The Case for Virtual Register Machines*

Brian Davis, Andrew Beatty, Kevin Casey, David Gregg and John Waldron
Department of Computer Science
Trinity College
Dublin 2, Ireland
David.Gregg@cs.tcd.ie

ABSTRACT

Virtual machines (VMs) are a popular target for language implementers. Conventional wisdom tells us that virtual stack architectures can be implemented with an interpreter more efficiently, since the location of operands is implicit in the stack pointer. In contrast, the operands of register machine instructions must be specified explicitly. In this paper, we present a working system for translating stack-based Java virtual machine (JVM) code to a simple register code. We describe the translation process, the complicated parts of the JVM which make translation more difficult, and the optimisations needed to eliminate copy instructions. Experimental results show that a register format reduces the number of executed instructions by 34.88%, while increasing the number of bytecode loads by an average of 44.81%. Overall, this corresponds to an increase of 2.32 loads for each dispatch removed. We believe that the high cost of dispatches makes register machines attractive even at the cost of increased loads.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors—*Interpreters*

Keywords

Interpreter, Virtual Machine, Register Architecture, Stack Architecture

1. MOTIVATION

Virtual machines (VMs) are a popular target for language implementers who wish to distribute programs in a portable, architecture-neutral format, which can easily be interpreted or compiled. Virtual machine code is also the intermediate format of choice for efficient, general-purpose virtual machine interpreters. The most popular virtual machines use a virtual stack architecture for evaluating expressions, rather than the register architectures that are commonly used in real processors.

Conventional wisdom tells us that stack architectures can be implemented with an interpreter more efficiently, since the location of operands is implicit in the stack pointer. In contrast, the operands of register machine instructions must be specified explicitly. The interpreter must fetch these operands from the virtual machine code, increasing the interpreter overhead, when compared with a stack architecture. It is also widely believed that stack architectures allow more compact virtual machine code, again because operands are specified implicitly. In addition, stack code is easier to generate in the compiler than register code. At the very least, a stack machine eliminates the need for a complicated register allocator. For these reasons, stack architectures have been used as the intermediate representations for a number of popular virtual machines, such as the Java VM, .NET VM, and Pascal P-code.

More recently, a number of authors and implementors of virtual machines have suggested that virtual register machines could be more efficient. Gregg et al. [8] mentioned the possibility in a general discussion of interpreter optimisations. Furthermore, the Parrot VM - the intermediate representation for Perl 6 - will use a register architecture because the implementers belief in the superiority of register machines. The Parrot VM has provoked a number of lively debates on newsgroups such as comp.compilers and comp.lang.perl on the relative merits of virtual stack and register machines. Despite the controversy, neither side has presented significant quantitative results comparing the two approaches, so no conclusion could be reached.

In this paper, we present a working system for translating stack-based Java virtual machine (JVM) code to a simple register code. We describe the translation process, the complicated parts of the JVM which make translation more difficult, and the optimisations needed to eliminate copy instructions. We also present a number of design choices, which can have a significant impact on the number of instructions in the resulting register machine program. We present quantitative results for real, large programs: the standard SPECjvm98 and Java Grande benchmark suites. We find that a virtual register architecture significantly reduces (34.88%) the number of executed instructions in Java programs , while increasing the number of bytecode fetches by only 44.81%.

The rest of this paper is organised as follows. Section 2 describes the basic functioning of a virtual machine in-

*This work was supported by Enterprise Ireland Research Innovation Fund, Grant IF/2001/350.

IVME'03, June 12, 2003, San Diego, California, USA.

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

void engine()
{
  static Bytecode program[] = { iadd /* ... */ };

  Bytecode *ip;
  int *sp;

  while (1)
    switch (*ip) {
    case iadd:
      dest = ip[1];
      s1 = ip[2];
      s2 = ip[3];
      reg[dest]=reg[s1]+reg[s2];
      ip+=4;
      break;
    /* ... */
    }
}
```

Figure 1: Instruction dispatch using `switch`

```
typedef void *Inst;

void engine()
{
  static Bytecode program[] = { iadd /* ... */ };

