Virtual Machine Showdown: Stack Versus Registers

Yunhe Shi, David Gregg, Andrew Beatty
Department of Computer Science
University of Dublin, Trinity College
Dublin 2, Ireland
{yshi, David.Gregg,
Andrew.Beatty}@cs.tcd.ie

M. Anton Ertl
Institut für Computersprachen
TU Wien
Argentinierstraße 8
A-1040 Wien, Austria
anton@complang.tuwien.ac.at

ABSTRACT

Virtual machines (VMs) are commonly used to distribute programs in an architecture-neutral format, which can easily be interpreted or compiled. A long-running question in the design of VMs is whether 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 two ways. Firstly, our translation from stack to register code is much more sophisticated. The result is that we eliminate an average of more than 47% of executed VM instructions, with the register machine bytecode size only 25% larger than that of the corresponding stack bytecode. Secondly we present an implementation of a register machine in a fully standard-compliant implementation of the Java VM. We find that, on the Pentium 4, the register architecture requires an average of 32.3% less time to execute standard benchmarks if dispatch is performed using a C switch statement. Even if more efficient threaded dispatch is available (which requires labels as first class values), the reduction in running time is still approximately 26.5% for the register architecture.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Language; D.3.4 [**Programming Language**]: Processor—*Interpreter*

General Terms

Performance, Language

Keywords

Interpreter, Virtual Machine, Register Architecture, Stack Architecture

1. MOTIVATION

Virtual machines (VMs) are commonly used to distribute programs in an architecture-neutral format, which can easily be interpreted or compiled. The most popular VMs, such as the Java VM, use a virtual stack architecture, rather than the register architecture that dominates in real processors.

A long-running question in the design of VMs is whether stack architecture or register architecture can be implemented more efficiently with an interpreter. On the one hand stack architectures allow smaller VM code so less code must be fetched per VM instruction executed. On the other hand, stack machines require more VM instructions for a given computation, each of which requires an expensive (usually unpredictable) indirect branch for VM instruction dispatch. Several authors have discussed the issue [12, 15, 11, 16] 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. [5, 10] who translated Java VM stack code to a corresponding register machine code. A straightforward translation strategy was used, with simple compiler optimizations to eliminate instructions which become unnecessary in register format. The resulting register code required around 35% fewer executed VM instructions to perform the same computation than the stack architecture. 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 using an interpreter. However, Davis et al's work did not include an implementation of the virtual register architecture, so no real running times could be presented.

This paper extends the work of Davis et al. in two respects. First, our translation from stack to register code is 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 constant load instructions, to eliminate common constant loads and move constant loads out of loops. The result is that an average of more than 47% of executed VM instructions are eliminated. The resulting register VM code is roughly 25% larger than the original stack code, compared with 45% for Davis et al. We find that the increased cost of fetching more VM code involves only 1.07 extra real machine loads per VM instruction eliminated. Given that VM dispatches are much more expensive than real machine loads, this indicates strongly that register VM code is likely to be much

VEE'05, June 11-12, 2005, Chicago, Illinois, USA.

more time-efficient when implemented with an interpreter, although at the cost of increased VM code size.

The second contribution of our work is an implementation of a register machine in a fully standard-compliant implementation of the Java VM. While implementing the register VM interpreter is simple, integrating it with the garbage collection, exception handling and threading systems is more complicated. We present experimental results on the behaviour of the stack and register versions of JVMs, including hardware performance counter results. We find that on the Pentium 4, the register architecture requires an average of 32.3% less time to execute standard benchmarks if dispatch is performed using a C switch statement. Even if more efficient threaded dispatch is available (which requires labels as first class values), the reduction in running time is still about 26.5% for the register architecture.

The rest of this paper is organised as follows. In section 2 we describe the main differences between virtual stack and virtual register machines from the point of view of the interpreter. In section 3, we show how stack-based Java bytecode is translated into register-based bytecode. In sections 4 and 5, our copy propagation and constant instruction optimization algorithms are presented. Finally, in section 6, we analyze the static and dynamic code behaviour before and after optimization, and we show the performance improvement in our register-based JVM when compared to original stack-based JVM.

2. STACK VERSUS REGISTERS

The cost of executing a VM instruction in an interpreter consists of three components:

- Dispatching the instruction
- Accessing the operands
- Performing the computation

Instruction dispatch involves fetching the next VM instruction from memory, and jumping to the corresponding segment of interpreter code that implements the VM 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.

In C, dispatch is typically implemented 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 addition, the indirect branch generated by most compilers is highly (around 95% [7]) unpredictable on current architectures.

The main alternative to the `switch` statement is *threaded dispatch*. Threaded dispatch takes advantage of languages with labels as first class values (such as GNU C and assembly language) to optimize the dispatch process. This allows the range check and additional unconditional branches to be eliminated, and allows the code to be restructured to improve the predictability of the dispatch indirect branch (to around 45% [7]).

More sophisticated approaches, such as Piumarta and Ricardi's [14] approach of copying executable code just-in-time further reduce dispatch costs, at a further cost in simplicity, portability and memory consumption. Context threading [2] uses subroutine threading to change indirect branch to call/return, which better exploits hardware's return-address stack, to reduce the cost of dispatches. As the cost of dispatches falls, any benefit from using a register VM instead of a stack VM falls. However, `switch` and simple threaded dispatch are the most commonly used interpreter techniques, and `switch` is the only efficient alternative if ANSI C must be used.

The second cost component of executing a VM instruction is 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 number of memory fetches are the main reasons why stack architectures are so popular for VMs.

The final cost component of executing a VM instruction is performing the computation. Given that most VM instructions perform a simple computation, such as an add or load, this is usually the smallest part of the cost. The basic computation has to be performed, regardless of the format of the intermediate representation. However, eliminating invariant and common expressions is much easier on a register machine, which we exploit to eliminate repeated loads of identical constants (see section 5).

