performance benefit they can provide to a caching hierarchy. STACKUNIREG performs slightly better than UNIREG; STACKHARREG performs slightly worse than HARREG.

With the latency times used in our experiments the register buffer made the most dramatic improvement to average access times despite having the worst miss rate of all the added caches. The lack of registers available to the JVM is clearly a serious performance handicap. Were register resources available to the JVM, the *Fair Hardware Rule* tells us that Java programs could be as fast as traditional natively compiled programs in accessing memory despite the *30/70 Java Memory Rule*.

Chapter 7

Conclusions

Very little of the research to improve the performance of JVM's has considered the benefits of hardware support for the Java runtime environment. Assuming the continued popularity of Java, hardware components to support Java (particularly as additions to Java native platforms) would be justifiable. We have looked at how effectively current cache topologies capture the active working set of Java programs and compared this performance to that of traditional compiled programs (represented by the SPEC92 benchmark suite). Understanding the performance discrepancy required that we look at the memory profile of Java programs. The subdivision of data memory by the JVM specification led us to examine the possible benefits of adding constant, register, and stack caches.

After reviewing our conclusions we address the difficulties of adding specialized caches in Section 7.2. Section 7.3 briefly discusses how this work applies to languages other than Java. We conclude with possible future work in Section 7.4.

7.1 Summary

First we looked at the personality of the Java Virtual machine architecture in Chapter 3—a personality significantly different from that of register based machines. Compact bytecode, a powerful instruction set, and support of object oriented programming combines to cause a much higher ratio of data accesses to memory accesses in Java programs than in traditional compiled programs. Data accesses have lower locality of reference than instruction accesses; changing the proportion of data to instruction accesses poses a challenge to the effectiveness of caching.

Next in Chapter 4 we investigated the opcode and memory profile for Java programs. Tracing the executed opcodes and the memory accesses they cause for the SPEC JVM98 benchmark suite and a Java implementation of `linpack` reveals that not all opcodes cause

equal numbers of memory accesses. In particular, MEM_LOAD, MEM_STORE, CALL_RET, and NEW_OBJ opcodes account for more than their share of memory accesses. Most opcodes cause the JVM to make multiple memory accesses, leading to the *30/70 Java Memory Rule*.

Chapter 5 looked at the performance of unified and Harvard caches servicing a JVM. The key to caching Java programs lies in efficiently caching the working set of data accesses. Despite the reduced importance of the instruction cache, a Harvard cache outperforms a unified cache when total cache size is 16K or greater (*16K Instruction Rule*). Unfortunately, the software JVM's used today do not make use of the instruction cache and are effectively serviced by a unified cache half the size of what it could be.

The division of memory by the JVM specification into heap memory, stack memory, and constant pool memory serves as an entry point to examining the data memory profile of Java programs. Due to using an operand stack as scratch space (in the absence of machine registers), the set of data accesses have a much higher locality as nearly a quarter of all memory accesses target the top of this stack.

In Chapter 6 we considered the effects of adding small specialized caches to capture stack and constant pool accesses. Additionally, a small cache was used to simulate the effect of making registers available to the JVM. Constant pool accesses were found to have very low locality of reference (*Constant Locality Rule*), making the effectiveness of a small constant cache marginal.

The stack cache successfully captured stack accesses with greater than 99% hit rate. Our experiments with stack caches support the *24 < 64 Stack Rule*: a very small (64 byte) cache can efficiently cache the 24% of memory accesses that are made to operand stacks. Adding a small dedicated stack cache that could be used by the JVM would be beneficial to JVM performance especially if the latency of the cache could be kept low.

Adding a register buffer made the most significant improvement in the average memory access time for Java programs. The vast improvement underscores the effect of depriving the JVM of register resources. If a way could be found to simulate registers for a JVM performance could only be improved. Indeed, adding a register buffer to unified and Harvard caches yielded access times competitive with cache access times for the SPEC92 suite (which had access to register resources). This is summarized by the *Fair Hardware Rule*.

7.2 Difficulties of Adding Specialized Caches

Caches have always been transparent architectural features. An executing user program that requests a value from memory has no idea of whether or not a cache sits between the CPU and main memory. Such an abstraction has always been beneficial in that it reduces the complexity of writing user programs.

The difficulty in providing specialized caches for a user program such as a software

implemented JVM is that the JVM does not know if the specialized cache exists; manipulating caches is not feasible with this model. On the other hand, machine registers are architectural features to which instruction sets grant access. With modifications an instruction set could allow a user program to enable specialized caches. Specifying the correct cache could be an explicit or implicit action. Explicitly, instructions could exist to send accesses to a given cache. Alternatively, the cache to use could be determined by a couple of bits in the instruction. Access could be granted implicitly by requiring that access to a particular cache is based on the memory address. For example, after enabling a stack cache any memory address that fell within a certain range would be looked for in the stack cache. This is no different than the job compilers do today of segmenting memory based on the type of values stored. Such a segmentation would be easy to support in hardware.

Java native platforms do not suffer this handicap. A hardware JVM chip knows what cache resources are available and can easily direct Java memory accesses to the correct cache.

Ultimately, adding new cache support requires architectural changes. In addition to potentially altering an instruction set to make caches visible, space needs to be allocated on the chip for these specialized caches. Such alterations are neither cheap nor quick to make.

7.3 Applicability Outside of Java

After being introduced in 1995 Java has grown in popularity to be one of the prevalent programming languages of the time. However, it is still a young language with a long way to go before it matures out of the fad stage. While Java may disappear into the annals of extinct languages, the work presented here can be applied to languages that target stack machines, apart from Java.

7.4 Future Work

All of the data presented in this work was collected using an interpreter. Now a look needs to be taken at just-in-time compilers and how they interact with memory. Additionally, the supporting programs that run while a JVM executes a Java program use the caches as much as the Java program. For software JVM's this means that the instructions constituting the JVM are cached in the instruction cache. For JIT JVM's the process compiling Java bytecode to native machine code are cached in the instruction cache. Likewise, both kinds of JVM's cause data values to be stored in the data cache. The competition between the JVM program and the Java program for the cache resources of the physical machine is an area that needs to be investigated before the large step of adding specialized caches can be taken with confidence.

Both software and hardware JVM's garbage collect memory. This process certainly causes multiple memory accesses. Work remains to be done to determine the locality of reference of these accesses under different garbage collection schemes and how these accesses affect the efficiency of caching Java programs.

Appendix A

Rules of Thumb

1. *Amdahl's Law:* The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used [26].

2. *90/10 Locality Rule:* A program executes about 90% of its instructions in 10% of its code.

3. *2:1 Cache Rule:* The miss rate of a direct-mapped cache of size N is about the same as a two-way set-associative cache of size $N/2$ [26].

4. *Fair Hardware Rule:* Under equal hardware resources Java programs have faster average memory accesses than traditional native programs (Chapter 6).

5. *30/70 Java Memory Rule* Whereas traditional compiled programs demonstrate a 75/25 instruction to data memory access ratio, Java programs follow a 30/70 instruction to data memory access ratio (Chapter 4).

6. *25 < 64 Stack Rule:* The 25% of memory accesses made by Java programs to operand stacks can be efficiently cached in 64 bytes (Chapter 6).

7. *16K Instruction Rule:* For the average Java program using direct mapped caches, separating instructions into a separate cache is only worthwhile when the total cache size is 16K or greater (Chapter 5).

8. *Constant Locality Rule:* Accessing constant pool data has extremely low locality of reference due to infrequent use of constants and the division of constant data by `class` (each `class` has its own constant pool) (Chapter 6).

Appendix B

Benchmark Distributions

Total Memory Access Counts

Benchmark	Total Memory Accesses	Percent Logged
compress	87,595,582,237	0.133%
db	26,361,174,579	0.442%
jack	24,231,330,920	0.481%
javac	Not Available	Unknown
jess	Not Available	Unknown
linpack	5,220,886,980	2.232%
mpegaudio	74,293,366,140	0.157%
mtrt	Not Available	Unknown

Table B.1: Benchmarks that abnormally terminated did not record the total number of memory accesses made; their counts are listed as Not Available. The right column gives the percentage of the memory accesses that was logged, where known.

Java Opcode Classifications

Classification	Bytecode
LVAR_LOAD	ALOAD, ALOAD_*, DLOAD, DLOAD_*, FLOAD, FLOAD_*, ILOAD, ILOAD_*, LLOAD, LLOAD_*
LVAR_STORE	ASTORE, ASTORE_*, DSTORE, DSTORE_*, FSTORE, FSTORE_*, IINC, ISTORE, ISTORE_*, LSTORE, LSTORE_*
MEM_LOAD	AALOAD, BALOAD, CALOAD, DALOAD, FALOAD, GETFIELD, GETSTATIC, IALOAD, LALOAD, SALOAD
MEM_STORE	AASTORE, BASTORE, CASTORE, DASTORE, FASTORE, IASTORE, LASTORE, PUTFIELD, PUTSTATIC, SASTORE
COMPUTE	D2F, D2I, D2L, DADD, DCMPG, DCMPL, DDIV, DMUL, DNEG, DREM, DSUB, F2D, F2I, F2L, FADD, FCMPG, FCMPL, FDIV, FMUL, FNEG, FREM, FSUB, I2B, I2C, I2D, I2F, I2L, I2S, IADD, IAND, IDIV, IMUL, INEG, IOR, IREM, ISHL, ISHR, ISUB, IUSHR, IXOR, WA, LDO, L2D, L2F, L2I, LADD, LAND, LCMP, LDIV, LMUL, LNEG, LOR, LREM, LSHL, LSHR, LSUB, LUSHR, LXOR
BRANCH	GOTO, GOTO_W, IF_ACMPEQ, IF_ACMPNE, IF_ICMPEQ, IF_ICMPNE, IF_ICMPLT, IF_ICMPGE, IF_ICMPGT, IF_ICMPLE, IFEQ, IFNE, IFLT, IFGE, IFGT, IFLE, IFNONNULL, IFNULL, JSR, JSR_W, LOOKUPSWITCH, RET, TABLESWITCH
CALL_RET	ARETURN, DRETURN, FRETURN, INVOKEINTERFACE, INVOKESPECIAL, INVOKESTATIC, INVOKEVIRTUAL, IRETURN, LRETURN, RETURN
PUSH_CONST	ACONST_NULL, BIPUSH, DCONST_*, FCONST_*, ICONST_*, LCONST_*, LDC, LDC_W, LDC2_W, SIPUSH
MISC_STACK	DUP, DUP_*, DUP2, DUP2_*, POP, POP2, SWAP
NEW_OBJ	ANEWARRAY, MULTIANEWARRAY, NEW, NEWARRAY
OTHER	ARRAYLENGTH, ATHROW, CHECKCAST, INSTANCEOF, MONITORENTER, MONITOREXIT, NOP, WIDE

Table B.2: Java opcode classifications. A * indicates multiple opcodes of the same type with different implied integer arguments.

compress: Distributions

Type	Opcode Dist.			Mem. Access Dist.			Mem/
	Freq.	%	bench.	Freq.	%	bench.	Ops
LVAR_LOAD	5,210,769	31.38%	32.98%	16,853,097	14.47%	15.28%	3.23
LVAR_STORE	1,483,183	8.93%	6.58%	6,770,690	5.81%	7.52%	4.56
MEM_LOAD	3,019,219	18.18%	15.59%	49,053,377	42.10%	31.50%	16.25
MEM_STORE	769,032	4.63%	4.79%	12,242,612	10.50%	9.75%	15.91
COMPUTE	2,015,318	12.14%	10.67%	7,980,845	6.85%	7.00%	3.92
BRANCH	1,056,936	6.37%	9.06%	4,612,772	3.96%	11.26%	4.36
CALL_RET	741,399	4.47%	6.08%	11,656,868	10.00%	12.37%	15.72
PUSH_CONST	1,370,877	8.27%	9.51%	3,635,236	3.12%	4.48%	2.65
MISC_STACK	931,089	5.61%	4.09%	3,405,769	2.92%	2.17%	3.66
NEW_OBJ	3,239	0.02%	0.25%	229,560	0.20%	1.16%	70.87
OTHER	3,373	0.02%	0.42%	67,174	0.06%	0.51%	19.92
TOTAL	16,604,434			116,508,000			7.02

Table B.3: Opcode and memory access distributions for the compress benchmark, classified by opcode types.

db: Distributions

Type	Opcode Dist.			Mem. Access Dist.			Mem/
	Freq.	%	bench.	Freq.	%	bench.	Ops
LVAR_LOAD	8,716,464	42.55%	32.98%	32,051,578	27.51%	15.28%	3.68
LVAR_STORE	1,357,670	6.63%	6.58%	6,696,199	5.75%	7.52%	4.93
MEM_LOAD	2,067,214	10.10%	15.59%	28,331,279	24.32%	31.50%	13.71
MEM_STORE	746,393	3.64%	4.79%	10,592,148	9.09%	9.75%	14.19
COMPUTE	1,658,068	8.10%	10.67%	6,171,458	5.30%	7.00%	3.72
BRANCH	3,056,178	14.92%	9.06%	14,343,841	12.31%	11.26%	4.69
CALL_RET	477,526	2.33%	6.08%	6,158,340	5.29%	12.37%	12.90
PUSH_CONST	1,955,483	9.55%	9.51%	6,302,470	5.41%	4.48%	3.22
MISC_STACK	197,172	0.96%	4.09%	776,815	0.67%	2.17%	3.94
NEW_OBJ	133,399	0.65%	0.25%	4,200,019	3.60%	1.16%	31.48
OTHER	120,070	0.59%	0.42%	883,853	0.76%	0.51%	7.36
TOTAL	20,485,637			116,508,000			5.69

Table B.4: Opcode and memory access distributions for the db benchmark, classified by opcode types.

`Empty`**: Distributions**

Type	**Opcode Dist.**			**Mem. Access Dist.**			**Mem/**
	Freq.	%	bench.	Freq.	%	bench.	**Ops**
LVAR_LOAD	104,888	35.71%	32.98%	340,431	14.67%	15.28%	3.25
LVAR_STORE	32,439	11.04%	6.58%	154,907	6.68%	7.52%	4.78
MEM_LOAD	41,849	14.25%	15.59%	693,925	29.91%	31.50%	16.58
MEM_STORE	10,914	3.72%	4.79%	162,136	6.99%	9.75%	14.86
COMPUTE	23,527	8.01%	10.67%	89,056	3.84%	7.00%	3.78
BRANCH	31,261	10.64%	9.06%	137,932	5.94%	11.26%	4.41
CALL_RET	19,272	6.56%	6.08%	427,624	18.43%	12.37%	22.19
PUSH_CONST	25,371	8.64%	9.51%	154,322	6.65%	4.48%	6.08
MISC_STACK	1,509	0.51%	4.09%	4,083	0.18%	2.17%	2.71
NEW_OBJ	1,351	0.46%	0.25%	102,527	4.42%	1.16%	75.89
OTHER	1,334	0.45%	0.42%	53,194	2.29%	0.51%	39.88
TOTAL	293,715			2,320,137			7.90

Table B.5: Opcode and memory access distributions for the an empty Java program, classified by opcode types.

jack: Distributions

Type	**Opcode Dist.**			**Mem. Access Dist.**			**Mem/**
	Freq.	%	bench.	Freq.	%	bench.	**Ops**
LVAR_LOAD	3,763,667	26.25%	32.98%	11,556,725	9.92%	15.28%	3.07
LVAR_STORE	306,664	2.14%	6.58%	1,422,750	1.22%	7.52%	4.64
MEM_LOAD	2,996,309	20.89%	15.59%	48,157,211	41.33%	31.50%	16.07
MEM_STORE	1,429,723	9.97%	4.79%	24,169,050	20.74%	9.75%	16.90
COMPUTE	790,945	5.52%	10.67%	3,174,474	2.72%	7.00%	4.01
BRANCH	1,634,844	11.40%	9.06%	6,740,479	5.79%	11.26%	4.12
CALL_RET	409,709	2.86%	6.08%	7,612,992	6.53%	12.37%	18.58
PUSH_CONST	915,590	6.38%	9.51%	2,133,207	1.83%	4.48%	2.32
MISC_STACK	2,008,609	14.01%	4.09%	9,918,682	8.51%	2.17%	4.94
NEW_OBJ	31,339	0.22%	0.25%	1,107,807	0.95%	1.16%	35.35
OTHER	52,473	0.37%	0.42%	514,623	0.44%	0.51%	9.81
TOTAL	14,339,872			116,508,000			8.12

Table B.6: Opcode and memory access distributions for the jack benchmark, classified by opcode types.

javac: Distributions

Type	**Opcode Dist.**			**Mem. Access Dist.**			**Mem/**
	Freq.	%	bench.	Freq.	%	bench.	**Ops**
LVAR_LOAD	2,803,876	32.17%	32.98%	8,749,987	7.51%	15.28%	3.12
LVAR_STORE	561,458	6.44%	6.58%	2,490,032	2.14%	7.52%	4.43
MEM_LOAD	1,583,739	18.17%	15.59%	26,098,365	22.40%	31.50%	16.48
MEM_STORE	569,968	6.54%	4.79%	9,494,570	8.15%	9.75%	16.66
COMPUTE	524,843	6.02%	10.67%	2,050,349	1.76%	7.00%	3.91
BRANCH	836,456	9.60%	9.06%	50,212,159	43.10%	11.26%	60.03
CALL_RET	723,767	8.30%	6.08%	11,949,674	10.26%	12.37%	16.51
PUSH_CONST	618,107	7.09%	9.51%	1,807,610	1.55%	4.48%	2.92
MISC_STACK	396,256	4.55%	4.09%	1,653,266	1.42%	2.17%	4.17
NEW_OBJ	25,429	0.29%	0.25%	1,318,799	1.13%	1.16%	51.86
OTHER	72,605	0.83%	0.42%	683,189	0.59%	0.51%	9.41
TOTAL	8,716,495			116,508,000			13.36

Table B.7: Opcode and memory access distributions for the javac benchmark, classified by opcode types.

jess: Distributions

Type	**Opcode Dist.**			**Mem. Access Dist.**			**Mem/**
	Freq.	%	bench.	Freq.	%	bench.	**Ops**
LVAR_LOAD	5,619,818	37.85%	32.98%	17,852,976	15.32%	15.28%	3.18
LVAR_STORE	1,054,238	7.10%	6.58%	4,931,284	4.23%	7.52%	4.68
MEM_LOAD	2,703,206	18.21%	15.59%	43,076,877	36.97%	31.50%	15.94
MEM_STORE	247,678	1.67%	4.79%	4,129,824	3.54%	9.75%	16.67
COMPUTE	341,615	2.30%	10.67%	1,485,035	1.27%	7.00%	4.35
BRANCH	1,947,029	13.11%	9.06%	9,432,115	8.10%	11.26%	4.84
CALL_RET	1,671,152	11.26%	6.08%	28,847,394	24.76%	12.37%	17.26
PUSH_CONST	936,114	6.31%	9.51%	2,615,663	2.25%	4.48%	2.79
MISC_STACK	93,490	0.63%	4.09%	317,290	0.27%	2.17%	3.39
NEW_OBJ	45,581	0.31%	0.25%	1,640,973	1.41%	1.16%	36.00
OTHER	186,544	1.26%	0.42%	2,178,569	1.87%	0.51%	11.68
TOTAL	14,846,465			116,508,000			7.85

Table B.8: Opcode and memory access distributions for the jess benchmark, classified by opcode types.

linpack: Distributions

Type	Opcode Dist.			Mem. Access Dist.			Mem/
	Freq.	%	bench.	Freq.	%	bench.	Ops
LVAR_LOAD	4,737,260	27.12%	32.98%	18,465,571	15.85%	15.28%	3.90
LVAR_STORE	997,029	5.71%	6.58%	6,712,717	5.76%	7.52%	6.73
MEM_LOAD	1,005,675	5.76%	15.59%	15,344,053	13.17%	31.50%	15.26
MEM_STORE	589,206	3.38%	4.79%	9,800,463	8.41%	9.75%	16.63
COMPUTE	5,035,349	28.85%	10.67%	29,581,704	25.39%	7.00%	5.87
BRANCH	417,951	2.39%	9.06%	1,887,829	1.61%	11.26%	4.52
CALL_RET	1,175,859	6.74%	6.08%	18,167,015	15.59%	12.37%	15.45
PUSH_CONST	3,495,494	20.03%	9.51%	16,347,199	14.03%	4.48%	4.68
MISC_STACK	1,511	0.01%	4.09%	4,089	0.00%	2.17%	2.71
NEW_OBJ	1,357	0.01%	0.25%	154,166	0.13%	1.16%	113.63
OTHER	1,334	0.01%	0.42%	53,194	0.05%	0.51%	39.88
TOTAL	17,453,525			116,508,000			6.68

Table B.9: Opcode and memory access distributions for the linpack benchmark, classified by opcode types.

mpegaudio: Distributions

Type	Opcode Dist.			Mem. Access Dist.			Mem/
	Freq.	%	bench.	Freq.	%	bench.	Ops
LVAR_LOAD	5,945,503	33.27%	32.98%	21,040,575	18.06%	15.28%	3.54
LVAR_STORE	1,488,636	8.33%	6.58%	7,193,560	6.17%	7.52%	4.83
MEM_LOAD	3,532,706	19.77%	15.59%	51,777,990	44.44%	31.50%	14.66
MEM_STORE	594,547	3.33%	4.79%	8,667,286	7.44%	9.75%	14.58
COMPUTE	2,684,887	15.03%	10.67%	10,711,951	9.19%	7.00%	3.99
BRANCH	734,579	4.11%	9.06%	3,300,697	2.83%	11.26%	4.49
CALL_RET	351,649	1.97%	6.08%	5,949,748	5.11%	12.37%	16.92
PUSH_CONST	2,306,763	12.91%	9.51%	6,435,544	5.52%	4.48%	2.79
MISC_STACK	206,959	1.16%	4.09%	847,176	0.73%	2.17%	4.09
NEW_OBJ	5,269	0.03%	0.25%	424,769	0.36%	1.16%	80.62
OTHER	16,422	0.10%	0.42%	158,704	0.14%	0.51%	9.66
TOTAL	17,867,920			116,508,000			6.52

Table B.10: Opcode and memory access distributions for the mpegaudio benchmark, classified by opcode types.

mtrt: Distributions

Type	**Opcode Dist.**			**Mem. Access Dist.**			**Mem/**
	Freq.	%	bench.	Freq.	%	bench.	**Ops**
LVAR_LOAD	4,892,076	33.21%	32.98%	15,831,006	13.59%	15.28%	3.25
LVAR_STORE	1,082,567	7.35%	6.58%	5,310,223	4.56%	7.52%	4.78
MEM_LOAD	2,010,082	13.65%	15.59%	32,360,418	27.78%	31.50%	16.58
MEM_STORE	757,661	5.14%	4.79%	11,808,401	10.14%	9.75%	14.86
COMPUTE	1,093,320	7.42%	10.67%	4,120,879	3.54%	7.00%	3.78
BRANCH	1,558,830	10.58%	9.06%	14,443,162	12.40%	11.26%	4.41
CALL_RET	1,579,581	10.72%	6.08%	24,935,980	21.40%	12.37%	22.19
PUSH_CONST	812,205	5.51%	9.51%	2,433,604	2.09%	4.48%	6.08
MISC_STACK	847,844	5.76%	4.09%	3,317,765	2.85%	2.17%	2.71
NEW_OBJ	71,690	0.49%	0.25%	1,743,497	1.50%	1.16%	75.89
OTHER	22,690	0.15%	0.42%	203,065	0.17%	0.51%	39.88
TOTAL	14,728,546			2,320,137			7.90

Table B.11: Opcode and memory access distributions for the mtrt benchmark, classified by opcode types.

