# The Structure and Performance of Interpreters

Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong,
Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

## Abstract

Interpreted languages have become increasingly popular due to demands for rapid program development, ease of use, portability, and safety. Beyond the general impression that they are "slow," however, little has been documented about the performance of interpreters as a class of applications.

This paper examines interpreter performance by measuring and analyzing interpreters from both software and hardware perspectives. As examples, we measure the MIPSI, Java, Perl, and Tcl interpreters running an array of micro and macro benchmarks on a DEC Alpha platform. Our measurements of these interpreters relate performance to the complexity of the interpreter's virtual machine and demonstrate that native runtime libraries can play a key role in providing good performance. From an architectural perspective, we show that interpreter performance is primarily a function of the interpreter itself and is relatively *independent* of the application being interpreted. We also demonstrate that high-level interpreters' demands on processor resources are comparable to those of other complex compiled programs, such as gcc. We conclude that interpreters, as a class of applications, do not currently motivate special hardware support for increased performance.

## 1 Introduction

Interpreted languages have become commonplace for a wide variety of computational tasks. For example, Java and Perl are now standard languages for building internet applications, while Tcl is commonly used for rapid development of interactive user interfaces. Interpreters also play a crucial role as binary emulators, enabling code to port directly from one architecture to another [Afzal et al. 96]. Such environments reflect the extent to which program function, ease of development, portability, and safety represent important concerns for a broad range of contemporary application domains.

Despite the recent emphasis on safety and portability, the performance of interpreters remains a primary concern. In low-end consumer-oriented systems, for example, where low price limits hardware resources, interpreted programs must still execute effectively. Successful design of such systems therefore requires an understanding of the overall structure and performance of interpreters. To date, there has been little analysis of the performance of interpreters as a class of applications.

This paper explores the performance implications of interpreted code executing on modern microprocessors. We measure and evaluate interpreters for the MIPS instruction set and the Tcl, Perl, and Java languages while running a wide array of micro and macro benchmarks on a DEC Alpha platform. Our goal is *not* to compare these interpreters to each other, but rather, to (1) understand the important parameters of interpreter performance, and (2) measure the demands of these interpreters on current processors. Using various instrumentation and tracing techniques, we provide detailed measurements of the interpreter implementations. We also use trace-driven simulation to profile processor resource utilization during interpreter execution. Our data allows comparison of the interpreted benchmarks to similar microbenchmarks written in C and to the SPECint92 benchmark suite.

Previous research has concentrated mainly on the analysis of programming languages and their interaction with architecture. Early measurement studies of commercial minicomputer systems looked at instruction set usage and showed that only a small percentage of the processor's instruction set accounted for more than 80% of the dynamic instructions used in typical programs [Foster et al. 71, Alexander & Wortman 75, Elshoff 76]. Similar studies on CISCs provided the rationale for moving to RISC processors [Clark & Levy 82, Hennessy & Patterson 90]. More recently, researchers have looked at the interaction of the memory system and various object-oriented and functional languages [Calder et al. 94, Diwan et al. 95, Goncalves & Appel 95, Holzle & Ungar 95]. Researchers have also studied the interaction of particular classes of applications with architecture: for example, [Maynard et al. 94] and [Uhlig et al. 95] studied the memory system behavior of commercial and productivity applications.

In a similar vein, we investigate the interaction of interpreters with modern architectures and show how interpreted applications use the facilities provided by interpreters. The execution time of an interpreted program depends upon the number of commands interpreted and the time to decode and execute each command; we show how the number of commands and their execution time vary widely from interpreter to interpreter and are directly related to the complexity of the virtual machine implemented. Of the interpreters we studied, for example, the Tcl interpreter supports the highest

This research was supported by grants from the National Science Foundation (CCR-9401689, CCR-9200832, CDA-9123308, CCR-9632769), the Office of Naval Research (N00014-94-1-0559), the Digital Equipment Corporation and the Intel Corporation. Bershad was partially supported by a National Science Foundation Presidential Faculty Fellowship Award. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the University of Washington, the Digital Equipment Corporation, the Intel Corporation, the National Science Foundation, the Office of Naval Research, or the U.S. Government.

level virtual machine; for an identical program, it thus executes fewer commands, but more native instructions per command, than the other interpreters. With our measurements, we show where each of the interpreters falls in this spectrum, and how their performance relates to such a characterization of their virtual machines. We also show that native runtime libraries can play a key role in providing good interpreter performance. The Java interpreter, for example, leverages native graphics runtime libraries to provide good performance for graphics-oriented applications.

From an architectural perspective, we find that interpreter performance is primarily a function of the interpreter itself and is independent of the application being interpreted. For example, the instruction and data cache performance of the Tcl interpreter is roughly equivalent when running any of our macro benchmarks. We also demonstrate that an interpreter's demands on the processor resources, particularly the instruction cache, are a function of the complexity of the virtual machine; and that the architectural behavior of high-level interpreters is comparable to that of other complex compiled translators, such as gcc. As a result, we conclude that interpreters, as a class of applications, do not currently motivate the need for special hardware support for increased performance.

In the rest of this paper we investigate the question of interpreter performance. Section 2 presents the interpreters that form the basis of this study. Section 3 analyzes and compares the performance of the interpreter implementations, and Section 4 extends the analysis by measuring the impact of interpreters on different components of a modern microprocessor architecture. Section 5 summarizes our measurements and results, and concludes the paper.

## 2 Interpreter Descriptions

We examined four interpreters in our study:

- *MIPSI* [Sirer 93] is an instruction-level emulator that executes MIPS R3000 [Kane & Heinrich 92] binaries compiled for the DEC Ultrix operating system. MIPSI has been used to investigate architectural alternatives for garbage collection [Goncalves & Appel 95] and multithreaded processors [Tullsen et al. 95], and as a teaching tool in architecture and operating system classes. The internal structure of the interpreter follows closely that of the initial stages of a CPU pipeline, with the fetch, decode and execute stages performed explicitly in software.

- *Java* is an object-oriented language and runtime environment designed to support the safe execution of portable programs downloaded via the World Wide Web. Java is similar to C++, but provides additional language features, such as automatic garbage collection, threads, and synchronization. Java also provides an extensive collection of native runtime libraries that define a high-level interface to routines for building graphical user interfaces. Instead of direct interpretation, Java source programs are compiled offline into byte codes defined by the Java Virtual Machine (JVM) [Sun Microsystems 95]. The Java interpreter operates directly on files containing JVM byte codes.

- *Perl* is a scripting language designed for manipulating text, data, files, and processes [Wall & Schwartz 90]. Perl supports a variety of advanced programming abstractions useful for string and file processing, including regular expressions, a high-level I/O interface, automatic memory management, and associative arrays. Perl programs are not interpreted directly, but are compiled at startup time into an internal representation of the source program. Perl performs this compilation step each time a program is invoked.

