USENIX Association

Proceedings of the 2nd Java^{TM} Virtual Machine Research and Technology Symposium (JVM '02)

San Francisco, California, USA
August 1-2, 2002

Cross-Architectural Performance Portability of a Java Virtual Machine Implementation

Matthias Jacob
Princeton University
mjacob@cs.princeton.edu

Keith Randall
Google, Inc.
keithr@alum.mit.edu

Abstract

This paper describes our experience in porting Compaq's Fast VM from the Alpha processor architecture to the Intel x86 processor architecture. We encountered several opportunities and pitfalls along the way in porting a JVM designed for a RISC architecture to a CISC architecture. Our goal was to preserve most of the FastVM's performance benefits already available on the Alpha platform, and modify or discover new optimizations as they were required for x86. We found that by porting a fast RISC JVM to x86, we could generate a JVM with performance competitive to state-of-the-art production JVM implementations.

1 Introduction

The Alpha processor architecture [23] and the Intel x86 processor architecture [1] have totally different design philosophies. Alpha, which is a RISC architecture [18], provides a minimal, simple instruction set which can be efficiently decoded. Intel x86 is a CISC architecture which is designed to run more complex operations within a single instruction, and thus includes more different instructions and formats. While porting Compaq's Fast VM [6] from Alpha to x86, we encountered several opportunities and pitfalls because of this change in architectural philosophy.

Fortunately, many parts of the JVM required little or no modification when switching from one architecture to another. These parts include the class loader, bytecode verifier, and most of the garbage collector. Other parts of the JVM were ported by others - we took advantage of Sun's port of the Java libraries to x86/Linux so we did not have to repeat that work. Instead, we concentrated on the major changes required in the just-in-time (JIT) compiler and closely related modules, like the stack unwinding mechanism. Crucial to a successful (i.e., fast) port of the JIT was maintaining the quality of generated code that was the result of many optimizations performed in the RISC JIT. We found that some optimizations were straightforward to port, other optimizations required major rework, and still others were simply unworkable in a CISC architecture. Finally, we also found that some additional optimizations not required at all by a RISC machine were of critical importance to fast CISC code.

The different design philosophies of the Alpha and x86 architectures impose different design constraints on a Java virtual machine:

- *Reduced number of registers:* The Alpha architecture has 31 registers, compared to the x86 architecture which has only 8. This differential makes it crucially important to do register allocation well on the x86.
- *Instructions contain multiple operations:* On a RISC architecture instructions either load values from memory, store values into memory, or execute arithmetic operations. In contrast, CISC architectures support complex instructions that integrate these different RISC functions into a single instruction. Selecting the optimal instruction for a certain task, therefore, becomes more difficult on the x86.
- *Different addressing modes:* Because x86 instructions decompose into multiple operations, similar instructions are built from slightly different primitive operations. For example, an addition can add a value in memory and a value in a register, or add two registers.
- *Non-orthogonality of instruction set:* Not all registers can be used with every instruction,

so CISC architectures impose additional constraints on how data is allocated to registers.

- *Source registers get overwritten:* Within an arithmetic instruction, a source register is often overwritten on a CISC architecture to store the result. If the old value of the source register is needed, an additional copy step before such an instruction is required.

In addition to these five general aspects, the x86 architecture has the following design differences with Alpha:

- *32-bit architecture:* Porting the JVM from a 64-bit architecture to a 32-bit architecture introduces several complications. Since the Java VM supports 64-bit integer operations, a 32-bit implementation must emulate these operations using multiple instructions. Furthermore, the 32-bit architecture limits the maximum feasible heap size to 4GB.

- *Limited set of registers per instruction:* RISC instructions support access to either all integer or floating-point registers, depending on the instruction. On the x86 architecture, certain instructions require their arguments to be in certain registers. For example the shift operations require the shift amount to be given in register %cl, whereas in the Alpha architecture, the shift amount can be in any register. These restrictions impose additional complexity on register allocation [9].

- *Floating-point stack versus floating-point registers:* In the x86 architecture, all floating-point operations are executed on a floating-point stack instead of floating-point registers. Operationally, an arithmetic operation on two floating-point values pops the first two elements on the floating-point stack, executes the operation, and pushes the result on the floating-point stack. The resulting stack has one less element than the original stack. The register allocator must take these movements into consideration.

- *Floating-point precision toggle:* On the Alpha architecture, the precision of the floating-point operation is always encoded in the instruction itself, whereas it needs to be explicitly set by an additional instruction in the x86 architecture before the instruction operates on two registers on the floating-point stack.