3. TRANSLATING STACK TO REGISTER

In this section we describe a system of translating JVM stack code to register code just-in-time. However, it is important to note that we do not advocate run-time translation from stack to register format as the best or only way to use virtual register machines. This is clearly a possibility, maybe even an attractive one, but our main intention in doing this work is to evaluate free-standing virtual register machines. Run-time translation is simply a mechanism we use to compare stack and register versions of the JVM easily. In a real system, we would use only the register machine, and compile for that directly.

Our implementation of the JVM pushes a new Java frame onto a run-time stack for each method call. The Java frame contains local variables, frame data, and the operand stack for the method(See figure 1). In the stack-based JVM, a local variable is accessed using an index, and the operand stack is accessed via the stack pointer. In the register-based 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 [9].

All values on the operand stack in a Java frame can be considered as temporary variables (registers) for a method and therefore are short-lived. Their scope of life is between the instructions that push them onto the operand stack and the instruction that consumes the value on the operand stack. On the other hand, local variables (also registers) are long-lived and their life scope is the time of method execution.

In the stack-based JVM, most operands of an instruction

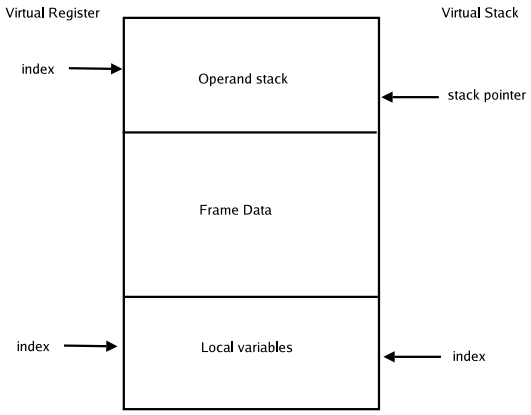

Figure 1: The structure of a Java frame

are implicit; they are found on the top of the operand stack. Most of the stack-based JVM instructions are translated into corresponding register-based virtual machine instructions, with implicit operands translated to explicit operand registers. The new register-based instructions use one byte for the opcode and one byte for each operand register (similar to the stack JVM).

Table 1 shows a simple example of bytecode translation. The function of the bytecode is to add two integers from two local variables and store the result back into another local variable.

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

- Operand stack pop instructions(*pop* and *pop2*) are translated into *nop* because they are not needed in register-based code.
- Instructions related to loading of a local variable onto operand stack and storing data from operand stack into a local variable are translated into *move* instructions
- Stack manipulation instructions(e.g. *dup*, *dup2* ...) are translated into appropriate sequences of *move* instructions by tracking the state of the operand stack

3.1 Parameter Passing

A common way to implement stack-based 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. The parameters on the stack in the current Java frame will become the start of the called method's local variables. Although this provides efficient parameter passing, it prevents us from copy propagating into the source registers (parameters) of a method call. To solve this problem, we change the parameter passing mechanism in the register VM to non-overlapping and copy all the parameters to the location where the new Java frame will start. The benefit is that we can eliminate more *move* instructions. The drawback is that we need to copy all the parameters before we push a new Java frame onto the Java stack.

3.2 Variable Length Instructions

Most of the instructions in Java bytecode are fixed-length. There are three variable-length instructions in stack-based JVM instruction set (*multianewarray*, *tableswitch*, and *lookupswitch*). In addition to the original three variable-length instructions, all method call instructions become variable in length after the translation to register-based bytecode format. Here is the instruction format for method call:

Table 1: Bytecode translation. Assumption: current stack pointer before the code shown below is 10. In most cases, the first operand in an instruction is the destination register

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

```
op cpi1 cpi2 ret_reg arg1 arg2 ...
```

op is the opcode of a method call. *cpi1* and *cpi2* are the two-byte constant-pool indexes. *ret_reg* is the return value register number. *arg1*, *arg2*, ... are the argument register numbers. The number of arguments, which can be determined when the method call instructions are executed in the interpreter loop, are not part of the instruction format. The main reason for doing so is to reduce the codesize.

4. COPY PROPAGATION

In the stack-based JVM, operands are pushed from local variables onto the operand stack before they can be used, and results must be stored from the stack to local variables. More than 40% of executed instructions in common Java benchmarks consist of loads and stores between local variables and the stack [5]. Most of these stack push and pop operations are redundant in our register-based JVM as instructions can directly use local variables (registers) without going through the stack. In the translation stage, all loads and stores from/to local variables are translated into register *move* instructions. In order to remove these redundant *move* instructions, we apply both forward and backward copy propagation.

We take advantage of the stack-based JVM's stack operation semantics to help implement both varieties of copy propagation. During copy propagation, we use the stack pointer information after each instruction, which tells us which values on the stack are still alive.

4.1 Forward Copy Propagation

The convention in Java is that the operand stack is usually empty at the end of each source statement, so the lifetimes of values on the stack are usually short. Values pushed onto the operand stack are almost immediately consumed by a following instruction. Thus, we mainly focus on copy propagation optimization on basic blocks.

We separate *move* instructions into different categories and apply different types of copy propagation depending on the location of the source and destination operands in the original JVM stack code. We apply forward propagation to the following categories of *move* instructions:

- Local variables → stack
- Local variables → local variables (these do not exist in the original translated code but will appear after forward or backward copy propagation)

Table 2: Forward copy propagation algorithm. X is a *move* instruction being copy propagated and Y is a following instruction in the same basic block. *src* and *dest* are source and destination registers of these instructions

Y	X	
	src	dest
src	X.src = Y.src Do nothing	X.dest = Y.src Replace Y.src with X.src
dest	X.src = Y.dest X.src redefined after Y Can't remove X / stop	X.dest = Y.dest X.dest redefined after Y Can remove X / stop

- Stack → stack (these originate from the translation of `dup` instrutions)

The main benefit of forward copy propagation is to collapse dependencies on *move* operations. In most cases, this allows the *move* to be eliminated as dead code.

While doing forward copy propagation, we try to copy forward and identify whether a *move* instruction can be removed (See Table 2). X is a *move* instruction which is being copied forward and Y is a following instruction in the same basic block. X.dest is the destination register of the *move* instruction and X.src is the source register of the *move* instruction. In a following Y instruction, Y.src represents all the source registers and Y.dest is the destination register.