- *Tcl* is an embeddable command language that allows applications to be customized with a single scripting language [Ousterhout 94]. Tcl is also used as a stand-alone programming environment, providing a programming language interface that includes basic functionality comparable to that found in Perl, as well as mechanisms for easily extending the interpreter with compiled application-specific commands. One popular extension to Tcl is the Tk toolkit, which provides a simple window system interface to enable rapid prototyping of graphical user interfaces. The Tcl interpreter is structured to ease the addition and execution of application-specific commands, and it executes source programs directly.

We chose these interpreters for several reasons. First, they have a diverse set of goals and implementation strategies, enabling us to explore how these different strategies are reflected in their performance. Second, they are available in source form, enabling us to attribute overhead to various aspects of the implementation. Third, the interpreters are all available on a platform with tools for collecting and processing address traces, enabling us to analyze their behavior from an architectural perspective using trace-driven simulation. Finally, the popularity of Java, Perl, and Tcl makes them of interest to a large user community, while MIPSI serves as a representative binary emulator.

## 3 Interpreter Performance

This section analyzes the performance of each of the interpreters running a variety of workloads. We begin by showing the behavior of simple microbenchmarks. Using several real programs, we then characterize some fundamental interpreter overheads, examine the distribution of commands, and measure the cost of command interpretation. Finally, we show how the memory model that each interpreter presents can affect overall performance.

All of our measurements were performed on 175-MHz DEC Alpha 3000/300X workstations running Digital Unix 3.2. Instrumentation data and traces were gathered using ATOM [Srivastava & Eustace 94], a binary-rewriting tool from DEC WRL. Explicit timings and cycle counts were gathered by modifying interpreter source to sample the Alpha cycle counter. Times and cycle counts include all system activity, while instruction counts exclude the behavior of both the operating system and the X window system server.

### 3.1 Microbenchmarks and virtual machines

Table 1 demonstrates the slowdown of various simple operations performed by each interpreter relative to equivalent operations implemented in a compiled C program. The table shows that all the interpreters are significantly slower than C, and that no single interpreter performs best across all the microbenchmarks. Furthermore, the operations that access operating system service routines (e.g., the `read` benchmark) are slowed less than the other operations, because most of the computation is done in precompiled code. The `string-concat` and `string-split` benchmarks show a

| Benchmark | Description | Slowdown relative to C | | | |
|---|---|---|---|---|---|
| | | MIPSI | Java 1.0.1 | Perl 4.036 | Tcl 7.4 |
| a=b+c | assign the sum of two memory locations to a third | 260 | 96 | 770 | 6500 |
| if | conditional assignment | 79 | 21 | 190 | 1500 |
| null-proc | null procedure call | 84 | 84 | 670 | 580 |
| string-concat | concatenate two strings | 186 | 504 | 19 | 78 |
| string-split | split a string into four component strings | 65 | 161 | 13 | 29 |
| read | read a 4K file from a warm buffer cache | 3.3 | 4.6 | 1.2 | 15 |

Table 1: *Microbenchmark results. This table shows the slowdown of each microbenchmark relative to the equivalent operation implemented in C and compiled using the version of the C compiler that comes with Digital UNIX. Each microbenchmark ran for at least five seconds per trial. Each number presented is the average of 20 runs. Standard deviations were no more than 10% and were usually under 5%.*

similar effect for those languages (Perl and Tcl) that provide string manipulation facilities in native runtime libraries.

The explanation for the variations in the microbenchmark results lies in the *virtual machine interface* implemented by each interpreter. The virtual machine interface defines a set of *virtual commands* which provide a portable interface between the program and the processor. The implementation of the virtual machine executes one virtual command on each trip through the main interpreter loop. The interpreter thus incurs an overhead for fetching and decoding each virtual command before performing the work specified by the command. The execution time of an interpreted program therefore depends on the number of commands interpreted, the fetching and decoding cost of each command, and the time spent actually executing the operation specified by the command.

These cost components of interpretation are not independent of one another: the number of commands required to accomplish a given task depends on the level of the virtual machine abstraction defined by the interpreter. A simple virtual machine might require the execution of a large number of commands, so the decoding cost of each command can be critical to program execution time. On the other hand, a complex virtual machine can execute a given program in fewer commands, so the aggregate decoding cost may be moderate.

The interpreters we measure define virtual machines ranging from simple to complex. For example, MIPSI and Java define simple virtual machines; the overhead of each virtual command is small and nearly fixed, but a large number of commands are required to accomplish a given task. In contrast, Perl and Tcl each define complex virtual machines and result in non-uniform slowdowns relative to the C implementations. Sometimes, as with string management, the virtual machines provide an efficient implementation of a service. Other times, as with variable summation and assignment, the virtual machine interface introduces substantial overheads.

## 3.2 Application performance

The microbenchmark numbers help explain but not predict program performance. To gain better insight, we measured a set of typical programs, selected for each language from publicly available sources. These programs are described in Table 2. The `des` program has been implemented in all four languages and provides a common reference point for the interpreters.

Table 2 also shows the baseline performance of the interpreters running the benchmark suite. For each program, the table shows the total number of virtual commands executed by the interpreter while running the program, the number of underlying native instructions executed by the interpreter, the ratio of native instructions to virtual commands (separated into fetch/decode and execute categories), and the program's total execution time in machine cycles. For Perl, we break out the number of instructions devoted to program precompilation in the Native Instructions column. We do not include this precompilation overhead when calculating the average number of native instructions per virtual command, as it represents a fixed overhead per program.

The table shows that the average number of instructions required to fetch and decode a single virtual command is low and roughly fixed for MIPSI and Java, which use a simple uniform representation of virtual commands. In contrast, Perl uses a more complex internal representation of virtual commands, and thus has increased decoding costs. Tcl has fetch and decode costs that are an order of magnitude higher than the other languages, primarily because Tcl interprets the ASCII source program directly. The importance of the fetch/decode component of command interpretation diminishes as the cost of executing the command increases. For Java and Perl, the average number of native instructions required to execute virtual command varies, and can dominate the fetch/decode component.