  Bytecode *ip;
  Inst dispatch_table = { &&nop, &&aload_null, .... };
  int *sp;

  goto dispatch_table[*ip];

 iadd:
  dest = ip[1];
  s1 = ip[2];
  s2 = ip[3];
  reg[dest]=reg[s1]+reg[s2];
  ip+=4;
  goto dispatch_table[*ip];
}
```

Figure 2: Instruction dispatch using token threading in GNU C

terpreter, and the most important types of instruction dispatch. In section 3 we describe the main differences between virtual stack and virtual register machines. Section 4 looks at the particular strengths and weaknesses of the two types of virtual architecture, and estimates the relative advantages of each. In section 5 we present our translation system to convert stack Java bytecode to an equivalent virtual register code. Section 6 examines techniques for eliminating move instructions from register code. Finally in section 7 we present results showing that a register format can significantly reduce the number of executed instructions for the same program.

2. VIRTUAL MACHINE INTERPRETERS

The Java Virtual Machine uses a stack-based bytecode to represent the program. Interpreting a bytecode instruction consists of accessing arguments, performing the function of the instruction, and dispatching (fetching, decoding and starting) the next instruction.

Instruction dispatch typically consumes most of the execution time in virtual machine interpreters. The reason is that most VM instructions require only a small amount of computation, such as adding two numbers or loading a number onto the stack, and can be implemented in a few machine code instructions. In contrast, instruction dispatch can require up to 10-12 machine code instructions, and involves a time consuming indirect branch. For this reason, dispatch consumes a large proportion of the running time of most efficient interpreters [3].

Switch dispatch is the simplest and most widely used approach. The main loop of the interpreter consists of a large `switch` statement with one `case` for each opcode in the JVM instruction set. Figure 1 shows how this approach is implemented in C.

Switch dispatch is simple to implement, but rather inefficient for a number of reasons. First, most compilers produce a range check to ensure that the opcode is within the range of valid values. In the JVM this is unnecessary, since the bytecode verifier already checks that bytecodes are valid. Secondly, the `break` is translated into an unconditional jump back to the start of the loop. Given that the loop already contains a jump, it would be better to structure the loop as a set of routines that jump to one another. A final source of inefficiency results from there being only a single indirect branch for dispatching instructions. On machines with programmer visible pipelines, such as the Philips Trimedia processor for embedded systems, it is difficult to overlap this branch with other instructions [9]. On processors with branch predictors, this branch is very unpredictable [3]

An alternative to using a `switch` statement is threaded dispatch. Threaded dispatch is based on making explicit the sequence of steps generated by a compiler to implement a `switch` statement. Once these steps appear at the source level, the programmer can optimize the code by removing unnecessary work. Unfortunately, it is not possible to break a `switch` statement into its component parts in ANSI C, because there is no facility for `goto` statements that can jump to multiple different locations. To implement threaded dispatch, one requires a language with labels as first class value, such as GNU C, the language accepted by the GCC compiler.

Figure 2 shows how *token threaded* dispatch can be implemented using GNU C. The range check has been eliminated, as has the jump back to the dispatch routine at the end of the code for each VM instruction. Instead, the dispatch code is appended to the end of the code for each virtual machine instruction. This increases the size of the interpreter slightly, although it is usually faster. Another effect of replicating the dispatch code is that it allows the dispatch branch to be scheduled more efficiently with the code to implement the bytecode instruction, and it also greatly increases the prediction accuracy of the indirect branch on processors with branch target buffers (45% versus 2%–20% for switch dispatch) [3].

3. STACK VERSUS REGISTERS

The cost of executing a virtual machine instruction consists of three components:

- Instruction dispatch
- Operand access
- Performing the computation

The cost of dispatching an instruction is essentially the same for virtual register and stack machines. However, a given computation 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 JVM code as `iload c, iload b, iadd, istore a`. In a virtual register machine, the same code would be the single instruction `iadd a, b, c`. Thus, virtual register machines have the potential to significantly reduce the number of executed instructions. By how much? It depends on how often values must be loaded to, stored from, or shuffled around the stack. If the computation can be organised so that operands can always be found on top of the stack, changing to a register architecture would give no reduction in executed instructions.