Appendix C

Unified and Harvard Cache Results

Direct Mapped Unified Cache Miss Rates								
Size (KB)	compress	db	jack	javac	jess	linpack	mpegaudio	mtrt
1	23.67%	16.40%	32.54%	17.44%	26.18%	11.57%	18.13%	23.82%
2	15.35%	11.83%	15.45%	12.35%	17.67%	6.65%	12.30%	16.81%
4	9.55%	5.12%	12.30%	8.25%	12.76%	2.04%	9.01%	7.14%
8	5.09%	2.89%	5.34%	4.86%	7.17%	1.17%	4.30%	3.51%
16	3.05%	2.01%	2.00%	3.26%	4.45%	0.62%	3.30%	1.85%
32	1.02%	1.45%	1.11%	2.13%	2.55%	0.48%	2.65%	1.23%
64	0.84%	1.09%	0.73%	1.17%	1.42%	0.40%	1.88%	0.78%
128	0.51%	1.05%	0.41%	0.81%	0.84%	0.39%	1.60%	0.69%
Eight-way Associative Unified Cache Miss Rates								
Size (KB)	compress	db	jack	javac	jess	linpack	mpegaudio	mtrt
1	17.83%	8.27%	6.94%	12.62%	19.92%	9.12%	12.40%	15.22%
2	10.06%	5.07%	4.23%	8.32%	12.28%	1.87%	5.20%	7.02%
4	3.69%	2.06%	2.96%	4.17%	6.26%	0.83%	2.40%	2.81%
8	1.10%	1.20%	1.98%	2.50%	3.22%	0.66%	1.05%	1.63%
16	0.68%	1.01%	0.99%	1.39%	1.67%	0.48%	0.63%	1.01%
32	0.56%	0.97%	0.47%	0.80%	0.96%	0.39%	0.45%	0.72%
64	0.47%	0.92%	0.31%	0.56%	0.63%	0.38%	0.39%	0.58%
128	0.43%	0.91%	0.26%	0.43%	0.49%	0.38%	0.29%	0.53%

Table C.1: Unified cache miss rates by benchmark.

Direct Mapped Split Cache Miss Rates #1				
Size (KB)	**compress**		**db**	
	Data	**Instruction**	**Data**	**Instruction**
1	26.40%	0.19%	16.93%	0.41%
2	17.34%	0.18%	10.19%	0.41%
4	11.84%	0.03%	5.93%	0.34%
8	6.03%	0.02%	3.21%	0.33%
16	3.69%	0.00%	2.36%	0.00%
32	1.23%	0.00%	1.87%	0.00%
64	1.10%	0.00%	1.63%	0.00%
128	0.66%	0.00%	1.56%	0.00%
Eight-way Assoc. Split Cache Miss Rates #1				
Size (KB)	**compress**		**db**	
	Data	**Instruction**	**Data**	**Instruction**
1	20.38%	0.01%	10.35%	0.01%
2	10.68%	0.01%	5.83%	0.01%
4	3.37%	0.00%	2.50%	0.00%
8	1.25%	0.00%	1.70%	0.00%
16	0.86%	0.00%	1.49%	0.00%
32	0.73%	0.00%	1.44%	0.00%
64	0.61%	0.00%	1.39%	0.00%
128	0.57%	0.00%	1.36%	0.00%

Table C.2: Split cache miss rates for compress and db. The listed sizes are for both the instruction and data caches.

Direct Mapped Split Cache Miss Rates #2				
Size (KB)	**jack**		**javac**	
	Data	**Instruction**	**Data**	**Instruction**
1	24.30%	0.98%	27.90%	2.08%
2	18.81%	0.66%	20.12%	0.80%
4	15.06%	0.45%	12.91%	0.58%
8	6.30%	0.19%	6.95%	0.28%
16	2.14%	0.07%	4.10%	0.11%
32	1.23%	0.01%	2.94%	0.05%
64	0.83%	0.01%	2.04%	0.03%
128	0.49%	0.01%	1.41%	0.02%

Eight-way Assoc. Split Cache Miss Rates #2				
Size (KB)	**jack**		**javac**	
	Data	**Instruction**	**Data**	**Instruction**
1	6.69%	0.83%	20.12%	1.48%
2	4.67%	0.36%	11.33%	0.77%
4	3.18%	0.08%	5.73%	0.33%
8	2.04%	0.01%	3.55%	0.09%
16	0.93%	0.01%	2.04%	0.01%
32	0.53%	0.01%	1.38%	0.00%
64	0.37%	0.01%	1.06%	0.00%
128	0.32%	0.01%	0.90%	0.00%

Table C.3: Split cache miss rates for jack and javac. The listed sizes are for both the instruction and data caches.

Direct Mapped Split Cache Miss Rates #3				
Size (KB)	**jess**		**linpack**	
	Data	**Instruction**	**Data**	**Instruction**
1	29.03%	2.25%	14.01%	0.02%
2	19.37%	1.59%	7.88%	0.01%
4	13.61%	0.63%	2.60%	0.01%
8	7.94%	0.40%	1.48%	0.00%
16	4.80%	0.19%	0.79%	0.00%
32	2.64%	0.09%	0.62%	0.00%
64	1.60%	0.07%	0.51%	0.00%
128	0.98%	0.03%	0.50%	0.00%
Eight-way Assoc. Split Cache Miss Rates #3				
Size (KB)	**jess**		**linpack**	
	Data	**Instruction**	**Data**	**Instruction**
1	22.25%	1.08%	8.88%	0.00%
2	13.29%	0.48%	2.00%	0.00%
4	6.62%	0.14%	1.02%	0.00%
8	3.43%	0.04%	0.83%	0.00%
16	1.82%	0.01%	0.62%	0.00%
32	1.13%	0.01%	0.51%	0.00%
64	0.77%	0.01%	0.50%	0.00%
128	0.62%	0.01%	0.49%	0.00%

Table C.4: Split cache miss rates for jess and linpack. The listed sizes are for both the instruction and data caches.

Direct Mapped Split Cache Miss Rates #4				
Size (KB)	**mpegaudio**		**mtrt**	
	Data	**Instruction**	**Data**	**Instruction**
1	20.40%	1.01%	28.96%	1.05%
2	13.97%	0.69%	20.14%	0.60%
4	10.21%	0.47%	9.03%	0.30%
8	4.56%	0.14%	4.21%	0.24%
16	3.49%	0.03%	2.09%	0.14%
32	2.71%	0.02%	1.52%	0.06%
64	2.37%	0.01%	1.05%	0.00%
128	2.01%	0.01%	0.96%	0.00%
Eight-way Assoc. Split Cache Miss Rates #4				
Size (KB)	**mpegaudio**		**mtrt**	
	Data	**Instruction**	**Data**	**Instruction**
1	13.66%	0.53%	16.90%	0.61%
2	5.30%	0.26%	7.19%	0.33%
4	2.29%	0.05%	3.12%	0.05%
8	1.10%	0.03%	1.87%	0.00%
16	0.72%	0.01%	1.22%	0.00%
32	0.56%	0.01%	0.96%	0.00%
64	0.47%	0.01%	0.80%	0.00%
128	0.41%	0.01%	0.75%	0.00%

Table C.5: Split cache miss rates for mpegaudio and mtrt. The listed sizes are for both the instruction and data caches.

Appendix D

Hybrid Cache Results

Eight-way Associative Cache Miss Rates								
Size	**compress**	**db**	**jack**	**javac**	**jess**	**linpack**	**mpegaudio**	**mtrt**
64	48.48%	44.70%	49.11%	48.39%	53.88%	37.08%	50.53%	51.29%
128	34.51%	22.88%	31.96%	32.74%	41.15%	29.54%	37.94%	38.21%
256	28.80%	14.53%	12.99%	27.56%	32.70%	22.05%	25.43%	31.34%
512	20.63%	11.11%	6.56%	22.06%	21.80%	11.54%	12.37%	15.76%

Table D.1: Register cache miss rates by benchmark. Cache sizes are given in bytes. Each cache line is 8 bytes; a write back, write fetch policy was used with LRU replacement. Only data accesses were cached.

Direct Mapped Stack Cache Miss Rates #1				
Size (bytes)	**compress**	**db**	**jack**	**javac**
64	0.000037%	0.000033%	0.000037%	0.005881%
128	0.000033%	0.000030%	0.000033%	0.000708%
256	0.000033%	0.000030%	0.000033%	0.000194%
512	0.000033%	0.000030%	0.000033%	0.000194%

Table D.2: Stack cache miss rates by benchmark. Cache sizes are given in bytes. Each cache line is 8 bytes; a write through, write around policy was used. Only operand stack accesses were cached.

Direct Mapped Stack Cache Miss Rates #2				
Size (bytes)	**jess**	**linpack**	**mpegaudio**	**mtrt**
64	0.000046%	0.000022%	0.000040%	0.000156%
128	0.000041%	0.000020%	0.000030%	0.000152%
256	0.000041%	0.000020%	0.000030%	0.000152%
512	0.000041%	0.000020%	0.000030%	0.000152%

Table D.3: Stack cache miss rates by benchmark. Cache sizes are given in bytes. Each cache line is 8 bytes; a write through, write around policy was used. Only operand stack accesses were cached.

Direct Mapped Constant Cache Miss Rates								
Size	**compress**	**db**	**jack**	**javac**	**jess**	**linpack**	**mpegaudio**	**mtrt**
64	28.96%	33.71%	7.37%	32.39%	29.64%	31.73%	21.87%	29.74%
128	24.57%	31.85%	6.90%	29.71%	24.16%	9.27%	15.76%	20.46%
256	18.42%	25.90%	5.99%	22.93%	20.24%	9.15%	11.18%	11.75%
512	10.76%	23.85%	4.98%	16.99%	13.72%	0.22%	6.36%	9.17%
1024	3.19%	6.63%	3.86%	13.04%	9.61%	0.17%	4.20%	6.34%
2048	3.17%	6.10%	2.70%	9.24%	6.21%	0.16%	2.37%	5.08%

Table D.4: Constant cache miss rates by benchmark. Cache sizes are given in bytes. Each cache line is 16 bytes; a write back, write fetch policy was used. Only constant pool accesses were cached.

Table D.5: compress Hybrid Cache Results

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
CONSTHARREG: Direct Mapped						
1	10.76%	n/a	28.11%	0.19%	61.69%	7.98
2	10.76%	n/a	28.36%	0.18%	48.98%	6.84
4	10.76%	n/a	28.36%	0.03%	17.87%	3.89
8	10.76%	n/a	28.36%	0.02%	6.75%	2.84
16	10.76%	n/a	28.36%	<0.00%	4.10%	2.59
32	10.76%	n/a	28.36%	<0.00%	2.31%	2.42
64	10.76%	n/a	28.36%	<0.00%	1.86%	2.38
128	10.76%	n/a	28.36%	<0.00%	1.62%	2.36
CONSTHARREG: Eight-way Associative						
1	10.76%	n/a	28.35%	0.01%	55.18%	8.44
2	10.76%	n/a	28.36%	<0.00%	31.32%	5.88
4	10.76%	n/a	28.36%	<0.00%	6.72%	3.23
8	10.76%	n/a	28.36%	<0.00%	4.48%	2.99
16	10.76%	n/a	28.36%	<0.00%	2.08%	2.74
32	10.76%	n/a	28.36%	<0.00%	1.74%	2.70
64	10.76%	n/a	28.36%	<0.00%	1.53%	2.68
128	10.76%	n/a	28.36%	<0.00%	1.30%	2.65
CONSTHARVARD: Direct Mapped						
1	10.76%	n/a	n/a	0.19%	25.54%	11.00
2	10.76%	n/a	n/a	0.18%	15.98%	7.82
4	10.76%	n/a	n/a	0.03%	9.91%	5.78
8	10.76%	n/a	n/a	0.02%	3.33%	3.60
16	10.76%	n/a	n/a	<0.00%	2.13%	3.20
32	10.76%	n/a	n/a	<0.00%	0.96%	2.81
64	10.76%	n/a	n/a	<0.00%	0.83%	2.77
128	10.76%	n/a	n/a	<0.00%	0.73%	2.73
CONSTHARVARD: Eight-way Associative						
1	10.76%	n/a	n/a	0.01%	18.60%	9.88
2	10.76%	n/a	n/a	<0.00%	9.19%	6.32
4	10.76%	n/a	n/a	<0.00%	3.07%	4.00
8	10.76%	n/a	n/a	<0.00%	1.28%	3.33
16	10.76%	n/a	n/a	<0.00%	0.93%	3.19
32	10.76%	n/a	n/a	<0.00%	0.80%	3.14

continued on next page

Table D.5: compress Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
64	10.76%	n/a	n/a	<0.00%	0.67%	3.10
128	10.76%	n/a	n/a	<0.00%	0.60%	3.07
CONSTSTACKHARREG: Direct Mapped						
1	10.76%	<0.00%	39.63%	0.19%	56.19%	6.98
2	10.76%	<0.00%	39.74%	0.18%	40.43%	5.65
4	10.76%	<0.00%	39.74%	0.03%	27.21%	4.50
8	10.76%	<0.00%	39.74%	0.02%	9.50%	2.99
16	10.76%	<0.00%	39.74%	<0.00%	5.31%	2.62
32	10.76%	<0.00%	39.74%	<0.00%	3.52%	2.47
64	10.76%	<0.00%	39.74%	<0.00%	3.05%	2.43
128	10.76%	<0.00%	39.74%	<0.00%	2.67%	2.40
CONSTSTACKHARREG: Eight-way Associative						
1	10.76%	<0.00%	39.69%	0.01%	53.71%	7.71
2	10.76%	<0.00%	39.74%	<0.00%	25.02%	4.91
4	10.76%	<0.00%	39.74%	<0.00%	8.90%	3.34
8	10.76%	<0.00%	39.74%	<0.00%	4.70%	2.93
16	10.76%	<0.00%	39.74%	<0.00%	3.40%	2.81
32	10.76%	<0.00%	39.74%	<0.00%	2.93%	2.76
64	10.76%	<0.00%	39.74%	<0.00%	2.47%	2.72
128	10.76%	<0.00%	39.74%	<0.00%	2.21%	2.69
CONSTSTACKHARVARD: Direct Mapped						
1	10.76%	<0.00%	n/a	0.19%	34.42%	9.69
2	10.76%	<0.00%	n/a	0.18%	21.79%	6.97
4	10.76%	<0.00%	n/a	0.03%	15.23%	5.54
8	10.76%	<0.00%	n/a	0.02%	5.10%	3.36
16	10.76%	<0.00%	n/a	<0.00%	3.27%	2.96
32	10.76%	<0.00%	n/a	<0.00%	1.47%	2.58
64	10.76%	<0.00%	n/a	<0.00%	1.28%	2.53
128	10.76%	<0.00%	n/a	<0.00%	1.12%	2.50
CONSTSTACKHARVARD: Eight-way Associative						
1	10.76%	<0.00%	n/a	0.01%	26.30%	9.03
2	10.76%	<0.00%	n/a	<0.00%	12.52%	5.65
4	10.76%	<0.00%	n/a	<0.00%	4.67%	3.72
8	10.76%	<0.00%	n/a	<0.00%	1.98%	3.06

continued on next page

Table D.5: compress Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
16	10.76%	<0.00%	n/a	<0.00%	1.44%	2.93
32	10.76%	<0.00%	n/a	<0.00%	1.20%	2.87
64	10.76%	<0.00%	n/a	<0.00%	1.06%	2.84
128	10.76%	<0.00%	n/a	<0.00%	0.93%	2.80
CONSTSTACKUNI: Direct Mapped						
1	10.76%	<0.00%	n/a	n/a	26.73%	12.14
2	10.76%	<0.00%	n/a	n/a	16.37%	8.64
4	10.76%	<0.00%	n/a	n/a	10.60%	6.70
8	10.76%	<0.00%	n/a	n/a	4.04%	4.48
16	10.76%	<0.00%	n/a	n/a	2.48%	3.96
32	10.76%	<0.00%	n/a	n/a	1.07%	3.48
64	10.76%	<0.00%	n/a	n/a	0.83%	3.40
128	10.76%	<0.00%	n/a	n/a	0.73%	3.36
CONSTSTACKUNI: Eight-way Associative						
1	10.76%	<0.00%	n/a	n/a	20.15%	11.31
2	10.76%	<0.00%	n/a	n/a	10.46%	7.58
4	10.76%	<0.00%	n/a	n/a	4.38%	5.24
8	10.76%	<0.00%	n/a	n/a	1.48%	4.12
16	10.76%	<0.00%	n/a	n/a	0.93%	3.91
32	10.76%	<0.00%	n/a	n/a	0.77%	3.85
64	10.76%	<0.00%	n/a	n/a	0.70%	3.82
128	10.76%	<0.00%	n/a	n/a	0.62%	3.80
CONSTUNI: Direct Mapped						
1	10.76%	n/a	n/a	n/a	22.87%	14.21
2	10.76%	n/a	n/a	n/a	14.15%	10.25
4	10.76%	n/a	n/a	n/a	7.90%	7.41
8	10.76%	n/a	n/a	n/a	3.01%	5.19
16	10.76%	n/a	n/a	n/a	1.85%	4.66
32	10.76%	n/a	n/a	n/a	0.80%	4.18
64	10.76%	n/a	n/a	n/a	0.62%	4.10
128	10.76%	n/a	n/a	n/a	0.54%	4.07
CONSTUNI: Eight-way Associative						
1	10.76%	n/a	n/a	n/a	16.45%	12.87
2	10.76%	n/a	n/a	n/a	8.69%	8.86

continued on next page

Table D.5: compress Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
4	10.76%	n/a	n/a	n/a	3.29%	6.06
8	10.76%	n/a	n/a	n/a	1.10%	4.92
16	10.76%	n/a	n/a	n/a	0.70%	4.72
32	10.76%	n/a	n/a	n/a	0.59%	4.66
64	10.76%	n/a	n/a	n/a	0.50%	4.61
128	10.76%	n/a	n/a	n/a	0.44%	4.58
CONSTUNIREG: Direct Mapped						
1	10.76%	n/a	28.11%	n/a	38.54%	10.88
2	10.76%	n/a	28.36%	n/a	30.00%	9.07
4	10.76%	n/a	28.36%	n/a	11.03%	4.97
8	10.76%	n/a	28.36%	n/a	4.71%	3.60
16	10.76%	n/a	28.36%	n/a	2.77%	3.18
32	10.76%	n/a	28.36%	n/a	1.45%	2.89
64	10.76%	n/a	28.36%	n/a	1.05%	2.81
128	10.76%	n/a	28.36%	n/a	0.91%	2.78
CONSTUNIREG: Eight-way Associative						
1	10.76%	n/a	28.36%	n/a	33.37%	11.17
2	10.76%	n/a	28.36%	n/a	20.83%	8.08
4	10.76%	n/a	28.36%	n/a	5.61%	4.33
8	10.76%	n/a	28.36%	n/a	1.96%	3.43
16	10.76%	n/a	28.36%	n/a	1.34%	3.27
32	10.76%	n/a	28.36%	n/a	0.99%	3.19
64	10.76%	n/a	28.36%	n/a	0.92%	3.17
128	10.76%	n/a	28.36%	n/a	0.79%	3.14
HARREG: Direct Mapped						
1	n/a	n/a	30.92%	0.19%	59.83%	8.72
2	n/a	n/a	30.89%	0.18%	46.67%	7.18
4	n/a	n/a	30.89%	0.03%	22.37%	4.32
8	n/a	n/a	30.89%	0.02%	12.38%	3.16
16	n/a	n/a	30.89%	<0.00%	7.60%	2.60
32	n/a	n/a	30.89%	<0.00%	2.81%	2.04
64	n/a	n/a	30.89%	<0.00%	2.43%	1.99
128	n/a	n/a	30.89%	<0.00%	1.47%	1.88
HARREG: Eight-way Associative						

