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

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

Experiences Porting the Jikes RVM to Linux/IA32

Bowen Alpern *Maria Butrico* *Anthony Cocchi*
Julian Dolby *Stephen Fink* *David Grove* *Ton Ngo*
IBM T. J. Watson Research Center
Yorktown Heights, NY, 10598

Abstract

This paper describes our experiences in porting the Jikes Research Virtual Machine from its first platform, AIX/PowerPC, to its second, Linux/IA32. We discuss the main issues in realizing both an initial functional port, and then tuning efforts to achieve competitive performance. The paper presents software engineering issues in building a portable runtime system and compilers, as well as specific optimizations to improve performance on IA32.

1 Introduction

In early 2001, developers of the Jikes^{TM} Research Virtual Machine (RVM) committed to releasing the software open-source by the end of 2001. At the time, the virtual machine ran on PowerPC^{TM} processors running AIX^{TM}.[1] In order to increase the utility of the open-source release, we decided to port the software to run on Linux© on the Intel 32-bit architecture (IA32), our second platform.

The porting activities focused on two milestones:

Functional port: establish a working version of the virtual machine on the second architecture as soon as possible, and

Performance port: achieve acceptable performance by the end of 2001.

This paper describes our experiences in working towards and reaching these milestones. The next section provides background on the Jikes RVM and an overview of the porting effort. Section 3 describes issues involved in getting the virtual machine up and running on a second platform. Section 4 presents the changes and innovations required to achieve good performance on that platform. Finally, section 5 concludes.

2 Overview

The Jikes RVM began life as the *Jalapeño virtual machine* in late 1997. The project had two design goals: 1) support high-performance Java^{TM} servers running on PowerPC multiprocessors under the AIX operating system, and 2) provide a flexible research platform "where novel virtual machine ideas can be explored, tested, and evaluated". Although it was written in the Java programming language, in the initial implementation portability was "*not* a design goal: where an obvious performance advantage can be achieved by exploiting the peculiarities of Jalapeño's target architecture ... we feel obliged to take it" [2].

Jikes RVM does not interpret bytecodes; rather it compiles each method to machine code and executes the machine code natively. In an adaptive Jikes RVM configuration, the *baseline* compiler performs the initial compilation of a method. Methods that are either frequently executed or computationally intensive are identified via a sampling mechanism and recompiled by the *optimizing* compiler [4].

The baseline compiler directly mimics the stack machine behavior of the JVM specification. The baseline compiler translates bytecodes to machine code quickly, but the resultant machine code typically runs slowly. The baseline compiler implementation de-

[1]There was also a partial port available for Linux/PowerPC, provided by collaborators from the University of Massachusetts.

pends heavily on the target instruction set architecture. Much of the work of a functional port lies in constructing a new baseline code generator, more or less from scratch.

The optimizing compiler expends more effort to produce high quality machine code for selected methods. The optimizing compiler implementation far exceeds the baseline compiler in size and complexity. However, most of the optimizing compiler does not depend on the instruction set architecture, reducing porting effort for a second architecture.

The Jikes RVM does not directly map Java threads of an application to operating system threads (POSIX pthreads). Instead, the system creates a virtual processor object for each pthread in use (normally one for each physical CPU). The Jikes RVM thread scheduler multiplexes the application's Java threads and the RVM's daemon threads onto these virtual processors.

Although the Jikes RVM is written in the Java programming language, it must perform actions (e.g. access registers and manipulate raw memory addresses) which cannot be expressed in the Java programming language [8]. The virtual machine provides a `VM_Magic` class to circumvent these restrictions [3]. The compilers do not translate the bytecodes of `VM_Magic` methods. Rather, the compilers recognize calls to these methods and inline custom machine code in place of the call.

The original implementation exploits the large PowerPC register set, with 32 general-purpose registers and 32 floating-point registers.[2] Adjusting to the register-scarce IA32 architecture presented a major challenge for this port. Differences in the instruction sets led to different calling and stack conventions for the two architectures. Writing in the Java programming language (explicitly big endian) shielded us from many, but not all endian problems[3] (For instance, with pre-loaded constants and when atomically updating bit and byte maps). Still, despite our initial indifference to the possibility of an eventual port, large portions of the Jikes RVM more or less worked on Linux/IA32 without modification.