The following two sections describe various optimizations we implemented in the x86 JVM. Section 2 describes modifications we made to existing optimization algorithms to port them from Alpha to x86. Section 3 describes new optimizations implemented specifically for the x86.

2 Modified Optimizations

The Fast VM for Alpha [6] is a fast, full-featured virtual machine for Alpha and Tru64 Unix. For the purposes of this paper, we will concentrate on the JIT inside this JVM, as the majority of modifications required to port to x86/Linux occur inside the JIT. In this section, we describe three JIT optimizations that are crucial for efficient performance on x86 and which, for various reasons, required some redesign for the x86 port.

We describe the optimizations here – we postpone our experimental analysis of these optimizations to the performance section (Section 4).

2.1 Register Allocation

A crucial difference between the x86 and Alpha architectures is the number of directly-addressable machine registers. In particular, the x86 architecture has only 8 integer registers (none of which are completely general-purpose), compared to 31 on the Alpha. This differential makes it crucially important to do register allocation well on the x86. Many algorithms for register allocation have been developed for use in standard compilers [11, 10, 12, 14, 21]. Although these allocators generate excellent allocations, they are typically too slow for use in a just-in-time setting. Much recent work has focused on faster just-in-time allocators [19]. The Alpha register allocator, which we adapted for use on the x86, does a simple global[1] allocation based on access frequency, and then uses a greedy allocator within each basic block.

The data structure used for the allocator in the FastVM is called an *lmap* ("location map") (see Figure 1). In the global allocation phase, each Java *entity* (Java stack location S_i or local variable L_i) is assigned a *home* machine location (H), either a register or stack slot. The register allocator dedicates every register either as a home location (H) for a particular Java entity, or a temporary location (T)

[1] *global* in this context means across a single procedure.

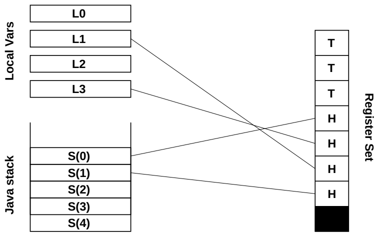

Figure 1: LMAP register allocation.

that can be used for storing intermediate results or call arguments. The allocation of home locations is a simple priority-based allocator, where the priority of a Java entity is just its estimated access frequency. Because interference analysis is expensive, the allocator assumes that every Java entity interferes with every other Java entity. The lack of interference analysis is not too detrimental as Java entities, particularly stack slots, encompass many live ranges. At each basic-block boundary, all live entities are forced to their home locations.

Within a basic block, the allocator uses a greedy allocation algorithm. The lmap maintains a mapping from Java entities to the locations where their values are currently stored to facilitate allocation. Methods of lmap are used to move arguments into registers, allocate additional temporary registers, and record the result registers. The lmap maintains separate allocation information for integer and floating-point registers.

Since the number of registers on the x86 is relatively small, it is important to use those registers during code generation effectively. We implemented a number of optimizations of the RISC register allocation algorithm to improve register utilization. First, the RISC allocator divides the available register set into two categories – home locations and temporary locations. For the x86 allocator, such a static partitioning was too restrictive, so we allowed the allocator to use home locations as temporaries when the home location was dead. This allowed us to increase the percentage of registers that are allowed to be home locations without overly constricting the temporary register set. Second, the x86 instruction set is not orthogonal – not all instructions can use all registers, so additional code was added to the allocator to allow allocation from a specific subset of possible registers. Finally, many instructions in the x86 architecture can accept arguments in memory locations instead of registers, so the allocator was also modified to understand that allocation of particular arguments to registers is optional, so that under high register-pressure conditions the memory form of the operation is used.

Wasteful static register allocations had to be eliminated as well. Because the Alpha architecture has lots of registers, the Alpha JVM assigned a few registers exclusively to particular tasks. One register was dedicated to holding a pointer to thread-local state, one register was reserved for interface method invocation, and one register was reserved as a scratch location for the code generator. Dedicating 3 registers to specific uses is tenable when 31 registers are available, but when only 8 registers are available, being carefree with your registers is not advisable.

Of particular importance to the performance of the RISC JVM is fast access to thread-local storage. The object allocator and mutex mechanism make heavy use of thread-local variables, so access to them must be fast. The RISC JVM assigned a register to point to the thread's own variables, so access to those variables required just a load or store with this register as a base. The CISC JVM cannot afford to spare a general-purpose register for this purpose. However, the x86 has other registers, one of which we “stole” to provide fast thread-local access without requiring any general-purpose registers. We store our thread-local pointer in a segment register on the x86 to avoid using a general-purpose register (fortunately, the %fs segment register is unused by current Linux x86 compilers and libraries).

