Virtual Machine Showdown: Stack Versus Registers

YUNHE SHI[1] and KEVIN CASEY
Trinity College Dublin
M. ANTON ERTL
Technische Universität Wien
and
DAVID GREGG
Trinity College Dublin

Virtual machines (VMs) enable the distribution of programs in an architecture-neutral format, which can easily be interpreted or compiled. A long-running question in the design of VMs is whether a stack architecture or register architecture can be implemented more efficiently with an interpreter. We extend existing work on comparing virtual stack and virtual register architectures in three ways. First, our translation from stack to register code and optimization are much more sophisticated. The result is that we eliminate an average of more than 46% of executed VM instructions, with the bytecode size of the register machine being only 26% larger than that of the corresponding stack one. Second, we present a fully functional virtual-register implementation of the Java virtual machine (JVM), which supports Intel, AMD64, PowerPC and Alpha processors. This register VM supports inline-threaded, direct-threaded, token-threaded, and switch dispatch. Third, we present experimental results on a range of additional optimizations such as register allocation and elimination of redundant heap loads. On the AMD64 architecture the register machine using switch dispatch achieves an average speedup of 1.48 over the corresponding stack machine. Even using the more efficient inline-threaded dispatch, the register VM achieves a speedup of 1.15 over the equivalent stack-based VM.

Categories and Subject Descriptors: D.3.4 [**Programming Language**]: Processor—*Interpreter*

[1]**Extension of Conference Paper**: An earlier version of the work in this paper appeared in the First ACM/Usenix Conference on Virtual Execution Environments (VEE'05) [Shi et al. 2005]. The new material in this paper consists of (1) a complete reimplementation of the register VM using the Cacao 0.95 JVM (2) SSA form intermediate representation (3) redundant load elimination using SSA (4) virtual register allocation to minimize the size of Java stack frame, (5) support of two additional VM instruction dispatch methods: direct threaded and inline threaded, (6) support additional architectures such as AMD64, Alpha, and PowerPC, (7) additional optimizations investigated, such as preliminary work on assessing the impact of redundant field and array load elimination

Corresponding author's address: David Gregg, Department of Computer Science, Trinity College Dublin, Dublin 2, Ireland. David.Gregg@cd.tcd.ie.

General Terms: Performance, Language

Additional Key Words and Phrases: Interpreter, virtual machine, register architecture, stack architecture

ACM Reference Format:

Shi, Y. Casey, K., Ertl, M. A., and Gregg, D. 2008. Virtual machine showdown: stack versus registers. ACM Trans. Architec. Code Optim. 4, 4, Article 21 (January 2008), 36 pages. DOI = 10.1145/1328195.1328197 http://doi.acm.org/10.1145/1328195.1328197

1. MOTIVATION

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

Interpreters are frequently used to implement virtual machines because they have several practical advantages over native code compilers. Interpreters are much slower than the native code produced by just-in-time compilers (even the fastest interpreters are currently about 5–10 times slower), but they are nonetheless widely used for lightweight language implementations. If written in a high-level language, interpreters are portable; they can simply be recompiled for a new architecture, whereas, generally, a just-in-time (JIT) compiler requires considerable porting effort. Interpreters also require little memory: the interpreter itself is typically much smaller than a JIT compiler [Radhakrishnan et al. 2000], and the interpreted bytecode is usually a fraction of the size of the corresponding executable native code. For this reason, interpreters are commonly found in embedded systems. Furthermore, interpreters avoid the compilation overhead in JIT compilers. For rarely executed code, interpreting is typically much faster than JIT compilation. The Hotspot JVMs [Sun-Microsystems 2001] take advantage of this by using a hybrid interpreter/JIT system. Code is initially interpreted, saving the time and space of JIT compilation. Sections of code are then JIT compiled only if they are found to be executed frequently. Interpreters are also dramatically simpler than compilers; they are easy to construct, and easy to debug. Finally, it is easy to provide tools such as debuggers and profilers when using an interpreter because it is easy to insert additional code into an interpreter loop. Providing such tools for native code is much more complex. Interpreters provide a range of attractive features for language implementation. In particular, most scripting languages are implemented using interpreters.

1.1 Previous Work

A long-running question in the design of VMs is whether a stack architecture or a register architecture can be implemented more efficiently with an interpreter. Stack architectures allow smaller VM code so less code must be fetched per VM instruction executed. However, stack machines require more VM instructions for a given computation, each of which requires an expensive (usually

unpredictable) indirect branch per VM instruction dispatch. Several authors have discussed the issue [Myers 1977; Schulthess and Mumprecht 1977; McGlashan and Bower 1999; Winterbottom and Pike 1997] and presented small examples where each architecture performs better, but no general conclusions can be drawn without a larger study.

The first large-scale quantitative results on this question were presented by Davis et al. [Davis et al. 2003; Gregg et al. 2005] who translated JVM stack code to a corresponding register machine code. A straightforward translation strategy was used with simple compiler optimizations to eliminate instructions that become unnecessary in register format. Of the resulting register code, around 35% fewer VM instructions were needed to perform the same computation than the stack code. However, the resulting register VM code was around 45% larger than the original stack code and resulted in a similar increase in bytecodes fetched. Given the high cost of unpredictable indirect branches, these results strongly suggest that register VMs can be implemented more efficiently than stack VMs with an interpreter. However, this work did not include an implementation of the virtual register architecture, so no real running times were presented.

1.2 Contribution

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

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

1.3 Other Factors

There are other factors to consider in the choice of code format. Compiling source code to stack-based bytecode is usually simpler than compiling to register code,

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

Code Size. One of the attractions of a stack VM is that the code is quite compact, due to the absence of explicit register operands. Later in this paper, we present work that shows that the bytecode for a register VM is only 26% larger than stack bytecode. In the case of Java, however, the bytecode only accounts for about 18% of a class file [Antonioli and Pilz 1998], the rest being occupied primarily by the constant-pool. This constant-pool is a table, used to store interface, class and field names and various constants used by the the class. Various techniques such as those employed by JAX [Tip et al. 2002] can be employed to reduce the constant-pool size. As a result bytecode can occupy as much as 75% of the memory footprint in some embedded systems [Clausen et al. 2000].

There are other options for the code format. For example, compressed syntax-tree based representations [Kistler and Franz 1999] are around twice as compact as stack-based bytecode, and are often considered a better source language for JIT compilation because they retain most of the high-level information from the source code. However, such tree based encodings are difficult to interpret efficiently, so they are most suitable when the VM will be implemented using only a JIT compiler.

Compressed Code Size. Code size is not only important because of the memory consumed, but also because programs may need to be sent over networks. In the case of Java, the contents tend to be easily compressed, repeating text, highly suitable for the jar file format commonly used for classfile transport. Typically classfiles are compressed to about 50% of their original size, and schemes have been proposed that compress classfiles even further (up to 10% to 25% of their original size [Pugh 1999]).

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

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

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

The choice of IR for the work presented in this paper was driven primarily by a different concern to those discussed above. To allow a meaningful comparison between a stack and register VMs, it was decided to keep the instruction sets as similar as possible. Where experimental issues are not the driving force, the choice of IR is likely to be made on the basis of a combination of these issues.

1.4 Paper Overview

The rest of this paper is organized as follows. In section 2, we describe the main differences between virtual stack machine and virtual register machines from the perspective of an interpreter. In section 3, we show how stack Java bytecode is translated into register bytecode and the optimizations applied on the new code. In section 4, we analyze the static and dynamic code behaviour before and after optimization, and we show the performance improvement in our register JVM when compared to the original stack JVM. In section 5, we examine other possible optimizations. In section 6, we discuss how our results can be extended to other VMs. Research related to our work is examined in section 7. Finally, in section 8, we conclude with a summary of the work presented in this paper.

2. STACK VERSUS REGISTER

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

2.1 Dispatching the Instruction

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

Instruction dispatch is typically implemented in C with a large `switch` statement, with one case for each opcode in the VM instruction set. Switch dispatch is simple to implement, but is rather inefficient. Most compilers produce a range check and an additional unconditional branch in the generated code for the `switch`. In processors using a branch target buffer (BTB) for indirect branch prediction, there is only one entry in the BTB for all indirect branch targets. Thus, the indirect branch generated by most compilers is highly unpredictable (around 95% [Ertl and Gregg 2003]) on architectures using a BTB for indirect branch prediction. The main advantages of switch dispatch are that the bytecode executed by the VM is compact, and it can be implemented using any ANSI C compiler.

An alternative to the `switch` statement is *token-threaded dispatch* [Klint 1981]. Threaded dispatch takes advantage of languages with labels as first

class values (such as GNU C and assembly language) to optimize the dispatch process, at the expense of the portability of the interpreter source code. Token-threaded dispatch uses the opcodes to lookup the target address of their implementation in a dispatch target address table. This enables the range check and additional unconditional branches to be eliminated, and permits the code to be restructured to improve the predictability of the indirect branch dispatch (to around 45% [Ertl and Gregg 2003]). On architectures with BTBs for indirect branch prediction, each instruction implementation has its own indirect branch instruction and thus, multiple entries of indirect branch targets can exist in the BTB.

Another alternative is *direct-threaded dispatch* [Bell 1973]. Direct-threaded code directly encodes the jump addresses as the opcodes of instructions and thus further reduces the cost of dispatch. The code to be interpreted is translated from bytecode into threaded code. In threaded code, VM opcodes are no longer bytes, but are instead addresses of the executable native code within the interpreter that performs the computation that corresponds to the original VM opcode. Thus the table lookup from token-threaded code can be eliminated, further reducing the cost of dispatch. Direct-threaded dispatch requires first class labels, a translation step, and the VM code size increases by up to a factor of four on a 32 bit machine or eight on a 64 bit machine.

An even more sophisticated approach is *inline-threaded dispatch* [Piumarta and Riccardi 1998] which copies executable machine code from the interpreter and relocates it to remove the dispatch code entirely. This requires an even more complicated translation from bytecode, much greater memory requirements, and is even less portable than the other forms of threaded dispatch. It is, however, the fastest VM instruction dispatch mechanism, and we present results for it in this paper.

Another alternative is context threading [Berndl et al. 2005]. The context threading approach uses subroutine threading to change indirect branches to call/returns, which better exploits the hardware return-address stack to reduce the cost of dispatches. However, this approach requires some mechanism to generate native executable machine code at run time. We have not implemented this dispatch mechanism, although we believe that it is slightly less efficient than inline threading, which eliminates indirect branches entirely.

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

2.2 Accessing the Operands

The location of the operands must appear explicitly in register code, whereas in stack code, operands are found relative to the stack pointer. Thus, the average register instruction is longer than the corresponding stack instruction, register

code is larger than stack code, and register code requires more memory fetches to execute. Small code size and small numbers of memory bytecode fetches are the main reasons why stack architectures are so popular for VMs.