The forward copy propagation algorithm is implemented with additional operand stack pointer information after each instruction to help to decide whether a register is alive or redefined. The following outlines our algorithm for copy propagation:

- Y.dest = X.dest. X.dest is redefined, stop copy propagation and remove instruction X.
- After instruction Y, stack pointer is below X.dest if X.dest is a register on stack. X.dest can be considered to be redefined, stop copy propagation, and remove instruction X.
- If Y is a return instruction, stop copy propagation and remove instruction X.
- If Y is an *athrow* and X.dest is on operand stack, stop copy propagation and remove instruction X because the operand stack will be cleared during exception handling.
- Y.dest = X.src. X.src is redefined and value in X.dest would still be be used after Y instruction. Stop copy propagation and don't remove instruction X. However, We can continue to find out whether X.dest is not used in the following instructions and then is redefined. If so, remove instruction X.
- After instruction Y, stack pointer is below X.src if X.src is a register on stack. X.src can be considered as being redefined, stop copy propagation and don't remove instruction X. We ignore this rules for the second run of forward copy propagation; it is quite similar to above rule.

Several techniques are used to improve the ability of our algorithm to eliminate *move* insructions.

- All dup (such as `dup`, `dup2_x2`) instructions are translated into one or more *move* instructions which allows them to be eliminated using our algorithm.
- All *iinc* instructions are moved as far towards the end of a basic block as possible because *iinc* instructions are commonly used to increment an index into an array. The push of the index onto the stack and *iinc* instruction used to increase the index are usually next to each other and thus prevent us from forward copy propagation.
- In a few special cases, forward copy propagation across basic block boundaries is used to eliminate more *move* instructions. If a *move* instruction's forward copy propagation reaches the end of a basic block and its destination operand is on the stack, we can follow its successor basic blocks to find all the usages of the operand and then trace back from the operand consumption instruction to the definition instruction. If we don't find any other instructions except the one instruction being copy propagated forward, then we can continue the cross basic block copy propagation.

4.2 Backward Copy Propagation

Backward copy propagation is used to backward copy and eliminate the following types of *move* instructions:

- Stack → local variables

Most stack JVM instructions put their result on the stack and a *stores* instruction stores the result into a local variable. The role of backward copy propagation is to store the result directly into the local variable without going through the operand stack. In reality, we can't copy forward this type of *move* instruction because after the instruction the source register is above the top of the stack pointer. Due to the characteristics of this type of *move* instruction, a lot of criteria required by backward copy propagation are already satisfied. Suppose Z is a *move* instruction being considered for backward copy propagation. Y is a previous instruction in the same basic block which has Y.dest = Z.src. Whether we can do the backward copy propagation and remove instruction Z depends on the following criteria:

1. Y.dest is a register
2. Z is a *move* instruction
3. Z.dest is a register
4. Z.src = Y.dest
5. Z.dest is not consumed between Y..Z
6. Z.dest is not redefined between Y..Z
7. Y.dest is not alive out of the basic block, which is satisfied because Y.dest = Z.src and Z.src is above the top of stack pointer after Z
8. After the copy propagation, original Y.dest(Z.src) is not used anymore. It is satisfied as long as 5 and 6 are satisfied because Y.dest(Z.src) is above the top of stack pointer after the instruction Z.

Another way to think of the backward copy propagation in our case is that some computation puts the result on the operand stack and then a *move* instruction stores the result from the stack to a local variable in the stack-based Java virtual machine. In a register-based Java virtual machine, we can shortcut the steps and save the result directly into a local variable.

A simple version of across basic-block backward copy propagation is also used. If a backward copy instruction reaches the beginning of a basic block, we need to find out whether we can backward copy to all its predecessors. If so, we backward copy to all its predecessors.

4.3 Example

The following example demonstrates both forward and backward copy propagation. We assume that the first operand register in each instruction is the destination register.

```
1. move r10, r1            //iload_1
2. move r11, r2            //iload_2
3. iadd r10, r10, r11      //iadd
4. move r3, r10            //istore_3
```

Instructions 1 and 2 move the values of registers r1 and r2 (local variables) to registers r10 and r11 (stack) respectively. Instruction 3 adds the values in register r10 and r11 (stack) and put the result back into register r10 (stack). Instruction 4 moves the register r10 (stack) into register r3 (local variable). This is typical of stack-based Java virtual machine code. We can apply forward copy propagation to instructions 1 and 2 and their source are copy propagated into instruction 3's sources. We can apply backward copy propagation to instruction 4 and backward copy progagate into instruction 3's destination which is replaced by instruction 4's destination. After both copy propagations, instructions 1, 2, and 4 can be removed. The only remaining instruction is:

```
3. iadd r3, r1, r2
```

5. CONSTANT INSTRUCTIONS

In stack-based Java virtual machine, there are a large number of constant instructions pushing immediate constant values or constant values from constant pool of a class onto the operand stack. For example, we have found that an average of more than 6% of executed instructions in the SPECjvm98 and Java Grande benchmarks push constants onto the stack. In many cases the same constant is pushed onto the stack every iteration of a loop. Unfortunately, it is difficult to reuse constants in a stack VM, because VM instructions which take values from the stack also destroy those values. Virtual register machines have no such problems. Once a value is loaded to a register, it can be used repeatedly until the end of the method. To remove redundant loads of constant values, we apply the following optimizations.

5.1 Combine Constant Instruction and iinc Instruction

In the stack-based JVM, the *iinc* instruction can only be used to increase a local variable by an immediate value. However, in the register machine we make no distinction between stack and local variables, so we can use the *iinc* instruction with all registers. This allows us to combine sequences of instructions which add a small integer to an value on the stack.

We scan the translated register-based instructions to find all those *iadd* and *isub* instructions which has one of its operands pushed by a constant instruction with a byte constant value , due to the byte immediate value in *iinc* instruction. Then we use an *iinc* instruction to replace an *iadd*(or *isub*) instruction and a constant instruction.