For the other two purposes that the RISC JVM reserves a register, we instead reserve a thread-local variable. Although typically slower than a register, the thread-local variables have the same semantics as a register and thus can be used in their stead. We also take advantage of the fact that most instructions that use these reserved registers in the RISC JVM have analogues in the x86 world that accept our thread-local memory locations (constant offsets from %fs) instead of registers. The CISC nature of the x86 is thus a two-edged sword – fewer registers require us to be crafty about our register use, but at the same time the x86 provides us with more instructions and addressing modes to be crafty with.

The combination of these register allocation improvements increased the JVM's spec rating by 68%. For details on our experiments, see Section 4.

2.2 Instruction Selection

Instructions on a RISC architecture are structured in a systematic way. They can be categorized into ALU operations, memory operations, and control operations. ALU operations always work on registers, and every register from the register set can be addressed by every ALU operation. Memory operations move values between the register set and memory. This structure makes instruction selection relatively easy, and the Java data types map 1:1 to the instruction architecture.

The optimal selection of instructions is more complex on x86 than it is on Alpha for the following reasons:

- *Different addressing modes:* Because a lot of operations exist in different addressing modes, unlike a RISC processor, the correct kind of instruction needs to be chosen in order to avoid any additional instructions for moving values. For example if the values for an addition operation are in a memory and in a register the instruction selection algorithm always picks an add instruction that adds a memory and a register location.

- *Limited set of registers per instruction:* When picking the next instruction, the code generator always checks in which registers the current values are in and chooses the instruction appropriately. If the current register allocation doesn't fit to the instruction at all, values need to be moved. This scheme could be improved with more global analysis, but at the expense of a larger compile-time cost.

- *Efficient 64-bit operations:* The Java bytecode contains 64-bit integer and floating-point operations that the x86 platform needs to support. For each of these bytecode operations the number of temporary registers and the amount of memory accesses need to be minimized. For example, the following code is one possible implementation of the `ladd` (64-bit integer addition) bytecode instruction.

```
mov    0x0(%esp,1),%eax
add    0x8(%esp,1),%eax
mov    0x4(%esp,1),%ecx
adc    0x10(%esp,1),%ecx
```

2.3 Instruction Patching

Because just-in-time compilation takes place in a single pass, the generated code needs to be fixed up in certain situations:

- *class initializers:* Java requires that a class be initialized before any of its methods or fields are accessed. For any reference in the code we are compiling that refers to an uninitialized class, the JIT must insert instructions before the reference to ensure that the class is initialized. After the class is initialized, these added instructions are superfluous and can be patched to NOPs for improved performance.

- *fix up branches:* When generating code, the target address of forward branches is not yet known, so these branches must be fixed up when the destination address is known.

- *copying registers:* In some situations, register copy operations can be avoided by renaming registers backward in the already-generated code.

- *inlining:* Small methods can be directly inlined at the call site.

Since the instruction length is fixed on RISC architectures, it is relatively easy to implement instruction patching efficiently and safely.

In certain cases (see Section 2) instructions need to be patched at runtime. Patching in a JVM can be a delicate operation, because often the code that is being patched is being executed concurrently by another thread or processor. Thus, all intermediate states of the patching operation need to be seen by all processors as valid and correct code.

On Alpha, patching is straightforward since every instruction is exactly 4 bytes long. Each instruction can be atomically replaced with a single write (and the JVM is designed to only require single-instruction patching). However, on the x86 architecture, instruction lengths may be different, which complicates the patching process. We need to make sure that we can atomically patch any instruction of reasonable length with another, possibly different length, instruction sequence.

To make patching possible, at code generation time we ensure that any patchable instruction contains enough bytes for any instruction which we might want to patch over it, and that the patch location is suitably aligned so that an atomic operation can be used to perform the switch. On Pentium-class processors, this means that the patch location must not straddle a cache-line (32-byte) boundary. Finally, at patch time we equalize the lengths of the sequences by padding the inserted sequence with nop's, and use an atomic compare-and-exchange operation to ensure the patch is performed exactly once.

2.3.1 Inlining

Small compiled methods that fit into a call site can easily be inlined using the code patching mechanism. The JVM inlines a method by patching the method body into a call site and pads the remaining space with nops. Interestingly, inlining in this manner allows us to inline a method *after* all call sites to it have been compiled. This kind of inlining typically happens because inlining at the time we compile the caller is not always possible, usually because of class initialization constraints. More sophisticated techniques for inlining can be found in [13].

2.3.2 Retargeting

The Alpha JIT uses *retargeting* to avoid register-to-register copies. When a Java entity needs to be moved from one register to another, the Alpha JIT scans backward in the code to find the instruction that generated the value (if it exists), and rewrites it to use the new destination.