From the viewpoint of a VM interpreter, a stack VM must keep track of the bytecode instruction pointer (IP), the stack pointer (SP), and the frame pointer (FP) while a register VM only needs the IP and FP. Thus, when the register VM is implemented using an interpreter on a real processor, one variable fewer is required in the inner loop of the interpreter than for the stack VM. This reduces real machine register pressure, and may result in less spilling and reloading of variables. On platforms with small numbers of architected registers[2], such as Intel x86 processors which have only eight general purpose registers, this reduction in register pressure may impact performance. Moreover, a stack VM must update the SP as values are pushed or popped.

2.3 Performing the Computation

Given that most VM instructions perform a simple computation, such as adding or loading, this is usually the smallest part of the cost. The basic computation has to be performed regardless of the instruction format. However, eliminating loop invariants and redundant loads (common subexpressions) is only possible on a register VM[3]. In Section 3.3, we exploit this property to eliminate repeated loads of identical values in a register VM.

3. TRANSLATION AND OPTIMIZATION

In this section, we describe a system of translating JVM stack code to virtual register code and its optimization in a just-in-time manner. This JIT translation was chosen as a useful mechanism to allow us to compare stack and register versions of the JVM easily. Whether or not JIT translation (or any particular run-time translation) from stack format to register format is the optimal way to use virtual register machines remains an open question. In a realistic system, it is most likely that one would use only the register machine, and compile for that directly. Finally, it should be noted that standard, well-known JIT compiler techniques are used for this run-time translation, given that the focus of this research is on the results of the translation, and not on the translation itself.

---

[2]Architected registers are those registers available to the programmer, and described in the instruction set architecture (ISA). For example, the Pentium III supports the x86 ISA, which has 8 general purpose, 8 floating point and several architected control registers. However, the Pentium III implementation of the x86 ISA uses 64 integer and 64 floating point physical registers which are used internally within the processor. The programmer has no direct access to these physical registers.

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

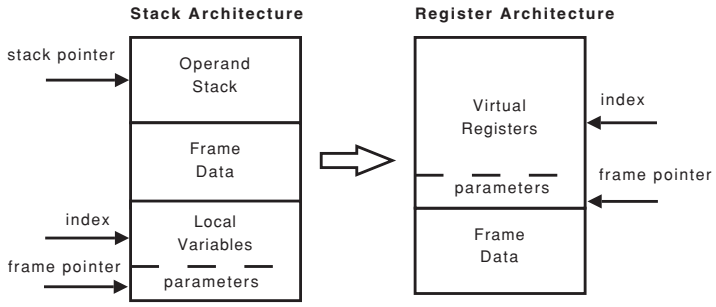

Fig. 1. The structure of a Java frame.

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

Fig. 2. Stack bytecode to register bytecode translation. Assumption: current stack pointer before the code shown above is 10. The registers after -> are destination registers.

3.1 Translation from Stack to Register

Our implementation of the JVM pushes a new Java frame onto a run-time stack for each method call. The Java frame for a stack architecture contains local variables, frame data, and the operand stack for the method (see Figure 1). In the stack JVM, a local variable is accessed using an index, and the operand stack is accessed via the stack pointer. In the register JVM, both the local variables and operand stack can be considered as virtual registers for the method. There is a simple mapping from stack locations to register numbers, because the height and contents of the JVM operand stack are known at any point in a program [Gosling 1995]. In practice, the number of virtual registers (local variables and stack slots) in a method will only be limited by the size of the operand used to specify the register number. Each method call has its own set of virtual registers on its Java frame.

In the stack JVM, most operands of an instruction are implicit; they are found on the top of the operand stack. Most of the stack JVM instructions are translated into corresponding register JVM instructions, with implicit operands translated to explicit operand registers. For example, the stack JVM instruction `if_icmpeq branchbyte1 branchbyte2` is encoded with three bytes, one byte for the opcode and two bytes for the branch offset. The instruction takes two operands from the operand stack and performs a comparison to decide the next execution path. After conversion to register code, the operands are no longer implicit. Thus, the instruction becomes `if_icmpeq r1 r2 branchbyte1 branchbyte2`, occupying a total of five bytes, one byte for the opcode, one byte for each register and two bytes for the branch offset. Figure 2 shows another simple example of bytecode translation. The bytecode adds two integers from two local variables and stores the result back into another local variable.

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

(1) `pop` and `pop2` can be eliminated immediately because they are not needed in the virtual register machine code. Many invoke instructions (method calls) push a return value that is not used by the following instruction onto the operand stack and the stack JVM also uses a number of `pop`/`pop2` instructions purely to maintain consistency of the operand stack.
(2) Instructions that load a local variable onto the operand stack or store a value from the operand stack in a local variable are translated into `move` instructions.
(3) Stack manipulation instructions (e.g. `dup`, `dup2`, ...) are translated into appropriate sequences of `move` instructions by tracking the state of the operand stack.
(4) Wide stack VM instructions which address local variables numbered above 255 can be handled through the addition of one or more `move_wide` instructions. These are register VM instructions with four bytes of operands (two bytes for the source and two for the destination). In the case of Java, this simple extension would enable up to 65535 local variables to be addressed, as permitted by the Java Standard.
(5) The `iinc` instruction in the stack JVM is used to increment a local variable by a constant value. `iinc` is an interesting VM instruction in stack JVMs because the computation is done without the operand stack. The computation should push an operand and a constant, add, and store the result to a local variable. It can be regarded as a type of register VM instruction that is available in the stack JVM. We translate an `iadd` or `isub` into an `iinc` VM instruction if one of its operands is a small integer constant (i.e. it is preceded by a VM instruction that pushes a small integer constant onto the stack).

3.2 Method Invocation

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

In a stack JVM, operands (parameters) always come from the top of the stack, and become the first local variables of the called method (see Figure 1). A common way to implement a stack JVM is to overlap the current Java frame's operand stack (which contains a method call's parameters) and a new Java frame's local variables.

In the register JVM, we do not overlap the Java frames to pass method parameters. Instead, we copy all the parameters from the virtual registers in the

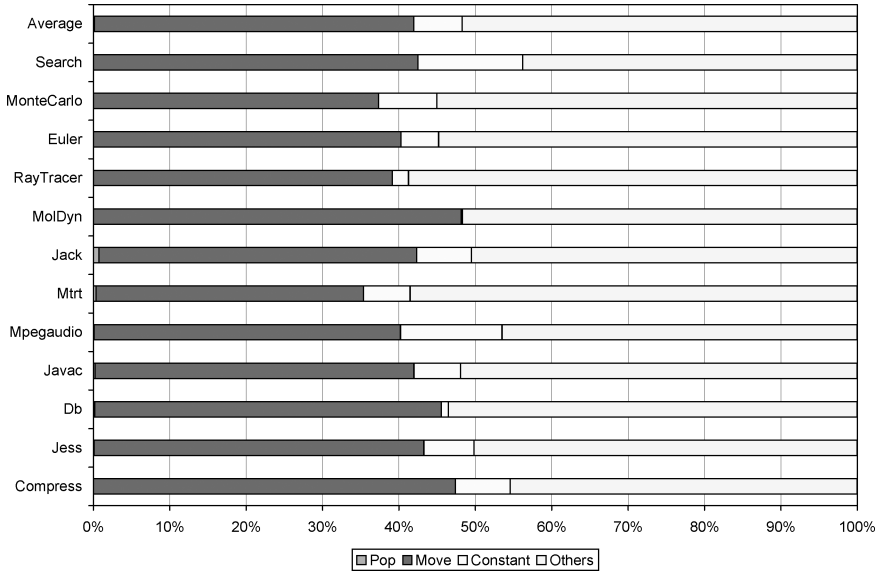

Fig. 3. Different categories of dynamically executed stack VM instructions.

calling method's Java frame into the virtual registers in the Java frame for the new (called) method. We considered a similar mechanism in our virtual register machine as in a stack JVM. We would place the parameters for a method invocation in consecutive registers, in the highest numbered registers for the method. Instead of copying the values of these registers into the stack frame of the called method, we could simply move the frame pointer to point to the first of these parameters. Although this would provide an efficient parameter passing mechanism, it prevents us from copy propagating into the source registers (parameters) of a method call. Even though the operands of method invocation VM instructions are contiguous after initial translation, once we have performed copy propagation and other optimizations this ordering is lost. However, the benefits of our optimizations are much greater than the small loss in efficiency of parameter passing.

3.3 Optimization

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

—Loads and stores between operand stack and local variables are translated into `move` instructions in register code. On average, more than 42% (see Figure 3) of executed VM instructions in the SPECjvm98 and Java Grande

benchmark suites (including library code) consist of loads and stores between local variables and the operand stack.

—Redundant loads of constant values and other arithmetic common subexpressions: In the stack architecture, constants are loaded onto the operand stack each time when needed for computation. The same constant could be loaded multiple times in a method, which is required on a stack-based architecture. An average of 6% (see Figure 3) of original executed instructions are constant load instructions.

In order to make a fair comparison, we try:

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

—to keep the instruction set and their implementation in the interpreter the same except for the adaptation to the new instruction format and those differences mentioned in the Section 3.1.

An important question is whether the resulting comparison is fair. If we applied the same optimizations to the stack code, would it also be improved? In fact, the Soot optimization framework [Vallée-Rai et al. 1999] was used to translate stack JVM code to three-address code. They applied more aggressive optimizations than we use, and translated the resulting code back to stack JVM code. In order to achieve any improvements in running time, inter-procedural optimizations (which we do not perform) were required. They concluded that *intra*-procedural optimizations generally have very little effect on Java bytecode, on the basis that these can only work on scalar operations. This strongly suggests that the differences in performance we measure are the result of inherent differences between stack and register code, rather than the result of applying optimizations to one and not the other.

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

A brief description of the optimizations is as follows:

—Copy propagation: Copy propagation [Muchnick 1997] is applied to eliminate `move` instructions in basic blocks. The stack pointer is used to find out whether an operand on the stack is alive or dead. Forward copy propagation is used to eliminate operand stack loads and backward copy propagation is used to eliminate operand stack stores.

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

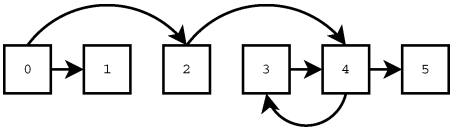

Fig. 4. The control flow of the example.

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

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

Fig. 5. Source code for the `hashCode()` method in the `java.lang.String`(GNU Classpath 0.90) class.

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

3.4 Putting it all together

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

(1) Translate original bytecode into virtual register intermediate representation and build a factored control flow graph [Choi et al. 1999]
(2) Apply local copy propagation on basic blocks [Muchnick 1997]
(3) Build a dominator tree [Lengauer and Tarjan 1979] and enhance the intermediate representation with SSA form [Cytron et al. 1991]
(4) Remove dead code [Cytron et al. 1991]
(5) Apply global copy propagation
(6) Apply global redundant load elimination
(7) Apply loop invariant code motion [Muchnick 1997]
(8) Virtual register allocation [Mössenböck 2000; Briggs et al. 1994] and remove SSA ϕ functions
(9) Write the optimized register code into virtual register bytecode in memory.