5.2 Move Constant Instructions out of Loop and Eliminate Duplicate Constant Instruction

Because the storage locations of most constant instructions are on the stack, they are temporary variables, and are quickly consumed by a following instruction. The only way that we can reuse the constant value is to allocate a dedicated register for the same constant value above the operand stack. We only optimize those constant instructions that store a constant values onto the stack locations and those constant values are consumed in the same basic block. The constant instructions that store value into local variables, which have wider scope, are not targeted by our optimization. A constant instruction which stores directly into a local variable can appear after backward copy propagation. The following steps are carried out to optimize constant instructions:

- Scan all basic blocks in a method to find (1) multiple constant VM instructions which push the same constant and (2) constant VM instructions that are inside a loop. All constant values pushed by these VM instructions onto the operand stack must be consumed by a following instruction in the same basic block for our optimization to be applied.

- A dedicated virtual register is allocated for each constant value used in the method[1]. The constant VM instruction's destination virtual register will be updated to the new dedicated virtual register, as will the VM instruction(s) that consume the constant.

- All load constant VM instructions are moved to the beginning of the basic block. All load constant VM instructions inside a loop are moved to a loop preheader.

- The immediate dominator tree is used to eliminate redundant initializations of dedicated constant registers.

The above procedure produces two benefits. First, redundant loads of the same constant are eliminated. If there are more than two constant instructions that try to initialize the same dedicated constant registers in the same basic block or in two basic blocks in which one dominates the other, the duplicate dedicated constant register initialization instruction can be removed. The other benefit is to allow us to move the constant instructions out of loops.

[1]Given that we use one byte indices to specify the virtual register, a method can use up to 256 virtual registers. Thus, our current implementation does not attempt to minimize register usage, because we have far more registers than we need. A simple register allocator could greatly reduce register requirements.

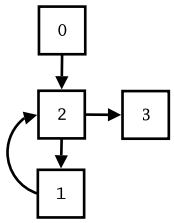

Figure 2: The control flow of the medium-size example

5.3 Putting it all together

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

- Find basic blocks, their predecessors and successors
- Translate stack-based bytecode into intermediate register-based bytecode representation.
- Find loops and build the dominator matrix.
- Apply forward copy propagation.
- Apply backward copy propagation.
- Combine constant instruction and *iadd*/*isub* instructions into *iinc* instructions.
- Move *iinc* instructions as far down their basic block as possible.
- Eliminate redundant constant load operations and move constant load operations from loops.
- Apply forward copy propagation again[2].
- Write the optimized register code into virtual register bytecode in memory.

In order to better demonstrate the effect of the optimizations, we present the following more complicated example with 4 basic blocks and one loop(See Figure 2). The number operands without r are either constant-pool indexes, immediate values, or branch offsets(absolute basic block number is used here instead to clearly indicate which basic block is the target of a jump):

The translated intermediate code with all operands explicitly specified before optimizations:

```
basic block(0):
 1. iconst_0 r17
 2. move r1, r17
 3. move r17, r0
 4. getfield_quick 3, 0, r17, r17
 5. move r2, r17
 6. move r17, r0
 7. agetfield_quick 2, 0, r17, r17
 8. move r3, r17
 9. move r17, r0
10. getfield_quick 4, 0, r17, r17
11. move r4, r17
12. iconst_0 r17
13. move r5, r17
14. goto 0, 2          //jump to basic block 2

basic block(1)
15. bipush r17, 31
16. move r18, r1
17. imul r17, r17, r18
18. move r18, r3
19. move r19, r2
20. iinc r2, r2, r1
21. caload r18, r18, r19
22. iadd r17, r17, r18,
23. move r1, r17
24. iinc r5, r5, 1,

basic block(2):
25. move r17, r5
26. move r18, r4
27. if_icmplt 0, 1, r17, r18
    // jump to basic block 1

basic block(3):
28. move r17, r1
29. ireturn r17
```

The intermediate code after optimizations:

```
basic block(0):
15. bipush r20, 31     //constant moved out of loop
 1. iconst_0 r1
 4. getfield_quick 3, 0, r2, r0
 7. agetfield_quick 2, 0, r3, r0
10. getfield_quick 4, 0, r4, r0
12. iconst_0 r5
14. goto 0, 2

basic block(1):
17. imul r17, r20, r1
21. caload operand: r18, r3, r2
22. iadd operand: r1, r17, r18
24. iinc operand: r5, r5, r1
20. iinc operand: r2, r2, r1

basic block(2):
27. if_icmplt operand: 0, 1, r5, r4

basic block(3):
29. ireturn r1
```

All the *move* instructions have been eliminated after the optimizations. Constant instruction 15 has been assigned a new dedicated register number 20 to store the constant value and has been moved out of loop to its preheader, which is then combined with its predecessors because it has only one predecessor. Instruction 20 has been moved down inside its basic block to provide more opportunities for copy propagation.

[2]We have found that we can eliminate a small percentage of *move* instruction by applying the forward copy propagation algorithm a second time. *dup* instructions generally shuffle the stack operands around the stack and redefine the values in those registers. This will stop the copy propagation. After first forward copy propagation and backward copy propagation, new opportunities for forward copy propation are created.

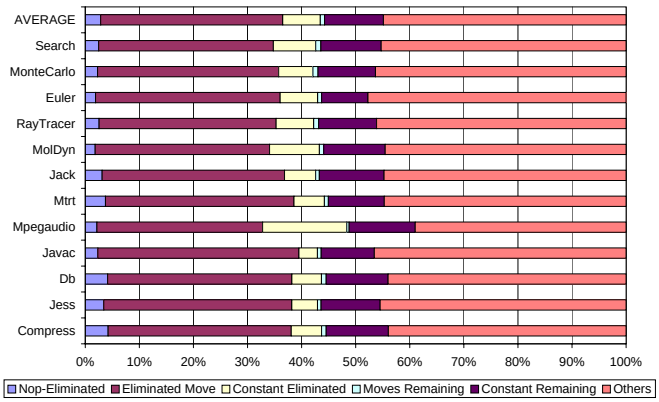

Figure 3: Breakdown of statically appearing VM instructions before and after optimization for all the benchmarks.