continued on next page

Table D.5: compress Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
1	n/a	n/a	30.82%	0.01%	59.90%	9.90
2	n/a	n/a	30.89%	<0.00%	33.83%	6.45
4	n/a	n/a	30.89%	<0.00%	7.56%	2.96
8	n/a	n/a	30.89%	<0.00%	3.02%	2.35
16	n/a	n/a	30.89%	<0.00%	1.93%	2.21
32	n/a	n/a	30.89%	<0.00%	1.63%	2.17
64	n/a	n/a	30.89%	<0.00%	1.39%	2.14
128	n/a	n/a	30.89%	<0.00%	1.23%	2.12
STACKHARREG: Direct Mapped						
1	n/a	<0.00%	42.71%	0.19%	58.91%	8.28
2	n/a	<0.00%	42.85%	0.18%	43.10%	6.53
4	n/a	<0.00%	42.85%	0.03%	31.92%	5.26
8	n/a	<0.00%	42.85%	0.02%	17.56%	3.66
16	n/a	<0.00%	42.85%	<0.00%	10.58%	2.88
32	n/a	<0.00%	42.85%	<0.00%	3.96%	2.13
64	n/a	<0.00%	42.85%	<0.00%	3.57%	2.09
128	n/a	<0.00%	42.85%	<0.00%	2.15%	1.93
STACKHARREG: Eight-way Associative						
1	n/a	<0.00%	42.69%	0.01%	54.31%	8.83
2	n/a	<0.00%	42.85%	<0.00%	28.46%	5.56
4	n/a	<0.00%	42.85%	<0.00%	9.18%	3.10
8	n/a	<0.00%	42.85%	<0.00%	4.01%	2.44
16	n/a	<0.00%	42.85%	<0.00%	2.83%	2.29
32	n/a	<0.00%	42.85%	<0.00%	2.29%	2.22
64	n/a	<0.00%	42.85%	<0.00%	1.99%	2.18
128	n/a	<0.00%	42.85%	<0.00%	1.85%	2.16
STACKHARVARD: Direct Mapped						
1	n/a	<0.00%	n/a	0.19%	33.93%	10.64
2	n/a	<0.00%	n/a	0.18%	22.74%	7.72
4	n/a	<0.00%	n/a	0.03%	17.09%	6.23
8	n/a	<0.00%	n/a	0.02%	8.70%	4.04
16	n/a	<0.00%	n/a	<0.00%	5.32%	3.15
32	n/a	<0.00%	n/a	<0.00%	1.77%	2.23
64	n/a	<0.00%	n/a	<0.00%	1.60%	2.18

continued on next page

Table D.5: compress Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
128	n/a	<0.00%	n/a	<0.00%	0.96%	2.02
STACKHARVARD: Eight-way Associative						
1	n/a	<0.00%	n/a	0.01%	26.97%	10.04
2	n/a	<0.00%	n/a	<0.00%	14.00%	6.18
4	n/a	<0.00%	n/a	<0.00%	4.81%	3.44
8	n/a	<0.00%	n/a	<0.00%	1.79%	2.55
16	n/a	<0.00%	n/a	<0.00%	1.26%	2.39
32	n/a	<0.00%	n/a	<0.00%	1.08%	2.33
64	n/a	<0.00%	n/a	<0.00%	0.89%	2.28
128	n/a	<0.00%	n/a	<0.00%	0.83%	2.26
STACKUNI: Direct Mapped						
1	n/a	<0.00%	n/a	n/a	27.26%	13.25
2	n/a	<0.00%	n/a	n/a	17.66%	9.58
4	n/a	<0.00%	n/a	n/a	12.43%	7.57
8	n/a	<0.00%	n/a	n/a	6.62%	5.35
16	n/a	<0.00%	n/a	n/a	3.97%	4.33
32	n/a	<0.00%	n/a	n/a	1.32%	3.32
64	n/a	<0.00%	n/a	n/a	1.10%	3.23
128	n/a	<0.00%	n/a	n/a	0.66%	3.06
STACKUNI: Eight-way Associative						
1	n/a	<0.00%	n/a	n/a	21.28%	12.50
2	n/a	<0.00%	n/a	n/a	12.04%	8.46
4	n/a	<0.00%	n/a	n/a	4.76%	5.28
8	n/a	<0.00%	n/a	n/a	1.43%	3.83
16	n/a	<0.00%	n/a	n/a	0.88%	3.59
32	n/a	<0.00%	n/a	n/a	0.74%	3.53
64	n/a	<0.00%	n/a	n/a	0.62%	3.47
128	n/a	<0.00%	n/a	n/a	0.56%	3.45
STACKUNIREG: Direct Mapped						
1	n/a	<0.00%	42.71%	n/a	33.41%	9.94
2	n/a	<0.00%	42.85%	n/a	24.22%	7.81
4	n/a	<0.00%	42.85%	n/a	16.83%	6.08
8	n/a	<0.00%	42.85%	n/a	9.68%	4.40
16	n/a	<0.00%	42.85%	n/a	5.73%	3.48

continued on next page

Table D.5: compress Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
32	n/a	<0.00%	42.85%	n/a	2.12%	2.63
64	n/a	<0.00%	42.85%	n/a	1.76%	2.55
128	n/a	<0.00%	42.85%	n/a	1.06%	2.39
STACKUNIREG: Eight-way Associative						
1	n/a	<0.00%	42.66%	n/a	30.57%	10.57
2	n/a	<0.00%	42.85%	n/a	17.84%	7.20
4	n/a	<0.00%	42.85%	n/a	6.82%	4.26
8	n/a	<0.00%	42.85%	n/a	2.25%	3.04
16	n/a	<0.00%	42.85%	n/a	1.43%	2.82
32	n/a	<0.00%	42.85%	n/a	1.19%	2.76
64	n/a	<0.00%	42.85%	n/a	1.02%	2.71
128	n/a	<0.00%	42.85%	n/a	0.91%	2.68
UNIREG: Direct Mapped						
1	n/a	n/a	30.92%	n/a	39.01%	11.50
2	n/a	n/a	30.89%	n/a	30.03%	9.35
4	n/a	n/a	30.89%	n/a	14.18%	5.57
8	n/a	n/a	30.89%	n/a	8.19%	4.13
16	n/a	n/a	30.89%	n/a	4.93%	3.36
32	n/a	n/a	30.89%	n/a	1.81%	2.61
64	n/a	n/a	30.89%	n/a	1.44%	2.52
128	n/a	n/a	30.89%	n/a	0.87%	2.39
UNIREG: Eight-way Associative						
1	n/a	n/a	30.78%	n/a	35.60%	12.16
2	n/a	n/a	30.89%	n/a	22.48%	8.60
4	n/a	n/a	30.89%	n/a	6.45%	4.24
8	n/a	n/a	30.89%	n/a	2.05%	3.04
16	n/a	n/a	30.89%	n/a	1.17%	2.80
32	n/a	n/a	30.89%	n/a	1.06%	2.77
64	n/a	n/a	30.89%	n/a	0.89%	2.72
128	n/a	n/a	30.89%	n/a	0.70%	2.67

Table D.6: `db` Hybrid Cache Results

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
CONSTHARREG: Direct Mapped						
1	23.85%	n/a	14.44%	0.41%	49.08%	4.21
2	23.85%	n/a	14.32%	0.41%	20.95%	2.90
4	23.85%	n/a	14.32%	0.34%	13.66%	2.55
8	23.85%	n/a	14.32%	0.33%	7.71%	2.28
16	23.85%	n/a	14.32%	<0.00%	6.03%	2.15
32	23.85%	n/a	14.32%	<0.00%	4.91%	2.10
64	23.85%	n/a	14.32%	<0.00%	4.53%	2.08
128	23.85%	n/a	14.32%	<0.00%	4.30%	2.07
CONSTHARREG: Eight-way Associative						
1	23.85%	n/a	14.31%	<0.00%	49.22%	4.70
2	23.85%	n/a	14.32%	<0.00%	27.43%	3.56
4	23.85%	n/a	14.32%	<0.00%	7.26%	2.51
8	23.85%	n/a	14.32%	<0.00%	5.12%	2.40
16	23.85%	n/a	14.32%	<0.00%	4.22%	2.35
32	23.85%	n/a	14.32%	<0.00%	4.35%	2.36
64	23.85%	n/a	14.32%	<0.00%	4.08%	2.35
128	23.85%	n/a	14.32%	<0.00%	3.83%	2.33
CONSTHARVARD: Direct Mapped						
1	23.85%	n/a	n/a	0.41%	16.33%	7.61
2	23.85%	n/a	n/a	0.41%	9.24%	5.35
4	23.85%	n/a	n/a	0.34%	5.10%	4.01
8	23.85%	n/a	n/a	0.33%	2.84%	3.29
16	23.85%	n/a	n/a	<0.00%	2.18%	3.02
32	23.85%	n/a	n/a	<0.00%	1.75%	2.89
64	23.85%	n/a	n/a	<0.00%	1.64%	2.85
128	23.85%	n/a	n/a	<0.00%	1.55%	2.82
CONSTHARVARD: Eight-way Associative						
1	23.85%	n/a	n/a	<0.00%	9.07%	5.96
2	23.85%	n/a	n/a	<0.00%	4.97%	4.46
4	23.85%	n/a	n/a	<0.00%	2.13%	3.43
8	23.85%	n/a	n/a	<0.00%	1.66%	3.26
16	23.85%	n/a	n/a	<0.00%	1.51%	3.20
32	23.85%	n/a	n/a	<0.00%	1.47%	3.19

continued on next page

Table D.6: db Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
64	23.85%	n/a	n/a	<0.00%	1.40%	3.16
128	23.85%	n/a	n/a	<0.00%	1.39%	3.16
CONSTSTACKHARREG: Direct Mapped						
1	23.85%	<0.00%	22.01%	0.41%	56.96%	4.31
2	23.85%	<0.00%	22.07%	0.41%	43.53%	3.75
4	23.85%	<0.00%	22.07%	0.34%	29.38%	3.14
8	23.85%	<0.00%	22.07%	0.33%	16.90%	2.62
16	23.85%	<0.00%	22.07%	<0.00%	13.54%	2.42
32	23.85%	<0.00%	22.07%	<0.00%	11.22%	2.33
64	23.85%	<0.00%	22.07%	<0.00%	10.71%	2.31
128	23.85%	<0.00%	22.07%	<0.00%	10.21%	2.28
CONSTSTACKHARREG: Eight-way Associative						
1	23.85%	<0.00%	21.94%	<0.00%	50.46%	4.52
2	23.85%	<0.00%	22.07%	<0.00%	29.88%	3.55
4	23.85%	<0.00%	22.07%	<0.00%	13.86%	2.78
8	23.85%	<0.00%	22.07%	<0.00%	10.92%	2.64
16	23.85%	<0.00%	22.07%	<0.00%	10.05%	2.60
32	23.85%	<0.00%	22.07%	<0.00%	9.52%	2.57
64	23.85%	<0.00%	22.07%	<0.00%	9.45%	2.57
128	23.85%	<0.00%	22.07%	<0.00%	9.12%	2.55
CONSTSTACKHARVARD: Direct Mapped						
1	23.85%	<0.00%	n/a	0.41%	20.50%	6.04
2	23.85%	<0.00%	n/a	0.41%	14.75%	4.94
4	23.85%	<0.00%	n/a	0.34%	8.19%	3.68
8	23.85%	<0.00%	n/a	0.33%	4.65%	3.01
16	23.85%	<0.00%	n/a	<0.00%	3.60%	2.75
32	23.85%	<0.00%	n/a	<0.00%	2.92%	2.62
64	23.85%	<0.00%	n/a	<0.00%	2.74%	2.59
128	23.85%	<0.00%	n/a	<0.00%	2.60%	2.56
CONSTSTACKHARVARD: Eight-way Associative						
1	23.85%	<0.00%	n/a	<0.00%	13.74%	5.34
2	23.85%	<0.00%	n/a	<0.00%	7.75%	4.04
4	23.85%	<0.00%	n/a	<0.00%	3.51%	3.12
8	23.85%	<0.00%	n/a	<0.00%	2.75%	2.95

continued on next page

Table D.6: db Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
16	23.85%	<0.00%	n/a	<0.00%	2.53%	2.91
32	23.85%	<0.00%	n/a	<0.00%	2.44%	2.89
64	23.85%	<0.00%	n/a	<0.00%	2.37%	2.87
128	23.85%	<0.00%	n/a	<0.00%	2.35%	2.87
CONSTSTACKUNI: Direct Mapped						
1	23.85%	<0.00%	n/a	n/a	18.16%	9.31
2	23.85%	<0.00%	n/a	n/a	14.80%	8.12
4	23.85%	<0.00%	n/a	n/a	5.99%	4.97
8	23.85%	<0.00%	n/a	n/a	3.52%	4.09
16	23.85%	<0.00%	n/a	n/a	2.54%	3.74
32	23.85%	<0.00%	n/a	n/a	1.84%	3.49
64	23.85%	<0.00%	n/a	n/a	1.47%	3.36
128	23.85%	<0.00%	n/a	n/a	1.39%	3.33
CONSTSTACKUNI: Eight-way Associative						
1	23.85%	<0.00%	n/a	n/a	9.10%	6.93
2	23.85%	<0.00%	n/a	n/a	5.43%	5.43
4	23.85%	<0.00%	n/a	n/a	2.31%	4.17
8	23.85%	<0.00%	n/a	n/a	1.54%	3.85
16	23.85%	<0.00%	n/a	n/a	1.37%	3.78
32	23.85%	<0.00%	n/a	n/a	1.28%	3.75
64	23.85%	<0.00%	n/a	n/a	1.26%	3.74
128	23.85%	<0.00%	n/a	n/a	1.23%	3.73
CONSTUNI: Direct Mapped						
1	23.85%	n/a	n/a	n/a	16.04%	11.41
2	23.85%	n/a	n/a	n/a	11.17%	9.04
4	23.85%	n/a	n/a	n/a	4.55%	5.82
8	23.85%	n/a	n/a	n/a	2.63%	4.88
16	23.85%	n/a	n/a	n/a	1.89%	4.52
32	23.85%	n/a	n/a	n/a	1.36%	4.27
64	23.85%	n/a	n/a	n/a	1.09%	4.13
128	23.85%	n/a	n/a	n/a	1.02%	4.10
CONSTUNI: Eight-way Associative						
1	23.85%	n/a	n/a	n/a	7.36%	8.19
2	23.85%	n/a	n/a	n/a	4.38%	6.54

continued on next page

Table D.6: db Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
4	23.85%	n/a	n/a	n/a	1.73%	5.07
8	23.85%	n/a	n/a	n/a	1.14%	4.74
16	23.85%	n/a	n/a	n/a	1.01%	4.67
32	23.85%	n/a	n/a	n/a	0.94%	4.63
64	23.85%	n/a	n/a	n/a	0.93%	4.62
128	23.85%	n/a	n/a	n/a	0.90%	4.61
CONSTUNIREG: Direct Mapped						
1	23.85%	n/a	14.44%	n/a	23.30%	7.02
2	23.85%	n/a	14.32%	n/a	11.70%	4.54
4	23.85%	n/a	14.32%	n/a	7.17%	3.58
8	23.85%	n/a	14.32%	n/a	4.22%	2.95
16	23.85%	n/a	14.32%	n/a	3.12%	2.72
32	23.85%	n/a	14.32%	n/a	2.29%	2.54
64	23.85%	n/a	14.32%	n/a	1.85%	2.45
128	23.85%	n/a	14.32%	n/a	1.76%	2.43
CONSTUNIREG: Eight-way Associative						
1	23.85%	n/a	14.33%	n/a	21.73%	7.61
2	23.85%	n/a	14.32%	n/a	15.06%	5.99
4	23.85%	n/a	14.32%	n/a	3.68%	3.23
8	23.85%	n/a	14.32%	n/a	2.26%	2.89
16	23.85%	n/a	14.32%	n/a	1.78%	2.77
32	23.85%	n/a	14.32%	n/a	1.66%	2.74
64	23.85%	n/a	14.32%	n/a	1.60%	2.73
128	23.85%	n/a	14.32%	n/a	1.59%	2.73
HARREG: Direct Mapped						
1	n/a	n/a	15.93%	0.41%	49.53%	4.24
2	n/a	n/a	15.87%	0.41%	23.09%	2.83
4	n/a	n/a	15.87%	0.34%	15.68%	2.43
8	n/a	n/a	15.87%	0.33%	8.58%	2.05
16	n/a	n/a	15.87%	<0.00%	6.40%	1.88
32	n/a	n/a	15.87%	<0.00%	5.15%	1.82
64	n/a	n/a	15.87%	<0.00%	4.41%	1.78
128	n/a	n/a	15.87%	<0.00%	4.26%	1.77
HARREG: Eight-way Associative						

continued on next page

Table D.6: db Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
1	n/a	n/a	15.90%	<0.00%	55.48%	5.11
2	n/a	n/a	15.87%	<0.00%	29.17%	3.52
4	n/a	n/a	15.87%	<0.00%	8.59%	2.28
8	n/a	n/a	15.87%	<0.00%	5.19%	2.07
16	n/a	n/a	15.87%	<0.00%	4.17%	2.01
32	n/a	n/a	15.87%	<0.00%	4.01%	2.00
64	n/a	n/a	15.87%	<0.00%	3.77%	1.99
128	n/a	n/a	15.87%	<0.00%	3.73%	1.99
STACKHARREG: Direct Mapped						
1	n/a	<0.00%	23.68%	0.41%	58.16%	4.40
2	n/a	<0.00%	23.65%	0.41%	46.04%	3.82
4	n/a	<0.00%	23.65%	0.34%	32.29%	3.14
8	n/a	<0.00%	23.65%	0.33%	17.85%	2.44
16	n/a	<0.00%	23.65%	<0.00%	13.55%	2.18
32	n/a	<0.00%	23.65%	<0.00%	11.05%	2.06
64	n/a	<0.00%	23.65%	<0.00%	9.76%	2.00
128	n/a	<0.00%	23.65%	<0.00%	9.48%	1.98
STACKHARREG: Eight-way Associative						
1	n/a	<0.00%	23.67%	<0.00%	54.14%	4.72
2	n/a	<0.00%	23.65%	<0.00%	32.31%	3.52
4	n/a	<0.00%	23.65%	<0.00%	14.95%	2.56
8	n/a	<0.00%	23.65%	<0.00%	10.29%	2.31
16	n/a	<0.00%	23.65%	<0.00%	9.14%	2.24
32	n/a	<0.00%	23.65%	<0.00%	8.74%	2.22
64	n/a	<0.00%	23.65%	<0.00%	8.45%	2.21
128	n/a	<0.00%	23.65%	<0.00%	8.49%	2.21
STACKHARVARD: Direct Mapped						
1	n/a	<0.00%	n/a	0.41%	21.34%	6.16
2	n/a	<0.00%	n/a	0.41%	15.93%	5.06
4	n/a	<0.00%	n/a	0.34%	9.32%	3.70
8	n/a	<0.00%	n/a	0.33%	5.13%	2.84
16	n/a	<0.00%	n/a	<0.00%	3.79%	2.52
32	n/a	<0.00%	n/a	<0.00%	3.03%	2.36
64	n/a	<0.00%	n/a	<0.00%	2.64%	2.28

continued on next page

Table D.6: db Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
128	n/a	<0.00%	n/a	<0.00%	2.55%	2.26
STACKHARVARD: Eight-way Associative						
1	n/a	<0.00%	n/a	<0.00%	15.31%	5.55
2	n/a	<0.00%	n/a	<0.00%	8.88%	4.05
4	n/a	<0.00%	n/a	<0.00%	4.01%	2.92
8	n/a	<0.00%	n/a	<0.00%	2.74%	2.62
16	n/a	<0.00%	n/a	<0.00%	2.44%	2.55
32	n/a	<0.00%	n/a	<0.00%	2.33%	2.53
64	n/a	<0.00%	n/a	<0.00%	2.25%	2.51
128	n/a	<0.00%	n/a	<0.00%	2.23%	2.50
STACKUNI: Direct Mapped						
1	n/a	<0.00%	n/a	n/a	18.64%	9.47
2	n/a	<0.00%	n/a	n/a	15.55%	8.32
4	n/a	<0.00%	n/a	n/a	6.70%	5.04
8	n/a	<0.00%	n/a	n/a	3.83%	3.98
16	n/a	<0.00%	n/a	n/a	2.68%	3.55
32	n/a	<0.00%	n/a	n/a	1.94%	3.28
64	n/a	<0.00%	n/a	n/a	1.47%	3.10
128	n/a	<0.00%	n/a	n/a	1.41%	3.08
STACKUNI: Eight-way Associative						
1	n/a	<0.00%	n/a	n/a	10.24%	7.24
2	n/a	<0.00%	n/a	n/a	6.29%	5.57
4	n/a	<0.00%	n/a	n/a	2.72%	4.07
8	n/a	<0.00%	n/a	n/a	1.60%	3.59
16	n/a	<0.00%	n/a	n/a	1.36%	3.49
32	n/a	<0.00%	n/a	n/a	1.30%	3.47
64	n/a	<0.00%	n/a	n/a	1.24%	3.44
128	n/a	<0.00%	n/a	n/a	1.20%	3.42
STACKUNIREG: Direct Mapped						
1	n/a	<0.00%	23.68%	n/a	18.42%	5.68
2	n/a	<0.00%	23.65%	n/a	15.53%	5.06
4	n/a	<0.00%	23.65%	n/a	9.99%	3.87
8	n/a	<0.00%	23.65%	n/a	5.77%	2.96
16	n/a	<0.00%	23.65%	n/a	4.18%	2.62

continued on next page

Table D.6: db Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
32	n/a	<0.00%	23.65%	n/a	3.09%	2.38
64	n/a	<0.00%	23.65%	n/a	2.41%	2.24
128	n/a	<0.00%	23.65%	n/a	2.34%	2.22
STACKUNIREG: Eight-way Associative						
1	n/a	<0.00%	23.68%	n/a	16.30%	5.96
2	n/a	<0.00%	23.65%	n/a	10.22%	4.47
4	n/a	<0.00%	23.65%	n/a	4.54%	3.07
8	n/a	<0.00%	23.65%	n/a	2.69%	2.62
16	n/a	<0.00%	23.65%	n/a	2.28%	2.52
32	n/a	<0.00%	23.65%	n/a	2.16%	2.49
64	n/a	<0.00%	23.65%	n/a	2.12%	2.48
128	n/a	<0.00%	23.65%	n/a	2.01%	2.45
UNIREG: Direct Mapped						
1	n/a	n/a	15.93%	n/a	24.06%	7.05
2	n/a	n/a	15.87%	n/a	12.96%	4.60
4	n/a	n/a	15.87%	n/a	8.25%	3.57
8	n/a	n/a	15.87%	n/a	4.74%	2.80
16	n/a	n/a	15.87%	n/a	3.41%	2.51
32	n/a	n/a	15.87%	n/a	2.51%	2.31
64	n/a	n/a	15.87%	n/a	1.87%	2.17
128	n/a	n/a	15.87%	n/a	1.80%	2.15
UNIREG: Eight-way Associative						
1	n/a	n/a	15.91%	n/a	24.60%	8.17
2	n/a	n/a	15.87%	n/a	15.79%	5.96
4	n/a	n/a	15.87%	n/a	4.31%	3.08
8	n/a	n/a	15.87%	n/a	2.25%	2.57
16	n/a	n/a	15.87%	n/a	1.80%	2.45
32	n/a	n/a	15.87%	n/a	1.78%	2.45
64	n/a	n/a	15.87%	n/a	1.65%	2.42
128	n/a	n/a	15.87%	n/a	1.64%	2.41