Another reason for pessimism with the number of executed instructions on register machines relates to register allocation. The number of virtual registers is always limited, and if there are more live values than registers, some values must be spilled to memory. Additional load and store instructions must be added to access spilled values, increasing the number of executed instructions rather than reducing them.

Our experience is that this argument is something of a red herring, at least for the Java VM. The most commonly used instructions for loading and storing local variables use a one-byte index, which specifies the number of the local variable. A comparable virtual register machine would use a one-byte index to specify each of its operands, allowing up to 256 virtual registers to be used in each method. Measurements show that no methods in the standard SPECjvm98 and Java Grande benchmarks contain anything like 256 local variables or stack values, the values that could be allocated to registers. On the contrary, most methods contain less than 25 such values [16].

The second component of the cost of executing a VM instruction is accessing the operands. This consists of two separates costs - finding the location of the operands, and accessing the operands themselves. Finding the operands' locations is expensive for a virtual register machine. The location of each operand must be fetched from the instruction stream, and used as an index into an array of virtual registers. In contrast the cost of locating operands is lower on stack machine, since most operands are found on the top of the stack. The main cost is updating the stack pointer, and even this is not always necessary.

Virtual registers and virtual stacks are usually implemented as arrays in memory, so the cost of accessing the operands themselves is similar for both types of virtual architecture. Stack caching can be used, however, to reduce the cost of accessing virtual stacks by keeping the top one or two items in a register. For example, Ertl [2] found that keeping the topmost stack item in a register reduced memory traffic for stack items by about 50%. It is very difficult to keep virtual register items in real machine registers, because real machine registers can not be accessed array-like, with an index. The final component of the cost of executing a VM instruction is actually performing the computation itself. Given that most VM instruction perform a simple computation, such as an add or a load, this is usually the smallest part of the cost of executing a VM instruction. Generally, the type of virtual machine will not make a difference to this cost. The basic computation has to be performed, regardless of the format of our intermediate representation. However, there are situations where a register architecture can allow slightly more efficient code. In particular, exploiting common subexpressions is easier on a register machine that does not destroy its operands when using them, as a stack machine normally does.

4. SOME ESTIMATES

Clearly, the difference in speed between a virtual stack machine and a corresponding register machine can depend on many factors, especially on modern out-of-order processors with branch prediction and caches that make performance difficult to predict. However, we believe that the difference between the two types of machine can be estimated by looking at four main factors. The running of a virtual register machine (VRM) might be compared to a virtual stack machine (VSM) as follows:

$T_{VRM} \approx T_{VSM} - \#dispatches \times T_{dispatch} + \#fetches \times T_{fetch}$

In other words, the running time of a program on a VRM will be approximately equal to the running time of the program on a corresponding stack machine, minus the reduction in dispatches times the cost of a dispatch, plus the increase in fetches of operand locations times the cost of each of those fetches.

Generally, we would expect that the increase in fetches is likely to be large, since most VM instructions need one or two extra immediate operands to specify register locations. The cost of each of these fetches is likely to be low, however, since each usually corresponds to just an additional load instruction.

The reduction in dispatches is much more difficult to estimate without looking at real programs. However the cost of each dispatch is likely to be large. Ertl and Gregg [3] found that virtual machine interpreters contain very large numbers of indirect branches (up to 13% of all executed real machine instructions). Furthermore, these branches are highly (60%-97%) unpredictable on current desktop and workstation processors. The cost of each indirect branch misprediction is high, because it requires that the entire pipeline be drained, consuming 6–20 cycles, depending on the length of the pipeline. Ertl and Gregg [3] found that more than half of the execution time of many efficient interpreters is spent on indirect branch mispredictions. Almost anything that reduces the number of dispatches has the potential to significantly improve performance.

Another complication is that the cost of all dispatch mechanisms is not the same. As outlined in section 2, threaded dispatch is about twice as fast as switch dispatch, although it cannot be implemented in ANSI C. Thus, register machines might prove more efficient where the interpreter must be written in ANSI C for maximum portability, while a stack architecture might have an edge where GNU C or assembly language is acceptable.