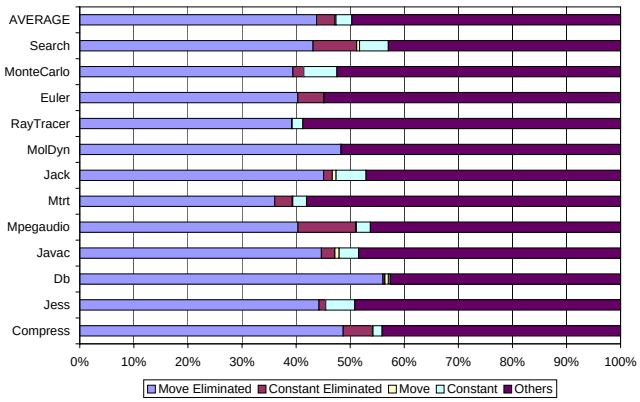

Figure 4: Breakdown of dynamically appearing VM instructions before and after optimization for all the benchmarks.

6. EXPERIMENTAL EVALUATION

Stack-based JVM bytecode is very compact because the location of operands is implicitly on the operand stack. Register-based bytecode needs to have all those implicit operands as part of an instruction. That means that the register-based code size will usually be much larger than stack-based bytecode. Larger code size means that more instruction bytecode must be fetched from memory as part of VM instruction execution, slowing down the register machine. On the other hand, virtual register machines can express computations using fewer VM instructions than a corresponding stack machine. Dispatching a VM instruction is expensive, so the reduction in executed VM instructions is likely to significantly improve the speed of the virtual register machine.

An important question is whether the increase in VM instruction fetches or the decrease in dispatches from using a virtual register machine has a greater effect on execution time. In this section we describe the experimental evaluation of two interpreter-based JVMs. The first is a conventional stack-based JVM (Sun's J2ME CDC 1.0 - foundation profile), and the second is a modified version of this JVM which translates stack code into register code just-in-time, and implements an interpreter for a virtual register machine.

We use the SPECjvm98 client benchmarks[1] (size 100 inputs) and Java Grande[3] (Section 3, data set size A) to benchmark both implementations of the JVM. 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 benchmark program code and Java library code executed by the VMs.

6.1 Static Instruction Analysis after Optimization

Generally, there is a one-to-one correspondence between stack VM instructions and the translated register VM instructions. However there are a couple of exceptions. First, the JVM includes some very complicated instructions for duplicating data on the stack, which we translate into a sequence of *move* VM instructions. The result of this transformation is that the number of static instructions in the translated code is about 0.35% larger than in the stack code. Secondly, some JVM stack manipulation instructions (such as `pop` and `pop2`) can simply be translated to `nop` instructions, and can be eliminated directly.

Figure 3 shows the breakdown of statically appearing VM instructions after translation and optimization. On average we can simply eliminate 2.84% of nop instructions (translated from *pop* and *pop2*) because they manipulate the stack but perform no computation. Applying copy propagation allows a further 33.67% of statically appearing instructions to be eliminated. Our copy propagation algorithm is so successful that the remaining *moves* instructs account for only an average 0.78% of the original instructions. Almost all *moves* are eliminated. Constant optimization allows a further average of 6.95% of statically appearing VM instructions to be eliminated. The remaining load constant VM instructions account for an average of 10.89% of the original VM instructions. However, these figures are for statically appearing code, so moving a constant load out of a loop to a loop preheader does not result in any static reduction. Overall, 43.47% of static VM instructions are eliminated.

6.2 Dynamic Instruction Analysis after Optimization

In order to study the dynamic (runtime) behaviour of our register-based JVM code, we counted the number of executed VM instructions run without any optimization as the starting point of our analysis. However, the stack VM instructions that translate to `nop` instructions have already been eliminated at this point and are not included in the analysis.

Figure 4 shows the breakdown of dynamically executed VM instructions before and after optimization. Interstingly *move* instructions account for a much greater percentage of executed VM instructions than static ones. This allows our copy propagation to eliminate *move* VM instructions accounting for 43.78% of dynamically executed VM instructions. The remaining *move*s account for only 0.53% of the original VM instructions. Applying constant optimizations allows a further reduction of 3.33% of original VM instructions to be eliminated. The remaining dynamically executed constant VM instructions account for 2.98%. However, there are far more static constant instructions(17.84%) than those dynamically run(6.26%) in the benchmarks. We discovered

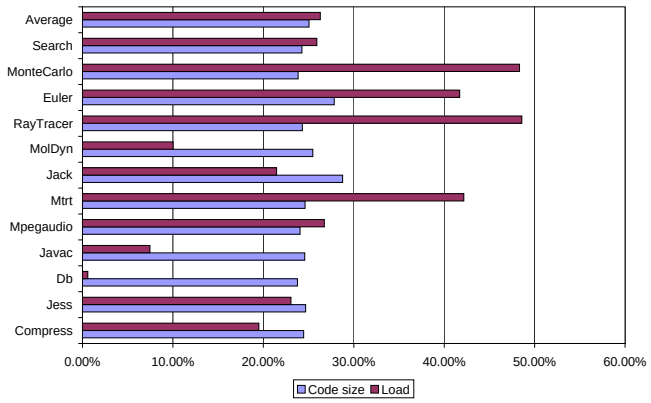

Figure 5: Increase in code size and resulting net increase in bytecode loads from using a register rather than stack architecture.