To demonstrate the effect of the optimizations, we present the following example (see Figure 5 for Java source code and Figure 6 for corresponding bytecodes) with 6 basic blocks and one loop (see Figure 4 for its control flow graph). In Figure 6:

—The VM instruction operands with # are immediate operands.
—Virtual register numbers are prefixed with the initial `r`.
—Field identifiers are shown using the names of the fields.

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

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

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

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

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

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

Fig. 6. Original stack VM code and corresponding register VM code for the `hashCode()` method in the `java.lang.String`(GNU Classpath 0.90) class.

—In each instruction, the register number after -> is the destination register.
—The stack VM instructions are numbered 1 to 37.
—The instruction numbers in the register code show the stack instruction from which each register instruction originated.

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

Table I. Hardware and Software Configuration

Processor	OS	Compiler
AMD Athlon(tm) 64 X2 Dual Core Processor 4400+	Linux 2.6.14	GCC 4.0.3
Intel(R) Pentium(R) 4 CPU 2.26GHz	Linux 2.6.13	GCC 2.95
DEC Alpha 800MHz 21264B	Linux 2.6.8	GCC 3.3.5
Motorola MPC 7447a (PowerPC) 1066MHz	Linux 2.6.18	GCC 4.0.2
Intel(R) Core(TM)2 CPU 2.13GHz	Linux 2.6.18	GCC 3.2.3

4. EXPERIMENTAL EVALUATION

4.1 Setup

For the present work, we used Cacao 0.95 (interpreter only with JIT disabled) as a base VM to implement the virtual register machine[4]. Cacao, released under the GPL, uses GNU Classpath as its class library and has a Boehm-Demers-Weiser garbage collector. Additionally, since version 0.93, Cacao has included a vmgen [Ertl et al. 2002] interpreter generator, which is used to define the virtual register machine instruction set and generate the interpreter. Both the virtual register and virtual stack interpreters support inline-threaded [Piumarta and Riccardi 1998; Ertl et al. 2006], direct-threaded, token-threaded, and switch dispatches.

We use the SPECjvm98 client benchmarks [SPEC 1998] (size 100 inputs) and Java Grande [Bull et al. 2000] (Section 3, data set size A). Methods are translated to register code the first time they are executed; thus all measurements in the following analysis include only methods that are executed at least once. The measurements include both the benchmark program code and the Java library code (GNU Classpath 0.90) executed by the VMs.

Table I shows the hardware and software configuration for the experiments. In the rest of the paper, we refer to these different processor architectures as AMD64, Intel Pentium 4, Alpha, PPC, and Intel Core 2 Duo. The choices of GCC compilers on different hardware platforms are based on their availability. We tested different GCC versions and selected the one that generated the interpreter with the best performance for each platform. We tried to make sure that the frequently used variables, such as the virtual IP, the virtual SP and the frame stack pointer are located in processor registers. Moreover, we had to avoid certain GCC versions because of GCC bugs: PR15242 and PR25285. These bugs could degrade the performance of interpreters up to 300% because of some optimizations performed on computed `goto`s. These bugs increase the dispatch cost of the interpreter. We believe that the increase in dispatch cost will affect stack-based VMs more than register-based VMs.

4.2 Static Instruction Analysis of Register Code

Figure 7 shows the breakdown of statically appearing VM instructions after converting to register code (translating and optimizing). On average 1.8% of VM

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

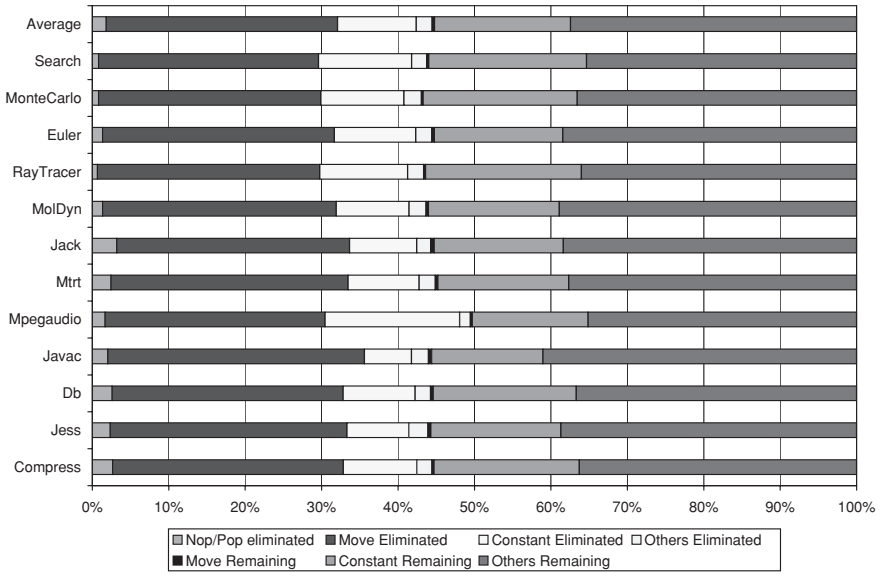

Fig. 7. Breakdown of statically appearing VM instructions before and after optimization for all the benchmarks.

instructions are `pop` or `pop2` instructions. These can simply be translated to `nop` instructions in the register VM and eliminated, because they move the stack pointer, but do not move any values or perform any computation. A significant number of statically appearing `move` instructions are eliminated. Originally, `move` instructions account for 31% of VM instructions, but this is reduced to only 0.32% (of the original instructions) after translation. Similarly, optimization results in the elimination of constant load instructions, from an average of 28% of total statically appearing VM instructions down to 18% (of the original instructions) after translation. This is achieved by reusing commonly occurring constants, thus allowing the constant loads to be removed. Eliminating other common subexpressions allows a further 2.1% of static VM instruction to be optimized away. Overall, an average of 44% of static VM instructions are eliminated.

4.3 Stack Frame Space

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

As part of the translation from stack to register code we apply a simple graph-colouring register allocation to pack the values which were previously split

Table II. The Comparison of Required Stack/Local Variable Slots (Virtual Registers) Between Stack and Register Architectures

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

between the locals and the evaluation stack into a smaller number of virtual registers. Table II shows the average number of stack frame slots required in a method for the locals and operand stack in the stack machine, and for the virtual registers in the register machine. On average, methods for the stack VM require 5.47 slots. The corresponding number for our register VM code is 4.61. It is important to note, however, that the register VM code normally has more live values. Eliminating redundant constant load instructions will keep more variables alive at the same time, which means more virtual registers are required. If we do not apply these redundancy elimination optimizations, we find that the register machine needs an average of only 4.02 slots. Even though a smaller stack frame size has little impact on the execution time of the VM interpreter, it may be beneficial to embedded or other small devices with tight memory constraints.

4.4 Dynamic Instruction Analysis of Register Code

In order to study the dynamic (runtime) behaviour of our register JVM code, we counted the number of VM instructions executed in the stack and register VMs. Figure 8 shows the breakdown of VM instructions dynamically executed before and after converting to register code.

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

(2) The second largest category of executed instruction elimination is constant load instructions (3.5% on average), which is much lower than the constant load instruction elimination (10% on average) in static code. The remaining dynamically executed constant VM instructions account for 2.9%. However, there are far more remaining constant load instructions (18%) in static

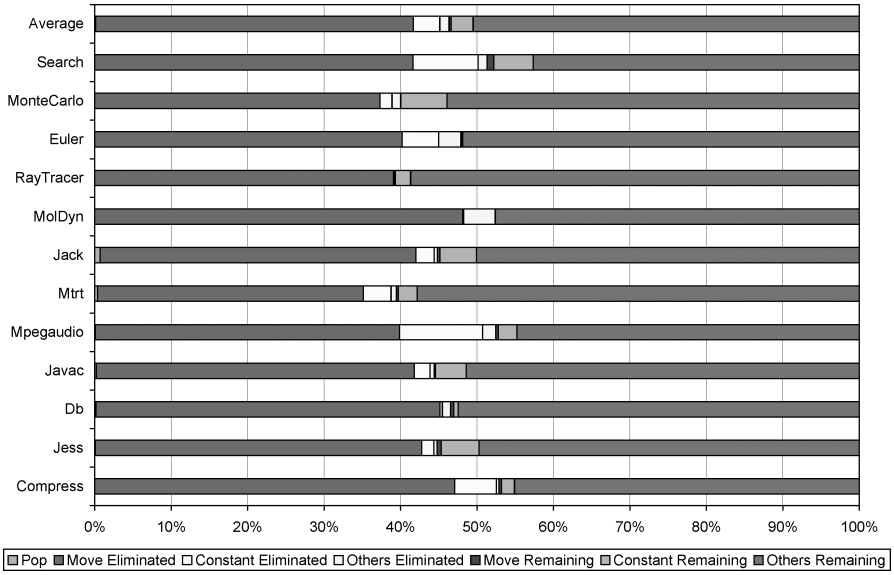

Fig. 8. Breakdown of dynamically appearing VM instructions before and after optimization for all the benchmarks.

code than those dynamically run (2.9%) in the benchmarks. We discovered that there are a large number of constant instructions in the initialization bytecode which are usually executed only once.

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

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

Overall, we eliminate an average of 46% of dynamically executed VM instructions. In general, copy propagations of `move` instructions produce the most effective result. Other optimizations are more dependent on the characteristics of the particular program. For example, in the benchmark *moldyn*, eliminated constant load VM instructions account for only 0.11% of total executed instructions, although such instructions account for 10% of static instructions.

4.5 Code Size

The register VM code size is usually larger than that of a stack VM. There are actually two effects in action here. Register machine instructions are larger than stack instructions because the locations of the operands must be expressed explicitly. On the other hand, register machines need fewer VM instructions to do the same work, so there are fewer VM instructions in the code. Figure 9 shows the increase in code size of our register machine code compared to the original stack code. On average, the register code size is 26% larger than that of the original stack code, despite the fact that the register machine requires 44% fewer static instructions than the stack architecture. This is a significant

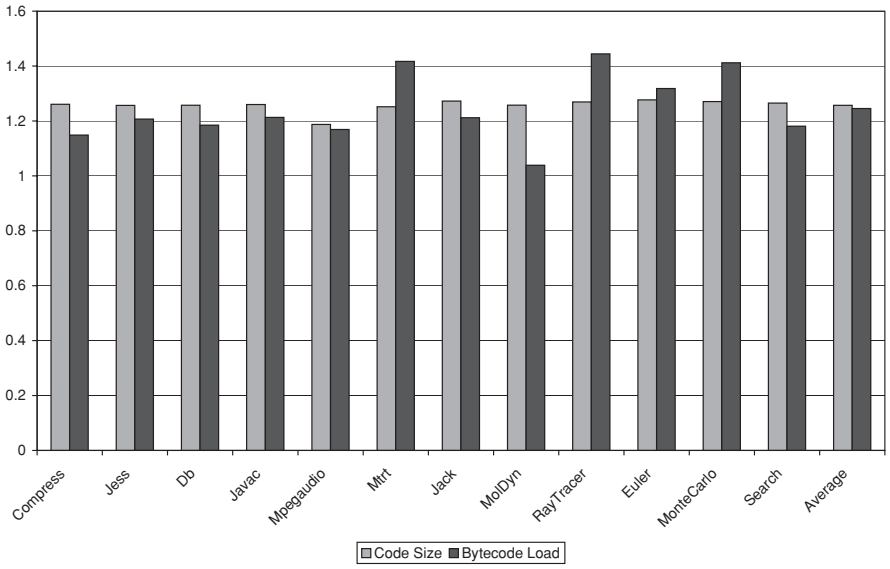

Fig. 9. Fractional increase in total code size (static) of executed methods and bytecode loads (dynamic) for register over stack architecture.

increase in code size, but it is far lower than the 45% increase reported by Davis et al. [2003].