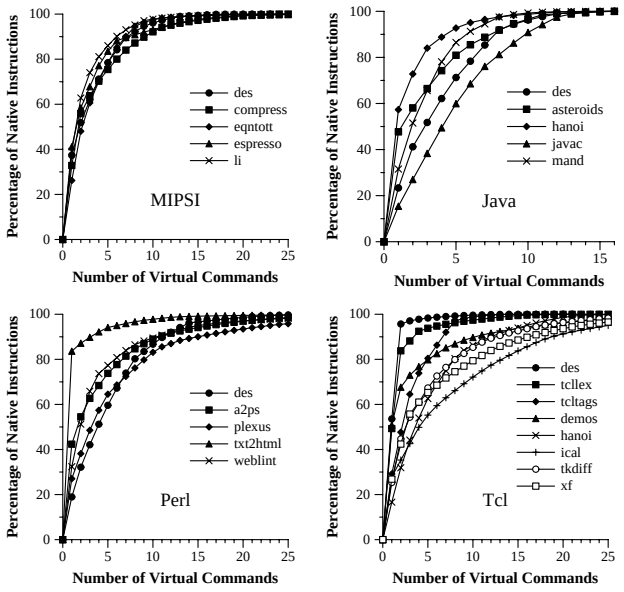


Figure 1: *Cumulative native instruction count distributions. Each point $(x, y)$ shows that the top $x$ virtual commands account for $y$% of the native instructions executed for the benchmark. The native instructions only include the execute component from Table 2 (fetch/decode instructions are excluded). Figure 2 shows the exact contribution of specific commands in more detail.*

| Language | Benchmark | Description | Size (KB) | Virtual Commands ($\times 10^3$) | Native Instructions ($\times 10^3$) | | Average Native Instructions per Virtual Command | | Total Cycles ($\times 10^3$) |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | Fetch/Decode | Execute | |
| C | des | DES encryption and decryption | 90 | 170 | 170 | | 0 | 1.0 | 530 |
| MIPSI | des | DES encryption and decryption | 29 | 190 | 13,000 | | 51 | 17 | 29,000 |
| | compress | Unix compress utility | 33 | 4,200 | 290,000 | | 49 | 20 | 570,000 |
| | eqntott | Equation to truth table conversion | 53 | 15,000 | 1,000,000 | | 47 | 19 | 1,800,000 |
| | espresso | Boolean minimization | 210 | 760 | 52,000 | | 49 | 20 | 100,000 |
| | li | Lisp interpreter | 94 | 3,400 | 240,000 | | 48 | 23 | 450,000 |
| Java | des | DES encryption and decryption | 35 | 320 | 14,000 | | 16 | 27 | 63,000 |
| | asteroids | Asteroids game | 18 | 2,800 | 120,000 | | 16 | 27 | 1,300,000 |
| | hanoi | Towers of Hanoi (5 disks) | 4.9 | 180 | 33,000 | | 16 | 170 | 340,000 |
| | javac | Espresso Java compiler (1.0 beta) | 280 | 5,700 | 240,000 | | 16 | 26 | 900,000 |
| | mand | Interactive Mandelbrot explorer | 7.0 | 5,600 | 190,000 | | 16 | 18 | 370,000 |
| Perl | des | DES encryption and decryption | 23 | 140 | (6,100) | 45,000 | 200 | 82 | 230,000 |
| | a2ps | Convert ASCII file to postscript | 24 | 590 | (4,200) | 290,000 | 130 | 350 | 2,500,000 |
| | plexus | HTTP server | 26 | 49 | (29,000) | 95,000 | 180 | 1,200 | 770,000 |
| | txt2html | Convert text to HTML | 33 | 39 | (5,900) | 100,000 | 150 | 2,300 | 360,000 |
| | weblint | HTML syntax checker | 39 | 120 | (7,600) | 130,000 | 150 | 870 | 650,000 |
| Tcl | des | DES encryption and decryption | 27 | 82 | 600,000 | | 2,100 | 5,200 | 2,000,000 |
| | tcllex | Lexical analysis tool | 7.8 | 31 | 220,000 | | 3,800 | 3,300 | 890,000 |
| | tcltags | Generate emacs tags file | 3.3 | 260 | 1,100,000 | | 2,700 | 1,500 | 4,300,000 |
| | demos | Tk widget demos | 98 | 12 | 95,000 | | 2,600 | 5,300 | 1,700,000 |
| | hanoi | Tk towers of Hanoi (5 disks) | 7.2 | 3.4 | 17,000 | | 2,100 | 3,000 | 300,000 |
| | ical | Tk interactive calendar program | 170 | 15 | 66,000 | | 2,100 | 2,300 | 1,300,000 |
| | tkdiff | Tk interface to diff | 45 | 5.9 | 37,000 | | 2,000 | 4,300 | 340,000 |
| | xf | Tk interface builder | 2,700 | 8.9 | 94,000 | | 5,200 | 5,400 | 3,300,000 |

Table 2: *This table shows baseline performance measurements of the interpreters running a set of representative programs, along with one benchmark program written in C. The Size column gives the size of the input to the interpreter. The Native Instructions column shows the total number of instructions executed by the program, excluding the operating system and the windowing system. For Perl, this column also shows in parentheses the number of instructions precompiling the program. The Average columns show the number of native instructions executed divided by the total number of virtual commands, split into Fetch/Decode and Execute components. For Perl, these ratios exclude precompilation instructions. The Tcl* `xf` *and* `demos` *programs have not been ported to version 7.4 of the interpreter, so we run those on version 7.3; all other Tcl programs run on version 7.4.*

For example, in Java's `hanoi` benchmark, most of the commands have long-running implementations in the native graphics runtime library, and as a result only 8.6% of all instructions are due to fetching and decoding virtual commands.

The execution time for different virtual commands depends on the complexity of those commands. This leads us to ask whether specific virtual commands account for a disproportionate amount of an interpreter's execution costs; such commands would be natural targets for optimization. Figure 1 gives an initial indication of the concentration of instructions among distinct virtual commands. For example, for the `des` benchmark in Tcl, just two virtual commands account for 96% of the execute component of native instructions. Figure 2 presents more detailed data for each of our benchmark applications. For each benchmark, we show two histograms: on the left, the white-barred histogram shows the distribution of *virtual commands*; on the right, the grey-barred histogram shows the percentage of *native instructions* due to command execution for each virtual command. For example, for the `txt2html` benchmark in Perl, the `match` command accounts for 9% of the virtual commands interpreted and 84% of the native instructions due to command execution.

For MIPSI, we see that for each program three virtual commands are responsible for over 60% of the instructions dedicated to virtual command execution. For the most part, these instructions manipulate the interpreted program's memory model (`lw` and `sw`), which we describe in more detail in the following section[1]. Recall from Table 2, however, that the majority of MIPSI's overall execution time is due to fetch/decode overhead rather than command execution. Thus, a good initial target of optimization for MIPSI would be the fetch/decode loop rather than any individual command.

The Java graphs show that some Java programs spend a large fraction of their execution time in native runtime libraries (`native`). For the benchmarks that use graphics heavily (e.g., `hanoi` and `asteroids`), up to 57% of instructions due to the execute component occur within these libraries. For these applications the interpreter itself is therefore not the primary performance bottleneck. Figure 2 also illustrates how Java applications that extensively use native libraries diminish the importance of primitive byte codes. In the `asteroids` benchmark, `st_load` (stack load) commands account for 30% of the virtual commands executed, but less than 7% of the execute instructions because the program spends almost half (48%) of its execute instructions within native library code.