```
ILOAD 4      IMOVE r10, r4        ; load local 4
ILOAD 5      IMOVE r11, r5        ; load local 5
IADD         IADD r10, r10, r11   ; integer add
ISTORE 6     IMOVE r6, r10        ; store TOS to local 6
ILOAD 6      IMOVE r10, r6        ; load local 6
IFEQ 7       IFEQ r10, 7          ; branch by 7 if TOS == 0
```

Figure 3: Example of stack and corresponding register code

5. FROM STACK TO REGISTER

To compare the relative benefits of virtual stack and register machines, we constructed a system for translating stack-based Java bytecode to a similar register code. Our translation scheme is based on mapping local variables and stack locations to a single set of virtual registers. In the JVM, all local variables are numbered, and we translate local variable numbers directly to virtual register numbers (so local variable zero is mapped to register zero).

Mapping stack locations to virtual register numbers is a little more complicated. Each stack location is given a number, and those numbers start just after the position of the last local variable. Mapping stack locations to register numbers is much simplified by Java's strict stack discipline. It is not possible to write code that, for example, increases the number of items on the stack on each iteration of a loop, as can be done in Forth. At every point in the program, the height of the stack must be fixed, and can be determined by simple static analysis. At control flow join points the height of the stack must be equal on both incoming control flow edges. Thus, by tracking the value of the stack pointer at each point in the program, it is possible to map stack locations to register numbers.

Figure 3 shows an example of Java stack bytecode and the corresponding register code. Note that in the register code, the first register operand is always the destination. We assume that there are ten local variable slots in this method (`r0..r9`), so the stack pointer for the initially empty stack will point to `r10`. Thus, when we translate an `ILOAD` instruction which copies the value in local variable 4 to the top of the stack, we translate this as an integer move (`IMOVE`) instruction from register `r4` to register `r10`.

Similarly, we translate the `IADD` stack instruction to a register `IADD` instruction that takes the topmost item in the stack (`r11`) and the second from top (`r10`), adds the two and places the result in the new topmost stack item (`r10`), which will be one lower than the previous top of stack because `IADD` reduces the height of the stack by one.

Using this scheme, it is relatively easy to translate any sequence of stack Java bytecode to an equivalent register format. It is important to note that the resulting code will often contain unnecessary and redundant `MOVE` instructions. For example, the original stack code contains the sequence `ISTORE 6, ILOAD 6`, which stores the topmost stack item to local variable number 6, and then reloads the value to the top of the stack. This type of sequence is actually extremely common in code produced by the *javac* compiler. Presuming that the value is stored to the local variable to allow it to remain live after the end of the basic block, it is not possible to express this in fewer instructions. In the corresponding register code, however, it is easy to remove many of these `IMOVE` instructions.

One type of instruction that needs special handling in the translation is method invocation instructions. Invoke instructions take their parameters from the top of the stack. These n topmost stack items become the first n local variables of the invoked method. Thus, invoke instructions can consume several values, and they destroy these values in the process.