Although static code size is an important issue, we also consider the size increase in executed bytecode when moving from stack to register VMs. This increase in executed bytecode is a direct result of the (static) increased code size of the register JVM. Because of this, more VM instruction bytecodes (both opcodes and operands) must be loaded, on average, from memory as the program is interpreted. Figure 9 also shows the resulting increase in bytecode loads. Interestingly, the increase in overall code size is often very different from the increase in instruction bytecodes loaded in the parts of the program that are executed most frequently. Nonetheless, the average increase in loads (25%) is similar to the average increase in code size (26%). An alternative to fetching each operand location separately is to use a four-byte VM instruction containing the opcode and three register indices. This entire VM instruction could be fetched in a single load. However, it would still be necessary to extract the opcode and register numbers inside the four-byte VM instruction. This would involve shifting and masking the loaded VM instruction. Clearly the cost of such operations varies from one processor to another. For example the Northwood-core Pentium 4 has no barrel shifter, so large shifts are expensive. In general, if a piece of code loads four successive bytes and does something with them, most compilers generate separate byte loads, rather than a single word load and using shifts and masks to extract the bytes[5].

[5]Preliminary experiments by the authors on the Pentium IV suggest that shifting/masking is the slower approach on that particular architecture.

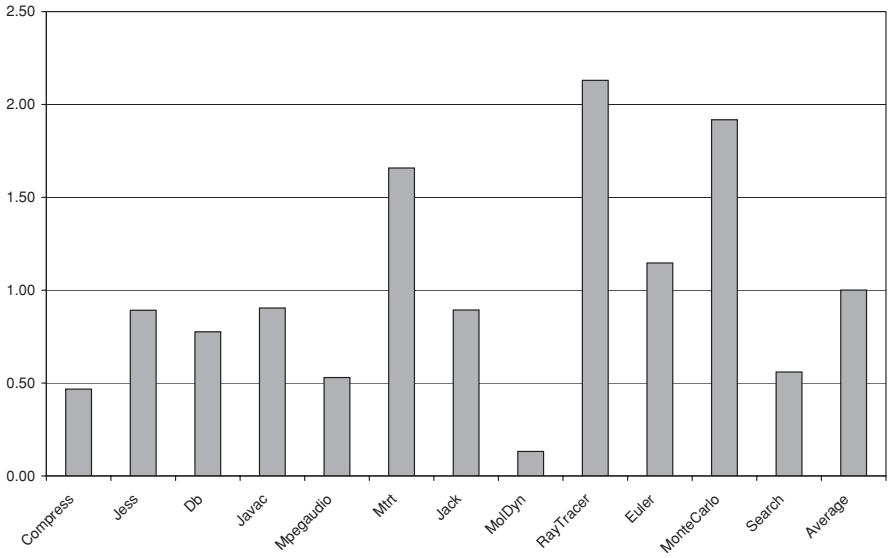

Fig. 10. Increase in dynamically loaded bytecode instructions per VM instruction dispatch eliminated by using a register rather than stack architecture. In other words, if the register VM uses one less dispatch, how many extra bytes of bytecode must it load?

Despite the cost of increased bytecode loads due to larger code, the fact that that fewer VM instructions are needed by the register VM gives a significant performance advantage. To measure the relative importance of these two factors, we compared the number of extra dynamic bytecode loads required by the register machine per dynamically executed VM instruction eliminated. Figure 10 shows that the number of additional byte loads per executed VM instruction eliminated is small at an average of only 1.00 loads. On most architectures, even one CPU load costs much less to execute than an instruction dispatch, with its difficult-to-predict indirect branch. This strongly suggests that register machines can be interpreted more efficiently on most modern architectures.

4.6 CPU Loads and Stores

Apart from CPU loads of instruction bytecodes, the main source of CPU loads in a JVM interpreter comes from moving data between the local variables and the stack. In most interpreter-based JVM implementations, the stack and the local variables are represented as arrays in memory. Thus, moving a value from a local variable to the stack (or vice versa) involves both a CPU load to read the value from one array, and a CPU store to write the value to the other array. A simple operation such as adding two numbers can involve large numbers of CPU loads and stores to implement the shuffling between the stack and registers.

In our register machine, the virtual registers are also represented as an array. However, VM instructions can access their operands in the virtual register array directly, without first moving the values to an operand stack array. Thus, the virtual register machine can actually require fewer CPU loads and stores

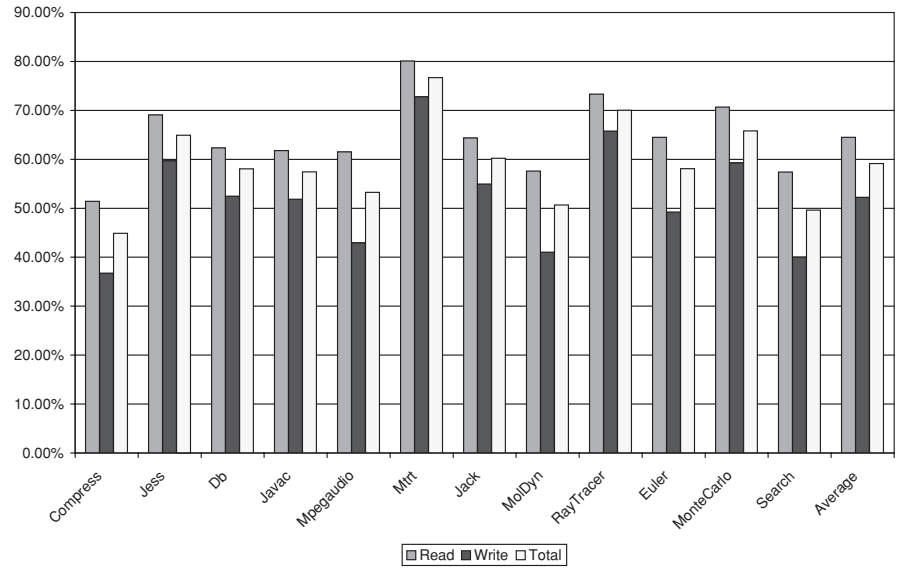

Fig. 11. Dynamic number of CPU loads and stores required to access virtual registers in our virtual register machine, expressed as a percentage of the corresponding loads and stores used to access the stack and local variables in a virtual stack machine.

to perform the same computation. Figure 11 shows (a simulated measure of) the number of dynamic CPU loads and stores required for accessing the virtual register array, as a percentage of the corresponding loads and stores for the stack JVM to access the local variable and operand stack arrays. The register VM requires only 65% as many CPU loads and 52% as many CPU writes, with an overall figure of 59%.

In order to compare these numbers with the number of additional loads required for fetching instruction bytecodes, we express these memory operations as a ratio to the dynamically executed VM instructions eliminated by using the virtual register machine. Figure 12 shows that on average, the register VM requires 1.74 fewer CPU memory operations to access such variables per instruction dispatch eliminated. This is much larger than the number of additional loads required due to the larger size of virtual register code (1.00). Thus, the interpreter for the register VM would execute fewer memory operations overall.

However, these measures of memory accesses for the local variables, the operand stack and the virtual registers depend entirely on the assumption that they are implemented as arrays in memory. In practice, we have little choice but to use an array for the virtual registers, because there is no way to index CPU registers like an array on most real architectures. However, stack caching [Ertl 1995] can be used to keep the topmost stack values in registers, and eliminate large numbers of associated CPU loads and stores. For example, around 50% of stack access CPU memory operations could be eliminated by keeping just the topmost stack item in a register [Ertl 1995]. Thus, in many implementations the virtual register architecture is likely to need more CPU loads and stores to access these kinds of values.

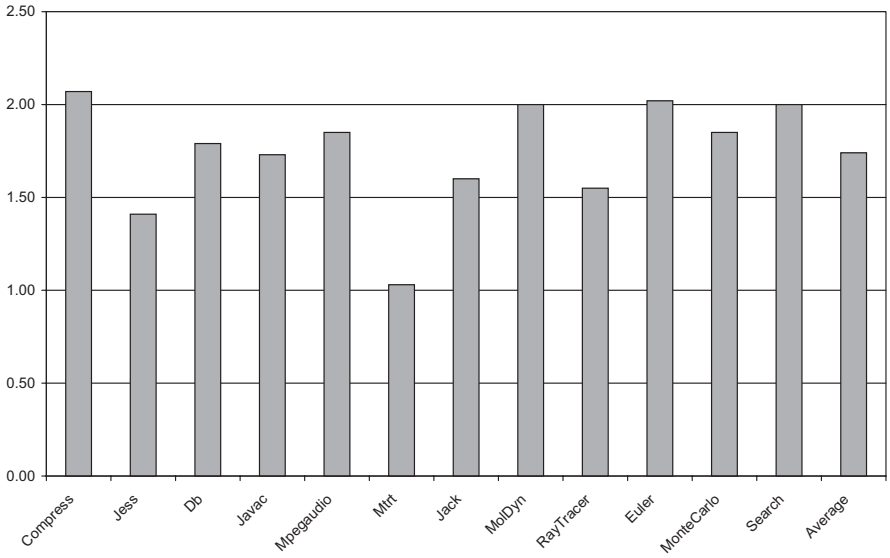

Fig. 12. The reduction of CPU memory accesses for each executed VM instruction eliminated by using a register VM rather than a stack VM. This is analogous to the measurement in Figure 10.

4.7 Timing Results

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

We compare the performance of stack JVM interpreters and register JVM interpreters with four different dispatch mechanism: (1) switch dispatch, (2) token-threaded dispatch, (3) direct-threaded dispatch and (4) inline-threaded dispatch [Piumarta and Riccardi 1998] (see Section 2). For fairness, we always compare the performance of stack and register interpreter implementations which use the same dispatch mechanism.