Table D.7: jack Hybrid Cache Results

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
CONSTHARREG: Direct Mapped						
1	4.98%	n/a	11.69%	0.98%	31.67%	3.10
2	4.98%	n/a	11.69%	0.66%	22.36%	2.70
4	4.98%	n/a	11.69%	0.45%	16.50%	2.44
8	4.98%	n/a	11.69%	0.19%	9.93%	2.16
16	4.98%	n/a	11.69%	0.07%	6.45%	2.01
32	4.98%	n/a	11.69%	<0.00%	3.73%	1.90
64	4.98%	n/a	11.69%	<0.00%	2.42%	1.84
128	4.98%	n/a	11.69%	<0.00%	1.58%	1.81
CONSTHARREG: Eight-way Associative						
1	4.98%	n/a	11.68%	0.82%	46.30%	4.17
2	4.98%	n/a	11.69%	0.37%	26.28%	3.21
4	4.98%	n/a	11.69%	0.08%	14.01%	2.63
8	4.98%	n/a	11.69%	0.01%	7.93%	2.35
16	4.98%	n/a	11.69%	<0.00%	3.35%	2.14
32	4.98%	n/a	11.69%	<0.00%	2.01%	2.08
64	4.98%	n/a	11.69%	<0.00%	1.37%	2.06
128	4.98%	n/a	11.69%	<0.00%	1.19%	2.05
CONSTHARVARD: Direct Mapped						
1	4.98%	n/a	n/a	0.98%	20.59%	3.10
2	4.98%	n/a	n/a	0.66%	14.36%	2.70
4	4.98%	n/a	n/a	0.45%	11.47%	2.44
8	4.98%	n/a	n/a	0.19%	6.43%	2.16
16	4.98%	n/a	n/a	0.07%	1.80%	2.01
32	4.98%	n/a	n/a	<0.00%	1.05%	1.90
64	4.98%	n/a	n/a	<0.00%	0.70%	1.84
128	4.98%	n/a	n/a	<0.00%	0.45%	1.81
CONSTHARVARD: Eight-way Associative						
1	4.98%	n/a	n/a	0.83%	6.10%	4.17
2	4.98%	n/a	n/a	0.36%	4.41%	3.21
4	4.98%	n/a	n/a	0.08%	2.89%	2.63
8	4.98%	n/a	n/a	0.01%	1.77%	2.35
16	4.98%	n/a	n/a	<0.00%	0.82%	2.14
32	4.98%	n/a	n/a	<0.00%	0.48%	2.08

continued on next page

Table D.7: jack Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
64	4.98%	n/a	n/a	<0.00%	0.37%	2.06
128	4.98%	n/a	n/a	<0.00%	0.35%	2.05
CONSTSTACKHARREG: Direct Mapped						
1	4.98%	<0.00%	15.97%	0.98%	61.02%	3.96
2	4.98%	<0.00%	15.97%	0.66%	47.66%	3.46
4	4.98%	<0.00%	15.97%	0.45%	34.28%	2.97
8	4.98%	<0.00%	15.97%	0.19%	22.31%	2.53
16	4.98%	<0.00%	15.97%	0.07%	14.91%	2.26
32	4.98%	<0.00%	15.97%	<0.00%	9.22%	2.05
64	4.98%	<0.00%	15.97%	<0.00%	6.22%	1.95
128	4.98%	<0.00%	15.97%	<0.00%	4.05%	1.87
CONSTSTACKHARREG: Eight-way Associative						
1	4.98%	<0.00%	15.98%	0.83%	54.19%	4.23
2	4.98%	<0.00%	15.97%	0.37%	40.52%	3.63
4	4.98%	<0.00%	15.97%	0.08%	27.19%	3.06
8	4.98%	<0.00%	15.97%	0.01%	16.48%	2.63
16	4.98%	<0.00%	15.97%	<0.00%	7.57%	2.27
32	4.98%	<0.00%	15.97%	<0.00%	4.47%	2.15
64	4.98%	<0.00%	15.97%	<0.00%	3.67%	2.12
128	4.98%	<0.00%	15.97%	<0.00%	3.10%	2.10
CONSTSTACKHARVARD: Direct Mapped						
1	4.98%	<0.00%	n/a	0.98%	29.72%	8.61
2	4.98%	<0.00%	n/a	0.66%	21.42%	6.76
4	4.98%	<0.00%	n/a	0.45%	17.24%	5.83
8	4.98%	<0.00%	n/a	0.19%	9.69%	4.16
16	4.98%	<0.00%	n/a	0.07%	2.70%	2.62
32	4.98%	<0.00%	n/a	<0.00%	1.60%	2.38
64	4.98%	<0.00%	n/a	<0.00%	1.07%	2.26
128	4.98%	<0.00%	n/a	<0.00%	0.69%	2.18
CONSTSTACKHARVARD: Eight-way Associative						
1	4.98%	<0.00%	n/a	0.82%	8.87%	4.62
2	4.98%	<0.00%	n/a	0.37%	6.49%	3.97
4	4.98%	<0.00%	n/a	0.08%	4.33%	3.40
8	4.98%	<0.00%	n/a	0.01%	2.68%	2.98

continued on next page

Table D.7: jack Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
16	4.98%	<0.00%	n/a	<0.00%	1.21%	2.61
32	4.98%	<0.00%	n/a	<0.00%	0.74%	2.50
64	4.98%	<0.00%	n/a	<0.00%	0.60%	2.46
128	4.98%	<0.00%	n/a	<0.00%	0.52%	2.44
ConstStackUni: Direct Mapped						
1	4.98%	<0.00%	n/a	n/a	21.12%	9.87
2	4.98%	<0.00%	n/a	n/a	15.34%	7.96
4	4.98%	<0.00%	n/a	n/a	12.28%	6.95
8	4.98%	<0.00%	n/a	n/a	7.06%	5.23
16	4.98%	<0.00%	n/a	n/a	2.29%	3.65
32	4.98%	<0.00%	n/a	n/a	1.26%	3.31
64	4.98%	<0.00%	n/a	n/a	0.82%	3.17
128	4.98%	<0.00%	n/a	n/a	0.50%	3.06
ConstStackUni: Eight-way Associative						
1	4.98%	<0.00%	n/a	n/a	6.89%	5.90
2	4.98%	<0.00%	n/a	n/a	5.18%	5.25
4	4.98%	<0.00%	n/a	n/a	3.56%	4.64
8	4.98%	<0.00%	n/a	n/a	2.30%	4.17
16	4.98%	<0.00%	n/a	n/a	1.10%	3.72
32	4.98%	<0.00%	n/a	n/a	0.55%	3.51
64	4.98%	<0.00%	n/a	n/a	0.43%	3.46
128	4.98%	<0.00%	n/a	n/a	0.34%	3.43
ConstUni: Direct Mapped						
1	4.98%	n/a	n/a	n/a	30.69%	17.33
2	4.98%	n/a	n/a	n/a	11.71%	8.84
4	4.98%	n/a	n/a	n/a	9.27%	7.75
8	4.98%	n/a	n/a	n/a	5.31%	5.98
16	4.98%	n/a	n/a	n/a	1.72%	4.37
32	4.98%	n/a	n/a	n/a	0.93%	4.02
64	4.98%	n/a	n/a	n/a	0.61%	3.88
128	4.98%	n/a	n/a	n/a	0.37%	3.77
ConstUni: Eight-way Associative						
1	4.98%	n/a	n/a	n/a	5.47%	6.90
2	4.98%	n/a	n/a	n/a	3.99%	6.14

continued on next page

Table D.7: jack Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
4	4.98%	n/a	n/a	n/a	2.69%	5.48
8	4.98%	n/a	n/a	n/a	1.73%	4.99
16	4.98%	n/a	n/a	n/a	0.81%	4.52
32	4.98%	n/a	n/a	n/a	0.40%	4.31
64	4.98%	n/a	n/a	n/a	0.31%	4.26
128	4.98%	n/a	n/a	n/a	0.26%	4.24
CONSTUNIREG: Direct Mapped						
1	4.98%	n/a	11.69%	n/a	39.05%	7.82
2	4.98%	n/a	11.69%	n/a	11.90%	3.71
4	4.98%	n/a	11.69%	n/a	8.90%	3.25
8	4.98%	n/a	11.69%	n/a	5.61%	2.76
16	4.98%	n/a	11.69%	n/a	3.75%	2.47
32	4.98%	n/a	11.69%	n/a	2.04%	2.21
64	4.98%	n/a	11.69%	n/a	1.30%	2.10
128	4.98%	n/a	11.69%	n/a	0.79%	2.03
CONSTUNIREG: Eight-way Associative						
1	4.98%	n/a	11.68%	n/a	24.70%	6.44
2	4.98%	n/a	11.69%	n/a	12.92%	4.40
4	4.98%	n/a	11.69%	n/a	7.85%	3.53
8	4.98%	n/a	11.69%	n/a	4.65%	2.97
16	4.98%	n/a	11.69%	n/a	2.06%	2.53
32	4.98%	n/a	11.69%	n/a	1.04%	2.35
64	4.98%	n/a	11.69%	n/a	0.61%	2.28
128	4.98%	n/a	11.69%	n/a	0.63%	2.28
HARREG: Direct Mapped						
1	n/a	n/a	15.47%	0.98%	30.49%	3.40
2	n/a	n/a	15.45%	0.66%	21.84%	2.85
4	n/a	n/a	15.45%	0.45%	16.54%	2.51
8	n/a	n/a	15.45%	0.19%	10.51%	2.12
16	n/a	n/a	15.45%	0.07%	7.14%	1.90
32	n/a	n/a	15.45%	<0.00%	4.12%	1.71
64	n/a	n/a	15.45%	<0.00%	2.71%	1.63
128	n/a	n/a	15.45%	<0.00%	1.60%	1.56
HARREG: Eight-way Associative						

continued on next page

Table D.7: jack Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
1	n/a	n/a	15.44%	0.83%	47.97%	5.05
2	n/a	n/a	15.45%	0.37%	23.93%	3.35
4	n/a	n/a	15.45%	0.08%	15.36%	2.73
8	n/a	n/a	15.45%	0.01%	8.10%	2.22
16	n/a	n/a	15.45%	<0.00%	3.70%	1.92
32	n/a	n/a	15.45%	<0.00%	1.86%	1.80
64	n/a	n/a	15.45%	<0.00%	1.24%	1.75
128	n/a	n/a	15.45%	<0.00%	1.06%	1.74
STACKHARREG: Direct Mapped						
1	n/a	<0.00%	17.36%	0.98%	54.41%	4.08
2	n/a	<0.00%	17.38%	0.66%	43.14%	3.51
4	n/a	<0.00%	17.38%	0.45%	32.29%	2.98
8	n/a	<0.00%	17.38%	0.19%	22.23%	2.48
16	n/a	<0.00%	17.38%	0.07%	15.42%	2.15
32	n/a	<0.00%	17.38%	<0.00%	9.38%	1.85
64	n/a	<0.00%	17.38%	<0.00%	6.37%	1.71
128	n/a	<0.00%	17.38%	<0.00%	3.77%	1.59
STACKHARREG: Eight-way Associative						
1	n/a	<0.00%	17.37%	0.82%	49.74%	4.38
2	n/a	<0.00%	17.38%	0.37%	37.39%	3.66
4	n/a	<0.00%	17.38%	0.08%	25.77%	3.00
8	n/a	<0.00%	17.38%	0.01%	16.48%	2.49
16	n/a	<0.00%	17.38%	<0.00%	7.54%	2.02
32	n/a	<0.00%	17.38%	<0.00%	4.24%	1.84
64	n/a	<0.00%	17.38%	<0.00%	3.17%	1.78
128	n/a	<0.00%	17.38%	<0.00%	2.56%	1.75
STACKHARVARD: Direct Mapped						
1	n/a	<0.00%	n/a	0.98%	33.22%	10.85
2	n/a	<0.00%	n/a	0.66%	26.41%	8.98
4	n/a	<0.00%	n/a	0.45%	21.25%	7.56
8	n/a	<0.00%	n/a	0.19%	8.86%	4.18
16	n/a	<0.00%	n/a	0.07%	3.02%	2.59
32	n/a	<0.00%	n/a	<0.00%	1.75%	2.24
64	n/a	<0.00%	n/a	<0.00%	1.18%	2.08

continued on next page

Table D.7: jack Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
128	n/a	<0.00%	n/a	<0.00%	0.69%	1.95
STACKHARVARD: Eight-way Associative						
1	n/a	<0.00%	n/a	0.83%	8.64%	4.78
2	n/a	<0.00%	n/a	0.36%	6.44%	4.04
4	n/a	<0.00%	n/a	0.08%	4.45%	3.39
8	n/a	<0.00%	n/a	0.01%	2.88%	2.90
16	n/a	<0.00%	n/a	<0.00%	1.34%	2.43
32	n/a	<0.00%	n/a	<0.00%	0.75%	2.24
64	n/a	<0.00%	n/a	<0.00%	0.56%	2.19
128	n/a	<0.00%	n/a	<0.00%	0.46%	2.16
STACKUNI: Direct Mapped						
1	n/a	<0.00%	n/a	n/a	24.81%	12.34
2	n/a	<0.00%	n/a	n/a	19.72%	10.39
4	n/a	<0.00%	n/a	n/a	15.82%	8.90
8	n/a	<0.00%	n/a	n/a	6.86%	5.47
16	n/a	<0.00%	n/a	n/a	2.57%	3.83
32	n/a	<0.00%	n/a	n/a	1.44%	3.40
64	n/a	<0.00%	n/a	n/a	0.95%	3.21
128	n/a	<0.00%	n/a	n/a	0.53%	3.05
STACKUNI: Eight-way Associative						
1	n/a	<0.00%	n/a	n/a	7.38%	6.46
2	n/a	<0.00%	n/a	n/a	5.28%	5.55
4	n/a	<0.00%	n/a	n/a	3.78%	4.89
8	n/a	<0.00%	n/a	n/a	2.54%	4.35
16	n/a	<0.00%	n/a	n/a	1.29%	3.81
32	n/a	<0.00%	n/a	n/a	0.61%	3.51
64	n/a	<0.00%	n/a	n/a	0.39%	3.41
128	n/a	<0.00%	n/a	n/a	0.33%	3.39
STACKUNIREG: Direct Mapped						
1	n/a	<0.00%	17.36%	n/a	18.49%	4.55
2	n/a	<0.00%	17.38%	n/a	14.88%	3.97
4	n/a	<0.00%	17.38%	n/a	11.35%	3.41
8	n/a	<0.00%	17.38%	n/a	8.01%	2.88
16	n/a	<0.00%	17.38%	n/a	5.64%	2.50

continued on next page

Table D.7: jack Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
32	n/a	<0.00%	17.38%	n/a	3.29%	2.12
64	n/a	<0.00%	17.38%	n/a	2.19%	1.95
128	n/a	<0.00%	17.38%	n/a	1.22%	1.80
STACKUNIREG: Eight-way Associative						
1	n/a	<0.00%	17.37%	n/a	16.97%	4.91
2	n/a	<0.00%	17.38%	n/a	13.18%	4.22
4	n/a	<0.00%	17.38%	n/a	9.29%	3.51
8	n/a	<0.00%	17.38%	n/a	6.18%	2.95
16	n/a	<0.00%	17.38%	n/a	3.11%	2.39
32	n/a	<0.00%	17.38%	n/a	1.42%	2.08
64	n/a	<0.00%	17.38%	n/a	0.97%	2.00
128	n/a	<0.00%	17.38%	n/a	0.78%	1.97
UNIREG: Direct Mapped						
1	n/a	n/a	15.44%	n/a	38.86%	8.40
2	n/a	n/a	15.45%	n/a	12.55%	3.87
4	n/a	n/a	15.45%	n/a	9.58%	3.35
8	n/a	n/a	15.45%	n/a	6.29%	2.79
16	n/a	n/a	15.45%	n/a	4.33%	2.45
32	n/a	n/a	15.45%	n/a	2.42%	2.12
64	n/a	n/a	15.45%	n/a	1.57%	1.97
128	n/a	n/a	15.45%	n/a	0.88%	1.86
UNIREG: Eight-way Associative						
1	n/a	n/a	15.44%	n/a	27.59%	7.36
2	n/a	n/a	15.45%	n/a	13.44%	4.58
4	n/a	n/a	15.45%	n/a	9.11%	3.73
8	n/a	n/a	15.45%	n/a	5.13%	2.95
16	n/a	n/a	15.45%	n/a	2.49%	2.43
32	n/a	n/a	15.45%	n/a	1.08%	2.15
64	n/a	n/a	15.45%	n/a	0.81%	2.10
128	n/a	n/a	15.45%	n/a	0.63%	2.07