In theory, this scheme allows extremely fast parameter passing, since making the topmost stack elements into the first local variables simply involves one assignment to the frame pointer. In practice, however, the parameters are rarely already on the stack, and most invoke instructions are preceded by one or more load instructions. Furthermore, once the parameters have been passed to the invoked method, they are in local variables and must be loaded to the stack before they can be used.

The simplest way to translate the parameter passing mechanism would be a completely literal translation, where the topmost registers of the caller become the first registers of the callee. In our first implementation we used this scheme, but found that it prevented us from removing very large number of load instructions when translating to register format. The problem is that invoke instructions consume several values, each of which must be in a specific register, and so cannot be moved.

Our new scheme uses an alternative scheme where each invoke takes as an immediate argument a list of the registers that it takes as parameters. Although this increases the number of loads necessary to identify the location of operands, it allows us to eliminate the great majority of `MOVE` instructions in register code.

6. ELIMINATING MOVES

Translating the Java bytecode to register code does not automatically reduce the number of executed instructions. The translation process outlined in the previous section simply converts each instruction directly from a stack to a register format. To eliminate unnecessary `MOVE` instructions, we apply a copy propagation algorithm that rewires the source and destination registers of instructions to bypass `MOVE`s. Once the sources and destinations have been changed, many of the `MOVE` instructions become dead code and can be eliminated.

We implemented two copy propagation algorithms, the first operating only on basic blocks and the second operating on the entire Java method. The basic block algorithm is both simple and efficient, and allows copy operations to be bypassed within a basic block. It is important to note that most values that are loaded to the stack are used very soon afterward, so a basic block algorithm can be very effective.

One complication that normally arises with single-basic-block copy propagation is that it is difficult to eliminate dead copies, because it is not clear which values are still alive at the end of the basic block. In Java bytecode, the problem is very much easier, since most destinations of `MOVE` instructions are on the stack. We can easily identify when most values on the stack become dead because the stack pointer moves below them (anything above the stack pointer is dead). Furthermore, the standard idiom used by the *javac* compiler is that the stack should be empty at the start and end of each statement. Thus, in the great majority of cases, the stack is empty at the end of each basic block, and all the items on the stack are dead.

Our second copy propagation algorithm operates on an entire method at a time. It uses classic dataflow analysis to compute liveness sets and propagate copies across basic block boundaries. Thus, it is much slower and more complicated that our basic block algorithm, and is probably not suitable for using in a scheme that translates from stack code to register code at load time. However, when we compared the two algorithms we found that there is a difference of less than 1% in the results. Clearly, a simple, efficient basic-block approach with liveness based on stack position is sufficient in most cases.

7. EXPERIMENTAL EVALUATION