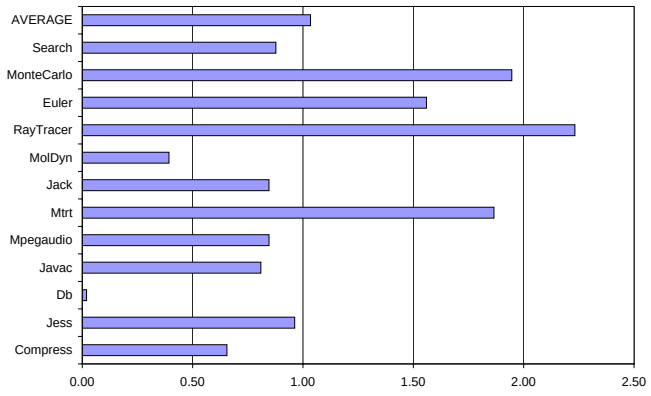

Figure 6: Increase in dynamically loaded bytecode instructions per VM instruction dispatch eliminated by using a register rather than stack architecture.

that there are a large number of constant instructions in the initialization bytecode which are usually executed only once. On average, our optimizations remove 47.21% of the dynamically executed original VM instructions.

6.3 Code Size

Generally speaking, the code size of register-based VM instructions is larger than that of the corresponding stack VM instructions. Figure 5 shows the percentage increase in code size of our register machine code compared to the original stack code. On average, the register code is 25.05% larger than the original stack code, despite the fact that the register machine requires 43% fewer static instructions than the stack architecture. This is a significant increase in code size, but it is far lower than the 45% increase reported by Davis et al. [5].

As a result of the increased code size of the register-based JVM, more VM instruction bytecodes must be fetched from memory as the program is interpreted. Figure 5 also shows the resulting increase in bytecode load. Interestingly, the increase in overall code size is often very different from the increase in instruction bytecode loaded in the parts of the program that are executed most frequently. Nonetheless, the

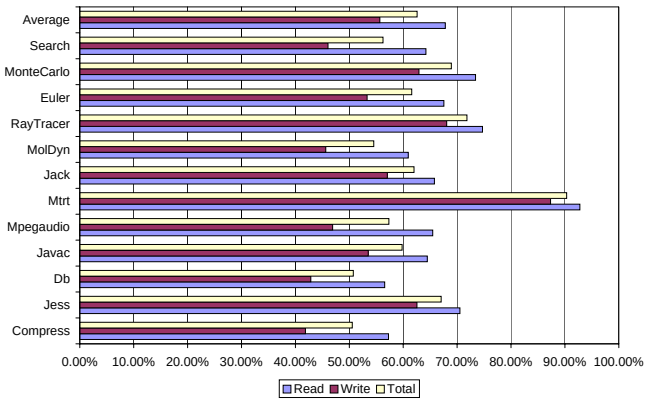

Figure 7: Dynamic number of real machine load and store required to access virtual registers in our virtual register machine as a percentage of the corresponding loads and stores to access the stack and local variables in a virtual stack machine

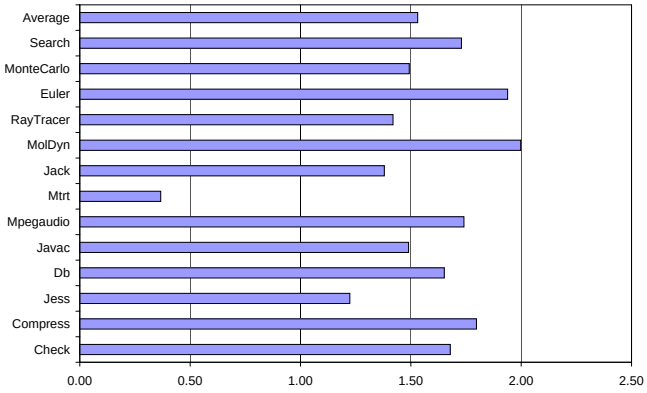

Figure 8: The reduction of real machine memory accesses for each register-based bytecode instruction eliminated

average increase in loads is similar to the average increase in code size, which is at 26.03%.

The performance advantage of using a register rather than stack VM is that fewer VM instructions are needed. On the other hand, this comes at the cost of increased bytecode loads due to larger code. To measure the relative importance of these two factors, we compared the number of extra dynamic bytecode loads required by the register machine, per dynamically executed VM instruction eliminated. Figure 6 shows that the number of additional bytecode loads per VM instruction eliminated is small at an average of 1.07%. On most architectures one load costs much less to execute than an instruction dispatch, with its difficult-to-predict indirect branch. This strongly suggests that register machines can be interpreted more efficiently on most modern architectures.

6.4 Dynamic Local Memory Access

Apart from real machine loads of instruction bytecodes, the main source of real machine 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 real machine load to read the value from one array, and a real machine store to write the value to the other array. Thus, adding a simple operation such as adding two numbers can involve large numbers of real machine 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 stack machine can actually require fewer real machine loads and stores to perform the same computation. Figure 7 shows (a simulated measure) the number of the dynamic real machine loads and stores required for accessing the virtual register array, as a percentage of the corresponding loads and stores for the stack JVM to access the local variable and operand stack arrays. The virtual register machine requires only 67.8% as many real machine loads and 55.07% as many real machine writes, with an overall figure of 62.58%.

In order to compare these numbers with the number of additional loads required for fetching instruction bytecodes, we expressed these memory operations as a ratio to the dynamically executed VM instructions eliminated by using the virtual register machine. Figure 8 shows that on average, the register VM requires 1.53 fewer real machine memory operations to access such variables. This is actually larger than the number of additional loads required due to the larger size of virtual register code.

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 real machine registers like an array on most real architectures. However, stack caching [6] can be used to keep the topmost stack values in registers, and eliminate large numbers of associated real machine loads and stores. For example, Ertl [6] found that around 50% of stack access real machine memory operations could be eliminated by keeping just the topmost stack item in a register. Thus, in many implementations, the virtual register architecture is likely to need more real machine loads and stores to access these kinds of values.

6.5 Timing Results

To measure the real running times of the stack and register-based implementations of the JVM, we ran both VMs on Pentium 3 and Pentium 4 systems. The stack-based JVM simply interprets standard JVM bytecode. The running time for the register-based JVM includes the time necessary to translate and optimize each method the first time it is executed. However, our translation routines are fast, and consume less than 1% of the execution time, so 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 four different interpreter implementations: (1) a stack-based JVM interpreter using switch dispatch (see section 2), (2) a stack-based JVM interpreter using threaded dispatch, (3) a virtual register-based JVM using switch dispatch and (4) a virtual register-based JVM using threaded dispatch. For fairness, we always compare implementations which use the same dispatch mechanism.