Table D.8: javac Hybrid Cache Results

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
CONSTHARREG: Direct Mapped						
1	16.99%	n/a	27.67%	2.08%	60.89%	6.28
2	16.99%	n/a	27.62%	0.80%	49.08%	5.28
4	16.99%	n/a	27.62%	0.58%	35.75%	4.49
8	16.99%	n/a	27.62%	0.28%	14.64%	3.25
16	16.99%	n/a	27.62%	0.11%	9.17%	2.91
32	16.99%	n/a	27.62%	0.05%	7.14%	2.78
64	16.99%	n/a	27.62%	0.03%	4.53%	2.63
128	16.99%	n/a	27.62%	0.02%	3.00%	2.55
CONSTHARREG: Eight-way Associative						
1	16.99%	n/a	27.46%	1.47%	52.33%	6.41
2	16.99%	n/a	27.62%	0.77%	27.33%	4.65
4	16.99%	n/a	27.62%	0.33%	13.30%	3.64
8	16.99%	n/a	27.62%	0.09%	8.37%	3.26
16	16.99%	n/a	27.62%	0.01%	4.52%	3.00
32	16.99%	n/a	27.62%	<0.00%	3.60%	2.94
64	16.99%	n/a	27.62%	<0.00%	2.60%	2.87
128	16.99%	n/a	27.62%	<0.00%	2.19%	2.85
CONSTHARVARD: Direct Mapped						
1	16.99%	n/a	n/a	2.08%	27.03%	8.47
2	16.99%	n/a	n/a	0.80%	18.76%	6.49
4	16.99%	n/a	n/a	0.58%	11.97%	5.08
8	16.99%	n/a	n/a	0.28%	6.17%	3.85
16	16.99%	n/a	n/a	0.11%	3.65%	3.31
32	16.99%	n/a	n/a	0.05%	2.86%	3.14
64	16.99%	n/a	n/a	0.03%	1.88%	2.94
128	16.99%	n/a	n/a	0.02%	1.27%	2.81
CONSTHARVARD: Eight-way Associative						
1	16.99%	n/a	n/a	1.47%	17.21%	7.25
2	16.99%	n/a	n/a	0.77%	9.29%	5.25
4	16.99%	n/a	n/a	0.33%	4.61%	4.05
8	16.99%	n/a	n/a	0.09%	2.89%	3.59
16	16.99%	n/a	n/a	0.01%	1.79%	3.32
32	16.99%	n/a	n/a	<0.00%	1.32%	3.21

continued on next page

Table D.8: javac Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
64	16.99%	n/a	n/a	<0.00%	1.05%	3.15
128	16.99%	n/a	n/a	<0.00%	0.95%	3.13
CONSTSTACKHARREG: Direct Mapped						
1	16.99%	<0.00%	36.66%	2.08%	59.67%	6.00
2	16.99%	<0.00%	36.75%	0.80%	48.35%	5.08
4	16.99%	<0.00%	36.75%	0.58%	33.09%	4.23
8	16.99%	<0.00%	36.75%	0.28%	19.47%	3.45
16	16.99%	<0.00%	36.75%	0.11%	12.00%	3.02
32	16.99%	<0.00%	36.75%	0.05%	9.42%	2.87
64	16.99%	<0.00%	36.75%	0.03%	5.97%	2.68
128	16.99%	<0.00%	36.75%	0.02%	4.64%	2.61
CONSTSTACKHARREG: Eight-way Associative						
1	16.99%	<0.00%	36.64%	1.47%	53.49%	6.29
2	16.99%	<0.00%	36.75%	0.77%	30.60%	4.73
4	16.99%	<0.00%	36.75%	0.33%	16.48%	3.77
8	16.99%	<0.00%	36.75%	0.09%	10.45%	3.34
16	16.99%	<0.00%	36.75%	0.01%	6.52%	3.08
32	16.99%	<0.00%	36.75%	<0.00%	4.83%	2.98
64	16.99%	<0.00%	36.75%	<0.00%	3.99%	2.93
128	16.99%	<0.00%	36.75%	<0.00%	3.41%	2.90
CONSTSTACKHARVARD: Direct Mapped						
1	16.99%	<0.00%	n/a	2.08%	33.11%	7.65
2	16.99%	<0.00%	n/a	0.80%	22.82%	5.86
4	16.99%	<0.00%	n/a	0.58%	14.69%	4.66
8	16.99%	<0.00%	n/a	0.28%	8.19%	3.67
16	16.99%	<0.00%	n/a	0.11%	4.72%	3.13
32	16.99%	<0.00%	n/a	0.05%	3.68%	2.97
64	16.99%	<0.00%	n/a	0.03%	2.31%	2.77
128	16.99%	<0.00%	n/a	0.02%	1.79%	2.70
CONSTSTACKHARVARD: Eight-way Associative						
1	16.99%	<0.00%	n/a	1.48%	22.98%	6.91
2	16.99%	<0.00%	n/a	0.77%	12.55%	5.03
4	16.99%	<0.00%	n/a	0.33%	6.38%	3.90
8	16.99%	<0.00%	n/a	0.09%	4.04%	3.46

continued on next page

Table D.8: javac Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
16	16.99%	<0.00%	n/a	0.01%	2.43%	3.18
32	16.99%	<0.00%	n/a	<0.00%	1.85%	3.08
64	16.99%	<0.00%	n/a	<0.00%	1.53%	3.03
128	16.99%	<0.00%	n/a	<0.00%	1.31%	2.99
CONSTSTACKUNI: Direct Mapped						
1	16.99%	<0.00%	n/a	n/a	15.61%	9.40
2	16.99%	<0.00%	n/a	n/a	10.95%	7.49
4	16.99%	<0.00%	n/a	n/a	7.30%	5.99
8	16.99%	<0.00%	n/a	n/a	4.45%	4.82
16	16.99%	<0.00%	n/a	n/a	2.80%	4.15
32	16.99%	<0.00%	n/a	n/a	1.92%	3.79
64	16.99%	<0.00%	n/a	n/a	1.06%	3.44
128	16.99%	<0.00%	n/a	n/a	0.81%	3.33
CONSTSTACKUNI: Eight-way Associative						
1	16.99%	<0.00%	n/a	n/a	11.94%	9.00
2	16.99%	<0.00%	n/a	n/a	7.36%	6.86
4	16.99%	<0.00%	n/a	n/a	3.65%	5.13
8	16.99%	<0.00%	n/a	n/a	2.25%	4.48
16	16.99%	<0.00%	n/a	n/a	1.28%	4.02
32	16.99%	<0.00%	n/a	n/a	0.79%	3.79
64	16.99%	<0.00%	n/a	n/a	0.59%	3.70
128	16.99%	<0.00%	n/a	n/a	0.47%	3.64
CONSTUNI: Direct Mapped						
1	16.99%	n/a	n/a	n/a	16.24%	10.94
2	16.99%	n/a	n/a	n/a	10.94%	8.46
4	16.99%	n/a	n/a	n/a	7.36%	6.79
8	16.99%	n/a	n/a	n/a	4.26%	5.34
16	16.99%	n/a	n/a	n/a	2.78%	4.65
32	16.99%	n/a	n/a	n/a	1.99%	4.28
64	16.99%	n/a	n/a	n/a	1.03%	3.83
128	16.99%	n/a	n/a	n/a	0.71%	3.68
CONSTUNI: Eight-way Associative						
1	16.99%	n/a	n/a	n/a	11.04%	9.70
2	16.99%	n/a	n/a	n/a	6.74%	7.41

continued on next page

Table D.8: javac Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
4	16.99%	n/a	n/a	n/a	3.27%	5.56
8	16.99%	n/a	n/a	n/a	2.00%	4.88
16	16.99%	n/a	n/a	n/a	1.13%	4.42
32	16.99%	n/a	n/a	n/a	0.69%	4.18
64	16.99%	n/a	n/a	n/a	0.52%	4.09
128	16.99%	n/a	n/a	n/a	0.41%	4.03
CONSTUNIREG: Direct Mapped						
1	16.99%	n/a	27.67%	n/a	19.41%	8.89
2	16.99%	n/a	27.62%	n/a	15.25%	7.54
4	16.99%	n/a	27.62%	n/a	11.76%	6.41
8	16.99%	n/a	27.62%	n/a	6.39%	4.67
16	16.99%	n/a	27.62%	n/a	4.61%	4.09
32	16.99%	n/a	27.62%	n/a	3.53%	3.74
64	16.99%	n/a	27.62%	n/a	1.38%	3.04
128	16.99%	n/a	27.62%	n/a	0.93%	2.90
CONSTUNIREG: Eight-way Associative						
1	16.99%	n/a	27.50%	n/a	17.41%	9.38
2	16.99%	n/a	27.62%	n/a	10.82%	6.96
4	16.99%	n/a	27.62%	n/a	5.36%	4.94
8	16.99%	n/a	27.62%	n/a	3.25%	4.16
16	16.99%	n/a	27.62%	n/a	1.72%	3.60
32	16.99%	n/a	27.62%	n/a	1.09%	3.36
64	16.99%	n/a	27.62%	n/a	0.66%	3.20
128	16.99%	n/a	27.62%	n/a	0.59%	3.18
HARREG: Direct Mapped						
1	n/a	n/a	29.68%	2.08%	59.39%	6.44
2	n/a	n/a	29.75%	0.80%	47.71%	5.30
4	n/a	n/a	29.75%	0.58%	34.36%	4.33
8	n/a	n/a	29.75%	0.28%	16.22%	3.00
16	n/a	n/a	29.75%	0.11%	9.91%	2.52
32	n/a	n/a	29.75%	0.05%	7.13%	2.32
64	n/a	n/a	29.75%	0.03%	4.77%	2.15
128	n/a	n/a	29.75%	0.02%	3.35%	2.05
HARREG: Eight-way Associative						

continued on next page

Table D.8: javac Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
1	n/a	n/a	29.65%	1.47%	54.35%	6.76
2	n/a	n/a	29.75%	0.77%	31.78%	4.79
4	n/a	n/a	29.75%	0.33%	15.75%	3.40
8	n/a	n/a	29.75%	0.09%	9.30%	2.82
16	n/a	n/a	29.75%	0.01%	5.39%	2.49
32	n/a	n/a	29.75%	<0.00%	3.46%	2.34
64	n/a	n/a	29.75%	<0.00%	2.55%	2.27
128	n/a	n/a	29.75%	<0.00%	2.10%	2.23
STACKHARREG: Direct Mapped						
1	n/a	<0.00%	37.60%	2.08%	61.25%	6.35
2	n/a	<0.00%	37.71%	0.80%	49.76%	5.26
4	n/a	<0.00%	37.71%	0.58%	34.08%	4.18
8	n/a	<0.00%	37.71%	0.28%	21.11%	3.25
16	n/a	<0.00%	37.71%	0.11%	12.85%	2.67
32	n/a	<0.00%	37.71%	0.05%	9.21%	2.41
64	n/a	<0.00%	37.71%	0.03%	6.22%	2.21
128	n/a	<0.00%	37.71%	0.02%	4.81%	2.12
STACKHARREG: Eight-way Associative						
1	n/a	<0.00%	37.59%	1.47%	59.56%	6.92
2	n/a	<0.00%	37.71%	0.77%	35.19%	4.91
4	n/a	<0.00%	37.71%	0.33%	18.79%	3.55
8	n/a	<0.00%	37.71%	0.09%	11.92%	2.97
16	n/a	<0.00%	37.71%	0.01%	6.98%	2.58
32	n/a	<0.00%	37.71%	<0.00%	4.70%	2.40
64	n/a	<0.00%	37.71%	<0.00%	3.75%	2.33
128	n/a	<0.00%	37.71%	<0.00%	3.01%	2.28
STACKHARVARD: Direct Mapped						
1	n/a	<0.00%	n/a	2.08%	33.03%	8.16
2	n/a	<0.00%	n/a	0.80%	23.85%	6.23
4	n/a	<0.00%	n/a	0.58%	15.42%	4.71
8	n/a	<0.00%	n/a	0.28%	8.84%	3.49
16	n/a	<0.00%	n/a	0.11%	5.13%	2.80
32	n/a	<0.00%	n/a	0.05%	3.62%	2.52
64	n/a	<0.00%	n/a	0.03%	2.44%	2.31

continued on next page

Table D.8: javac Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
128	n/a	<0.00%	n/a	0.02%	1.88%	2.22
STACKHARVARD: Eight-way Associative						
1	n/a	<0.00%	n/a	1.48%	25.54%	7.64
2	n/a	<0.00%	n/a	0.77%	14.50%	5.25
4	n/a	<0.00%	n/a	0.33%	7.51%	3.73
8	n/a	<0.00%	n/a	0.09%	4.67%	3.10
16	n/a	<0.00%	n/a	0.01%	2.73%	2.69
32	n/a	<0.00%	n/a	<0.00%	1.82%	2.51
64	n/a	<0.00%	n/a	<0.00%	1.46%	2.44
128	n/a	<0.00%	n/a	<0.00%	1.20%	2.39
STACKUNI: Direct Mapped						
1	n/a	<0.00%	n/a	n/a	16.97%	10.08
2	n/a	<0.00%	n/a	n/a	12.53%	8.12
4	n/a	<0.00%	n/a	n/a	8.30%	6.25
8	n/a	<0.00%	n/a	n/a	5.12%	4.84
16	n/a	<0.00%	n/a	n/a	3.34%	4.06
32	n/a	<0.00%	n/a	n/a	2.08%	3.50
64	n/a	<0.00%	n/a	n/a	1.21%	3.11
128	n/a	<0.00%	n/a	n/a	0.92%	2.98
STACKUNI: Eight-way Associative						
1	n/a	<0.00%	n/a	n/a	13.62%	9.81
2	n/a	<0.00%	n/a	n/a	9.06%	7.51
4	n/a	<0.00%	n/a	n/a	4.61%	5.27
8	n/a	<0.00%	n/a	n/a	2.80%	4.35
16	n/a	<0.00%	n/a	n/a	1.55%	3.72
32	n/a	<0.00%	n/a	n/a	0.90%	3.39
64	n/a	<0.00%	n/a	n/a	0.61%	3.25
128	n/a	<0.00%	n/a	n/a	0.48%	3.18
STACKUNIREG: Direct Mapped						
1	n/a	<0.00%	37.58%	n/a	16.71%	7.64
2	n/a	<0.00%	37.71%	n/a	13.62%	6.62
4	n/a	<0.00%	37.71%	n/a	9.67%	5.30
8	n/a	<0.00%	37.71%	n/a	6.34%	4.18
16	n/a	<0.00%	37.71%	n/a	4.32%	3.51

continued on next page

Table D.8: javac Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
32	n/a	<0.00%	37.71%	n/a	2.72%	2.97
64	n/a	<0.00%	37.71%	n/a	1.59%	2.59
128	n/a	<0.00%	37.71%	n/a	1.20%	2.46
STACKUNIREG: Eight-way Associative						
1	n/a	<0.00%	37.58%	n/a	16.40%	8.60
2	n/a	<0.00%	37.71%	n/a	11.14%	6.60
4	n/a	<0.00%	37.71%	n/a	5.87%	4.59
8	n/a	<0.00%	37.71%	n/a	3.61%	3.73
16	n/a	<0.00%	37.71%	n/a	2.04%	3.13
32	n/a	<0.00%	37.71%	n/a	1.17%	2.79
64	n/a	<0.00%	37.71%	n/a	0.82%	2.66
128	n/a	<0.00%	37.71%	n/a	0.63%	2.59
UNIREG: Direct Mapped						
1	n/a	n/a	29.69%	n/a	20.61%	9.05
2	n/a	n/a	29.75%	n/a	16.50%	7.66
4	n/a	n/a	29.75%	n/a	12.39%	6.28
8	n/a	n/a	29.75%	n/a	6.83%	4.39
16	n/a	n/a	29.75%	n/a	4.79%	3.71
32	n/a	n/a	29.75%	n/a	3.26%	3.19
64	n/a	n/a	29.75%	n/a	1.58%	2.62
128	n/a	n/a	29.75%	n/a	1.11%	2.46
UNIREG: Eight-way Associative						
1	n/a	n/a	29.65%	n/a	19.01%	9.70
2	n/a	n/a	29.75%	n/a	12.93%	7.36
4	n/a	n/a	29.75%	n/a	6.58%	4.92
8	n/a	n/a	29.75%	n/a	3.84%	3.86
16	n/a	n/a	29.75%	n/a	2.09%	3.18
32	n/a	n/a	29.75%	n/a	1.21%	2.84
64	n/a	n/a	29.75%	n/a	0.86%	2.71
128	n/a	n/a	29.75%	n/a	0.64%	2.62

Table D.9: jess Hybrid Cache Results

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
CONSTHARREG: Direct Mapped						
1	13.72%	n/a	32.53%	2.25%	66.26%	9.85
2	13.72%	n/a	32.28%	1.59%	49.61%	7.93
4	13.72%	n/a	32.28%	0.63%	39.36%	6.72
8	13.72%	n/a	32.28%	0.40%	24.92%	5.14
16	13.72%	n/a	32.28%	0.19%	12.22%	3.76
32	13.72%	n/a	32.28%	0.09%	7.96%	3.29
64	13.72%	n/a	32.28%	0.07%	5.88%	3.06
128	13.72%	n/a	32.28%	0.03%	1.90%	2.63
CONSTHARREG: Eight-way Associative						
1	13.72%	n/a	32.50%	1.08%	57.75%	10.02
2	13.72%	n/a	32.28%	0.49%	41.48%	7.90
4	13.72%	n/a	32.28%	0.14%	24.41%	5.76
8	13.72%	n/a	32.28%	0.04%	15.42%	4.65
16	13.72%	n/a	32.28%	0.01%	8.45%	3.80
32	13.72%	n/a	32.28%	<0.00%	4.87%	3.36
64	13.72%	n/a	32.28%	<0.00%	2.26%	3.04
128	13.72%	n/a	32.28%	<0.00%	1.42%	2.94
CONSTHARVARD: Direct Mapped						
1	13.72%	n/a	n/a	2.25%	29.06%	12.58
2	13.72%	n/a	n/a	1.59%	18.81%	9.09
4	13.72%	n/a	n/a	0.63%	12.92%	7.02
8	13.72%	n/a	n/a	0.40%	7.07%	5.06
16	13.72%	n/a	n/a	0.19%	4.29%	4.11
32	13.72%	n/a	n/a	0.09%	2.30%	3.43
64	13.72%	n/a	n/a	0.07%	1.46%	3.15
128	13.72%	n/a	n/a	0.03%	0.90%	2.96
CONSTHARVARD: Eight-way Associative						
1	13.72%	n/a	n/a	1.08%	21.12%	11.17
2	13.72%	n/a	n/a	0.49%	11.98%	7.63
4	13.72%	n/a	n/a	0.14%	5.71%	5.21
8	13.72%	n/a	n/a	0.04%	2.98%	4.17
16	13.72%	n/a	n/a	0.01%	1.61%	3.64
32	13.72%	n/a	n/a	<0.00%	1.03%	3.42

continued on next page

Table D.9: jess Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
64	13.72%	n/a	n/a	<0.00%	0.79%	3.33
128	13.72%	n/a	n/a	<0.00%	0.63%	3.27
CONSTSTACKHARREG: Direct Mapped						
1	13.72%	<0.00%	43.99%	2.25%	59.00%	8.89
2	13.72%	<0.00%	44.01%	1.59%	42.81%	7.11
4	13.72%	<0.00%	44.01%	0.63%	30.29%	5.68
8	13.72%	<0.00%	44.01%	0.40%	19.53%	4.52
16	13.72%	<0.00%	44.01%	0.19%	12.37%	3.74
32	13.72%	<0.00%	44.01%	0.09%	6.64%	3.12
64	13.72%	<0.00%	44.01%	0.07%	4.26%	2.87
128	13.72%	<0.00%	44.01%	0.03%	2.70%	2.70
CONSTSTACKHARREG: Eight-way Associative						
1	13.72%	<0.00%	44.03%	1.08%	53.94%	9.37
2	13.72%	<0.00%	44.01%	0.49%	33.70%	6.86
4	13.72%	<0.00%	44.01%	0.14%	17.03%	4.81
8	13.72%	<0.00%	44.01%	0.04%	9.06%	3.85
16	13.72%	<0.00%	44.01%	0.01%	4.95%	3.35
32	13.72%	<0.00%	44.01%	<0.00%	3.13%	3.13
64	13.72%	<0.00%	44.01%	<0.00%	2.40%	3.04
128	13.72%	<0.00%	44.01%	<0.00%	1.99%	2.99
CONSTSTACKHARVARD: Direct Mapped						
1	13.72%	<0.00%	n/a	2.25%	34.70%	11.03
2	13.72%	<0.00%	n/a	1.59%	23.27%	8.22
4	13.72%	<0.00%	n/a	0.63%	15.57%	6.27
8	13.72%	<0.00%	n/a	0.40%	9.55%	4.80
16	13.72%	<0.00%	n/a	0.19%	5.83%	3.89
32	13.72%	<0.00%	n/a	0.09%	3.14%	3.23
64	13.72%	<0.00%	n/a	0.07%	2.02%	2.96
128	13.72%	<0.00%	n/a	0.03%	1.25%	2.77
CONSTSTACKHARVARD: Eight-way Associative						
1	13.72%	<0.00%	n/a	1.08%	27.54%	10.47
2	13.72%	<0.00%	n/a	0.49%	15.86%	7.20
4	13.72%	<0.00%	n/a	0.14%	7.73%	4.94
8	13.72%	<0.00%	n/a	0.04%	4.06%	3.93

continued on next page

Table D.9: jess Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
16	13.72%	<0.00%	n/a	0.01%	2.24%	3.43
32	13.72%	<0.00%	n/a	<0.00%	1.40%	3.20
64	13.72%	<0.00%	n/a	<0.00%	1.05%	3.11
128	13.72%	<0.00%	n/a	<0.00%	0.88%	3.06
ConstStackUni: Direct Mapped						
1	13.72%	<0.00%	n/a	n/a	27.81%	13.41
2	13.72%	<0.00%	n/a	n/a	19.04%	10.26
4	13.72%	<0.00%	n/a	n/a	13.10%	8.13
8	13.72%	<0.00%	n/a	n/a	7.88%	6.26
16	13.72%	<0.00%	n/a	n/a	4.96%	5.21
32	13.72%	<0.00%	n/a	n/a	2.82%	4.44
64	13.72%	<0.00%	n/a	n/a	1.60%	4.00
128	13.72%	<0.00%	n/a	n/a	0.95%	3.77
ConstStackUni: Eight-way Associative						
1	13.72%	<0.00%	n/a	n/a	22.06%	12.93
2	13.72%	<0.00%	n/a	n/a	13.08%	9.26
4	13.72%	<0.00%	n/a	n/a	6.64%	6.63
8	13.72%	<0.00%	n/a	n/a	3.42%	5.31
16	13.72%	<0.00%	n/a	n/a	1.77%	4.63
32	13.72%	<0.00%	n/a	n/a	1.04%	4.33
64	13.72%	<0.00%	n/a	n/a	0.73%	4.21
128	13.72%	<0.00%	n/a	n/a	0.59%	4.15
ConstUni: Direct Mapped						
1	13.72%	n/a	n/a	n/a	25.64%	15.57
2	13.72%	n/a	n/a	n/a	16.69%	11.53
4	13.72%	n/a	n/a	n/a	11.67%	9.26
8	13.72%	n/a	n/a	n/a	6.41%	6.88
16	13.72%	n/a	n/a	n/a	4.01%	5.80
32	13.72%	n/a	n/a	n/a	2.27%	5.01
64	13.72%	n/a	n/a	n/a	1.28%	4.57
128	13.72%	n/a	n/a	n/a	0.75%	4.33
ConstUni: Eight-way Associative						
1	13.72%	n/a	n/a	n/a	18.72%	14.19
2	13.72%	n/a	n/a	n/a	10.93%	10.18