Although retargeting is very effective on the Alpha, it is difficult to implement on the x86 for three reasons. First, it is difficult in general to walk backward in code when instructions are not fixed size. Second, many x86 instructions generate their results in particular registers and thus are not retargetable. Finally, for many x86 instructions the output register is also an input register, so rewriting the output register alone is not possible. For these reasons, we added instead a forward-looking heuristic into our register allocator. The allocator computes the preferred register into which each Java entity should be placed, based on the requirements of its nearest future use. The allocator then uses these suggested registers, if possible, to satisfy allocation requests. By preferentially using the suggested registers, the allocator often ensures that the entity is in the correct location when the use of the entity is later encountered.

register	use	type
eax	1st int arg, first int return	scratch
ecx	method ptr	scratch
edx	2nd int arg, second int return	scratch
ebx		preserved
esi		preserved
edi		preserved
ebp		preserved
esp	stack pointer	preserved

Figure 2: Optimized Calling Convention

3 Optimizations for x86

Simply updating the JVM optimizations to take into account the different properties of x86 was not enough, however. The x86 platform has additional peculiarities that required the implementing of some additional optimizations. We describe these optimizations here.

3.1 Calling Convention

There are four problems with the x86 calling convention that make it difficult to port the RISC JVM to x86. First, the x86 argument passing and (some) return value passing are done on the stack instead of in registers. Second, the x86 dedicates two registers to stack management, a frame pointer and a stack pointer, when it is possible to use only a single register. Third, we need to be able to unwind the stack to implement the Java exception model, and changes to the x86 calling convention were required to simplify and speed up this unwinding. In addition, Java requires precise detection of stack overflow, which is difficult in the standard calling convention because almost any instruction can cause a stack overflow. Finally, the x86 calling convention enforces only 4-byte alignment of stack frames, which can be a performance problem because 8-byte stack operations might be unaligned.

In order to solve all of these problems, we developed a new calling convention as shown in Figure 2. This register assignment gives us 3 scratch (caller-save) registers and 4 preserved (callee-save) registers, plus a stack pointer.

We modified the calling convention to use a fixed stack pointer over the life of a method, as opposed to the standard x86 convention which encourages the use of push and pop instructions which modify the stack pointer. Local stack variables can be accessed at constant offsets from the stack pointer. The optimized stack scheme of our implementation is shown in Figure 3. The prolog/epilog and a sample callsite of the optimized calling convention can be found in Figures 4 and 5 respectively.

By allocating a callee-saved register slot at the bottom of the stack frame, the prolog of a method can immediately check whether a stack overflow has occurred by storing a callee-saved register (or any value, if there aren't any registers that need to be saved) to the bottom of the stack frame. Thus, the only instructions that can cause a stack overflow are the first store in the method prolog, and call instructions (which push their return address). At both of these locations, stack overflow exceptions are simple to deal with.

We also took the opportunity while changing the calling convention to align stack frames to 8-byte boundaries for faster stack operations on the `double` type.

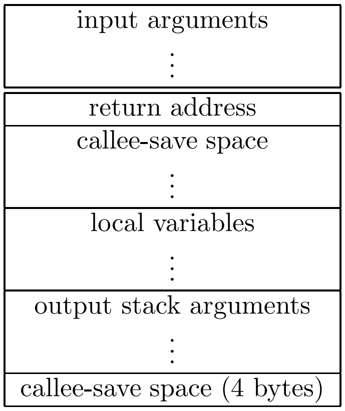

Figure 3: Optimized stack frame layout

```
subl $24, %esp
movl %ebx, (%esp)  # save %ebx,
                     stack check
...body of method...
movl (%esp), %ebx  # restore %ebx
addl $24, %esp
ret
```

Figure 4: Method prolog/epilog of the optimized calling convention.

```
movl $1, %eax    % 1st arg
movl $2, %edx    % 2nd arg
movl $3, 4(%esp) % 3rd arg
call method
                 % return value
                   in %eax
```

Figure 5: Example call site in the optimized calling convention.

3.2 Floating-Point Modes

Switching the floating-point precision mode on x86 inserts an additional instruction into the code and also causes stalls in the processor pipeline. For Java methods using 32-bit and 64-bit floating-point operations, switching precision is inevitable. However, our experiments have shown that while running the benchmarks, only one method in the Java class libraries and one method in the benchmark programs (mpegaudio) use two precision modes within a single method. Even in these cases the less frequent precision is used at most two times. To take advantage of this fact, we choose a default precision mode for each method and set the precision mode to the default at the beginning of each method that uses floating-point operations. Thus, default precision operations require no additional instructions, while the precision mode needs to be set and reset around the occasional non-default precision operation. Our calling convention considers the precision mode to be a preserved value, so it must be reset at the end of the method.