Figure 13 shows the speedup in running time of our implementation of the virtual register machine compared to the virtual stack machine on the AMD64 machine using the various dispatch mechanisms. With `switch` dispatch, the register VM has the highest average speedup (1.48) because `switch` dispatch is the most expensive. Even with the efficient inline-threaded dispatch, the register VM still has an average speedup of 1.15.

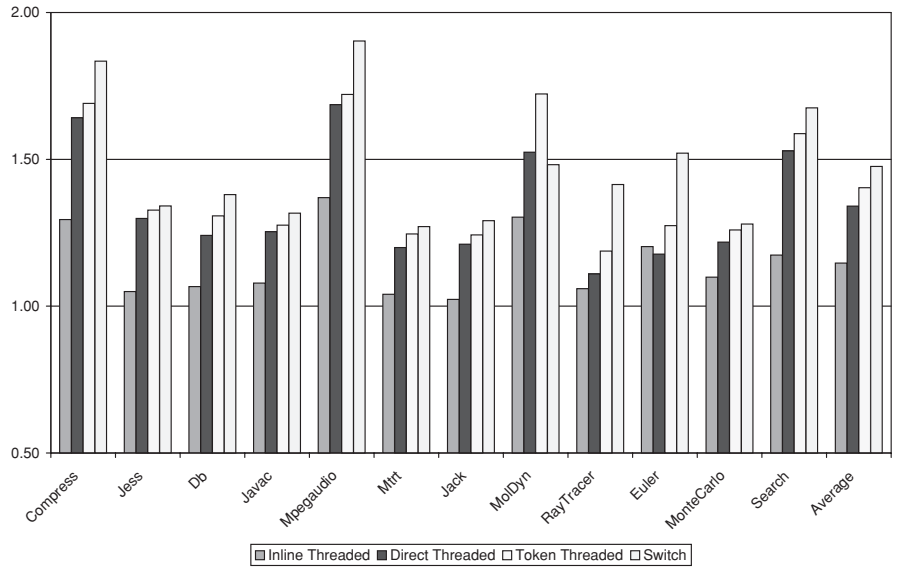

Fig. 13. AMD64: register VM speedups over stack VM of same dispatch (based on average real running time of two runs).

Figure 14 shows the same figures for a Pentium 4 machine, whose processor utilizes a trace cache. With inline-threaded dispatch, the register VM has an average speedup of 1.00, and some benchmarks are very close to or worse than on the stack VM. The `switch` register VM has highest speedup (1.46). The `mtrt` benchmark performs very poorly for various dispatches, which may be due to high cost of threading using GCC 2.95 compiler.

Figure 15 shows the speedup of register VMs over stack VMs on the Intel Core 2 Duo processor. The average speedups of register over stack-based VMs are 1.15 (inline-threaded), 1.32 (direct-threaded), 1.29 (token-threaded), and 1.65 (switch).

Figure 16 shows the speedups of register VMs over stack VMs on the IBM PowerPC processor. The average speedups for the four dispatch mechanisms (inline-threaded, direct-threaded, token-threaded and switch) are 1.16, 1.30, 1.29, and 1.41 respectively.

Figure 17 shows the speedup of register VMs over stack VMs on the Alpha processor. The inline-threaded dispatch is not working for Alpha and there are still bugs which prevent `javac` from running correctly (and thus is excluded from the benchmark results). The average speedups are 1.22 (direct-threaded), 1.25 (token-threaded), and 1.64 (switch).

4.8 Performance Counter Results

To obtain a finer grained view of benchmark performance, we use AMD64 hardware performance counters to measure various processor events during the execution of the programs. Figures 18 and 19 show performance counter results for the SPECjvm98 benchmarks *Compress* and *Jack*. We measure the data cache accesses, data cache misses, instruction cache fetches, instruction cache misses,

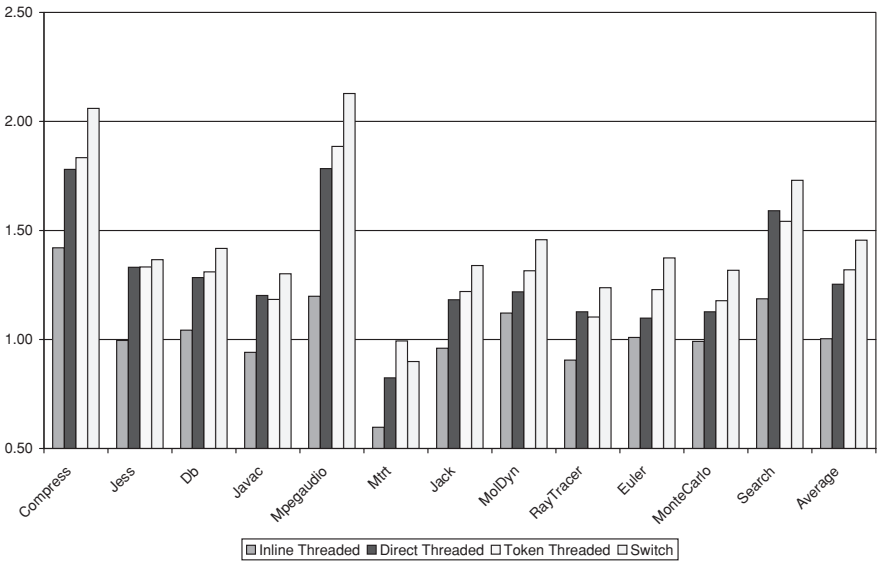

Fig. 14. Intel Pentium 4: register VM speedups over stack VM of same dispatch (based on average real running time of five runs).

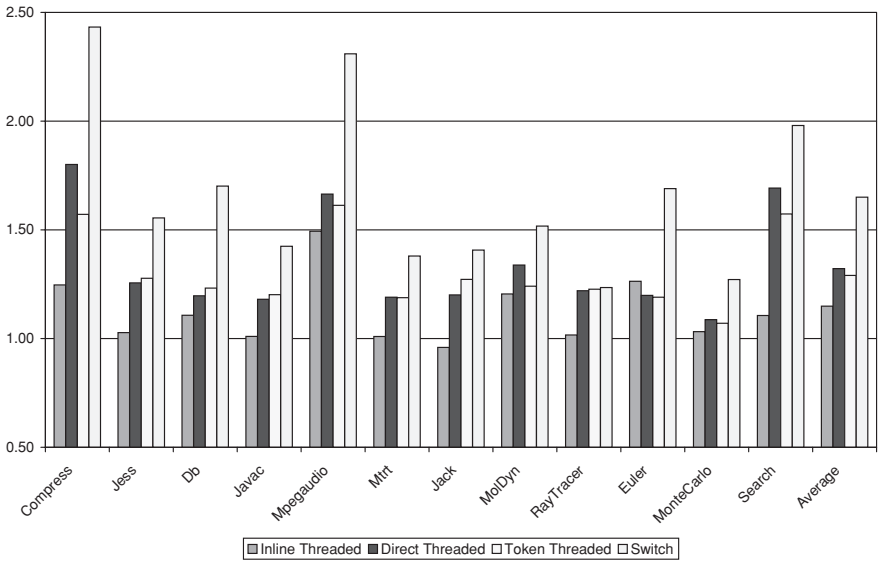

Fig. 15. Intel Core 2 Duo: register VM speedups over stack VM of same dispatch (based on average real running time of three runs).

retired taken branches (which include indirect branches; unfortunately there is no way to measure indirect branches alone by using AMD64's performance counters), retired taken branches mispredicted (indirect branches are the main source of misprediction), and retired instructions[6].

[6]On out-of-order processors, a retired instruction is one that has been executed and completed.

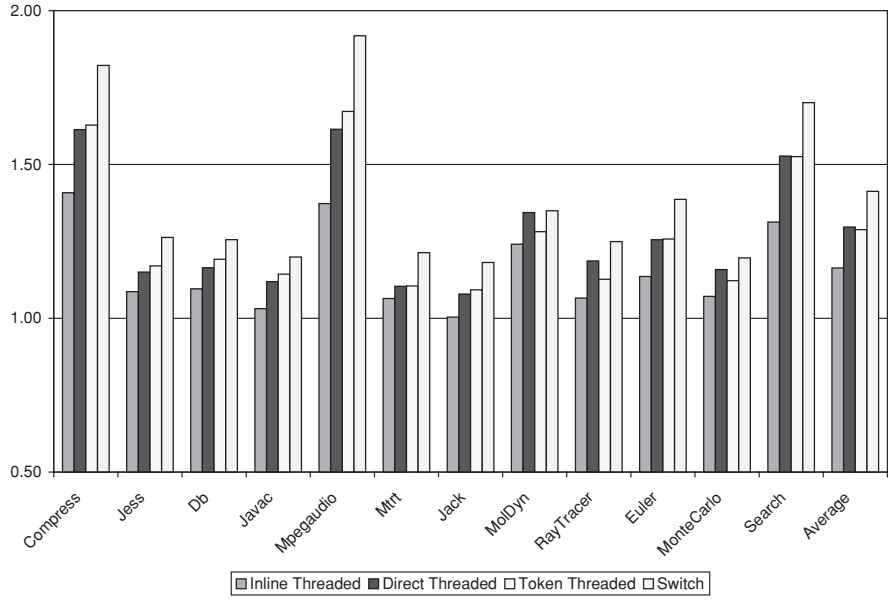

Fig. 16. IBM PowerPC: register VM speedups over stack VM of same dispatch (based on average real running time of three runs).

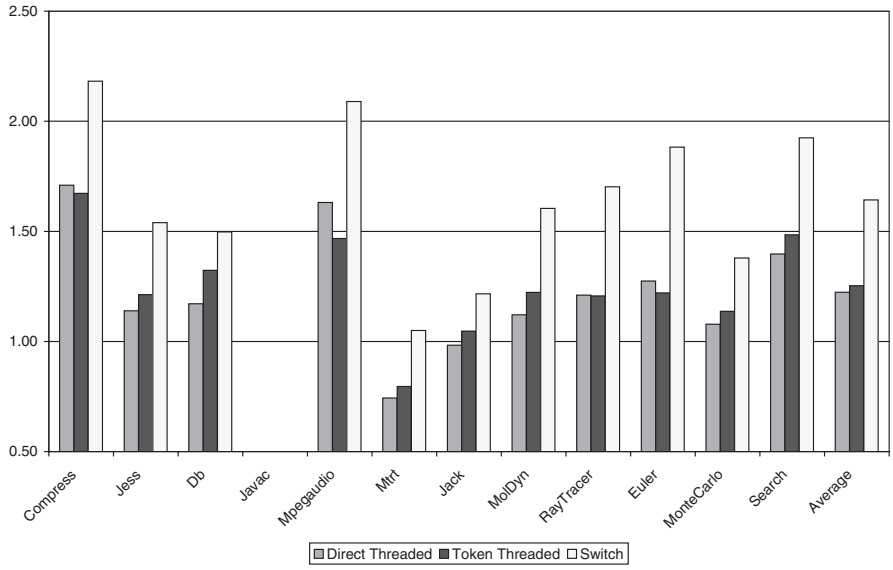

Fig. 17. Alpha: register VM speedups over stack VM of same dispatch (based on average real running time of five runs).