continued on next page

Table D.9: jess Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
4	13.72%	n/a	n/a	n/a	5.42%	7.34
8	13.72%	n/a	n/a	n/a	2.76%	5.97
16	13.72%	n/a	n/a	n/a	1.43%	5.28
32	13.72%	n/a	n/a	n/a	0.85%	4.98
64	13.72%	n/a	n/a	n/a	0.58%	4.85
128	13.72%	n/a	n/a	n/a	0.46%	4.78
CONSTUNIREG: Direct Mapped						
1	13.72%	n/a	32.54%	n/a	42.34%	12.51
2	13.72%	n/a	32.28%	n/a	31.74%	10.06
4	13.72%	n/a	32.28%	n/a	25.03%	8.54
8	13.72%	n/a	32.28%	n/a	15.87%	6.46
16	13.72%	n/a	32.28%	n/a	8.15%	4.70
32	13.72%	n/a	32.28%	n/a	5.30%	4.06
64	13.72%	n/a	32.28%	n/a	3.64%	3.68
128	13.72%	n/a	32.28%	n/a	1.22%	3.13
CONSTUNIREG: Eight-way Associative						
1	13.72%	n/a	32.50%	n/a	37.60%	13.03
2	13.72%	n/a	32.28%	n/a	26.96%	10.23
4	13.72%	n/a	32.28%	n/a	16.42%	7.50
8	13.72%	n/a	32.28%	n/a	9.77%	5.78
16	13.72%	n/a	32.28%	n/a	5.39%	4.65
32	13.72%	n/a	32.28%	n/a	3.03%	4.04
64	13.72%	n/a	32.28%	n/a	1.25%	3.58
128	13.72%	n/a	32.28%	n/a	0.78%	3.45
HARREG: Direct Mapped						
1	n/a	n/a	34.78%	2.25%	63.62%	10.45
2	n/a	n/a	34.48%	1.59%	46.87%	8.10
4	n/a	n/a	34.48%	0.63%	36.64%	6.64
8	n/a	n/a	34.48%	0.40%	19.04%	4.31
16	n/a	n/a	34.48%	0.19%	10.20%	3.12
32	n/a	n/a	34.48%	0.09%	5.67%	2.52
64	n/a	n/a	34.48%	0.07%	3.36%	2.21
128	n/a	n/a	34.48%	0.03%	2.06%	2.04
HARREG: Eight-way Associative						

continued on next page

Table D.9: jess Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
1	n/a	n/a	34.54%	1.08%	55.75%	10.50
2	n/a	n/a	34.48%	0.49%	37.78%	7.72
4	n/a	n/a	34.48%	0.14%	18.31%	4.77
8	n/a	n/a	34.48%	0.04%	8.26%	3.25
16	n/a	n/a	34.48%	0.01%	4.14%	2.63
32	n/a	n/a	34.48%	<0.00%	2.47%	2.38
64	n/a	n/a	34.48%	<0.00%	1.64%	2.26
128	n/a	n/a	34.48%	<0.00%	1.41%	2.22
STACKHARREG: Direct Mapped						
1	n/a	<0.00%	44.70%	2.25%	59.07%	9.60
2	n/a	<0.00%	44.75%	1.59%	43.40%	7.52
4	n/a	<0.00%	44.75%	0.63%	31.60%	5.89
8	n/a	<0.00%	44.75%	0.40%	21.02%	4.50
16	n/a	<0.00%	44.75%	0.19%	13.20%	3.47
32	n/a	<0.00%	44.75%	0.09%	7.30%	2.70
64	n/a	<0.00%	44.75%	0.07%	4.48%	2.34
128	n/a	<0.00%	44.75%	0.03%	2.78%	2.11
STACKHARREG: Eight-way Associative						
1	n/a	<0.00%	44.72%	1.07%	54.83%	10.17
2	n/a	<0.00%	44.75%	0.48%	35.48%	7.26
4	n/a	<0.00%	44.75%	0.14%	18.51%	4.73
8	n/a	<0.00%	44.75%	0.04%	9.72%	3.43
16	n/a	<0.00%	44.75%	0.01%	5.27%	2.77
32	n/a	<0.00%	44.75%	<0.00%	3.28%	2.48
64	n/a	<0.00%	44.75%	<0.00%	2.24%	2.33
128	n/a	<0.00%	44.75%	<0.00%	1.80%	2.26
STACKHARVARD: Direct Mapped						
1	n/a	<0.00%	n/a	2.25%	33.68%	11.75
2	n/a	<0.00%	n/a	1.59%	23.24%	8.67
4	n/a	<0.00%	n/a	0.63%	16.01%	6.48
8	n/a	<0.00%	n/a	0.40%	10.27%	4.81
16	n/a	<0.00%	n/a	0.19%	6.25%	3.63
32	n/a	<0.00%	n/a	0.09%	3.44%	2.81
64	n/a	<0.00%	n/a	0.07%	2.11%	2.43

continued on next page

Table D.9: jess Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
128	n/a	<0.00%	n/a	0.03%	1.29%	2.19
STACKHARVARD: Eight-way Associative						
1	n/a	<0.00%	n/a	1.08%	27.76%	11.30
2	n/a	<0.00%	n/a	0.48%	16.90%	7.66
4	n/a	<0.00%	n/a	0.14%	8.58%	4.89
8	n/a	<0.00%	n/a	0.04%	4.47%	3.54
16	n/a	<0.00%	n/a	0.01%	2.40%	2.85
32	n/a	<0.00%	n/a	<0.00%	1.50%	2.56
64	n/a	<0.00%	n/a	<0.00%	1.05%	2.41
128	n/a	<0.00%	n/a	<0.00%	0.86%	2.35
STACKUNI: Direct Mapped						
1	n/a	<0.00%	n/a	n/a	28.19%	14.43
2	n/a	<0.00%	n/a	n/a	19.94%	11.08
4	n/a	<0.00%	n/a	n/a	14.25%	8.76
8	n/a	<0.00%	n/a	n/a	8.63%	6.47
16	n/a	<0.00%	n/a	n/a	5.40%	5.16
32	n/a	<0.00%	n/a	n/a	3.10%	4.22
64	n/a	<0.00%	n/a	n/a	1.73%	3.67
128	n/a	<0.00%	n/a	n/a	1.02%	3.38
STACKUNI: Eight-way Associative						
1	n/a	<0.00%	n/a	n/a	23.07%	14.08
2	n/a	<0.00%	n/a	n/a	14.43%	10.07
4	n/a	<0.00%	n/a	n/a	7.50%	6.86
8	n/a	<0.00%	n/a	n/a	3.90%	5.18
16	n/a	<0.00%	n/a	n/a	2.04%	4.32
32	n/a	<0.00%	n/a	n/a	1.17%	3.92
64	n/a	<0.00%	n/a	n/a	0.78%	3.74
128	n/a	<0.00%	n/a	n/a	0.62%	3.66
STACKUNIREG: Direct Mapped						
1	n/a	<0.00%	44.70%	n/a	36.26%	11.27
2	n/a	<0.00%	44.75%	n/a	26.99%	8.97
4	n/a	<0.00%	44.75%	n/a	20.05%	7.25
8	n/a	<0.00%	44.75%	n/a	12.98%	5.49
16	n/a	<0.00%	44.75%	n/a	8.32%	4.33

continued on next page

Table D.9: jess Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
32	n/a	<0.00%	44.75%	n/a	4.73%	3.44
64	n/a	<0.00%	44.75%	n/a	2.72%	2.94
128	n/a	<0.00%	44.75%	n/a	1.63%	2.67
STACKUNIREG: Eight-way Associative						
1	n/a	<0.00%	44.72%	n/a	33.26%	11.99
2	n/a	<0.00%	44.75%	n/a	22.39%	8.92
4	n/a	<0.00%	44.75%	n/a	11.96%	5.97
8	n/a	<0.00%	44.75%	n/a	6.29%	4.37
16	n/a	<0.00%	44.75%	n/a	3.31%	3.52
32	n/a	<0.00%	44.75%	n/a	1.89%	3.12
64	n/a	<0.00%	44.75%	n/a	1.22%	2.93
128	n/a	<0.00%	44.75%	n/a	1.02%	2.87
UNIREG: Direct Mapped						
1	n/a	n/a	34.78%	n/a	43.03%	13.15
2	n/a	n/a	34.48%	n/a	31.86%	10.28
4	n/a	n/a	34.48%	n/a	24.94%	8.55
8	n/a	n/a	34.48%	n/a	13.10%	5.58
16	n/a	n/a	34.48%	n/a	7.33%	4.13
32	n/a	n/a	34.48%	n/a	4.17%	3.34
64	n/a	n/a	34.48%	n/a	2.33%	2.87
128	n/a	n/a	34.48%	n/a	1.39%	2.64
UNIREG: Eight-way Associative						
1	n/a	n/a	34.54%	n/a	38.07%	13.51
2	n/a	n/a	34.48%	n/a	26.83%	10.29
4	n/a	n/a	34.48%	n/a	14.05%	6.63
8	n/a	n/a	34.48%	n/a	6.06%	4.34
16	n/a	n/a	34.48%	n/a	2.95%	3.45
32	n/a	n/a	34.48%	n/a	1.65%	3.08
64	n/a	n/a	34.48%	n/a	1.04%	2.91
128	n/a	n/a	34.48%	n/a	0.85%	2.85

Table D.10: linpack Hybrid Cache Results

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
CONSTHARREG: Direct Mapped						
1	0.22%	n/a	19.11%	0.02%	26.97%	3.37
2	0.22%	n/a	19.35%	<0.00%	14.80%	2.58
4	0.22%	n/a	19.35%	<0.00%	3.92%	1.85
8	0.22%	n/a	19.35%	<0.00%	2.26%	1.73
16	0.22%	n/a	19.35%	<0.00%	1.70%	1.70
32	0.22%	n/a	19.35%	<0.00%	1.45%	1.68
64	0.22%	n/a	19.35%	<0.00%	1.13%	1.66
128	0.22%	n/a	19.35%	<0.00%	1.11%	1.66
CONSTHARREG: Eight-way Associative						
1	0.22%	n/a	19.34%	<0.00%	26.40%	3.83
2	0.22%	n/a	19.35%	<0.00%	8.17%	2.43
4	0.22%	n/a	19.35%	<0.00%	2.58%	2.00
8	0.22%	n/a	19.35%	<0.00%	2.07%	1.96
16	0.22%	n/a	19.35%	<0.00%	1.51%	1.92
32	0.22%	n/a	19.35%	<0.00%	1.32%	1.90
64	0.22%	n/a	19.35%	<0.00%	1.20%	1.90
128	0.22%	n/a	19.35%	<0.00%	1.09%	1.89
CONSTHARVARD: Direct Mapped						
1	0.22%	n/a	n/a	0.02%	13.52%	6.71
2	0.22%	n/a	n/a	<0.00%	6.75%	4.35
4	0.22%	n/a	n/a	<0.00%	1.84%	2.65
8	0.22%	n/a	n/a	<0.00%	1.10%	2.39
16	0.22%	n/a	n/a	<0.00%	0.83%	2.30
32	0.22%	n/a	n/a	<0.00%	0.67%	2.24
64	0.22%	n/a	n/a	<0.00%	0.55%	2.20
128	0.22%	n/a	n/a	<0.00%	0.55%	2.20
CONSTHARVARD: Eight-way Associative						
1	0.22%	n/a	n/a	<0.00%	8.57%	5.69
2	0.22%	n/a	n/a	<0.00%	2.08%	3.11
4	0.22%	n/a	n/a	<0.00%	1.08%	2.72
8	0.22%	n/a	n/a	<0.00%	0.89%	2.64
16	0.22%	n/a	n/a	<0.00%	0.66%	2.55
32	0.22%	n/a	n/a	<0.00%	0.55%	2.51

continued on next page

Table D.10: linpack Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
64	0.22%	n/a	n/a	<0.00%	0.54%	2.50
128	0.22%	n/a	n/a	<0.00%	0.53%	2.50
CONSTSTACKHARREG: Direct Mapped						
1	0.22%	<0.00%	40.25%	0.02%	47.98%	4.47
2	0.22%	<0.00%	38.66%	<0.00%	36.02%	3.65
4	0.22%	<0.00%	38.66%	<0.00%	8.80%	2.06
8	0.22%	<0.00%	38.66%	<0.00%	5.34%	1.86
16	0.22%	<0.00%	38.66%	<0.00%	4.07%	1.78
32	0.22%	<0.00%	38.66%	<0.00%	3.46%	1.75
64	0.22%	<0.00%	38.66%	<0.00%	2.88%	1.71
128	0.22%	<0.00%	38.66%	<0.00%	2.85%	1.71
CONSTSTACKHARREG: Eight-way Associative						
1	0.22%	<0.00%	38.08%	<0.00%	32.77%	3.90
2	0.22%	<0.00%	38.66%	<0.00%	9.37%	2.38
4	0.22%	<0.00%	38.66%	<0.00%	5.53%	2.13
8	0.22%	<0.00%	38.66%	<0.00%	4.64%	2.07
16	0.22%	<0.00%	38.66%	<0.00%	3.46%	1.99
32	0.22%	<0.00%	38.66%	<0.00%	2.88%	1.95
64	0.22%	<0.00%	38.66%	<0.00%	2.84%	1.95
128	0.22%	<0.00%	38.66%	<0.00%	2.80%	1.95
CONSTSTACKHARVARD: Direct Mapped						
1	0.22%	<0.00%	n/a	0.02%	26.52%	5.61
2	0.22%	<0.00%	n/a	<0.00%	15.33%	3.93
4	0.22%	<0.00%	n/a	<0.00%	4.11%	2.23
8	0.22%	<0.00%	n/a	<0.00%	2.51%	1.99
16	0.22%	<0.00%	n/a	<0.00%	1.91%	1.90
32	0.22%	<0.00%	n/a	<0.00%	1.53%	1.84
64	0.22%	<0.00%	n/a	<0.00%	1.27%	1.81
128	0.22%	<0.00%	n/a	<0.00%	1.26%	1.80
CONSTSTACKHARVARD: Eight-way Associative						
1	0.22%	<0.00%	n/a	<0.00%	16.16%	4.62
2	0.22%	<0.00%	n/a	<0.00%	4.27%	2.57
4	0.22%	<0.00%	n/a	<0.00%	2.42%	2.26
8	0.22%	<0.00%	n/a	<0.00%	2.03%	2.19

continued on next page

Table D.10: linpack Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
16	0.22%	<0.00%	n/a	<0.00%	1.52%	2.10
32	0.22%	<0.00%	n/a	<0.00%	1.26%	2.06
64	0.22%	<0.00%	n/a	<0.00%	1.24%	2.05
128	0.22%	<0.00%	n/a	<0.00%	1.23%	2.05
CONSTSTACKUNI: Direct Mapped						
1	0.22%	<0.00%	n/a	n/a	16.50%	6.60
2	0.22%	<0.00%	n/a	n/a	9.79%	4.82
4	0.22%	<0.00%	n/a	n/a	2.40%	2.86
8	0.22%	<0.00%	n/a	n/a	1.48%	2.61
16	0.22%	<0.00%	n/a	n/a	1.12%	2.51
32	0.22%	<0.00%	n/a	n/a	0.88%	2.45
64	0.22%	<0.00%	n/a	n/a	0.73%	2.41
128	0.22%	<0.00%	n/a	n/a	0.72%	2.41
CONSTSTACKUNI: Eight-way Associative						
1	0.22%	<0.00%	n/a	n/a	12.69%	6.37
2	0.22%	<0.00%	n/a	n/a	2.75%	3.36
4	0.22%	<0.00%	n/a	n/a	1.44%	2.96
8	0.22%	<0.00%	n/a	n/a	1.19%	2.89
16	0.22%	<0.00%	n/a	n/a	0.88%	2.79
32	0.22%	<0.00%	n/a	n/a	0.72%	2.75
64	0.22%	<0.00%	n/a	n/a	0.71%	2.74
128	0.22%	<0.00%	n/a	n/a	0.70%	2.74
CONSTUNI: Direct Mapped						
1	0.22%	n/a	n/a	n/a	11.01%	8.49
2	0.22%	n/a	n/a	n/a	5.70%	6.04
4	0.22%	n/a	n/a	n/a	1.42%	4.06
8	0.22%	n/a	n/a	n/a	0.86%	3.80
16	0.22%	n/a	n/a	n/a	0.65%	3.70
32	0.22%	n/a	n/a	n/a	0.51%	3.63
64	0.22%	n/a	n/a	n/a	0.42%	3.59
128	0.22%	n/a	n/a	n/a	0.41%	3.59
CONSTUNI: Eight-way Associative						
1	0.22%	n/a	n/a	n/a	8.57%	8.39
2	0.22%	n/a	n/a	n/a	1.78%	4.82

continued on next page

Table D.10: linpack Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
4	0.22%	n/a	n/a	n/a	0.85%	4.32
8	0.22%	n/a	n/a	n/a	0.69%	4.24
16	0.22%	n/a	n/a	n/a	0.51%	4.14
32	0.22%	n/a	n/a	n/a	0.41%	4.09
64	0.22%	n/a	n/a	n/a	0.41%	4.09
128	0.22%	n/a	n/a	n/a	0.40%	4.09
CONSTUNIREG: Direct Mapped						
1	0.22%	n/a	19.11%	n/a	17.39%	5.00
2	0.22%	n/a	19.35%	n/a	9.83%	3.64
4	0.22%	n/a	19.35%	n/a	2.40%	2.29
8	0.22%	n/a	19.35%	n/a	1.40%	2.11
16	0.22%	n/a	19.35%	n/a	1.04%	2.04
32	0.22%	n/a	19.35%	n/a	0.87%	2.01
64	0.22%	n/a	19.35%	n/a	0.68%	1.97
128	0.22%	n/a	19.35%	n/a	0.67%	1.97
CONSTUNIREG: Eight-way Associative						
1	0.22%	n/a	19.34%	n/a	16.02%	5.44
2	0.22%	n/a	19.35%	n/a	5.34%	3.22
4	0.22%	n/a	19.35%	n/a	1.57%	2.44
8	0.22%	n/a	19.35%	n/a	1.27%	2.37
16	0.22%	n/a	19.35%	n/a	0.83%	2.28
32	0.22%	n/a	19.35%	n/a	0.80%	2.28
64	0.22%	n/a	19.35%	n/a	0.66%	2.25
128	0.22%	n/a	19.35%	n/a	0.65%	2.25
HARREG: Direct Mapped						
1	n/a	n/a	24.88%	0.02%	27.36%	4.24
2	n/a	n/a	23.82%	<0.00%	14.44%	2.92
4	n/a	n/a	23.82%	<0.00%	5.17%	2.07
8	n/a	n/a	23.82%	<0.00%	2.89%	1.86
16	n/a	n/a	23.82%	<0.00%	1.49%	1.73
32	n/a	n/a	23.82%	<0.00%	1.23%	1.71
64	n/a	n/a	23.82%	<0.00%	1.02%	1.69
128	n/a	n/a	23.82%	<0.00%	1.00%	1.69
HARREG: Eight-way Associative						

continued on next page

Table D.10: linpack Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
1	n/a	n/a	23.60%	<0.00%	22.72%	4.17
2	n/a	n/a	23.82%	<0.00%	5.88%	2.44
4	n/a	n/a	23.82%	<0.00%	2.29%	2.06
8	n/a	n/a	23.82%	<0.00%	1.71%	2.00
16	n/a	n/a	23.82%	<0.00%	1.29%	1.96
32	n/a	n/a	23.82%	<0.00%	1.02%	1.93
64	n/a	n/a	23.82%	<0.00%	1.00%	1.92
128	n/a	n/a	23.82%	<0.00%	0.98%	1.92
STACKHARREG: Direct Mapped						
1	n/a	<0.00%	38.62%	0.02%	45.42%	4.82
2	n/a	<0.00%	39.07%	<0.00%	34.79%	4.08
4	n/a	<0.00%	39.07%	<0.00%	11.61%	2.38
8	n/a	<0.00%	39.07%	<0.00%	6.66%	2.01
16	n/a	<0.00%	39.07%	<0.00%	3.49%	1.78
32	n/a	<0.00%	39.07%	<0.00%	2.89%	1.74
64	n/a	<0.00%	39.07%	<0.00%	2.39%	1.70
128	n/a	<0.00%	39.07%	<0.00%	2.37%	1.70
STACKHARREG: Eight-way Associative						
1	n/a	<0.00%	38.62%	<0.00%	31.15%	4.31
2	n/a	<0.00%	39.07%	<0.00%	8.24%	2.43
4	n/a	<0.00%	39.07%	<0.00%	4.74%	2.13
8	n/a	<0.00%	39.07%	<0.00%	3.91%	2.07
16	n/a	<0.00%	39.07%	<0.00%	2.92%	1.98
32	n/a	<0.00%	39.07%	<0.00%	2.40%	1.94
64	n/a	<0.00%	39.07%	<0.00%	2.37%	1.94
128	n/a	<0.00%	39.07%	<0.00%	2.32%	1.93
STACKHARVARD: Direct Mapped						
1	n/a	<0.00%	n/a	0.02%	24.93%	6.30
2	n/a	<0.00%	n/a	<0.00%	15.94%	4.60
4	n/a	<0.00%	n/a	<0.00%	5.21%	2.59
8	n/a	<0.00%	n/a	<0.00%	3.01%	2.17
16	n/a	<0.00%	n/a	<0.00%	1.60%	1.91
32	n/a	<0.00%	n/a	<0.00%	1.26%	1.84
64	n/a	<0.00%	n/a	<0.00%	1.04%	1.80