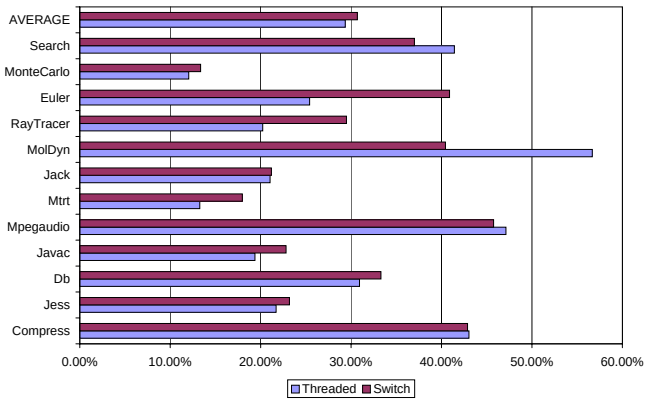

Figure 9: Register-based virtual machine reduction in running time (based on average real running time of five runs): switch and threaded (Pentium 3)

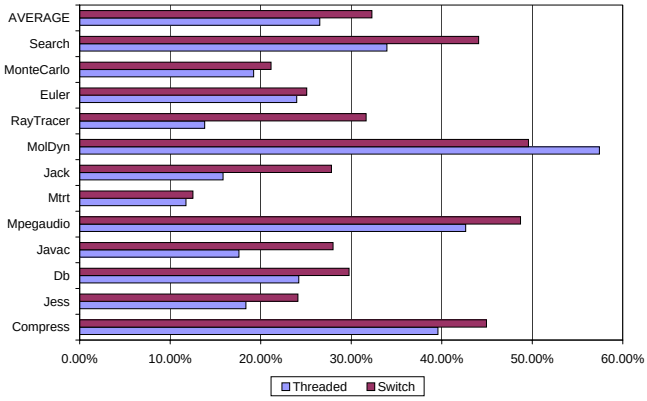

Figure 10: Register-based virtual machine performance improvement in terms of performance counter running time: switch and threaded (Pentium 4: performance counter)

Figure 9 shows the percentage reduction in running time of our implementation of the virtual register machine compared to the virtual stack machine for variations of both interpreters using both switch and threaded dispatch. Switch dispatch is more expensive, so the reduction in running time is slightly larger (30.69%) than the threaded versions of the interpreters (29.36%). Nonetheless, a reduction in the running time of around 30% for both variants of the interpreters is a very significant improvement. There are few interpreter optimizations that give a 30% reduction in running time.

Figure 10 shows the same figures for a Pentium 4 machine. The Pentium 4 has a very deep pipeline (20 stages) so the cost of branch mispredictions is very much higher than that of the Pentium 3. The result is that switch dispatch is very slow on the Pentium 4 due to the large number of indirect branch mispredictions it causes. On average, the switch-dispatch register machine requires 32.28% less exe-

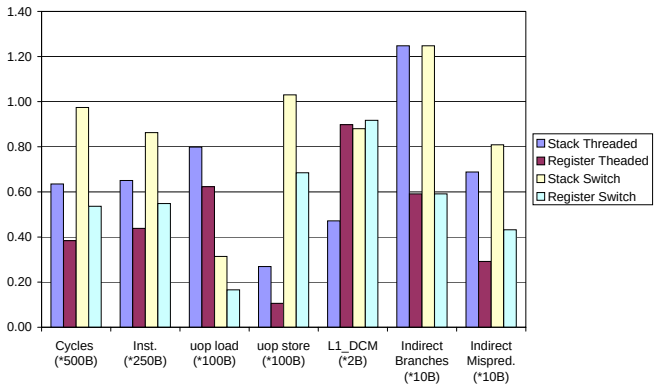

Figure 11: Compress (Pentium 4 performance counter results)

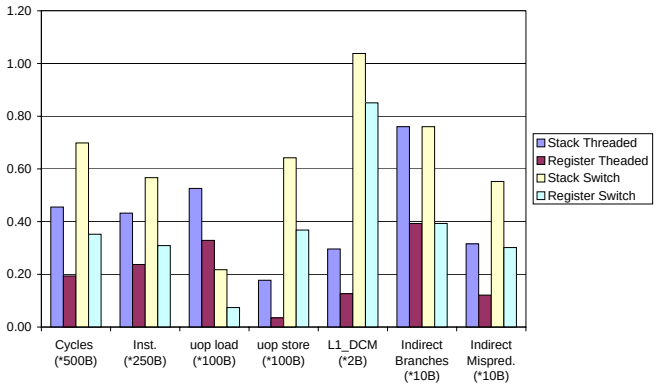

Figure 12: Moldyn (Pentium 4 performance counter results)

cution time than the switch-dispatch stack machine. The corresponding figure for the threaded-dispatch JVMs is only 26.53%.

To more deeply explore the reasons for the relative performance, we use the Pentium 4's hardware performance counters to measure various processor events during the execution of the programs. Figures 11 and 12 show performance counter results for the SPECjvm98 benchmark *compress* and Java Grande benchmark *moldyn*. We measure the number of cycles of execution time, number of retired Pentium 4 instructions, numbers of retired Pentium 4 load and store micro-operations, number of level 1 data cache misses, number of indirect branches retired and number of retired indirect branches mispredicted.

Figures 11 and 12 show that threaded dispatch is much more efficient than switch dispatch. The interpreters that use switch dispatch require far more cycles and executed instructions, and the indirect branch misprediction rate is significantly higher.

When we compare the stack and register versions of our JVM, we see that the register JVM is much more efficient in several respects. It requires significantly less executed instructions than the stack-based JVM. More significantly, for the *compress* benchmark, it requires less than half the number of indirect branches. Given the large rate of indirect branch misprediction, and the high cost of indirect branches, it is not surprising that the virtual register implementation of the JVM is faster.