Because non-default precision operations are rare, this strategy significantly decreases the number of switches required and thus increases performance. Precisely analyzing the control-flow and optimizing the number of switches in a given method would help, but may be too expensive for just-in-time compilation.

4 Performance

In this section we show performance results for the FastVM implementation on x86 and compare it to other state-of-the-art JVM implementations on x86. The second part presents improvements of the different single optimization methods which we have explained in the previous sections.

To compare the different JVM implementations on x86 we ran the SpecJVM98 [4] benchmark suite (large size) on a Compaq Deskpro machine (Pentium III 866MHz) with 256MB main memory running Linux 2.4.3. The heap size of the JVM is 128MB since we want to avoid as many side effects of garbage collection activity as possible during the measurements. Setting the heap size to 128MB eliminates most garbage collections.

For performance measurements we set up a Java wrapper program that invokes every single benchmark three times. At the end we compare the geometric mean as well as the best and the worst of all three benchmark times.

With a just-in-time compiler the first run usually takes the most time and any succeeding run is shorter since later runs invoke methods that have already been compiled to native machine code. We found that after three runs most JVM implementations do not improve much further by using already precompiled code. We also chose to restart the JVM after a single benchmark, so that benchmarks cannot take advantage of a method that has been compiled by a benchmark invoked earlier.

The FastVM uses a simple heuristic approach to decide whether a method needs to be compiled or not. During execution the FastVM counts how often a method gets invoked, and if the count exceeds an upper limit the methods gets compiled and optimized. In contrast, in a feedback-based compiler, optimization takes place only in the parts of the code that are frequently executed. To find out which parts the JVM needs to optimize, the compiler gets program profiling information as feedback from the runtime system. The advantage of feedback-based systems is that frequently executed parts of the program get well optimized. On the other hand, profiling and optimizing code imposes an additional runtime penalty and often code can be efficiently optimized using lightweight optimization methods as described in this paper.

4.1 Performance comparison of different JVM implementations

In this performance evaluation we compare the FastVM to Sun's JRE 1.3.1 (HotSpot client and server) and IBM JRE 1.3.0. These two JVM implementations are the leaders in the SpecJVM benchmark results and therefore a reasonable performance indicator.

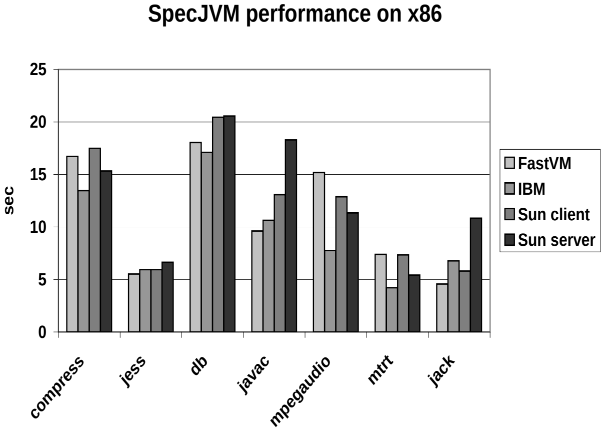

Figure 6: Average runtimes of SpecJVM on x86.

Figure 6 shows the average runtime of the four JVMs over three runs. The FastVM is the fastest JVM on jess, javac, and jack, and comes last in mpegaudio and mtrt. Mpegaudio and mtrt mainly use floating point operations, and so far we have not spent much effort in optimizing floating point operations apart from switching precision modes as explained in section 3.2. The FastVM is faster by 5-30% on some benchmarks and can be slower as much as 95% in mpegaudio.

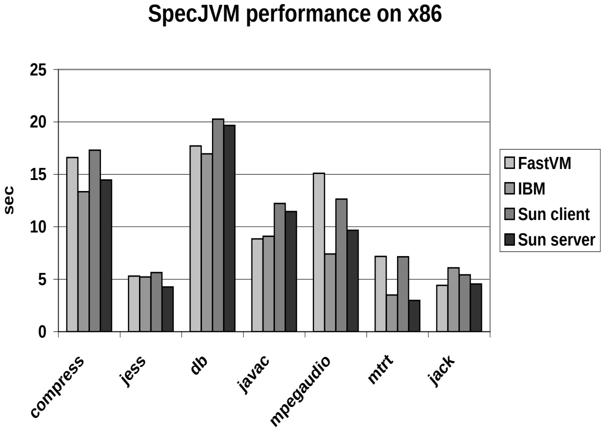

Figure 7: Best-case runtimes of SpecJVM on x86.