Our basic thesis is that virtual register machines have the potential to be interpreted more efficiently than stack machines by reducing the number of executed instructions. To test this thesis, we implemented a system for translating Java bytecode to a corresponding register format, and measured the differences using the SPECjvm98 [14] and Java Grande [1] benchmarks. These benchmarks consists of several large programs with real data, which are intended to be representative of a wide range of Java applications.

Our translation system was built into CVM, a small implementation of the Java 2 Micro Edition (J2ME) standard. It supports the full JVM instruction set, as well as full system-level threads. We made a number of small additions to CVM to enable it to run the SPEC benchmarks and to allow us to safely compile it at a higher level of optimization than the standard distribution. All quoted figures are for the basic block implementation of copy propagation and dead copy elimination, as we believe this to be the most practical scheme. The whole-method copy propagation gives only slightly (less than 1%) better results.

The first time each method is invoked we translate it to register format. Thus, we present measurements only for methods that are executed at least once. Table 1 and figure 4 show the breakdown of instructions after translation to register format, based on statically appearing code. Overall, an average of 34.11% of statically appearing instructions are `MOVE` instructions. 28.56% of total instructions are `MOVE`s that can be eliminated with copy propagation and dead code elimination.

Table 2 and figure 5 show the breakdown of dynamically executed instructions. Interestingly, 41.28% of dynamically executed instructions are `MOVE`s. Clearly, local loads and stores are not distributed evenly throughout the programs, and code with larger numbers of such instructions tends to be executed more frequently. An average of 34.88% of executed instructions can be eliminated by translating to a register format. At more than one third of executed instructions, this is a very large number and strongly suggests that virtual register machines could be interpreted more efficiently than stack machines. This is especially likely to be true where the interpreter uses `switch` dispatch (see section 2), such as where the interpreter must be written in ANSI C.

In the last two columns of figure 2 and in figure 6 we examine the net increase in bytecode loads in the interpreter caused by the register format. These figures assume that each opcode and each register operand occupies one byte, each of which must be loaded separately. There are two effects at work here. First, translating to a register format increase the number of operands in the bytecode. Secondly, applying copy propagation and dead copy elimination allows us to eliminate a large number of instructions, thus reducing both the number of opcodes and operands. Overall, the register format requires an average of 44.81% extra bytecode loads. Clearly this is a large number, but loads are usually very much less costly than the indirect branches in instruction dispatches.

We also examined the ratio of the increase in the number of loads to the reduction in dispatches. We found that for most of the SPEC benchmarks the register format increased the number of loads required by an average of about two extra bytecode loads for every dispatch eliminated. The Grande benchmarks required quite a few more loads for each dispatch removed, such as a ratio of more than five to one for the *Euler* benchmark. We believe that this is probably caused by the Grande benchmarks being more mathematically oriented, and thus using more complicated expressions which can be expressed more elegantly in stack code. Over all the SPECjvm98 and Grande benchmarks, translating to bytecode increased the number of bytecode loads by an average of 2.32 for every dispatch eliminated. We believe that for `switch` based interpreters running on modern pipelined processors where the cost of branch mispredictions is very high, even 2.32 extra loads for each dispatch removed will still result in a significant benefit to the virtual register machine interpreter.