Figures 11 and 12 show that *uop load* for threaded VM is much higher than switch VM. The most likely reasons for such case is that it is more difficult for compiler to optimize the registers allocation for threaded interpreter loop than switch-based one. It is easier for compiler to recognize the switch-based interpreter loop and different segment of switch statement. On the other hand, the threaded interpreter loop consists lots of code segments with labels and the execution of bytecode jumps around those labels. Obviously, it is much easier for compiler to optimize register allocation in switch-based interpreter loop than a threaded one.

6.6 Discussion

Our implementation of the register-based VM translates the stack-based bytecode into register-based bytecode at the runtime. We don't propose to do so in the real-life implementation. The purpose of our implementation is to evaluate the register-based VM. Our register-based JVM implementation came from the stack-based JVM implementation. Except for the necessary adaption of interpreter loop, garbage-collection and exception handling to the new instruction format, there is very little change to its original code segments to interpret bytecode instructions. The objective of doing so is to provide a fair comparison between the stack-based and the register-based JVM.

Another technique to eliminate redundant stack-load and -store *move* instructions would be to use register coalescing. However, the technique is less efficient and more complex than our simple copy propagation algorithm because it involves repeatedly doing data flow analysis and building interference graph. Moreover, our copy propagation is so effective that only less than 2% of *move* instructions are remaining in the static code while the results reported in [13] are only about 96% of *move* instructions removed by the most aggressive register coalescing and 86% *move* instructions removed in [8].

Super-instruction is an another technique to reduce the number of indirect branches and to eliminate intermediate storage of result on the stack. In most cases, the performance improvement are quite modest[4]. Our preliminary study estimates that around 512 extra superinstructions must be added to the interpreter to achieve the same static instruction reduction presented in this paper.

The arithmetic instruction format in our register-based JVM use three-addresses. Another obvious way to reduce code size is to use two-address instruction format for these instructions. We choose to use three-address instruction format in order to improve the chances of our copy propagation. Moreover, the static arithmetic instructions consist of, on average, only 6.28% of all instructions in SPECjvm98 client benchmarks. Most of the individual arithmetic instructions are statically less than 1%. The contribution of using two addresses arithmetic instruction format to code size reduction is very small.

After the copy propagation, most of the stack slots are not used anymore. One area of improvements that we can make is to do the dataflow analysis and try to compact the virtual register usage so that the size of Java frame can become smaller. This will probably have small impact on memory usage and performance.

Given a computation task, a register-based VM inherently needs far fewer instructions than a stack-based VM. In our case, our register-based JVM implementation can reduce the static number of bytecode instructions by 43.47% and the dynamic number of executed bytecode instructions by 47.21% when compared to those of the stack-based JVM. The reduction of executed bytecode instructions leads to fewer real machine instructions for the benchmarks and significant smaller number of indirect branches, which is very costly when mispredictions of indirect branches happen. On the other hand, the larger codesize (25.05% larger) could results in possible higher level-1 data cache misses and load/store operations for processor. In terms of running time, the benchmark results show that our register-based JVM has an average 30.69%(switch) & 29.36%(threaded) improvement on Pentium 3 and 32.28%(switch) & 26.53%(threaded) on Pentium 4. This is a very strong indication that the register architecture is more superior for implementing interpreter-based virtual machine that the stack architecture.

7. CONCLUSIONS

A long standing question has been whether virtual stack or virtual register VMs can be executed more efficiently using an interpreter. Virtual register machines can be an attractive alternative to stack architectures because they allow the number of executed VM instructions to be substantially reduced. In this paper we have built on the previous work on Davis et al [5], 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 code, which we believe gives a more accurate measure of the potential of virtual register architectures. We have also presented results for a real implementation in a fully-featured, standard-compliant JVM.

We found that a register architecture requires an average of 47% fewer executed VM instructions, and that the resulting register code is 25% larger than the correpsonding stack code. The increased cost of fetching more VM code due to larger code size involves only 1.07% extra real machine loads per VM instruction eliminated. On a Pentium 4 machine, the register machine required 32.3% less time to execute standard benchmarks if dispatch is performed using a C switch statement. Even if more efficient threaded dispatch is available (which requires labels as first class values), the reduction in running time is still around 26.5% for the register architecture.

8. REFERENCES

[1] Spec release spec jvm98, first industry-standard benchmark for measuring java virtual machine performance. *Press Release*, page http://www.specbench.org/osg/jvm98/press.html, August 19 1998.

[2] M. Berndl, B. Vitale, M. Zaleski, and A. D. Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *2005 International Symposium on Code Generation and Optimization*, March 2005.

[3] M. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. Benchmarking java grande application. In *Second Ineternational Conference and Exhibtion on the Practical Application of Java*. Manchester, UK, April 2000.

[4] K. Casey, D. Gregg, M. A. Ertl, and A. Nisbet. Towards superinstructions for java interpeters. In *Proceedings of the 7th International Workshoop on Software and Compilers for Embedded Systems (SCOPES 03)*, pages 329–343, September 2003.

[5] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The case for virtual register machines. In *Interpreters, Virtual Machines and Emulators (IVME '03)*, pages 41–49, 2003.

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

[7] M. A. Ertl and D. Gregg. The behaviour of efficient virtual machine interpreters on modern architectures. In *Euro-Par 2001*, pages 403–412. Springer LNCS 2150, 2001.

[8] L. George and A. W. Appel. Iterated register coalescing. Technical Report TR-498-95, Princeton University, Computer Science Department, Aug. 1995.

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

[10] D. Gregg, A. Beatty, K. Casey, B. Davis, and A. Nisbet. The case for virtual register machines. *Science of Computer Programming, Special Issue on Interpreters Virtual Machines and Emulators*, 2005. To appear.

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

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

[13] J. Park and S. mook Moon. Optimistic register coalescing, Mar. 30 1999.

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

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

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