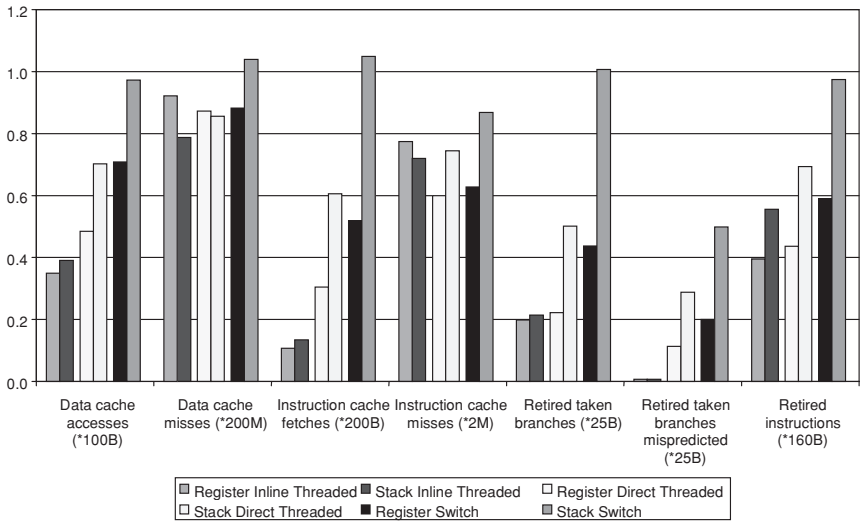

Fig. 18. Compress: AMD64 performance counters.

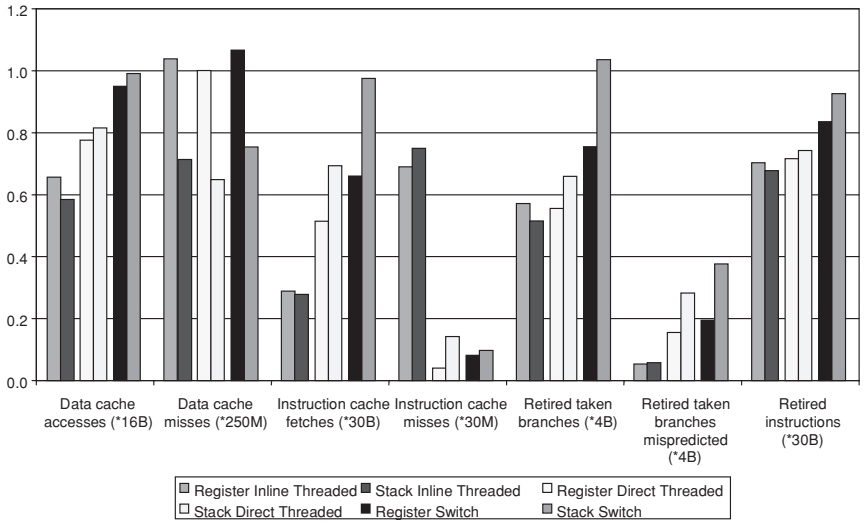

Fig. 19. Jack: AMD64 performance counters.

Figure 18 shows the measured performance counters for inline-threaded, direct-threaded and switch dispatches for the `compress` benchmark. From Figure 8, we know that, for the `compress` benchmark, 54% of executed VM instructions are eliminated compared to the stack architecture. As the dispatch method becomes more efficient, the difference between corresponding performance counters for register VMs and stack VMs becomes smaller. For inline-threaded dispatch, retired taken branches are almost the same for register and stack VMs. The main source of advantage is fewer retired instructions, which gives the register VM a speedup of 1.15 over the stack VM for inline-threaded dispatch.

For the compress benchmark, the register version of the machine always executes fewer real machine instructions. As we saw in Figure 10, translation to register format actually results in less than 0.5 extra bytecode loads per VM instruction eliminated. However, compress is the benchmark with the greatest reduction in real machine memory operations for manipulating local values (see Figure 11). This accounts for the much lower number of retired real machine instructions.

Figure 19 shows the measured performance counters of the jack benchmark for inline-threaded, direct-threaded and switch dispatches. From Figure 8, we know that 44% of executed instructions are eliminated from jack in the register VM. The data cache miss ratio and instruction cache miss ratio are much higher than those of the compress benchmark. For inline-threaded dispatch, the register VM shows more data cache accesses, data cache misses, retired taken branches, and retired instructions than those of the stack VM. Nonetheless, instruction cache misses and retired taken branches mispredicted are lower. The inline-threaded dispatch speedup of register VM over stack VM for the jack benchmark is only 1.02. For inline-threaded dispatch, both stack and register VMs show very high numbers of instruction cache misses when compared with other dispatch mechanisms because of binary executable code replication.

4.9 Dispatch Comparison

All the comparisons to this point have been between stack and register architecture pairs using the same dispatch mechanism. For example, we have shown performance of the register VM interpreter using token-threaded dispatch as a speedup over the performance of the corresponding stack VM. In this section, we compare differences between the dispatch mechanisms. The performance of the stack VM interpreter using switch dispatch is the baseline value (speedup=1.0) and all other variants are shown relative to that value (see Figure 20). Sun's JDK 1.6.0 (interpreter mode only) gives an indication of the speed of Cacao's stack and register VMs and should be treated with caution because of the different implementation.

We see that the more complex, less portable dispatch mechanisms give the greatest speedups. We also observe that, at least for the benchmark results presented, the register machine has a significant edge. For example, if one has to choose between using direct-threaded dispatch on a stack VM and switch dispatch on a register VM, it should be noted that there is little difference in execution speed between the two implementations. However, switch dispatch is simpler to implement and much more portable. Furthermore interpreted bytecode is a fraction (typically 25%–50%) of the size of threaded code, so there is also a significant space saving. Therefore, the register VM interpreter with switch dispatch is preferable. If one is purely concerned with code size, a token-threaded stack VM is still the most compact option.

4.10 Discussion

Although our implementation of the register JVM translates the stack bytecode into register bytecode at runtime, we do not envision this in a real-life

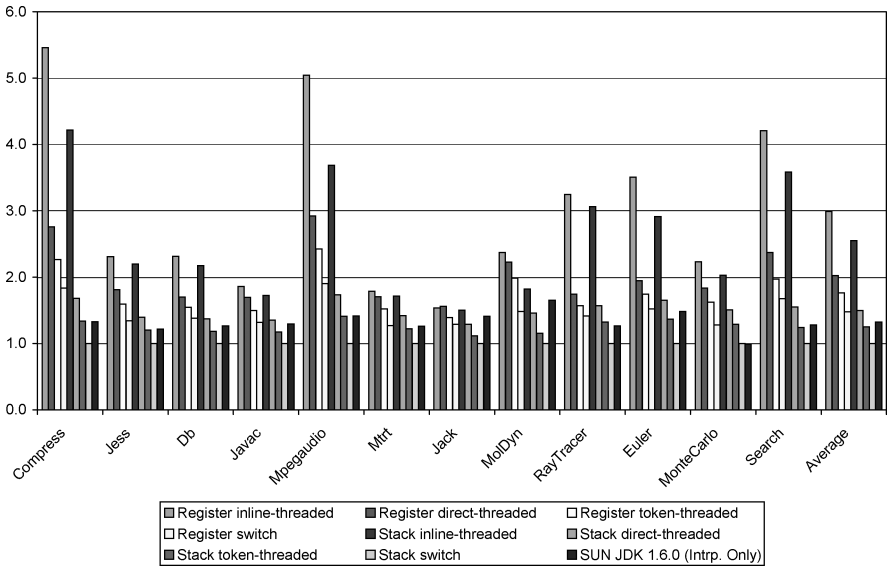

Fig. 20. AMD64: speedups over the stack switch interpreter.

implementation. The purpose of our implementation is to evaluate a virtual register JVM against an equivalent stack based one. Our register JVM implementation originates directly from a modification of a stack-based JVM implementation, thus giving us two VMs identical in every other regard. Apart from the necessary adaptation of the interpreter loop, along with some garbage-collection and exception handling modifications, there are very few changes to the original code segments responsible for interpreting bytecode instructions. The objective is to provide a fair comparison between the stack-based JVM and the register-based JVM.

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

An important question is whether we would generate better register code if we were to compile directly from Java source code, rather than translating from stack code. The *javac* compiler generates optimized stack code, but the

optimizations may not suit register code. Furthermore, eliminating (partially) redundant expressions in stack code is rarely worthwhile, because the common expression must be stored and later recovered, which is often more expensive than recomputing the expression. Although eliminating simple redundant computations in stack code is easy, we might find it easier to eliminate more redundancy if we were working from source code. In particular, eliminating some kinds of redundant expressions, such as those described in the next section, depends on pointer analysis to ensure that the transformation is safe. Pointer analysis may be also be easier to perform on source code rather than after its translation to register code.

5. MORE OPTIMIZATIONS

5.1 Redundant Heap Load Elimination

As we saw in Section 3.3, register machines can take advantage of redundant computations more easily than stack machines. This is because (unlike stack VMs) register VMs do not destroy operands to VM instructions as they use them. The results presented in Section 4 are for register machine code where redundant loads of constants and some simple common subexpressions involving local variables were eliminated.

There is another category of redundant loads—the loads from class or object fields and array elements, and there has been some work on eliminating these redundant loads in compilers [Fink et al. 2000]. However, it is very important to note that eliminating such loads from heap data structures requires sophisticated pointer alias analysis to ensure that the object or array element is not modified between apparently redundant loads [Diwan et al. 1998]. In particular, we need to know whether a reference to an object has escaped into another thread, which may modify the object. Alias analysis is complex and slow; we have not implemented it in our translation.

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

Figure 21 shows that an average of 5% of original executed VM instructions can be eliminated by removing redundant `getfield` VM instructions. The corresponding figure for array loads is 2%. All benchmarks benefit from redundant `getfield` elimination, while only a few benchmarks benefit from redundant array load elimination. In the Euler benchmark, eliminated redundant array loads account for 13% of original executed VM instructions. After all optimizations, the register machine requires only 23% of the original stack machine instructions.

Figure 22 shows the same dispatch speedup results for the AMD64. The average speedup for inline-threaded goes from 1.15 to 1.29 and that of switch dispatch from 1.48 to 1.74 as this optimization is added.