8. RELATED WORK

Recent important developments in interpreters include the following. Interpreter generators simplify construction and maintenance of interpreters and can allow automatic VM instruction combining [12] and stack optimizations [4]. Stack caching [2] is a general technique for storing the topmost elements of the stack in registers. Ertl and Gregg [3] showed that interpreters (especially those using switch dispatch) spend most of their time in branch mispredictions on modern desktop architectures. Interpreter software pipelining [9] is a valuable technique for architectures with delayed branches (e.g. Philips Trimedia) or prepare to branch instructions (e.g. PowerPC), which makes the target of the dispatch branch available earlier by moving much of the dispatch code into the previous VM instruction. Costa [13] discusses various smaller optimizations.

The Sable VM [6] is an interpreter-based research JVM. This interpreter uses a run-time code generation system [11], not dissimilar from a just-in-time compiler. Sable uses a novel system of *preparation sequences* [7, 5] to deal with bytecode instructions that perform initialisations the first time they are executed, which make code generation difficult.

Myers [10] 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. However, 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.

The controversy between stack and register code has arisen again recently because of the decision to make the the Parrot

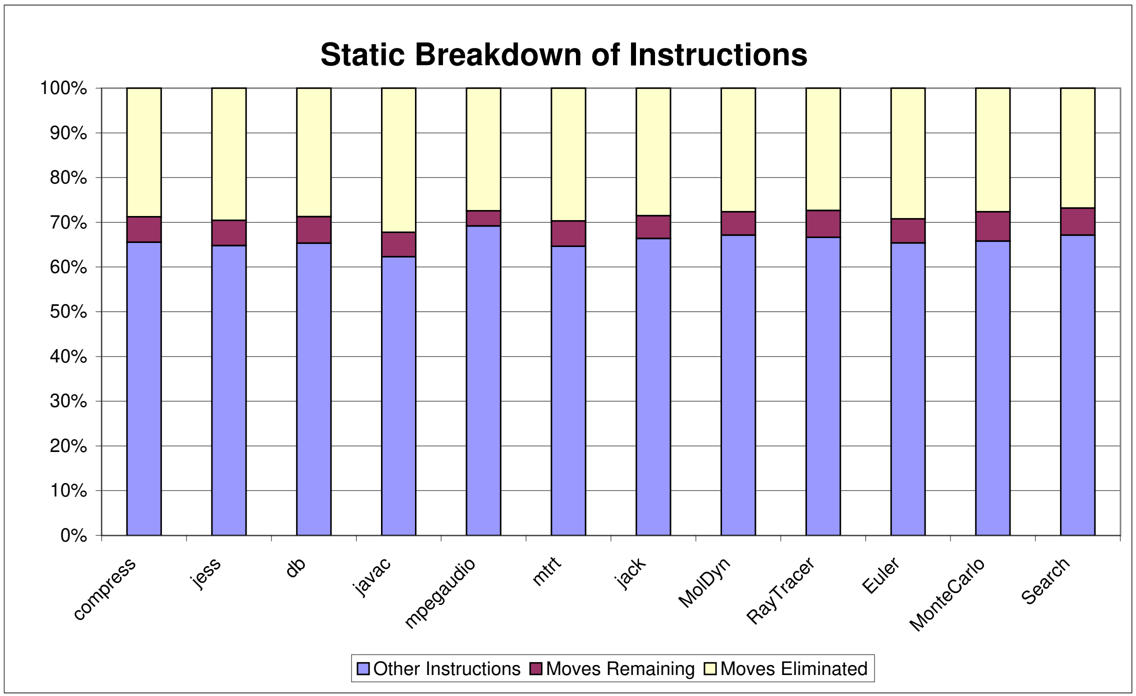

Figure 4: Breakdown of statically appearing register code instructions into moves eliminated, moves that could not be eliminated and other instructions.

Benchmark	Instructions	Moves	%	Eliminated	%
compress	28,612	9,852	34.43%	8,227	28.75%
jess	38,537	13,557	35.18%	11,392	29.56%
db	29,365	10,167	34.62%	8,434	28.72%
javac	59,545	22,442	37.69%	19,179	32.21%
mpegaudio	58,823	18,126	30.81%	16,135	27.43%
mtrt	33,969	12,004	35.34%	10,079	29.67%
jack	44,709	15,027	33.61%	12,737	28.49%
MolDyn	31,873	10,465	32.83%	8,811	27.64%
RayTracer	20,999	7,006	33.36%	5,739	27.33%
Euler	28,003	9,684	34.58%	8,178	29.20%
MonteCarlo	23,442	8,010	34.17%	6,477	27.63%
AlphaBetaSearch	21,328	7,005	32.84%	5,717	26.81%
Average	34,636	11,838.85	34.11%	9,982	28.56%

Table 1: The number of static instructions that can be potentially removed and the the number that are actually removed, compared with the total number of instructions.

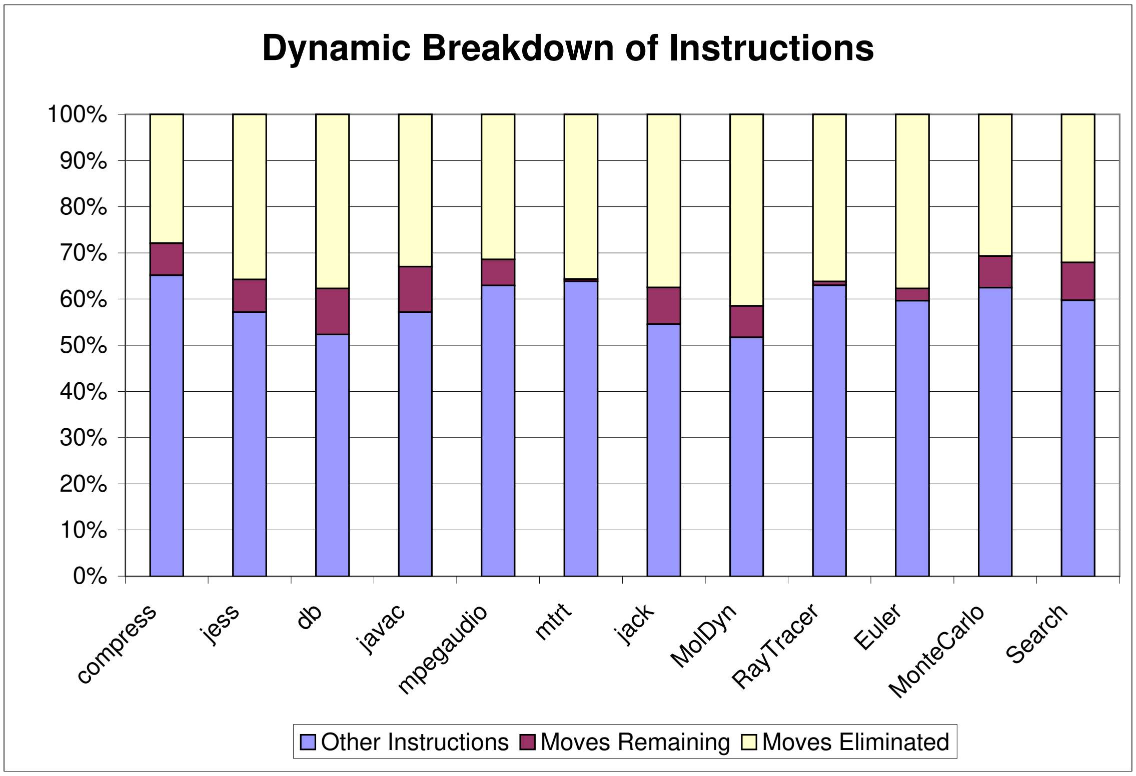

Figure 5: Breakdown of dynamic register code instructions into moves eliminated, moves that could not be eliminated and other instructions.

Benchmark	Instructions	Moves	%	Eliminated	%	extra loads	loads/dispatch
compress	4,917	1,713	34.84%	1372	27.91%	5280	3.85
jess	979	419	42.82%	349	35.73%	657	1.88
db	1,135	541	47.68%	428	37.69%	750	1.75
javac	1,335	571	42.82%	440	32.95%	952	2.17
mpegaudio	4,805	1,779	37.04%	1509	31.40%	4387	2.91
mtrt	970	350	36.13%	346	35.63%	667	1.93
jack	611	277	45.40%	229	37.48%	347	1.52
MolDyn	7,589	3,663	48.26%	3147	41.48%	2883	0.92
RayTracer	7,177	2,654	36.98%	2596	36.18%	7594	2.92
Euler	10,162	4,100	40.35%	3830	37.69%	9082	2.37
MonteCarlo	1,625	609	37.52%	498	30.67%	1717	3.45
AlphaBetaSearch	4,780	1,924	40.26%	1531	32.04%	4563	2.98
Average	3,545	1,431	41.28%	1252	34.88%	2991	2.32