Finally, for Perl and Tcl, a small number of virtual commands dominate the execution time of each application (Figure 1). However, the specific set of dominant virtual commands varies from program to program (Figure 2). The reason is that the high-level virtual machines give programmers a great deal of flexibility in

[1] The counts for `sll` are inflated because the assembler encodes no-ops to fill delay slots as `sll` instructions. For `eqntott`, `espresso`, and `li`, more than 90% of the `sll` instructions are no-ops, while for `des` and `compress`, 33% and 65% of the `sll` instructions are no-ops respectively.

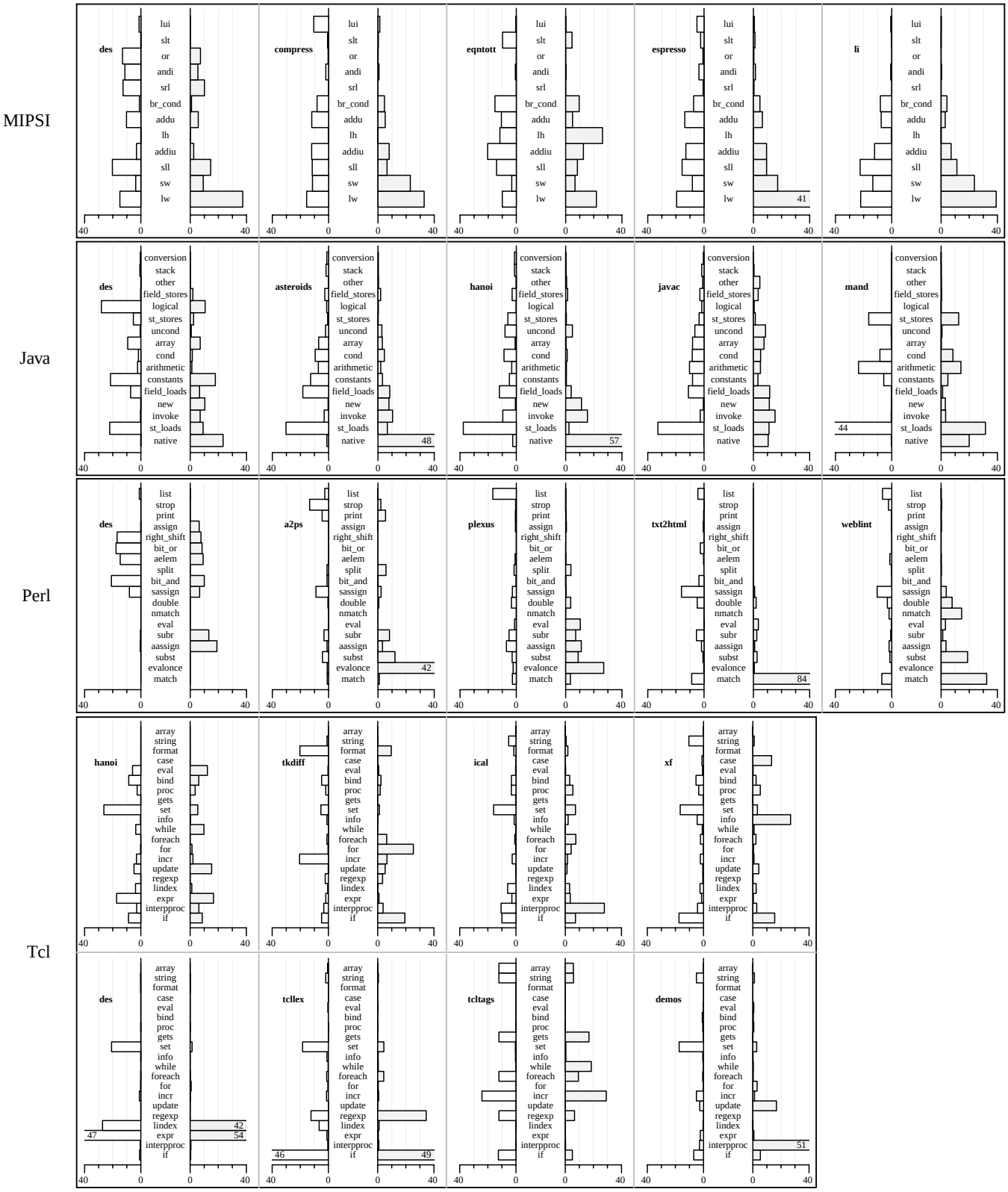


Figure 2: *Native instruction and virtual command count distributions. For each benchmark, we show two histograms. On the left (in white), the histogram shows the distribution of of virtual command counts from Table 2. On the right (in grey), the histogram shows the distribution of native instructions across virtual commands for the "execute" component of command interpretation (fetch/decode instructions are excluded). We omit infrequently executed virtual commands, and group the Java commands into primary categories. Bars higher than 40% are clipped, and the exact percentage is shown at the end of the bar.*

choosing which set of commands to use to implement any given program. Thus the set of commands used by each program varies. The result is that optimizing a particular command or set of commands will not result in broad performance improvements for these two languages.

### 3.3 Memory Model

In the previous subsection we discussed the virtual command usage of interpreted programs. We now discuss how interpreted programs access data. The virtual machine implemented by each interpreter provides a mechanism for accessing memory. This includes a way of naming data in memory and primitives to fetch and store values. Since many virtual commands access "memory," the complexity of the virtual memory model can affect performance. Specifically:

- MIPSI translates each memory access using in-core simulated page tables. The average per-access memory cost for our benchmarks is 62 native instructions; the percentage of total instructions executed for memory model implementation ranges from 13% to 18%. As with other aspects of performance, the MIPSI low-level virtual machine exhibits uniform behavior.

- Java data can be stored either temporarily on thread stacks or more permanently in object fields. Stack data can be accessed implicitly during the execution of a Java bytecode to fetch operands or save results (e.g., `iadd`), or explicitly using specific stack bytecodes. Object fields can only be accessed using specific bytecodes (e.g., `getfield` and `putfield`). Each stack reference costs 2 instructions on average, and each object field reference costs 11 instructions on average. Together, they account for 7% to 13% of total instructions executed across the Java benchmarks.

- Perl data can be stored in scalars, arrays, and associative arrays. Perl uses a symbol table to translate variable names, but the preprocessing phase compiles away most of these translations for scalars and arrays. For associative arrays, Perl always requires a hash table translation, which on average costs 210 native instructions. The percentage of total instructions executed to support the memory model ranges from 0.16% to 3.8%. These results illustrate one of the benefits of a preprocessing (or compilation) phase.