Figure 7 shows the best-case scenario. Here the feedback-based JVM code generators have an advantage since they can spend more time on optimizing code. However, the FastVM is still among the fastest JVMs for jess, jack, and javac. The lead of the FastVM is only around 2-3%.

In the worst-case scenario (Figure 8) feedback-based compilers, especially the Sun Hotspot server, lag behind since they spend a lot of time during the

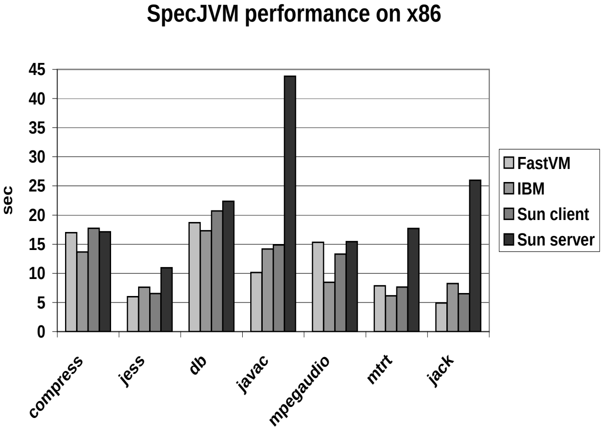

Figure 8: Worst-case runtimes of SpecJVM on x86.

first run in order to optimize native code. Here the FastVM has a lead in jess, javac, and jack. In javac the FastVM can be up to 39% faster than other implementations.

It is difficult to compare feedback-based code generators and non-feedback-based code generators. If a Java method that is difficult to optimize gets called frequently it is better to use a feedback-based compiler that applies heavy optimization techniques. On the other hand profiling code adds a runtime penalty, and therefore a simple approach for code optimization leads to a good performance. Furthermore, static compilers such as Swift [20] can be used to generate highly optimized native code in advance.

Simple code optimizations of a non-feedback-based code generator can compete with a feedback-based code generator, and these optimizations could even take place in the feedback-based system to improve runtime before any strong and time-consuming optimization starts. Furthermore, a simple heuristic approach for making decisions about whether or not to compile methods works reasonably well on the client applications of SpecJVM.

4.2 Optimizations for the FastVM on x86

This section investigates the effects of the different code optimizations we implemented for the FastVM on x86. These optimizations are register allocation, method inlining, and floating-point precision mode optimizations as we described in section 3. All three optimization techniques impose a negligible amount of additional compilation time to the VM.

4.2.1 Register Allocation

In this section we test the performance of the optimized register allocation scheme, and furthermore, how the benchmarks behave when we reduce the number of available registers.

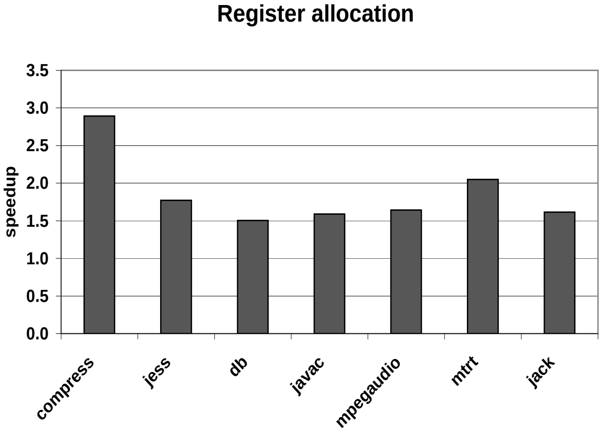

Figure 9: Benchmark speedup when enabling register allocation. The performance number represents the ratio between the measured runtime with the non-optimized compiler and the optimized compiler running register allocation only.

Figure 9 shows the overall performance improvement when we use register allocation instead of copying values from and to memory at each operation. The maximum speedup factor we achieve by enabling register allocation is about 3 for compress, but generally every benchmark profits from passing the arguments in registers.

The reason that compress gets more speedup than the other benchmarks is that it frequently runs sequential operations within one method that can be optimized well.