continued on next page

Table D.10: linpack Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
128	n/a	<0.00%	n/a	<0.00%	1.02%	1.80
STACKHARVARD: Eight-way Associative						
1	n/a	<0.00%	n/a	<0.00%	15.11%	5.07
2	n/a	<0.00%	n/a	<0.00%	3.68%	2.62
4	n/a	<0.00%	n/a	<0.00%	2.04%	2.27
8	n/a	<0.00%	n/a	<0.00%	1.68%	2.19
16	n/a	<0.00%	n/a	<0.00%	1.25%	2.10
32	n/a	<0.00%	n/a	<0.00%	1.04%	2.05
64	n/a	<0.00%	n/a	<0.00%	1.01%	2.05
128	n/a	<0.00%	n/a	<0.00%	1.00%	2.05
STACKUNI: Direct Mapped						
1	n/a	<0.00%	n/a	n/a	16.75%	7.43
2	n/a	<0.00%	n/a	n/a	10.85%	5.65
4	n/a	<0.00%	n/a	n/a	3.30%	3.36
8	n/a	<0.00%	n/a	n/a	1.92%	2.94
16	n/a	<0.00%	n/a	n/a	1.02%	2.67
32	n/a	<0.00%	n/a	n/a	0.79%	2.60
64	n/a	<0.00%	n/a	n/a	0.65%	2.56
128	n/a	<0.00%	n/a	n/a	0.64%	2.55
STACKUNI: Eight-way Associative						
1	n/a	<0.00%	n/a	n/a	13.09%	7.21
2	n/a	<0.00%	n/a	n/a	2.78%	3.65
4	n/a	<0.00%	n/a	n/a	1.33%	3.15
8	n/a	<0.00%	n/a	n/a	1.07%	3.06
16	n/a	<0.00%	n/a	n/a	0.79%	2.96
32	n/a	<0.00%	n/a	n/a	0.64%	2.91
64	n/a	<0.00%	n/a	n/a	0.63%	2.91
128	n/a	<0.00%	n/a	n/a	0.62%	2.90
STACKUNIREG: Direct Mapped						
1	n/a	<0.00%	39.06%	n/a	21.28%	5.82
2	n/a	<0.00%	39.07%	n/a	15.85%	4.80
4	n/a	<0.00%	39.07%	n/a	4.90%	2.74
8	n/a	<0.00%	39.07%	n/a	2.83%	2.35
16	n/a	<0.00%	39.07%	n/a	1.48%	2.10

continued on next page

Table D.10: linpack Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
32	n/a	<0.00%	39.07%	n/a	1.21%	2.05
64	n/a	<0.00%	39.07%	n/a	1.00%	2.01
128	n/a	<0.00%	39.07%	n/a	0.99%	2.00
STACKUNIREG: Eight-way Associative						
1	n/a	<0.00%	38.62%	n/a	17.35%	5.77
2	n/a	<0.00%	39.07%	n/a	4.18%	2.97
4	n/a	<0.00%	39.07%	n/a	2.07%	2.52
8	n/a	<0.00%	39.07%	n/a	1.67%	2.43
16	n/a	<0.00%	39.07%	n/a	1.23%	2.34
32	n/a	<0.00%	39.07%	n/a	1.00%	2.29
64	n/a	<0.00%	39.07%	n/a	0.97%	2.28
128	n/a	<0.00%	39.07%	n/a	0.96%	2.28
UNIREG: Direct Mapped						
1	n/a	n/a	24.88%	n/a	18.40%	5.87
2	n/a	n/a	23.82%	n/a	9.96%	4.02
4	n/a	n/a	23.82%	n/a	3.29%	2.64
8	n/a	n/a	23.82%	n/a	1.86%	2.35
16	n/a	n/a	23.82%	n/a	0.96%	2.16
32	n/a	n/a	23.82%	n/a	0.78%	2.12
64	n/a	n/a	23.82%	n/a	0.64%	2.10
128	n/a	n/a	23.82%	n/a	0.63%	2.09
UNIREG: Eight-way Associative						
1	n/a	n/a	23.60%	n/a	17.00%	6.22
2	n/a	n/a	23.82%	n/a	4.18%	3.22
4	n/a	n/a	23.82%	n/a	1.49%	2.59
8	n/a	n/a	23.82%	n/a	1.10%	2.50
16	n/a	n/a	23.82%	n/a	0.80%	2.43
32	n/a	n/a	23.82%	n/a	0.64%	2.39
64	n/a	n/a	23.82%	n/a	0.63%	2.39
128	n/a	n/a	23.82%	n/a	0.62%	2.38

Table D.11: mpegaudio Hybrid Cache Results

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
CONSTHARREG: Direct Mapped						
1	6.36%	n/a	23.53%	1.01%	48.06%	5.86
2	6.36%	n/a	23.52%	0.69%	29.63%	4.31
4	6.36%	n/a	23.52%	0.47%	20.04%	3.50
8	6.36%	n/a	23.52%	0.14%	7.14%	2.40
16	6.36%	n/a	23.52%	0.03%	4.95%	2.21
32	6.36%	n/a	23.52%	0.02%	3.23%	2.07
64	6.36%	n/a	23.52%	0.01%	2.37%	2.00
128	6.36%	n/a	23.52%	0.01%	1.44%	1.92
CONSTHARREG: Eight-way Associative						
1	6.36%	n/a	23.51%	0.53%	41.21%	5.97
2	6.36%	n/a	23.52%	0.27%	18.94%	3.86
4	6.36%	n/a	23.52%	0.05%	7.72%	2.78
8	6.36%	n/a	23.52%	0.03%	3.62%	2.40
16	6.36%	n/a	23.52%	0.01%	2.03%	2.25
32	6.36%	n/a	23.52%	<0.00%	1.53%	2.20
64	6.36%	n/a	23.52%	<0.00%	1.11%	2.16
128	6.36%	n/a	23.52%	<0.00%	0.83%	2.13
CONSTHARVARD: Direct Mapped						
1	6.36%	n/a	n/a	1.01%	18.56%	8.76
2	6.36%	n/a	n/a	0.69%	12.47%	6.60
4	6.36%	n/a	n/a	0.47%	8.95%	5.34
8	6.36%	n/a	n/a	0.14%	4.26%	3.67
16	6.36%	n/a	n/a	0.03%	3.41%	3.36
32	6.36%	n/a	n/a	0.02%	2.67%	3.10
64	6.36%	n/a	n/a	0.01%	2.32%	2.98
128	6.36%	n/a	n/a	0.01%	2.05%	2.89
CONSTHARVARD: Eight-way Associative						
1	6.36%	n/a	n/a	0.53%	11.81%	7.24
2	6.36%	n/a	n/a	0.27%	4.28%	4.21
4	6.36%	n/a	n/a	0.05%	1.87%	3.22
8	6.36%	n/a	n/a	0.03%	1.01%	2.88
16	6.36%	n/a	n/a	0.01%	0.68%	2.75
32	6.36%	n/a	n/a	<0.00%	0.56%	2.70

continued on next page

Table D.11: mpegaudio Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
64	6.36%	n/a	n/a	<0.00%	0.43%	2.65
128	6.36%	n/a	n/a	<0.00%	0.34%	2.61
CONSTSTACKHARREG: Direct Mapped						
1	6.36%	<0.00%	32.17%	1.01%	42.93%	4.89
2	6.36%	<0.00%	32.18%	0.69%	32.86%	4.15
4	6.36%	<0.00%	32.18%	0.47%	23.80%	3.48
8	6.36%	<0.00%	32.18%	0.14%	10.22%	2.49
16	6.36%	<0.00%	32.18%	0.03%	7.33%	2.27
32	6.36%	<0.00%	32.18%	0.02%	5.01%	2.11
64	6.36%	<0.00%	32.18%	0.01%	3.52%	2.00
128	6.36%	<0.00%	32.18%	0.01%	2.47%	1.93
CONSTSTACKHARREG: Eight-way Associative						
1	6.36%	<0.00%	32.16%	0.53%	34.81%	4.86
2	6.36%	<0.00%	32.18%	0.27%	17.89%	3.47
4	6.36%	<0.00%	32.18%	0.05%	8.43%	2.68
8	6.36%	<0.00%	32.18%	0.03%	4.67%	2.38
16	6.36%	<0.00%	32.18%	0.01%	3.16%	2.26
32	6.36%	<0.00%	32.18%	<0.00%	2.56%	2.21
64	6.36%	<0.00%	32.18%	<0.00%	2.08%	2.17
128	6.36%	<0.00%	32.18%	<0.00%	1.69%	2.14
CONSTSTACKHARVARD: Direct Mapped						
1	6.36%	<0.00%	n/a	1.01%	25.19%	7.53
2	6.36%	<0.00%	n/a	0.69%	18.49%	6.03
4	6.36%	<0.00%	n/a	0.47%	14.01%	5.02
8	6.36%	<0.00%	n/a	0.14%	6.70%	3.39
16	6.36%	<0.00%	n/a	0.03%	5.40%	3.09
32	6.36%	<0.00%	n/a	0.02%	4.24%	2.84
64	6.36%	<0.00%	n/a	0.01%	3.69%	2.72
128	6.36%	<0.00%	n/a	0.01%	3.27%	2.63
CONSTSTACKHARVARD: Eight-way Associative						
1	6.36%	<0.00%	n/a	0.53%	14.65%	5.89
2	6.36%	<0.00%	n/a	0.27%	6.31%	3.78
4	6.36%	<0.00%	n/a	0.05%	2.87%	2.90
8	6.36%	<0.00%	n/a	0.03%	1.59%	2.58

continued on next page

Table D.11: mpegaudio Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
16	6.36%	<0.00%	n/a	0.01%	1.07%	2.45
32	6.36%	<0.00%	n/a	<0.00%	0.89%	2.40
64	6.36%	<0.00%	n/a	<0.00%	0.69%	2.35
128	6.36%	<0.00%	n/a	<0.00%	0.54%	2.32
CONSTSTACKUNI: Direct Mapped						
1	6.36%	<0.00%	n/a	n/a	19.70%	9.54
2	6.36%	<0.00%	n/a	n/a	14.34%	7.70
4	6.36%	<0.00%	n/a	n/a	10.99%	6.55
8	6.36%	<0.00%	n/a	n/a	5.48%	4.66
16	6.36%	<0.00%	n/a	n/a	4.41%	4.30
32	6.36%	<0.00%	n/a	n/a	3.59%	4.01
64	6.36%	<0.00%	n/a	n/a	2.50%	3.64
128	6.36%	<0.00%	n/a	n/a	2.21%	3.54
CONSTSTACKUNI: Eight-way Associative						
1	6.36%	<0.00%	n/a	n/a	12.30%	7.98
2	6.36%	<0.00%	n/a	n/a	5.52%	5.33
4	6.36%	<0.00%	n/a	n/a	2.63%	4.20
8	6.36%	<0.00%	n/a	n/a	1.25%	3.66
16	6.36%	<0.00%	n/a	n/a	0.79%	3.48
32	6.36%	<0.00%	n/a	n/a	0.57%	3.40
64	6.36%	<0.00%	n/a	n/a	0.52%	3.37
128	6.36%	<0.00%	n/a	n/a	0.37%	3.32
CONSTUNI: Direct Mapped						
1	6.36%	n/a	n/a	n/a	16.77%	11.50
2	6.36%	n/a	n/a	n/a	11.12%	8.82
4	6.36%	n/a	n/a	n/a	8.13%	7.41
8	6.36%	n/a	n/a	n/a	4.03%	5.47
16	6.36%	n/a	n/a	n/a	3.23%	5.09
32	6.36%	n/a	n/a	n/a	2.61%	4.80
64	6.36%	n/a	n/a	n/a	1.82%	4.42
128	6.36%	n/a	n/a	n/a	1.61%	4.32
CONSTUNI: Eight-way Associative						
1	6.36%	n/a	n/a	n/a	10.97%	9.98
2	6.36%	n/a	n/a	n/a	4.32%	6.39

continued on next page

Table D.11: mpegaudio Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
4	6.36%	n/a	n/a	n/a	1.97%	5.13
8	6.36%	n/a	n/a	n/a	0.91%	4.55
16	6.36%	n/a	n/a	n/a	0.57%	4.37
32	6.36%	n/a	n/a	n/a	0.40%	4.28
64	6.36%	n/a	n/a	n/a	0.35%	4.25
128	6.36%	n/a	n/a	n/a	0.26%	4.20
CONSTUNIREG: Direct Mapped						
1	6.36%	n/a	23.52%	n/a	30.07%	8.35
2	6.36%	n/a	23.52%	n/a	18.89%	6.04
4	6.36%	n/a	23.52%	n/a	13.21%	4.86
8	6.36%	n/a	23.52%	n/a	5.32%	3.23
16	6.36%	n/a	23.52%	n/a	3.85%	2.93
32	6.36%	n/a	23.52%	n/a	2.82%	2.71
64	6.36%	n/a	23.52%	n/a	1.47%	2.44
128	6.36%	n/a	23.52%	n/a	0.93%	2.32
CONSTUNIREG: Eight-way Associative						
1	6.36%	n/a	23.51%	n/a	27.10%	8.82
2	6.36%	n/a	23.52%	n/a	13.34%	5.57
4	6.36%	n/a	23.52%	n/a	6.08%	3.86
8	6.36%	n/a	23.52%	n/a	2.51%	3.02
16	6.36%	n/a	23.52%	n/a	1.23%	2.72
32	6.36%	n/a	23.52%	n/a	0.78%	2.61
64	6.36%	n/a	23.52%	n/a	0.63%	2.58
128	6.36%	n/a	23.52%	n/a	0.45%	2.54
HARREG: Direct Mapped						
1	n/a	n/a	26.30%	1.01%	44.12%	6.12
2	n/a	n/a	26.33%	0.69%	30.45%	4.74
4	n/a	n/a	26.33%	0.47%	21.26%	3.80
8	n/a	n/a	26.33%	0.14%	9.20%	2.57
16	n/a	n/a	26.33%	0.03%	6.89%	2.33
32	n/a	n/a	26.33%	0.02%	5.16%	2.16
64	n/a	n/a	26.33%	0.01%	4.34%	2.07
128	n/a	n/a	26.33%	0.01%	3.39%	1.98
HARREG: Eight-way Associative						

continued on next page

Table D.11: mpegaudio Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
1	n/a	n/a	26.29%	0.53%	37.74%	6.19
2	n/a	n/a	26.33%	0.27%	17.34%	3.86
4	n/a	n/a	26.33%	0.05%	7.45%	2.72
8	n/a	n/a	26.33%	0.03%	3.49%	2.27
16	n/a	n/a	26.33%	0.01%	2.09%	2.11
32	n/a	n/a	26.33%	<0.00%	1.26%	2.02
64	n/a	n/a	26.33%	<0.00%	1.19%	2.01
128	n/a	n/a	26.33%	<0.00%	0.81%	1.97
STACKHARREG: Direct Mapped						
1	n/a	<0.00%	35.75%	1.01%	46.35%	5.78
2	n/a	<0.00%	35.77%	0.69%	36.21%	4.86
4	n/a	<0.00%	35.77%	0.47%	27.31%	4.05
8	n/a	<0.00%	35.77%	0.14%	10.49%	2.54
16	n/a	<0.00%	35.77%	0.03%	7.27%	2.24
32	n/a	<0.00%	35.77%	0.02%	4.60%	2.01
64	n/a	<0.00%	35.77%	0.01%	3.29%	1.89
128	n/a	<0.00%	35.77%	0.01%	2.41%	1.81
STACKHARREG: Eight-way Associative						
1	n/a	<0.00%	35.75%	0.53%	37.65%	5.66
2	n/a	<0.00%	35.77%	0.26%	19.13%	3.77
4	n/a	<0.00%	35.77%	0.05%	9.03%	2.73
8	n/a	<0.00%	35.77%	0.03%	4.47%	2.28
16	n/a	<0.00%	35.77%	0.01%	2.88%	2.11
32	n/a	<0.00%	35.77%	<0.00%	2.32%	2.06
64	n/a	<0.00%	35.77%	<0.00%	1.90%	2.02
128	n/a	<0.00%	35.77%	<0.00%	1.41%	1.97
STACKHARVARD: Direct Mapped						
1	n/a	<0.00%	n/a	1.01%	27.26%	8.54
2	n/a	<0.00%	n/a	0.69%	20.11%	6.75
4	n/a	<0.00%	n/a	0.47%	15.36%	5.56
8	n/a	<0.00%	n/a	0.14%	6.87%	3.44
16	n/a	<0.00%	n/a	0.03%	5.28%	3.04
32	n/a	<0.00%	n/a	0.02%	4.11%	2.75
64	n/a	<0.00%	n/a	0.01%	3.59%	2.62

continued on next page

Table D.11: mpegaudio Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
128	n/a	<0.00%	n/a	0.01%	3.05%	2.49
STACKHARVARD: Eight-way Associative						
1	n/a	<0.00%	n/a	0.53%	16.95%	6.79
2	n/a	<0.00%	n/a	0.27%	7.53%	4.12
4	n/a	<0.00%	n/a	0.05%	3.40%	2.94
8	n/a	<0.00%	n/a	0.03%	1.66%	2.45
16	n/a	<0.00%	n/a	0.01%	1.10%	2.29
32	n/a	<0.00%	n/a	<0.00%	0.88%	2.23
64	n/a	<0.00%	n/a	<0.00%	0.75%	2.19
128	n/a	<0.00%	n/a	<0.00%	0.57%	2.14
STACKUNI: Direct Mapped						
1	n/a	<0.00%	n/a	n/a	21.31%	10.60
2	n/a	<0.00%	n/a	n/a	15.70%	8.53
4	n/a	<0.00%	n/a	n/a	11.96%	7.14
8	n/a	<0.00%	n/a	n/a	5.72%	4.84
16	n/a	<0.00%	n/a	n/a	4.41%	4.35
32	n/a	<0.00%	n/a	n/a	3.56%	4.04
64	n/a	<0.00%	n/a	n/a	2.52%	3.65
128	n/a	<0.00%	n/a	n/a	2.14%	3.51
STACKUNI: Eight-way Associative						
1	n/a	<0.00%	n/a	n/a	13.99%	9.00
2	n/a	<0.00%	n/a	n/a	6.50%	5.84
4	n/a	<0.00%	n/a	n/a	3.14%	4.42
8	n/a	<0.00%	n/a	n/a	1.41%	3.70
16	n/a	<0.00%	n/a	n/a	0.84%	3.45
32	n/a	<0.00%	n/a	n/a	0.61%	3.36
64	n/a	<0.00%	n/a	n/a	0.53%	3.32
128	n/a	<0.00%	n/a	n/a	0.37%	3.26
STACKUNIREG: Direct Mapped						
1	n/a	<0.00%	35.75%	n/a	23.62%	6.97
2	n/a	<0.00%	35.77%	n/a	18.81%	5.95
4	n/a	<0.00%	35.77%	n/a	14.28%	4.99
8	n/a	<0.00%	35.77%	n/a	6.34%	3.30
16	n/a	<0.00%	35.77%	n/a	4.54%	2.92

continued on next page

Table D.11: mpegaudio Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
32	n/a	<0.00%	35.77%	n/a	3.31%	2.65
64	n/a	<0.00%	35.77%	n/a	1.63%	2.30
128	n/a	<0.00%	35.77%	n/a	1.22%	2.21
STACKUNIREG: Eight-way Associative						
1	n/a	<0.00%	35.75%	n/a	19.96%	7.06
2	n/a	<0.00%	35.77%	n/a	10.50%	4.77
4	n/a	<0.00%	35.77%	n/a	5.33%	3.52
8	n/a	<0.00%	35.77%	n/a	2.44%	2.82
16	n/a	<0.00%	35.77%	n/a	1.46%	2.58
32	n/a	<0.00%	35.77%	n/a	1.02%	2.47
64	n/a	<0.00%	35.77%	n/a	0.89%	2.44
128	n/a	<0.00%	35.77%	n/a	0.67%	2.39
UNIREG: Direct Mapped						
1	n/a	n/a	26.31%	n/a	28.80%	8.48
2	n/a	n/a	26.33%	n/a	20.01%	6.52
4	n/a	n/a	26.33%	n/a	14.19%	5.22
8	n/a	n/a	26.33%	n/a	6.72%	3.54
16	n/a	n/a	26.33%	n/a	5.08%	3.18
32	n/a	n/a	26.33%	n/a	4.00%	2.94
64	n/a	n/a	26.33%	n/a	2.66%	2.63
128	n/a	n/a	26.33%	n/a	2.08%	2.51
UNIREG: Eight-way Associative						
1	n/a	n/a	26.30%	n/a	25.89%	8.93
2	n/a	n/a	26.33%	n/a	12.63%	5.55
4	n/a	n/a	26.33%	n/a	6.00%	3.85
8	n/a	n/a	26.33%	n/a	2.50%	2.96
16	n/a	n/a	26.33%	n/a	1.27%	2.65
32	n/a	n/a	26.33%	n/a	0.83%	2.54
64	n/a	n/a	26.33%	n/a	0.65%	2.49
128	n/a	n/a	26.33%	n/a	0.52%	2.46