- Tcl data is named by strings and can be stored in scalars, lists, and (associative) arrays. All variable references require a symbol-table lookup that translates the variable name to a storage location. The average per-access memory cost ranges from 206 (for `des`) to 514 (for `xf`) native instructions. The cost varies due to the number of entries in the symbol table. The percentage of total instructions executed to support the memory model ranges from 3.4% to 14%, and is on average 9.3%.

These measurements show that while the memory model can be a significant source of overhead, preprocessing the input program as Perl does can reduce the subsequent runtime overhead of the memory model.

### 3.4 Summary

Simple interpreters such as MIPSI have the advantage of providing core functionality with nearly fixed overhead per virtual command. At the other extreme, complex interpreters such as Tcl and Perl provide highly expressive virtual command sets with relatively high interpretation overhead per virtual command. Finally, Java offers a compromise approach, with a reasonably efficient core set of functionality and a means of accessing native library code. Applications that make extensive use of native libraries can substantially reduce their reliance on interpreted code and its associated overheads.

## 4 The Architectural Impact of Interpreters

In this section we use trace-driven simulation to analyze the effect of interpreter execution on architectural resources, such as the cache and execution units. Using our benchmark suite, we first simulate instructions and memory references to identify the source and nature of all processor stalls. Using this same simulation technique for several compiled programs allows us to compare those programs to our interpreters; the goal is to determine whether significant differences exist between interpreted and directly-executed programs with respect to hardware resource utilization. We then focus on the behavior of the memory system and explore how different cache parameters affect interpreter performance.

### 4.1 Simulation Results

To evaluate overall execution behavior, we use a detailed instruction-level simulator of a modern microprocessor based on the design of the DEC Alpha 21064 [Tullsen et al. 95]. The simulator processes all instructions and memory references of an executing program, and accounts for the sources of all processor stalls during instruction execution. Table 3 lists the sources of these stalls and the penalties they impose[2].

With the simulator, we measured the overall execution behavior of the four interpreters running the programs in our benchmark suite. We also simulated a subset of the SPECint92 benchmarks as a basis for comparing the performance of well-understood compiled programs to interpreted programs.

Figure 3 shows the results of our simulations. For each benchmark, we show the percentage of issue slots that are filled (processor busy) and the distribution of unfilled issue slots due to delays imposed by various architectural components. The larger the size of a component bar, the more the component contributed to delays in instruction execution. For example, the Tcl interpreter running the `ical` benchmark had a processor utilization of 32%, while 16% of its issue slots were unfilled because of instruction cache misses.

From this figure we draw three conclusions:

1. For each interpreter, performance was generally independent of the benchmark being interpreted.

2. The interpreters that define high-level virtual machines (Perl and Tcl) have relatively poor instruction locality, while those

[2] The simulator predicts slightly fewer stalls than a real Alpha 21064 system for three reasons. First, only the user-level instructions of the program are simulated, thus ignoring the effects of executing operating system instructions or context switches to other programs. Second, the execution units in the simulator are uniform and can execute any instruction, whereas the issuing rules of the 21064 are less flexible. Finally, to support the simultaneous issuing of two load instructions, the first-level data cache is modeled as a banked cache as opposed to one that restricts the processor to one access per cycle. We do not believe that these minor differences affect our conclusions.

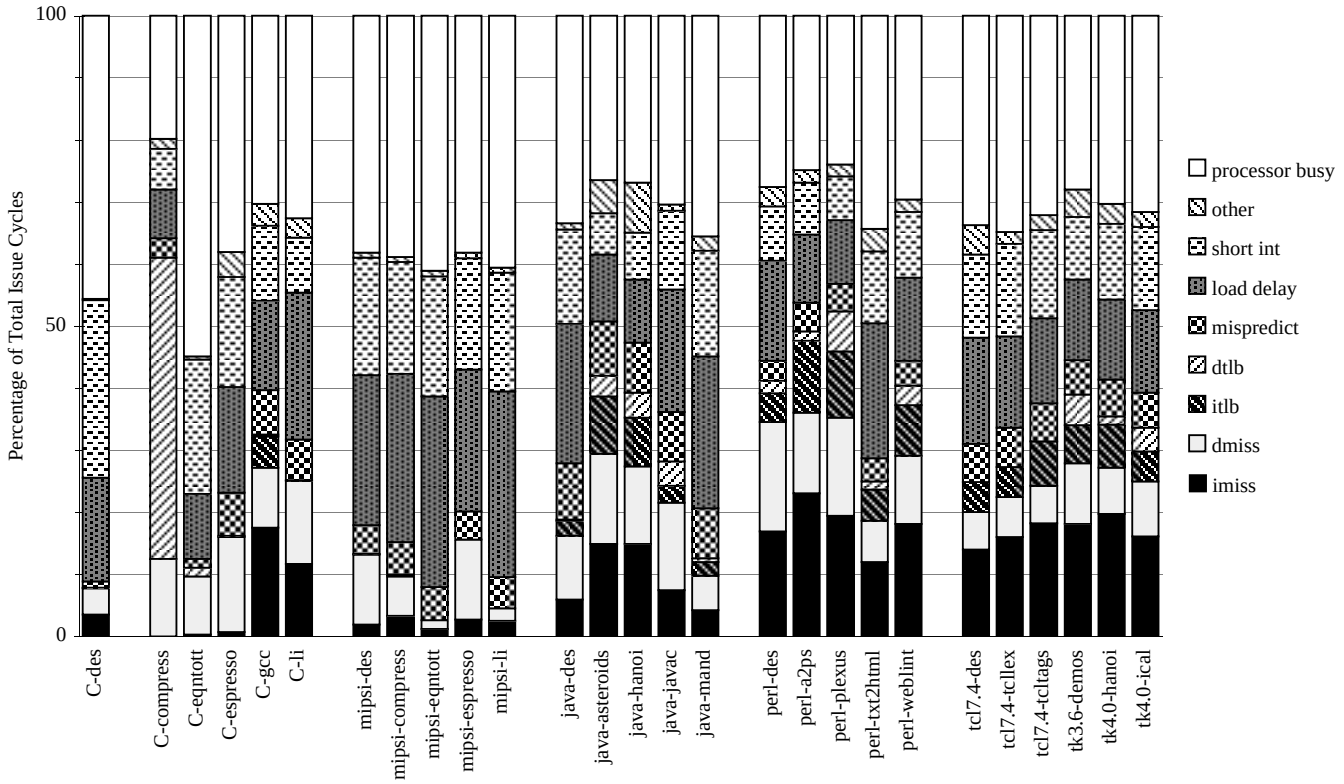


Figure 3: *Overall execution behavior. This figure shows the distribution of stall cycles in a 2-issue superscalar processor. Processor busy show the percentage of filled issue slots. The remaining categories show the percentage of unfilled issue slots.*

that define low-level virtual machines (MIPSI and Java) tend to have good instruction locality.