This effect becomes clearer in Figure 10 where we decrease the number of registers to a minimum of three and measure the performance. The y-axis indicates the speedup to the same code optimization that uses only three registers. In benchmarks that profit a lot from register allocation like compress we get a gradual speedup from three to seven available registers. In other benchmarks performance may even slow down a bit. The reason is that by using our simple local register allocation scheme we do not always pick the theoretically optimal allocation, and therefore performance may slow down by a nuance if the register allocation that uses only one register less was closer to the optimal solution. Benchmarks

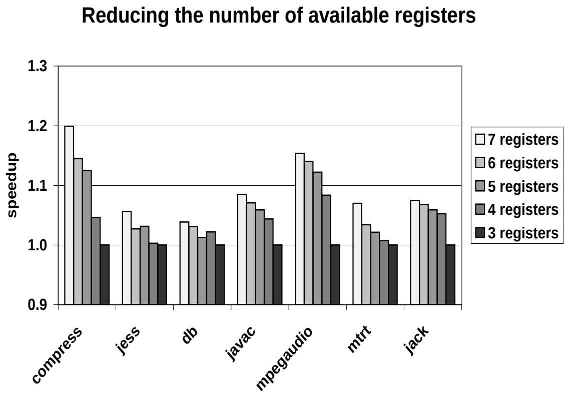

Figure 10: Reducing the number of available registers in the register allocator. The performance number represents the ratio between the measured runtime with four, five, six, and seven registers and the measured runtime with three registers.

that show only little improvement when we increase the number of available registers mostly profit from passing method arguments in registers.

4.2.2 Method inlining

This section shows the impact of inlining small methods. Before register allocation takes place we inline short methods before register allocation takes place. Figure 11 shows that inlining of short methods generally has only a marginal influence on performance for most benchmarks. Javac is the only exceptions with a speedup of 1.15.

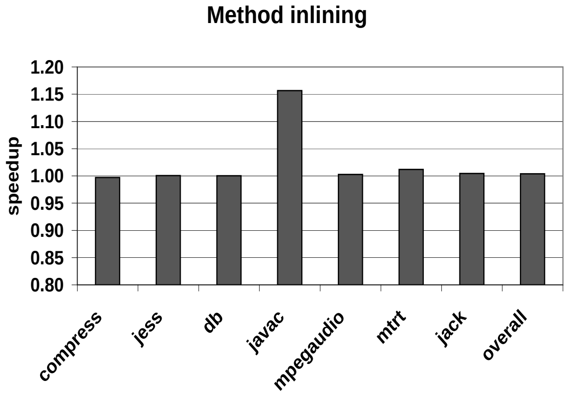

Figure 11: Benchmark speedup for inlining of short methods. The performance number represents the ratio between the measured runtime with method inlining enabled and the measured runtime with method inlining disabled.

4.2.3 Floating-Point precision toggle

When we set the floating-point precision at every single floating-point instruction explicitly, a significant number of memory operations gets added to the instruction stream that stalls the processor pipeline and thus degrades performance. By using a simple heuristic for determining whether a method uses single or double precision predominantly (as described in section 3), we are able to increase performance by a factor of up to 1.8 as shown in Figure 12.

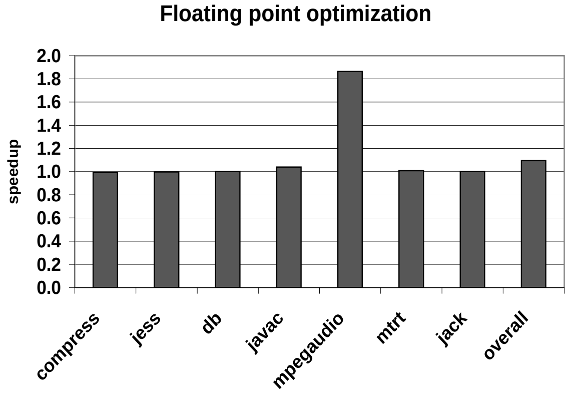

Figure 12: Benchmark performance when enabling optimization for toggling floating-point precision. The performance number represents the ratio between the measured runtime with floating-point mode optimization enabled and the measured runtime with floating-point mode optimization disabled.

In SpecJVM, only mpegaudio and mtrt use floating point operations frequently. However, in mpegaudio the compiler generates more register to register floating point operations that require the precision mode to be switched.

4.2.4 Overall performance improvement

Figure 13 shows the complete picture of the optimized code generator versus a non-optimized naive code generator.

Benchmarks profiting most from the optimizations are compress and mpegaudio. Each of them improves by a different optimization technique, which demonstrates the necessity of multiple optimization passes. The optimization techniques are simple and inexpensive and can be widely applied.

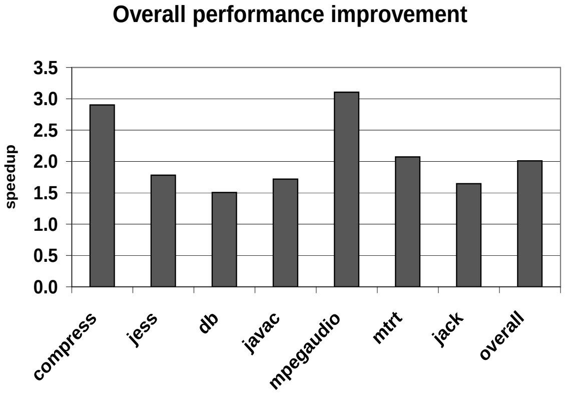

Figure 13: Benchmark speedup of optimized version using register allocation, floating-point optimization, and method inlining compared to the non-optimized implementation of the FastVM on x86.