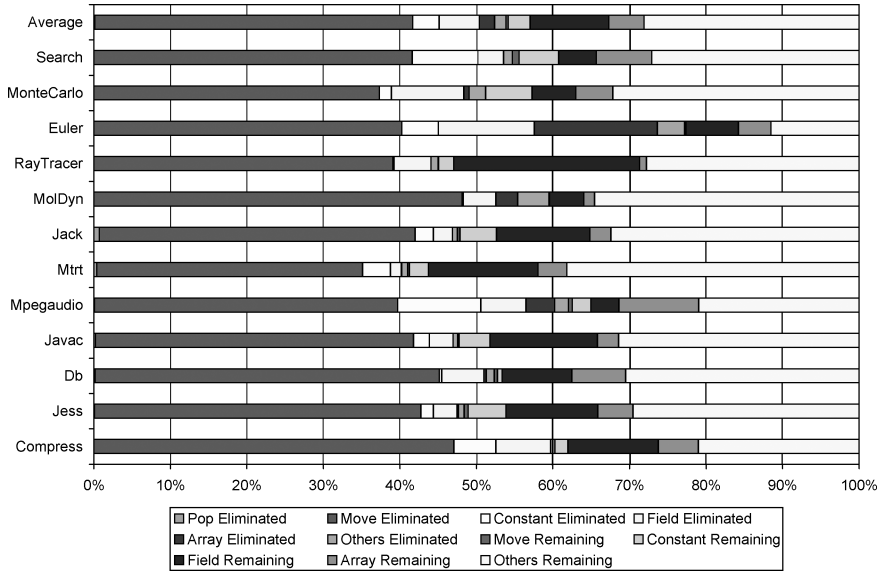

Fig. 21. Breakdown of dynamically appearing VM instructions with additional redundant heap load elimination for all the benchmarks. These results are indicative only, because our translator makes unsafe assumptions about aliasing.

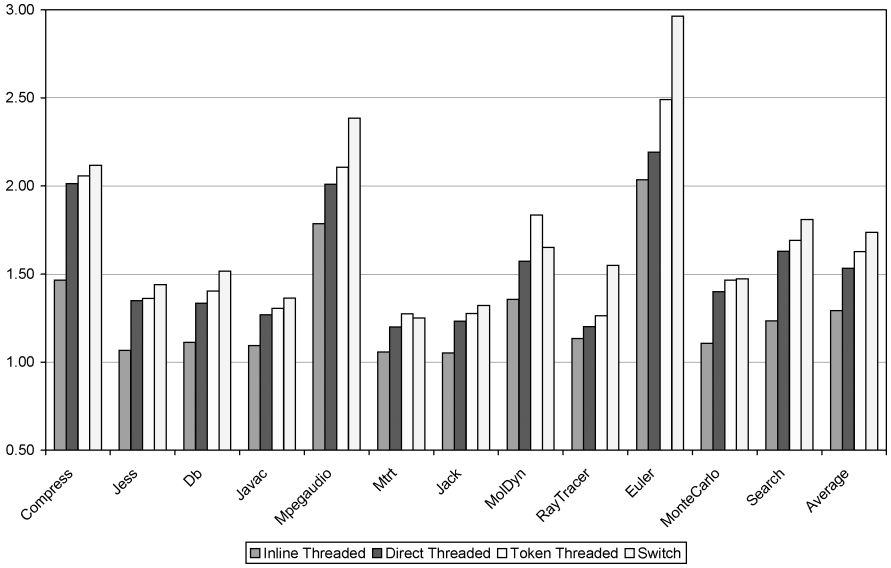

Fig. 22. AMD64: Register VM speedups with additional redundant heap load elimination (based on average real running time of two runs). These results are indicative only, because our translator makes unsafe assumptions about aliasing.

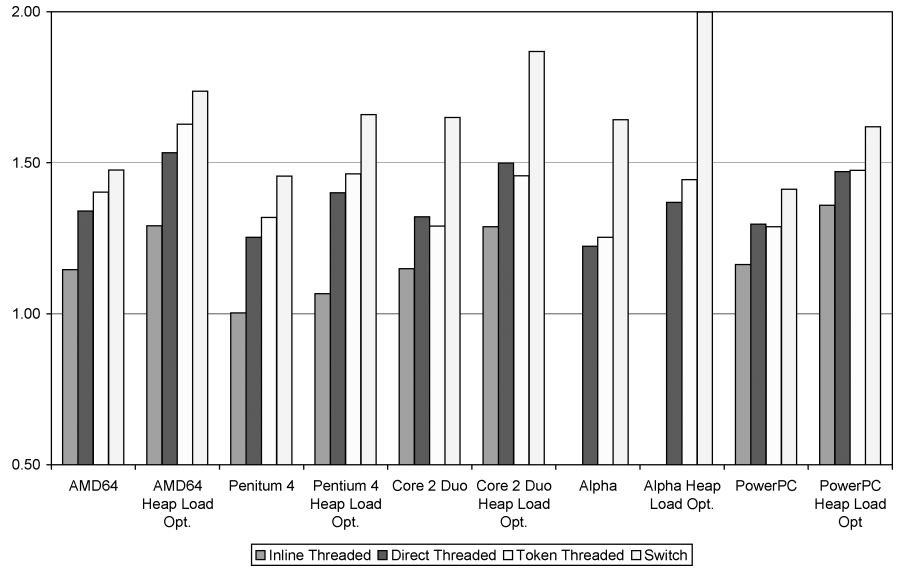

Fig. 23. The average speedups of register VM over stack VM using the same dispatch for different processors. The results which include heap load optimization are indicative only, because our translator makes unsafe assumptions about aliasing.

Figure 23 summarizes the average speedups of register VM over stack VM using the same dispatches with/without redundant heap load elimination. The register VM could potentially benefit significantly from eliminating these loads, but a real implementation of this optimization would require very sophisticated alias and escape analysis.

5.2 Stack Caching for Stack VM

Stack caching [Ertl 1995] can be used to keep the topmost stack values in registers, and eliminate large numbers of associated CPU loads and stores. Take, for example, the real machine memory operations required for the operand stack access. Around 50% of these real machine memory operations could be eliminated by keeping just the topmost stack item in a register [Ertl 1995]. Figure 24 shows the speedup over the stack VM with/without stack caching[7] using the same dispatch mechanisms. Stack caching did show improvements for the stack VMs. This improvement results in the speedups of the register VM going from 1.16 and 1.30 (over stack VM with no caching) down to 1.14 and 1.26 (over stack VM with caching) for inline-threaded and direct-threaded dispatch respectively.

5.3 Static Superinstructions

One way to reduce the number of VM interpreter dispatches is to add static *superinstructions* to the instruction set of the VM. These are new VM

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

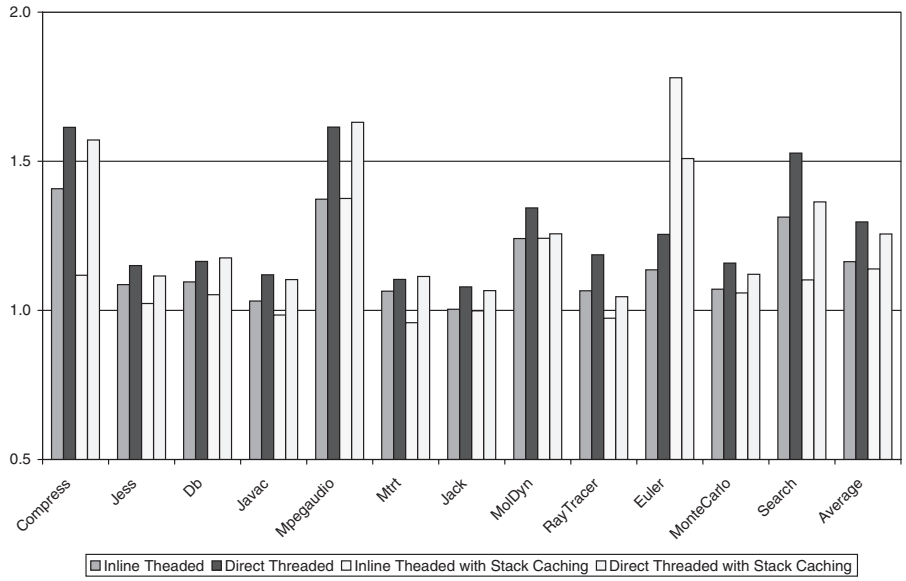

Fig. 24. PowerPC: register VM speedups over stack VM (with and without stack caching) of same dispatch (based on average real running time of two runs).

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

The main problem with superinstructions is choosing appropriate sequences. The superinstructions must be hardwired into the interpreter, at a time when the program to be run is usually unknown. Perhaps the best strategy for selecting sequences is to look at a large variety of programs and identify the most important sequences of VM instructions in those programs. Eller [2005] investigated using SPECjvm98 benchmarks to select superinstructions using a wide variety of selection strategies. He found that superinstructions could be added to a stack-based VM which would reduce the number of dispatches by up to 40%. However, to achieve that reduction, 1000 superinstructions were needed. This means that it is no longer possible to encode the instruction opcode in a single byte. Furthermore, the interpreter code to implement the superinstructions becomes significantly larger than that of the original interpreter. In contrast, the register machine does not require any additional VM instructions.

5.4 Two-Address Instructions

Our register JVM uses a three-address instruction format for arithmetic instructions. An obvious way to reduce code size would be to use a two-address

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

6. APPLICABILITY OF RESULTS TO RELATED QUESTIONS

Although our experiments in this work have been limited to the JVM, we believe that the results will extend to other VMs which employ an interpreter. Already, the conversion of the Lua VM from stack machine to register machine (the upgrade from version 4.0 to version 5.0) has yielded a substantial improvement in performance. Ierusalimschy et al [2005] have compared the stack machine implementation of version 4.0 to an equivalent register machine implementation (with no additional optimizations). Across their selected benchmarks, the register machine was an average of 1.30 times faster than the stack machine. More significantly, on the benchmark they specifically selected to test the execution engine, a speedup of 2.28 was reported.

The main benefit of the transition from stack VM to register VM is the reduction in the number of VM instruction dispatches, and consequently a reduction in branch mispredictions. There are other benefits which we have observed such as a reduction in real machine instructions at the CPU level. For coarse-grained VMs with higher-level instruction sets which have a significantly lower number of instruction dispatches to begin with, the transition to a register VM will not yield the same speedups. For example, Vitale and Abdelrahman [2004] found that inline threading had little benefit for a Tcl virtual machine, because each VM instruction performed a lot of work. Hence, dispatch accounted for only a small proportion of running time. It is likely that we would see similar results in a comparison of stack and register VMs for Tcl. Where the cost of dispatch is a small proportion of total time, there will be little benefit in any optimization to reduce dispatches.

Another interesting question is whether a stack or register VM is more suitable as a source language for JIT compilation. Winterbottom and Pike [1997] suggest that a register IR may be easier to compile to native code because it is closer to the register architecture used by real processors. Others argue that a stack machine is better, because stack code does not make assumptions about the number of available registers.