3. Data cache behavior for the interpreted programs is roughly similar to that of the SPEC programs.

The rest of this section addresses these points in more detail.

**General trends**

Considering the performance of the benchmarks for each interpreter as a group, it is evident that while the interpreters differ from one another, each interpreter exhibits similar performance behavior across benchmarks. In other words, application-specific behavior is overwhelmed by the performance of the interpreter itself. This can be seen, for example, by comparing the performance of the SPEC programs when run directly on the Alpha, and when interpreted by MIPSI. When run directly, there is a great deal of variation in the impact on the architecture (for example, contrast `C-compress` and `C-li`). But when interpreted by MIPSI, the performance profiles for all of the programs are quite similar. The large number of native instructions executed to interpret one virtual command dilute the performance behavior of the original application instruction stream.

The performance profiles for Perl and Tcl are also similar across benchmarks. Comparing their behavior to that of the SPEC benchmarks, we see that these two high-level interpreters behave much like gcc. They have relatively poor instruction and data locality, and similar instruction TLB behavior.

| Cause | Latency (cycles) | Description |
|---|---|---|
| other | variable | control hazards, memory bank conflicts, floating point and integer multiply instructions |
| short int | 2 | integer shift and byte instructions |
| load delay | 3 | pipeline delay with first-level cache hit |
| mispredict | 4 | branch misprediction |
| dtlb | 40 | miss in the data tlb |
| itlb | 40 | miss in the instruction tlb |
| dmiss | 6 or 30 | miss in first-level data cache or the second-level unified cache |
| imiss | 6 or 30 | miss in first-level instruction cache or the second-level unified cache |

Table 3: *Causes of processor stalls. This table describes the sources of stall cycles in our machine simulator. The simulated memory system uses 8 KB pages, first level direct-mapped 8 KB caches for instructions and data, a unified second level direct-mapped 512 KB cache, an 8 entry instruction TLB, and a 32 entry data TLB. The simulated branch logic includes a 256 entry 1-bit branch history table, a 12 entry return stack, and a 32 entry branch target cache.*

Java occupies an intermediate point in the spectrum between the low- and high-level interpreters, and this is reflected in its architectural behavior. When applications spend relatively little time in the native runtime libraries (`des`, `javac`, `mand`), they behave much like programs interpreted by MIPSI. On the other hand, the applications that use native runtime libraries heavily (`asteroids`, `hanoi`) have a performance profile similar to gcc and the high-level interpreters.

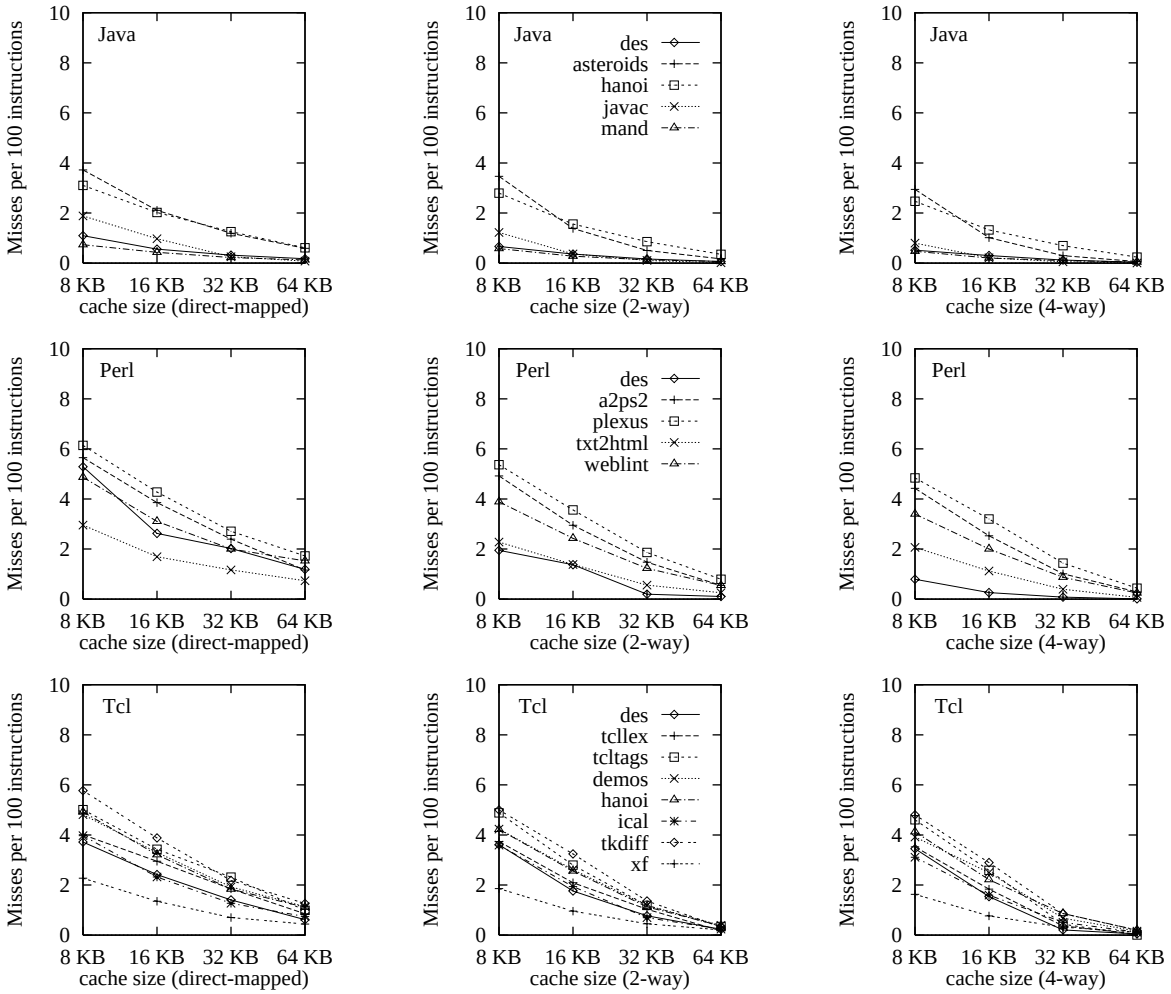


Figure 4: *Instruction cache behavior. This figure shows the instruction miss rates of Java, Perl, and Tcl programs as a function of cache size and associativity. Miss rate is the number of misses per 100 instructions. The miss rate is shown as a function of cache size within a graph and as a function of cache associativity among graphs.*

Examining Figure 3 more closely reveals that major sources of stalls for some interpreters include the instruction and data caches. Load delays are also a significant source of stalls; these stall cycles primarily reflect the effectiveness of the compiler and processor in instruction scheduling. Other important architectural components, such as the branch prediction hardware and TLB, do not have as large an impact on performance. The poor instruction TLB behavior exhibited by the interpreters and the SPEC benchmarks is due to the Alpha 21064's exceptionally small 8-entry iTLB; increasing the size of the simulated iTLB to 32 entries (the same size as the dTLB), effectively eliminates iTLB stalls. The stalls in the "short int" category are also primarily Alpha-specific, resulting from the absence of byte load/store instructions on the 21064.