5 Related Work

There has been a lot of work on writing JVMs in general [25], on JVMs for 64-bit RISC [16], and JVMs for x86 in particular [22]. Many of these JVMs are not well documented in the research literature because of competitive concerns. Two notable exceptions are IBM's Jalapeño JVM[8] and Intel's Open Research Platform(ORP)[7]. IBM's JVM is similar to ours in that it started out life as a RISC JVM on PowerPC, and has recently been ported to x86 [24]. Intel's JVM is of course native to the x86, but may experience similar porting pains when moving the IA64[2]. [17] describes the architecture of the HotSpot Virtual Machine. [6] describes the FastVM for Alpha that we ported to x86. In addition to just-in-time compilers several static compilers exist that generate native code from Java source code and use standard compiler backends for optimization [3, 5, 20] Unlike much of the published literature on x86 JVMs, our work focuses on the gritty, low-level design considerations required to make a JVM use a CISC instruction set effectively.

6 Conclusion

We have shown that it is possible to port a JVM implementation from a 64-bit RISC architecture to a 32-bit CISC architecture spending minimal effort and without losing much performance. The achieved performance is competitive with state-of-the-art Java Just-in-time compilers. Nevertheless, there were some pitfalls to get around, including floating-point precision mode, register allocation, and calling convention. On the other hand we had opportunities for further improvement in instruction selection. After all, from our experience it is generally simpler to generate efficient RISC code because of the complexity in addressing modes and instructions of CISC. Disregarding the architectural issues, more architecture-neutral compiler optimization techniques such as [19] and [15] can be implemented to further improve performance.

7 Acknowledgments

We thank the people at SRC, especially Rustan Leino and Mike Burrows, for help with implementation and insightful discussions on Java bytecode and architecture issues.

References

[1] *IA-32 Intel Architecture Software Developer's Manual.*

[2] *Intel Itanium^{TM}Architecture Software Developer's Manual.*

[3] NaturalBridge, BulletTrain.
`http://www.naturalbridge.com`.

[4] SpecJVM98.
`http://www.spec.org/osg/jvm98`.

[5] Toba.
`http://www.cs.arizona.edu/sumatra/toba`.

[6] The Compaq Fast Virtual Machine, June 1999.
`http://www.compaq.com/java/FastVM.html`.

[7] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 280–290, June 1998.

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

[9] A. Appel and L. George. Optimal spilling for CISC machines with few registers. *ACM SIGPLAN Notices*, 36(5):243–253, May 2001.

[10] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, May 1994.

[11] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, 1982.

[12] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, Oct. 1990.

[13] D. Detlefs and O. Agesen. Inlining of virtual methods. In R. Guerraoui, editor, *Proceedings ECOOP'99*, volume 1628 of *LNCS*, pages 258–278, Lisbon, Portugal, June 1999. Springer-Verlag.

[14] L. George and A. W. Appel. Iterated register coalescing. *ACM transacations on programming languages and systems.*, 18(3):300–324, May 1996. also in POPL'96.

[15] M. Kawahito, H. Komatsu, and T. Nakatani. Effective null pointer check elimination utilizing hardware trap. *ACM SIGPLAN Notices*, 35(11):139–149, Nov. 2000.

[16] A. Krall and R. Grafl. CACAO – A 64 bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[17] M. Paleczney, C. Vick, and C. Click. The Java HotSpot$^{\mathrm{TM}}$Server Compiler. In *Proceedings of First Usenix Java^{TM}Virtual Machine Research and Technology Symposium*, Apr. 2001.

[18] D. A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, Jan. 1985.

[19] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, Sept. 1999.

[20] D. J. Scales, K. H. Randall, and S. G. J. Dean. The Swift Java Compiler: Design and Implementation. Technical Report 2, Compaq Western Research Laboratory, April 2000.

[21] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, 1970.

[22] K. Shudo. shuJIT. `http://www.shudo.net/jit`.

[23] R. L. Sites and R. L. Witek. *Alpha AXP architecture reference manual.* Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, second edition, 1995.

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

[25] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioğlu, and E. Altman. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 128–138, Newport Beach, California, Oct. 12–16, 1999. IEEE Computer Society Press.