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

However, an optimizing JIT compiler is typically a complex piece of software, and translating the VM bytecode to a format more useful to the compiler is likely to be only a small part of compilation regardless of whether a stack or register VM is used. On the other hand, our results indicate that for a simple code-inlining JIT, there are some significant gains to be made in choosing a register IR rather than a stack IR. For a more aggressive JIT, the choice is not so clear. Finally, for mixed mode JIT compilers such as Sun's Hotspot VM [Sun-Microsystems 2001], interpretation speed is still important, and therefore a register VM may be used to improve the performance of interpretation.

7. RELATED WORK

A large number of JIT compilers have been constructed for stack-based VMs such as the JVM. These include Cacao [Krall and Grafl 1997] and Jikes RVM [Arnold et al. 2002]. Our translator uses the same standard, well-known techniques that are used in these JIT compilers. The HotSpot JVM [Sun-Microsystems 2001] uses a mixed-mode interpreter and JIT compiler. Interpreting the large amount of rarely executed code in many Java programs avoids the time and memory overhead of compilation and can be faster than JIT compilation.

Myers [1977] attempts to refute the idea that stack machines will necessarily result in smaller code, with lower cost to access operands. The argument is based on measurements of real programs which show that the expression in most assignment statements is extremely simple. Thus, in most cases, operands must be loaded to the stack for use, rather than already being there as part of the evaluation of a complex expression. Beyond measurements of the complexity of expressions, Myers presents only a handful of small examples showing situations where register code is superior to stack code. Schulthess and Mumprecht [Schulthess and Mumprecht 1977] argue that Myers' work is inconclusive, because programs contain features other than expressions that are better expressed using stacks (e.g. subroutine calls, parameter passing and multitasking). No quantitative data is provided.

The controversy between stack and register code has arisen again recently because of the decision to make the Parrot VM, the intermediate representation for the Perl 6 language, a register rather than stack machine. Arguments for this design decision [Sugalski 2002] have been based on just a couple of small examples, rather than studies of real programs. The Lua VM [R. Ierusalimschy et al. 1996] was also switched from a stack to a register machine, with the release of version 5.0 in 2003. Similar suggestions were proposed for the JVM [McGlashan and Bower 1999] and the Inferno VM [Winterbottom and Pike 1997], again without studies of real programs.

Much of the early work on RISC architectures was based on systematic studies of programs that examined the real, rather than presumed, frequency of instruction usage. Studies on the IBM 360 [Shustek 1978; Alexander and Wortman 1975] and the VAX [Wiecek 1982] caused researchers to rethink the complex instruction sets of the time, and led to the first RISC architectures [Patterson and Ditzel 1980]. Today, basing design decisions on measured, rather than presumed, frequencies of instruction usage has become widely accepted as sound engineering practice.

8. CONCLUSIONS

A long standing question has been whether virtual stack or virtual register VMs can be executed more efficiently using an interpreter. Register VMs can be an attractive alternative to stack architectures because they enable the number of executed VM instructions to be substantially reduced. In this paper we have built on the previous work of Davis et al. [Davis et al. 2003; Gregg et al. 2005], which counted the number of instructions for the two architectures using a simple translation scheme. We have presented a much more sophisticated translation and optimization scheme for translating stack VM code to register VM code, which we believe gives a more accurate measure of the potential of virtual register machine architectures. We have also presented experimental results for a fully-featured register JVM.

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

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers of VEE 2005 and ACM TACO, whose comments greatly improved earlier versions of this paper.

REFERENCES

ALEXANDER, W. AND WORTMAN, D. 1975. Static and dynamic chracteristics of XPL programs. *Computer 8,* 11 (Nov.), 41–46.

ANTONIOLI, D. N. AND PILZ, M. 1998. Analysis of the Java class file format. Tech. rep.

ARNOLD, M., HIND, M., AND RYDER, B. G. 2002. Online feedback-directed optimization of Java. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, New York. 111–129.

BELL, J. R. 1973. Threaded code. *Commun. ACM 16*, 6, 370–372.

BERNDL, M., VITALE, B., ZALESKI, M., AND BROWN, A. D. 2005. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *2005 International Symposium on Code Generation and Optimization*.

BRIGGS, P., COOPER, K. D., AND TORCZON, L. 1994. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems 16*, 3 (May), 428–455.

BULL, M., SMITH, L., WESTHEAD, M., HENTY, D., AND DAVEY, R. 2000. Benchmarking Java Grande applications. In *Second Ineternational Conference and Exhibtion on the Practical Application of Java*. Manchester, UK.

CHOI, J.-D., GROVE, D., HIND, M., AND SARKAR, V. 1999. Efficient and precise modeling of exceptions for the analysis of Java programs. In *PASTE '99: Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM Press, New York. 21–31.

CLAUSEN, L. R., SCHULTZ, U. P., CONSEL, C., AND MULLER, G. 2000. Java bytecode compression for low-end embedded systems. *ACM Trans. Program. Lang. Syst. 22*, 3, 471–489.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13*, 4 (Oct.), 451–490.

DAVIS, B., BEATTY, A., CASEY, K., GREGG, D., AND WALDRON, J. 2003. The case for virtual register machines. In *Interpreters, Virtual Machines and Emulators (IVME '03)*. 41–49.

DIWAN, A., MCKINLEY, K. S., AND MOSS, J. E. B. 1998. Type-based alias analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*. 106–117.

ELLER, H. 2005. Optimizing interpreters with superinstructions. M.S. thesis, Institut für Computersprachen, Technische Universität Wien. http://www.complang.tuwien.ac.at/ Dimplomarbeiten/eller05.ps.gz.

ERTL, M. A. 1995. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*. 315–327.

ERTL, M. A. AND GREGG, D. 2003. The structure and performance of *efficient* interpreters. *The Journal of Instruction-Level Parallelism 5*. http://www.jilp.org/vol5/.

ERTL, M. A., GREGG, D., KRALL, A., AND PAYSAN, B. 2002. vmgen—A generator of efficient virtual machine interpreters. *Software—Practice and Experience 32*, 3, 265–294.

ERTL, M. A., THALINGER, C., AND KRALL, A. 2006. Superinstructions and replication in the Cacao JVM interpreter. *Journal of .NET Technologies 4*, 25–32. Journal papers from *.NET Technologies 2006* conference.

FINK, S., KNOBE, K., AND SARKAR, V. 2000. Unified analysis of array and object references in strongly typed languages. *Lecture Notes in Computer Science* Volume 1824 (Feb.), 155–174.

GOSLING, J. 1995. Java Intermediate Bytecodes. In *Proc. ACM SIGPLAN Workshop on Intermediate Representations*. ACM Sigplan Notices, vol. 30:3. San Francisco, CA. 111–118.

GREGG, D., BEATTY, A., CASEY, K., DAVIS, B., AND NISBET, A. 2005. The case for virtual register machines. *Science of Computer Programming, Special Issue on Interpreters Virtual Machines and Emulators 57*, 319–338.

IERUSALIMSCHY, R. L., DE FIGUEIREDO, H., AND CELES, W. 1996. Lua—an extensible extension language. *Software: Practice and Experience 26*, 6, 635–652.

IERUSALIMSCHY, R., DE FIGUEIREDO, L., AND CELES, W. 2005. The implementation of Lua 5.0. *Journal of Universal Computer Science 11*, 7, 1159–1176. `http://www.jucs.org/jucs_11_7/the_implementation_of_lua`.

KISTLER, T. AND FRANZ, M. 1999. A tree-based alternative to Java byte-codes. *International Journal of Parallel Programming 27*, 1, 21–33.

KLINT, P. 1981. Interpretation techniques. *Software—Practice and Experience 11*, 963–973.

KRALL, A. AND GRAFL, R. 1997. Cacao—a 64-bit JavaVM just-in-time compiler. *Concurrency—Practice and Experience 9*, 11, 1017–1030.

LENGAUER, T. AND TARJAN, R. E. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1*, 1, 121–141.

MCGLASHAN, B. AND BOWER, A. 1999. The interpreter is dead (slow). Isn't it? In *OOPSLA'99 Workshop: Simplicity, Performance and Portability in Virtual Machine Design.*

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

MUCHNICK, S. S. 1997. *Advanced compiler design and implementation*. Morgan Kaufmann, San Francisco, CA.

MYERS, G. J. 1977. The case against stack-oriented instruction sets. *Computer Architecture News 6*, 3 (Aug.), 7–10.

PATTERSON, D. AND DITZEL, D. 1980. The case for the reduced instruction set computer. *Computer Architecture News 8*, 6 (Oct.), 25–33.

PIUMARTA, I. AND RICCARDI, F. 1998. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*. 291–300.

PUGH, W. 1999. Compressing Java class files. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. ACM Press, New York. 247–258.

RADHAKRISHNAN, R., VIJAYKRISHNAN, N., JOHN, L. K., AND SIVASUBRAMANIAM, A. 2000. Architectural issues in Java runtime systems. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture* (8–12 Jan.). Toulouse, France. 387–398.

SCHULTHESS, P. AND MUMPRECHT, E. 1977. Reply to the case against stack-oriented instruction sets. *Computer Architecture News 6*, 5 (Dec.), 24–27.

SHI, Y., GREGG, D., BEATTY, A., AND ERTL, M. A. 2005. Virtual machine showdown: stack versus registers. In *ACM/SIGPLAN Conference on Virtual Execution Environments*. ACM Press, New York. 153–163.

SHUSTEK, L. 1978. Aanalysis and performance of computer instruction sets. Ph.D. thesis, Stanford University.

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

SUGALSKI, D. 2002. Parrot in detail. In *Yet Another Perl Conference (YAPC 02)*. Saint Louis, Missouri. http://www.parrotcode.org/talks/ParrotInDetail2.pdf.

SUN-MICROSYSTEMS. 2001. The Java Hotspot virtual machine. Tech. rep., Sun Microsystems Inc.

TIP, F., SWEENEY, P. F., LAFFRA, C., EISMA, A., AND STREETER, D. 2002. Practical extraction techniques for Java. *ACM Trans. Program. Lang. Syst. 24*, 6, 625–666.

VALLÉE-RAI, R., HENDREN, L., SUNDARESAN, V., LAM, P., GAGNON, E., AND CO, P. 1999. Soot—a Java optimization framework. In *Proceedings of CASCON 1999*. 125–135.

VITALE, B. AND ABDELRAHMAN, T. S. 2004. Catenation and specialization for Tcl virtual machine performance. In *IVME '04: Proceedings of the 2004 workshop on Interpreters, Virtual Machines and Emulators*. ACM Press, New York. 42–50.

WIECEK, C. 1982. A case study of the VAX 11 instruction set usage for compiler execution. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operation Systems*. IEEE/ACM, Palo Alto, California. 177–184.

WINTERBOTTOM, P. AND PIKE, R. 1997. The design of the Inferno virtual machine. In *IEEE Compcon 97 Proceedings*. San Jose, California. 241–244.

Received August 2006; revised February 2007; accepted April 2007