**Instruction Cache**

In Figure 3, we see that the instruction cache performance corresponds to the type of the virtual machine implemented by an interpreter. For MIPSI, with a low-level virtual machine, only 2% of the issue slots are lost to instruction cache misses, whereas for Perl and Tcl, 18% and 17% of issue slots are lost to instruction cache misses, respectively. The Java interpreter also implements a low-level virtual machine, and hence many of the Java programs have good instruction cache behavior. The `asteroids` and `hanoi` benchmarks are exceptions, but this is explained by Figure 2, which shows that these benchmarks spend the majority of their time in native runtime libraries. Hence, the graph reflects more the performance of the native runtime library rather than the Java interpreter.

The measurements in Section 3.2 provide insight into the relationship between the level of the virtual machine and instruction cache behavior. Recall that as the level of abstraction of virtual commands increases, interpreters execute more native instructions to interpret the virtual command (Table 2). The interpretation loops of MIPSI and Java execute on average fewer than 100 instructions per virtual command, and therefore each iteration of the loop easily fits inside the instruction cache. Moreover, the instructions that are used for fetching and decoding are shared on each iteration, and remain in the cache and contribute to the low miss rate for these two interpreters. Thus MIPSI, with a higher percentage of instructions devoted to fetching and decoding than Java, has more shared instructions between iterations and a lower cache miss rate. Perl

and Tcl, on the other hand, execute thousands of instructions per virtual command, and consequently one iteration of the interpreter loop flushes the cache of the shared instructions from the previous iteration.

Figure 3 shows that Perl, Tcl, and the `asteroids` and `hanoi` benchmarks in Java have a large number of instruction cache stalls for the 8 KB direct-mapped instruction cache in our baseline architecture. To explore how the instruction cache performance improves with increasing first-level cache size, we simulated these interpreters executing on four cache sizes (8, 16, 32, and 64 KB) each with three types of associativity (direct-mapped, 2, and 4 way set associative).

Figure 4 shows the results of this experiment. The graphs indicate that the instruction working set size of the Perl interpreter is in the 32 KB to 64 KB range, and the working set size of the Tcl interpreter is in the 16 KB to 32 KB range. Interestingly, the Tcl applications and one Perl application (`des`) also benefit significantly from higher associativity. Once the cache is large enough to hold much of the working set of the interpreters (16 KB and larger), the i-cache continues to suffer from conflict misses. This effect can be seen by comparing the graphs for 2-way associativity to 4-way associativity. For example, with the 2-way set-associative 32 KB cache, `tcltags` has a miss rate of 1.2 misses per 100 instructions, but with the 4-way set-associative 32 KB cache the miss rate drops to 0.4 misses per 100 instructions. The explanation for this behavior is that the average number of instructions per virtual command executed by the interpreters for these applications is in the 4,000-11,000 range, which corresponds to 16-44 KB of instruction data for each loop iteration. With low associativity, once the cache is large enough to hold this many instructions, some of the instructions shared across iterations are being flushed out of the cache due to conflicts. With increased associativity, however, the conflicts are removed and the interpreters achieve good instruction cache performance.

#### Data Cache and TLB

Figure 3 shows that the percentage of wasted issue slots due to data cache stalls for the interpreters typically is as high as 18%, but is not significantly larger than for the compiled SPEC benchmarks. This suggests that storing the program as data, as occurs in an interpreted environment, does not carry a penalty in terms of data cache performance. Unlike compiled programs, where code is accessed on every cycle, the code for interpreted programs is accessed once every iteration of the interpreter loop. Compared to the total number of instruction and data references during a loop iteration, the accesses to program code are a small percentage of all data accesses.

The data TLB measurements emphasize that the behavior of the interpreter can overwhelm the behavior of the program being interpreted. The `compress` program provides a good example: the native version has a data working set too large for the 32 entry data TLB, resulting in 49% of issue slots going unfilled, while when interpreted by MIPSI, dTLB misses are inconsequential, accounting for less than 1% of the unfilled issue slots. (Of course, the program runs much more slowly when interpreted.)

### 4.2 Architecture Summary

In this section we explored the interaction of interpreted programs with architecture. The importance of the instruction cache depends on the complexity of the virtual machine defined by the interpreter. MIPSI and Java define low-level virtual machines, with an interpreter loop that generally fits well within even a 8 KB instruction cache. For Perl and Tcl, a 64 KB first-level instruction cache is sufficient to effectively capture the working set. Finally, from an architectural perspective, interpreter performance is largely independent of the program being interpreted.

## 5 Conclusions

With increased processor performance and demand for portability, security, and ease of use, interpreted languages have become a major part of today's computing environment. In this paper, we studied the behavior and performance of four interpreters executing a range of programs and microbenchmarks. We showed that the performance of an interpreter cannot be attributed solely to the frequently executed command dispatch loop. Performance is also linked to (1) the expressiveness of the virtual command set and how effectively these virtual commands are used, (2) the use of native runtime libraries, and (3) the way that the virtual machine names and accesses memory. We also showed that the "architectural footprint" of an interpreted program is primarily a function of the interpreter itself and not of the programs being interpreted, and that the high-level interpreters behave similarly to large SPECint92 applications, such as gcc.

It is always tempting to propose specialized hardware to support specific language environments, interpreted or not, as has been done in the past with mixed success [Smith et al. 71, Ditzel & Patterson 80, Flynn 80, Meyers 82, Moon 87, Ungar & Patterson 87]. For the interpreters we studied, however, it is clear that significant potential still exists for improvement through software means. For example, future implementations of Java and Tcl may involve more sophisticated compiling and runtime code generation [Symantec Corporation 96, Ousterhout 96]. Instruction fetch/decode overhead could be reduced by using threaded interpretation, by dynamically compiling portions of the interpreted program into native code, or by compiling to host machine level during load-time or through binary translation, thereby eliminating the fetch/decode overhead altogether [Bell 73, Klint 81, Deutsch & Schiffman 84, Andrews & Sand 92, Sites et al. 92, Cmelik & Keppel 94, Adl-Tabatabai et al. 96, Afzal et al. 96, Wilkinson 96]. These optimizations will have varying degrees of success, depending on the interpreter and interpreted program. We therefore believe that efforts to build specialized hardware for interpreters may be premature; the greatest advance will come as the designers of interpreters realize that performance, as well as portability, flexibility and safety, are crucial goals.

More measurements, including those of Java on the Intel x86 architecture running the Windows NT operating system, are available at http://www.cs.washington.edu/research/interpreters.