As of February 1, 2002, the source code for the Jikes RVM itself contains approximately 203,000 lines of Java code, 18,000 lines of "meta" source files that are the inputs to several code generation tools, 6,000 lines of C++ code to interface with the operating system and to get the RVM started, and about 50 lines of assembly code to effectuate the initial transition from C++ to Java.[4] Most of this source code is independent of the target platform. The Java source files contain 162,000 lines of platform-independent code, 22,000 lines of PowerPC-specific code, and 19,000 lines of IA32-specific code. Approximately 6,000 lines of the "meta" source files are platform-independent, 3,600 are PowerPC-specific, and 8,400 are IA32-specific. The optimizing compiler comprises the largest subsystem of Jikes RVM, with 100,000 lines of Java source code; 78,100 are platform-independent, 14,200 lines are PowerPC-specific and 7,700 are IA32-specific. About 900 lines of the C++ operating system interface code are IA32-specific; 1200 are PowerPC-specific (these support both Linux and AIX). The assembly code is completely architecture-dependent. The *FullAdaptiveSemispace* configuration on Linux/IA32 represents a typical RVM build: it contains 821 Java classes comprising 225,000 lines of code (66,000 are machine generated).

3 Establishing functionality

A central aim of the functional port was to achieve a clean decomposition of the RVM into architecture-independent and architecture-specific pieces. The RVM was designed to be a *family* of virtual machines with different variants incorporating different memory management and compilation schemes as well as more minor peculiarities. It uses three mechanisms to achieve this variety: `static final` variables called *controls*, subdirectories of the file system, and a minimal preprocessor that allows limited conditional exclusion

[2]These registers are treated as dedicated, scratch, volatile (caller-save), or non-volatile (callee-save). Parameters are passed in volatile registers. Intermediate results can be accumulated in volatile or scratch registers. Non-volatile registers retain their value across method calls.

[3]PowerPC is "big endian" — the high-order byte of a word is at the lowest address within it — while IA32 is "little endian".

[4]In addition, the Jikes RVM source tree contains several tools used in the build process (5,600 lines) and the jdp debugger (33,300 lines).

of blocks of source text.[5] The port uses these mechanisms to support variants that run on Linux/IA32.

The remainder of this section discusses the specific issues that needed to be addressed to construct such variants.

3.1 IA32 Assembler

An IA32 assembler must contain a large body of complex and tedious code, full of odd special cases and strange idioms, simply because it must reflect the nature of the IA32 instruction set. For the IA32 port, this yields two obvious consequences: the two compilers should share a single assembler backend,[6] and this single backend should, in so far as is practical, be generated from specifications of the ISA to simplify attaining complete and correct coverage.

To accommodate the vastly different structures of the baseline and optimizing compilers, the shared assembler consists of two parts. The first consists of low-level code that generates binary code for specific IA32 instructions and operands; for example, it provides a function to emit a 32-bit add of a register and an immediate. It also has low-level support for other code generation miscellany, e.g. convenient support to generate forward branches. This low-level interface naturally suits the baseline compiler, which invokes it directly. The second part of the assembler processes the optimizing compiler's MIR (machine-dependent intermediate representation) instructions, and calls the appropriate low-level assembler routine for each one. This process involves examining the opcode and each operand of an MIR instruction, and, from them, determining which low-level assembler primitive to call.

Both levels of the assembler would be tedious and error-prone to write by hand. For the low-level functionality, we introduced a semi-automated approach. We first divided the IA32 instructions into equivalence classes based on instructions having similar legal sets of operands and similar generated bit patterns; for example, binary ALU operations such as ADD, SUB, AND, and XOR all fit similar formats. We then wrote a template for the low-level functions to generate each such equivalence class, and instantiated the template for each instruction in the class. This saved effort, and facilitated debugging, since an error in a template would occur in all its instructions and thus be more likely to show up in tests.

The higher-level assembler for the optimizing compiler is, if anything, even more complex; it consists of nested case statements depending on the operator of each instruction, and on properties of each of its operands. A stand-alone program generates this code fully automatically at build time, as follows. Text files holding tables define the MIR instruction formats. For each MIR operator, the program examines the low-level assembler for functions generating that opcode, and generates a table of the operand types those functions support. It then generates a tree of queries of the MIR instruction operands to determine which low-level function to call. The generator also inserts error-checking code to catch any instruction that cannot be assembled.