Table D.12: mtrt Hybrid Cache Results

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
CONSTHARREG: Direct Mapped						
1	9.17%	n/a	31.43%	1.05%	56.46%	7.79
2	9.17%	n/a	28.98%	0.60%	47.12%	6.43
4	9.17%	n/a	28.98%	0.30%	31.61%	5.00
8	9.17%	n/a	28.98%	0.24%	9.33%	2.99
16	9.17%	n/a	28.98%	0.14%	4.78%	2.57
32	9.17%	n/a	28.98%	0.06%	3.46%	2.44
64	9.17%	n/a	28.98%	<0.00%	2.31%	2.33
128	9.17%	n/a	28.98%	<0.00%	2.07%	2.31
CONSTHARREG: Eight-way Associative						
1	9.17%	n/a	31.07%	0.60%	43.92%	7.35
2	9.17%	n/a	28.98%	0.33%	29.69%	5.51
4	9.17%	n/a	28.98%	0.05%	7.71%	3.22
8	9.17%	n/a	28.98%	<0.00%	4.76%	2.91
16	9.17%	n/a	28.98%	<0.00%	2.73%	2.70
32	9.17%	n/a	28.98%	<0.00%	2.34%	2.66
64	9.17%	n/a	28.98%	<0.00%	1.93%	2.62
128	9.17%	n/a	28.98%	<0.00%	1.80%	2.61
CONSTHARVARD: Direct Mapped						
1	9.17%	n/a	n/a	1.05%	29.18%	11.55
2	9.17%	n/a	n/a	0.60%	19.88%	8.61
4	9.17%	n/a	n/a	0.30%	7.80%	4.84
8	9.17%	n/a	n/a	0.24%	4.07%	3.68
16	9.17%	n/a	n/a	0.14%	1.97%	3.01
32	9.17%	n/a	n/a	0.06%	1.47%	2.85
64	9.17%	n/a	n/a	<0.00%	1.05%	2.71
128	9.17%	n/a	n/a	<0.00%	0.94%	2.67
CONSTHARVARD: Eight-way Associative						
1	9.17%	n/a	n/a	0.61%	15.34%	8.22
2	9.17%	n/a	n/a	0.33%	6.79%	5.16
4	9.17%	n/a	n/a	0.05%	2.75%	3.69
8	9.17%	n/a	n/a	<0.00%	1.68%	3.31
16	9.17%	n/a	n/a	<0.00%	1.17%	3.13
32	9.17%	n/a	n/a	<0.00%	0.97%	3.06

continued on next page

Table D.12: mtrt Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
64	9.17%	n/a	n/a	<0.00%	0.84%	3.01
128	9.17%	n/a	n/a	<0.00%	0.82%	3.01
CONSTSTACKHARREG: Direct Mapped						
1	9.17%	<0.00%	40.63%	1.05%	56.39%	7.14
2	9.17%	<0.00%	39.85%	0.60%	37.41%	5.36
4	9.17%	<0.00%	39.85%	0.30%	20.61%	3.89
8	9.17%	<0.00%	39.85%	0.24%	12.84%	3.23
16	9.17%	<0.00%	39.85%	0.14%	5.96%	2.63
32	9.17%	<0.00%	39.85%	0.06%	4.62%	2.51
64	9.17%	<0.00%	39.85%	<0.00%	3.61%	2.41
128	9.17%	<0.00%	39.85%	<0.00%	3.24%	2.38
CONSTSTACKHARREG: Eight-way Associative						
1	9.17%	<0.00%	40.44%	0.61%	44.15%	6.83
2	9.17%	<0.00%	39.85%	0.33%	21.70%	4.55
4	9.17%	<0.00%	39.85%	0.05%	9.36%	3.31
8	9.17%	<0.00%	39.85%	<0.00%	5.80%	2.96
16	9.17%	<0.00%	39.85%	<0.00%	4.07%	2.79
32	9.17%	<0.00%	39.85%	<0.00%	3.39%	2.73
64	9.17%	<0.00%	39.85%	<0.00%	3.02%	2.69
128	9.17%	<0.00%	39.85%	<0.00%	2.82%	2.67
CONSTSTACKHARVARD: Direct Mapped						
1	9.17%	<0.00%	n/a	1.05%	31.15%	8.97
2	9.17%	<0.00%	n/a	0.60%	20.50%	6.63
4	9.17%	<0.00%	n/a	0.30%	10.73%	4.51
8	9.17%	<0.00%	n/a	0.24%	5.62%	3.42
16	9.17%	<0.00%	n/a	0.14%	2.60%	2.76
32	9.17%	<0.00%	n/a	0.06%	1.97%	2.62
64	9.17%	<0.00%	n/a	<0.00%	1.53%	2.52
128	9.17%	<0.00%	n/a	<0.00%	1.36%	2.48
CONSTSTACKHARVARD: Eight-way Associative						
1	9.17%	<0.00%	n/a	0.61%	20.37%	7.53
2	9.17%	<0.00%	n/a	0.33%	9.10%	4.76
4	9.17%	<0.00%	n/a	0.05%	3.89%	3.45
8	9.17%	<0.00%	n/a	<0.00%	2.42%	3.08

continued on next page

Table D.12: mtrt Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
16	9.17%	<0.00%	n/a	<0.00%	1.64%	2.90
32	9.17%	<0.00%	n/a	<0.00%	1.41%	2.84
64	9.17%	<0.00%	n/a	<0.00%	1.26%	2.80
128	9.17%	<0.00%	n/a	<0.00%	1.18%	2.78
ConstStackUni: Direct Mapped						
1	9.17%	<0.00%	n/a	n/a	23.07%	11.39
2	9.17%	<0.00%	n/a	n/a	15.71%	8.73
4	9.17%	<0.00%	n/a	n/a	7.31%	5.69
8	9.17%	<0.00%	n/a	n/a	4.02%	4.50
16	9.17%	<0.00%	n/a	n/a	2.03%	3.78
32	9.17%	<0.00%	n/a	n/a	1.37%	3.54
64	9.17%	<0.00%	n/a	n/a	0.95%	3.38
128	9.17%	<0.00%	n/a	n/a	0.83%	3.34
ConstStackUni: Eight-way Associative						
1	9.17%	<0.00%	n/a	n/a	15.98%	10.06
2	9.17%	<0.00%	n/a	n/a	7.56%	6.59
4	9.17%	<0.00%	n/a	n/a	3.00%	4.71
8	9.17%	<0.00%	n/a	n/a	1.78%	4.20
16	9.17%	<0.00%	n/a	n/a	1.14%	3.94
32	9.17%	<0.00%	n/a	n/a	0.88%	3.83
64	9.17%	<0.00%	n/a	n/a	0.76%	3.78
128	9.17%	<0.00%	n/a	n/a	0.69%	3.75
ConstUni: Direct Mapped						
1	9.17%	n/a	n/a	n/a	23.47%	14.37
2	9.17%	n/a	n/a	n/a	16.30%	11.09
4	9.17%	n/a	n/a	n/a	6.06%	6.39
8	9.17%	n/a	n/a	n/a	3.32%	5.14
16	9.17%	n/a	n/a	n/a	1.73%	4.41
32	9.17%	n/a	n/a	n/a	1.16%	4.15
64	9.17%	n/a	n/a	n/a	0.75%	3.96
128	9.17%	n/a	n/a	n/a	0.66%	3.92
ConstUni: Eight-way Associative						
1	9.17%	n/a	n/a	n/a	13.74%	11.30
2	9.17%	n/a	n/a	n/a	6.53%	7.53

continued on next page

Table D.12: mtrt Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
4	9.17%	n/a	n/a	n/a	2.44%	5.40
8	9.17%	n/a	n/a	n/a	1.42%	4.87
16	9.17%	n/a	n/a	n/a	0.91%	4.60
32	9.17%	n/a	n/a	n/a	0.69%	4.48
64	9.17%	n/a	n/a	n/a	0.60%	4.44
128	9.17%	n/a	n/a	n/a	0.55%	4.41
CONSTUNIREG: Direct Mapped						
1	9.17%	n/a	31.45%	n/a	33.22%	10.74
2	9.17%	n/a	28.98%	n/a	26.57%	8.83
4	9.17%	n/a	28.98%	n/a	16.42%	6.41
8	9.17%	n/a	28.98%	n/a	5.36%	3.76
16	9.17%	n/a	28.98%	n/a	2.92%	3.18
32	9.17%	n/a	28.98%	n/a	1.93%	2.94
64	9.17%	n/a	28.98%	n/a	1.18%	2.76
128	9.17%	n/a	28.98%	n/a	1.03%	2.73
CONSTUNIREG: Eight-way Associative						
1	9.17%	n/a	30.61%	n/a	27.86%	10.63
2	9.17%	n/a	28.98%	n/a	18.05%	7.75
4	9.17%	n/a	28.98%	n/a	4.69%	4.11
8	9.17%	n/a	28.98%	n/a	2.78%	3.59
16	9.17%	n/a	28.98%	n/a	1.55%	3.25
32	9.17%	n/a	28.98%	n/a	1.19%	3.15
64	9.17%	n/a	28.98%	n/a	0.91%	3.08
128	9.17%	n/a	28.98%	n/a	0.93%	3.08
HARREG: Direct Mapped						
1	n/a	n/a	32.98%	1.05%	55.92%	8.38
2	n/a	n/a	33.13%	0.60%	38.59%	6.33
4	n/a	n/a	33.13%	0.30%	25.52%	4.77
8	n/a	n/a	33.13%	0.24%	9.20%	2.87
16	n/a	n/a	33.13%	0.14%	4.83%	2.35
32	n/a	n/a	33.13%	0.06%	3.40%	2.17
64	n/a	n/a	33.13%	<0.00%	2.23%	2.02
128	n/a	n/a	33.13%	<0.00%	2.03%	2.00
HARREG: Eight-way Associative						

continued on next page

Table D.12: mtrt Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
1	n/a	n/a	32.88%	0.61%	43.92%	7.88
2	n/a	n/a	33.13%	0.33%	19.62%	4.66
4	n/a	n/a	33.13%	0.05%	7.32%	2.99
8	n/a	n/a	33.13%	<0.00%	4.19%	2.57
16	n/a	n/a	33.13%	<0.00%	2.81%	2.38
32	n/a	n/a	33.13%	<0.00%	2.03%	2.28
64	n/a	n/a	33.13%	<0.00%	1.68%	2.23
128	n/a	n/a	33.13%	<0.00%	1.57%	2.22
STACKHARREG: Direct Mapped						
1	n/a	<0.00%	42.77%	1.05%	54.95%	7.87
2	n/a	<0.00%	41.53%	0.60%	37.33%	5.75
4	n/a	<0.00%	41.53%	0.30%	20.56%	3.94
8	n/a	<0.00%	41.53%	0.24%	12.35%	3.06
16	n/a	<0.00%	41.53%	0.14%	5.99%	2.37
32	n/a	<0.00%	41.53%	0.06%	4.44%	2.20
64	n/a	<0.00%	41.53%	<0.00%	3.32%	2.07
128	n/a	<0.00%	41.53%	<0.00%	3.04%	2.04
STACKHARREG: Eight-way Associative						
1	n/a	<0.00%	42.49%	0.61%	42.56%	7.32
2	n/a	<0.00%	41.53%	0.33%	20.96%	4.54
4	n/a	<0.00%	41.53%	0.05%	9.74%	3.14
8	n/a	<0.00%	41.53%	<0.00%	5.92%	2.68
16	n/a	<0.00%	41.53%	<0.00%	3.95%	2.44
32	n/a	<0.00%	41.53%	<0.00%	3.06%	2.33
64	n/a	<0.00%	41.53%	<0.00%	2.63%	2.28
128	n/a	<0.00%	41.53%	<0.00%	2.36%	2.25
STACKHARVARD: Direct Mapped						
1	n/a	<0.00%	n/a	1.05%	30.51%	9.72
2	n/a	<0.00%	n/a	0.60%	20.74%	7.17
4	n/a	<0.00%	n/a	0.30%	11.93%	4.88
8	n/a	<0.00%	n/a	0.24%	5.55%	3.25
16	n/a	<0.00%	n/a	0.14%	2.66%	2.51
32	n/a	<0.00%	n/a	0.06%	1.95%	2.31
64	n/a	<0.00%	n/a	<0.00%	1.45%	2.18

continued on next page

Table D.12: mtrt Hybrid Cache Results (*continued*)

Size (KB)	const %	stack %	reg %	instr %	data %	Time (cycles)
128	n/a	<0.00%	n/a	<0.00%	1.32%	2.14
STACKHARVARD: Eight-way Associative						
1	n/a	<0.00%	n/a	0.61%	21.71%	8.46
2	n/a	<0.00%	n/a	0.33%	9.21%	4.79
4	n/a	<0.00%	n/a	0.05%	4.20%	3.29
8	n/a	<0.00%	n/a	<0.00%	2.55%	2.80
16	n/a	<0.00%	n/a	<0.00%	1.70%	2.55
32	n/a	<0.00%	n/a	<0.00%	1.31%	2.44
64	n/a	<0.00%	n/a	<0.00%	1.13%	2.39
128	n/a	<0.00%	n/a	<0.00%	1.03%	2.36
STACKUNI: Direct Mapped						
1	n/a	<0.00%	n/a	n/a	23.53%	12.33
2	n/a	<0.00%	n/a	n/a	16.39%	9.44
4	n/a	<0.00%	n/a	n/a	8.51%	6.26
8	n/a	<0.00%	n/a	n/a	4.19%	4.52
16	n/a	<0.00%	n/a	n/a	2.15%	3.69
32	n/a	<0.00%	n/a	n/a	1.44%	3.41
64	n/a	<0.00%	n/a	n/a	0.96%	3.21
128	n/a	<0.00%	n/a	n/a	0.86%	3.17
STACKUNI: Eight-way Associative						
1	n/a	<0.00%	n/a	n/a	17.60%	11.32
2	n/a	<0.00%	n/a	n/a	8.01%	6.91
4	n/a	<0.00%	n/a	n/a	3.40%	4.79
8	n/a	<0.00%	n/a	n/a	1.99%	4.14
16	n/a	<0.00%	n/a	n/a	1.24%	3.79
32	n/a	<0.00%	n/a	n/a	0.89%	3.63
64	n/a	<0.00%	n/a	n/a	0.70%	3.54
128	n/a	<0.00%	n/a	n/a	0.64%	3.52
STACKUNIREG: Direct Mapped						
1	n/a	<0.00%	42.77%	n/a	29.03%	9.67
2	n/a	<0.00%	41.53%	n/a	20.15%	7.29
4	n/a	<0.00%	41.53%	n/a	10.17%	4.74
8	n/a	<0.00%	41.53%	n/a	6.26%	3.74
16	n/a	<0.00%	41.53%	n/a	3.26%	2.98

continued on next page

Table D.12: mtrt Hybrid Cache Results (*continued*)

Size (KB)	**const %**	**stack %**	**reg %**	**instr %**	**data %**	**Time (cycles)**
32	n/a	<0.00%	41.53%	n/a	2.20%	2.71
64	n/a	<0.00%	41.53%	n/a	1.48%	2.52
128	n/a	<0.00%	41.53%	n/a	1.33%	2.48
STACKUNIREG: Eight-way Associative						
1	n/a	<0.00%	42.57%	n/a	23.54%	9.39
2	n/a	<0.00%	41.53%	n/a	12.09%	5.96
4	n/a	<0.00%	41.53%	n/a	5.28%	3.98
8	n/a	<0.00%	41.53%	n/a	3.11%	3.35
16	n/a	<0.00%	41.53%	n/a	1.94%	3.01
32	n/a	<0.00%	41.53%	n/a	1.37%	2.84
64	n/a	<0.00%	41.53%	n/a	1.12%	2.77
128	n/a	<0.00%	41.53%	n/a	1.01%	2.74
UNIREG: Direct Mapped						
1	n/a	n/a	32.97%	n/a	34.00%	11.24
2	n/a	n/a	33.13%	n/a	24.29%	8.68
4	n/a	n/a	33.13%	n/a	14.57%	6.10
8	n/a	n/a	33.13%	n/a	5.68%	3.74
16	n/a	n/a	33.13%	n/a	3.13%	3.06
32	n/a	n/a	33.13%	n/a	2.04%	2.77
64	n/a	n/a	33.13%	n/a	1.23%	2.55
128	n/a	n/a	33.13%	n/a	1.10%	2.52
UNIREG: Eight-way Associative						
1	n/a	n/a	32.87%	n/a	28.08%	11.01
2	n/a	n/a	33.13%	n/a	14.54%	6.94
4	n/a	n/a	33.13%	n/a	4.85%	4.01
8	n/a	n/a	33.13%	n/a	2.73%	3.37
16	n/a	n/a	33.13%	n/a	1.70%	3.05
32	n/a	n/a	33.13%	n/a	1.14%	2.88
64	n/a	n/a	33.13%	n/a	1.02%	2.85
128	n/a	n/a	33.13%	n/a	0.89%	2.81

Bibliography

[1] W. P. Alexander, R. F. Berry, F. E. Levine, and R. J. Urquhart. A unifying approach to performance analysis in the Java environment. *IBM Systems Journal*, 39(1):118–134, 2000.

[2] B. Alpern, C. R. Attanasio, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russel, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[3] G. M. Amdahl. In *AFIPS Conference Proceedings*, pages 483–485, April 1967.

[4] S. J. Baylor, M. Devarakonda, S. J. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. J. Munroe. Java server benchmarks. *IBM Systems Journal*, 39(1):57–81, 2000.

[5] R. Christ, S. L. Halter, K. Lynne, S. Meizer, S. J. Munroe, and M. Pasch. Sanfrancisco performance: A case study in performance of large-scale Java applications. *IBM Systems Journal*, 39(1):4–20, 2000.

[6] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–26, 2000.

[7] D. Dillenberger, R. Bordawekar, C. W. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. F. Oliver, F. Samuel, and R. W. St. John. Building a Java virtual machine for server applications: The Jvm on OS/390. *IBM Systems Journal*, 39(1):194–211, 2000.

[8] R. Dimpsey, R. Arora, and K. Kuiper. Java server performance: A case study of building efficient, scalable Jvms. *IBM Systems Journal*, 39(1):151–174, 2000.

[9] M. A. Ertl. Stack caching for interpreters. In *SIGPLAN Notices*, volume 30, pages 315–327. ACM, June 1995.

[10] R. J. Fowler and J. E. Veenstra. A performance evaluation of optimal hybrid cache coherency protocols. In *Proceedings of the 5th international conference on architectural support for programming languages and operating systems*, volume V, pages 149–157, October 1992.

[11] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith. Cache performance of the SPEC92 benchmark suite. *IEEE Micro*, pages 17–27, August 1993.

[12] W. Gu, N. A. Burns, M. T. Collins, and W. Y. P. Wong. The evolution of a high-performing Java virtual machine. *IBM Systems Journal*, 39(1):135–150, 2000.

[13] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, C-38(12):1612–1630, 1989.

[14] W. W. Hsu, Y. Lee, and J. Peir. Capturing dynamic memory reference behavior with adaptive cache topology. In *Proceedings of the 8th international conference on architectural support for programming languages and operating systems*, volume VIII, pages 240–250, 1998.

[15] *IBM Systems Journal*, 39(1), 2000.

[16] B. L. Jacob and T. N. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *Proceedings of the 8th international conference on architectural support for programming languages and operating systems*, volume VIII, pages 295–306, October 1998.

[17] I. H. Kazi, H. H. Chen, B. Stanley, and D. J. Lilja. Techniques for obtaining high performance in Java programs. *ACM Computing Surveys*, 32(3):213–240, 2000.

[18] I. H. Kazi, D. P. Jose, B. Ben-Hamida, C. J. Hescott, C. Kwok, J. A. Konstan, D. J. Lilja, and P.-C. Yew. JaViz: A client/server Java profiling tool. *IBM Systems Journal*, 39(1):96–118, 2000.

[19] H. M. Levy and C. A. Thekkath. Hardware and software support for efficient exception handling. In *Proceedings of the 6th international conference on architectural support for programming languages and operating systems*, volume VI, pages 110–119, October 1994.

[20] T. Lindholm and F. Yellin. *The Java virtual machine specification*, chapter 6, pages 151–338. Addison-Wesley, Reading, Massachusetts, first edition, 1997.

[21] J. S. Liptay. Structural aspects of the System/360 model 85, part II: The cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[22] H. McGhan and M. O'Connor. PicoJava: A direct execution engine for Java bytecode. *Computer*, pages 22–30, October 1998.

[23] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1):21–57, 2000.

[24] K. D. Nilsen and W. J. Schmidt. Performance of a hardware-assisted real-time garbage collector. In *Proceedings of the 6th international conference on architectural support for programming languages and operating systems*, volume VI, pages 76–85, October 1994.

[25] J. M. O'Connor and M. Tremblay. PicoJava-I: The Java virtual machine in hardware. *IEEE Micro*, pages 45–53, March 1997.

[26] D. A. Patterson and J. L. Hennessy. *Computer architecture: A quantitative approach*, chapter 5.1–5.6, pages 373–438. Morgan Kaufmann Publishers, San Francisco, second edition, 1996.

[27] D. Ripps and B. Mushinsky. Benchmarks contrast 68020 cache-memory operations. *EDN*, 3:177–202, 1985.

[28] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time compiler. *IBM Systsms Journal*, 39(1):175–193, 2000.

[29] J. Torrellas, C. Xia, and R. L. Daigle. Optimizing the instruction cache performance of the operating system. *IEEE Transactions on Computers*, 47(12):1363–1381, 1998.

[30] B. Venners. *Inside the JAVA 2 Virtual Machine*. McGraw-Hill, second edition, 1999.

[31] D. Viswanathan and S. Liang. Java virtual machine profiler interface. *IBM Systems Journal*, 39(1):82–95, 2000.