### Acknowledgments

This work benefited from discussions with many of our colleagues, including Jeff Dean, Dave Grove, and E. Lewis. David Becker and Warren Jessop helped us overcome some last-minute technical hurdles. Alan Eustace, as ever, was an invaluable source of assistance in using Atom. Larry Sendlosky at DEC and Scott Rautmann and John Hale at Sun were instrumental in providing a version of Java for the Alpha. Gün Sirer and Dean Tullsen taught us how to use and modify MIPSI and the Alpha simulator.

## References


[Adl-Tabatabai et al. 96] Adl-Tabatabai, A., Langdale, G., Lucco, S., and Wahbe, R. Efficient and Language-Independent Mobile Programs. In *Proceedings of the 1996 ACM Symposium on Programming Languages Design and Implementation*, pages 127–136, May 1996.

[Afzal et al. 96] Afzal, T., Brenternitz, M., Kacher, M., Menyher, S., Ommerman, M., and Su, W. Motorola PowerPC Migration Tools – Emulation and Transition. In *Digest of Papers, COMPCON '96*, pages 145–150, February 1996.

[Alexander & Wortman 75] Alexander, W. G. and Wortman, D. B. Static and Dynamic Characteristics of XPL Programs. *IEEE Computer*, 8(11):41–46, November 1975.

[Andrews & Sand 92] Andrews, K. and Sand, D. Migrating a CISC Computer Family onto RISC via Object Code Translation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 213–222, October 1992.

[Bell 73] Bell, J. Threaded Code. *Communications of the ACM*, 16(6):370–372, June 1973.

[Calder et al. 94] Calder, B., Grunwald, D., and Zorn, B. Quantifying Behavioral Differences Between C and C++ Programs. Technical Report CU-CS-698, University of Colorado-Boulder, January 1994.

[Clark & Levy 82] Clark, D. W. and Levy, H. M. Measurement and Analysis of Instruction Use on the VAX-11/780. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, April 1982.

[Cmelik & Keppel 94] Cmelik, R. F. and Keppel, D. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[Deutsch & Schiffman 84] Deutsch, L. P. and Schiffman, A. M. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, Salt Lake City, Utah, January 1984.

[Ditzel & Patterson 80] Ditzel, D. and Patterson, D. Retrospective on High-level Language Computer Architecture. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, pages 97–104, La Baule, France, June 1980.

[Diwan et al. 95] Diwan, A., Tarditi, D., and Moss, E. Memory System Performance of Programs with Intensive Heap Allocation. *ACM Transactions on Computer Systems*, 13(3):244–273, August 1995.

[Elshoff 76] Elshoff, J. An Analysis of Some Commercial PL/1 Programs. *IEEE Transactions on Software Engineering*, 2:113–120, June 1976.

[Flynn 80] Flynn, M. J. Directions and Issues in Architecture and Language. *IEEE Computer*, 13(10):5–22, October 1980.

[Foster et al. 71] Foster, C. C., Gonter, R. H., and Riseman, E. M. Measures of Opcode Utilizations. *IEEE Transactions on Computers*, 13:582–584, May 1971.

[Goncalves & Appel 95] Goncalves, M. and Appel, A. Cache Performance of Fast-Allocating Programs. In *Proceedings of the Seventh International Conference of Functional Programming and Computer Architecture*, pages 293–305, June 1995.

[Hennessy & Patterson 90] Hennessy, J. L. and Patterson, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Palo Alto, CA, 1990.

[Holzle & Ungar 95] Holzle, U. and Ungar, D. Do Object-Oriented Languages Need Special Hardware Support? In *ECOOP '95 - Object-Orented Programming*, pages 283–202. Springer-Verlag, August 1995.

[Kane & Heinrich 92] Kane, G. and Heinrich, J. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[Klint 81] Klint, P. Interpretation Techniques. *Software – Practice and Experience*, 11(9):963–973, September 1981.

[Maynard et al. 94] Maynard, A. G., Donnelly, C. M., and Olszewski, B. R. Contrasting Characteristics and Cache Performance of Technical and Multi-user Commercial Workloads. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, San Jose, California, October 1994.

[Meyers 82] Meyers, G. J. *Advances in Computer Architecture*. Wiley, NY, 1982.

[Moon 87] Moon, D. A. Symbolics Architecture. *IEEE Computer*, pages 43–54, January 1987.

[Ousterhout 94] Ousterhout, J. *Tcl and the Tk Toolkit*. Addision-Wesley, Reading, MA, 1994.

[Ousterhout 96] Ousterhout, J. What's Happening at Sun Labs. http://www.sunlabs.com/research/tcl/team.html, April 1996.

[Sirer 93] Sirer, E. G. Measuring Limits of Fine-Grain Parallelism. Princeton University Senior Project, June 1993.

[Sites et al. 92] Sites, R. L., Chernoff, A., Kirck, M. B., Marks, M. P., and Robinson, S. G. Binary Translation. *Digital Technical Journal*, 4(4):137–152, 1992.

[Smith et al. 71] Smith, W. R., Rice, R. R., Chesley, G. D., Laliotis, T. A., Lundstrom, S. F., Chalhoun, M. A., Gerould, L. D., and Cook, T. C. SYMBOL: A Large Experimental System Exploring Major Hardware Replacement of Software. In *Proceedings AFIPS Spring Joint Computer Conference*, pages 601–616, 1971.

[Srivastava & Eustace 94] Srivastava, A. and Eustace, A. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the 1994 ACM Symposium on Programming Languages Design and Implementation*, pages 196–205. ACM, 1994.

[Sun Microsystems 95] Sun Microsystems. The Java Virtual Machine Specification. http://java.sun.com/doc/vmspec/html/vmspec-1.html, 1995.

[Symantec Corporation 96] Symantec Corporation. Symantec Cafe. http://cafe.symantec.com, June 1996.

[Tullsen et al. 95] Tullsen, D., Eggers, S., and Levy, H. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual Symposium on Computer Architecture*, pages 392–403, June 1995.

[Uhlig et al. 95] Uhlig, R., Nagle, D., Mudge, T., Sechrest, S., and Emer, J. Instruction Fetching: Coping with Code Bloat. In *Proceedings of the 22nd Annual Symposium on Computer Architecture*, pages 345–356, June 1995.

[Ungar & Patterson 87] Ungar, D. and Patterson, D. At What Price Smalltalk. *IEEE Computer*, 20(1):67–74, January 1987.

[Wall & Schwartz 90] Wall, L. and Schwartz, R. *Programming Perl*. O'Reilly and Associates, Inc., Sebastopol, CA, 1990.

[Wilkinson 96] Wilkinson, T. KAFFE – A Virual Machine to Run Java(tm) Code. http://web.soi.city.ac.uk/homes/tim/kaffe/kaffe.html, 1996.