Table 2: The number of executed instructions (in millions) that can be potentially removed and that are actually removed, compared with the original total. The net increase in bytecode fetches (in millions) is shown in second rightmost column. The rightmost column shows the number of extra bytecode loads for each VM instruction dispatch eliminated.

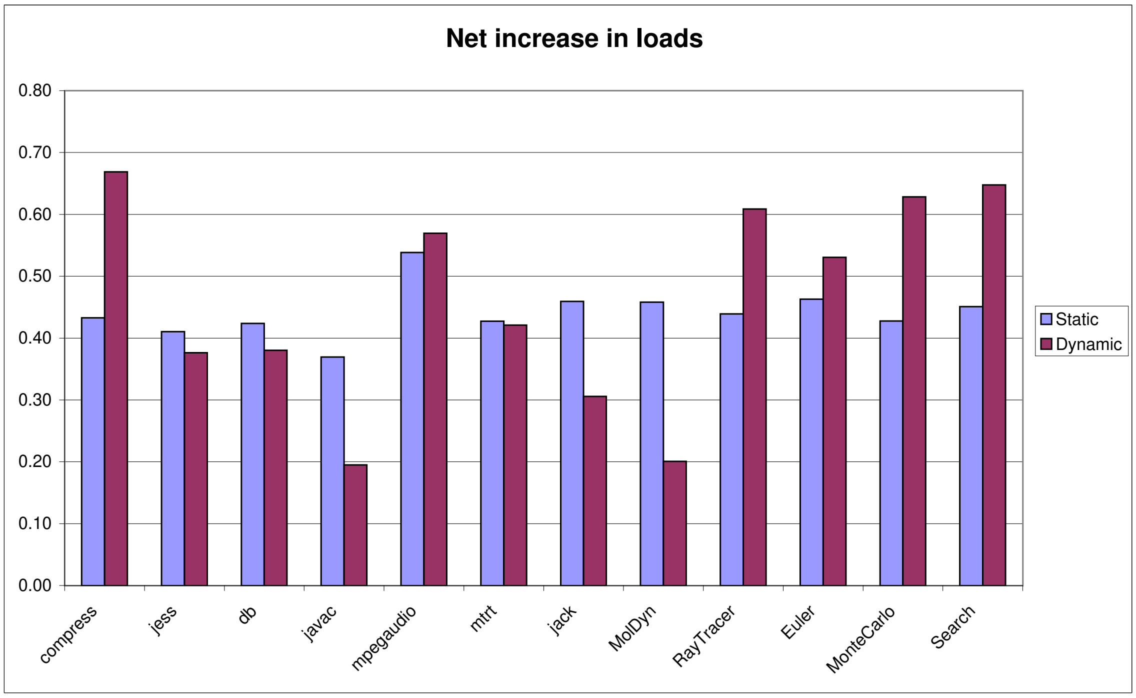

Figure 6: Net increase in bytecode loads from using a register rather than stack architecture.

VM, the intermediate representation for the Perl 6 language, a register rather than stack machine. Again, arguments for this design decision [15] have been based on just a couple of small examples, rather than any study of real programs.

9. CONCLUSION

Virtual register machines are an attractive alternative to virtual stack architectures because they allow the number of executed instructions to be reduced by eliminating large number of loads to and stores from the stack. This is especially important for interpreters modern pipelined processors, where the cost of instruction dispatch is very high.

We have described a system for translating Java bytecode to a corresponding register format. We have implemented this system in a real JVM and used it to collect data on the the effect of translating the SPECjvm98 and Java Grande benchmarks to register format. We believe that ours is the first quantitative data that measures hard numbers in real programs, rather than basing arguments on small examples. We found that translating to a register format decreases the number of executed instructions by an average of 34.88%, while increasing the number of bytecode loads by an average of 44.81%. Overall, this corresponds to an increase of 2.32 loads for each dispatch removed. We believe that the high cost of dispatches makes register machines attractive even at the cost of increased loads.

10. REFERENCES

[1] M. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. Benchmarking Java Grande applications. In *Second International Conference and Exhibition on the Practical Application of Java*, Manchester, UK, April 2000.

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

[3] 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.

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

[5] E. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, Mc Gill University, December 2002.

[6] E. Gagnon and L. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *First USENIX Java Virtual Machine Research and Technology Symposium*, Monterey, California, April 2001.

[7] E. Gagnon and L. Hendren. Effective inline-threaded interpretation of java bytecode using preparation sequences. In *Proceedings of the 12th International Conference on Compiler Construction*, LNCS 2622, pages 170–184, April 2003.

[8] D. Gregg, A. Ertl, and A. Krall. A fast java interpreter. In *Proceedings of the Workshop on Java optimisaton strategies for embedded systems (JOSES)*, Genoa, April 2001.

[9] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel. A code compression system based on pipelined interpreters. *Software—Practice and Experience*, 29(11):1005–1023, Sept. 1999.

[10] G. J. Myers. The case against stack-oriented

instruction sets. *Computer Architecture News*, 6(3):7–10, 1977.

[11] 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.

[12] T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.

[13] V. Santos Costa. Optimising bytecode emulation for Prolog. In *LNCS 1702, Proceedings of PPDP'99*, pages 261–267. Springer-Verlag, September 1999.

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

[15] D. Sugalski. *http://www.parrotcode.org/*.

[16] J. Waldron. Dynamic bytecode usage by object oriented java programs. In *Proceedings of the Technology of Object-Oriented Languages and Systems 29th International Conference and Exhibition*, Nancy, France, June 7-10 1999.