3.2 Baseline Compiler

A baseline compiler actually consists of two main components of roughly equal size: a target-independent portion that is responsible for generating GC maps and other descriptive information and a target-specific code generator. The heart of baseline compiler code generation executes a switch statement that emits code for each Java bytecode. Most of these 209 cases present simple exercises that are "solved" by emitting a dozen or fewer straight line assembler instructions. For those cases that are more complicated — for instance, the seven bytecodes that sometimes entail class loading — the internal structure of the case was imported from the PowerPC baseline compiler. We were careful to verify that each case (except for some of the "wide" variants) were exercised against a library of bytecode tests developed in conjunction with that original baseline compiler.

[5] To keep down instances of this rather ugly feature, we try to limit it usage to three cases: to define a control, to reference a class that would not otherwise be loaded, or to define or reference a field that would not otherwise be needed. In the 162,000 lines of platform independent Java code, there are 38 preprocessor blocks impacting 479 lines of source code that are related to the choice between AIX/PowerPC or Linux/IA32.

[6] The original PowerPC compilers each had their own assemblers.

3.3 Optimizing Compiler

Because writing an optimizing compiler represents a significant investment, from the beginning we designed the optimizing compiler for portability. A key aspect of this design divides the intermediate representation (IR) into three levels of operators:

1. **High Level IR (HIR)**: The HIR operator set is architecture independent and resembles the bytecode instruction set, although the IR uses a register transfer language in place of the bytecode stack abstraction.
2. **Low Level IR (LIR)**: The LIR operator set is architecture independent and resembles the instruction set of a typical RISC machine. The main difference between HIR and LIR is that complex HIR operators such as `new` or `call virtual` are expanded into the appropriate sequence of primitive operations.
3. **Machine Level IR (MIR)**: The MIR operator set is architecture-specific; with the exception of a few pseudo-operators that are expanded as part of final assembly, the MIR provides a one-to-one mapping between operators and the target ISA.

Translation from bytecodes to HIR, HIR optimizations, translation from HIR to LIR, and LIR optimizations are all architecture independent.

At system build-time, the builder generates classes representing the IR operators and helper functions from template files. Thus, the task of defining MIR operators corresponding to IA32 instructions consisted of defining new instruction formats and operator characteristics in two machine-dependent template files. The MIR operator definitions for IA32 consist of about 1300 lines in two files, defining 153 operators falling into 29 formats.

After defining the MIR, the main tasks in porting the optimizing compiler to a new ISA involve implementing the three major architecture-specific compiler phases: translation from LIR to MIR (aka instruction selection), register allocation, and final assembly. Each of these stages actually contains both architecture-independent and architecture-specific portions; we adopted a common idiom to define abstract classes that implement the shared functionality in terms of abstract methods defined by architecture-specific subclasses.

The instruction selection phase translates the machine-independent LIR into architecture-specific MIR. This translation phase partitions the dependence graph of each basic block into a forest of trees, and feeds the forest to a Bottom-Up Rewrite System (BURS)-based tree-pattern matching system [7]. Thus, the major porting task is to define the tree patterns and associated actions that serve as the input grammar to the BURS engine.

The register allocator maps the infinite set of symbolic registers onto a finite set of physical registers and spill locations. In addition, this phase generates prologues, epilogues, and calling sequences that respect the Jikes RVM and/or native OS calling conventions. The optimizing compiler relies on a variant of linear-scan [9] register allocation. The core of the register allocator should be machine-independent, but the implementation handles a fair number of low-level architecture-specific issues. Unfortunately, the original PowerPC implementation deeply intertwined the machine-dependent and machine-independent register allocator code. Rather than tackle a major refactoring problem with the old implementation, we decided to re-implement the register allocator from scratch for IA32. The new implementation cleanly separates machine-dependent and machine-independent code and includes expanded functionality and heuristics to suit both register-scarce and register-rich architectures. We have since back-ported the new implementation to PowerPC, so both platforms now share the machine-independent portion of the register allocator.

Final assembly generates executable machine code from the MIR and finalizes descriptive data structures such as exception tables and GC maps. The code for generating the descriptive data structures does not depend on the target architecture. Furthermore, as described above, the build system mechanically generates code that interfaces the MIR to the assembler. Therefore this stage introduced little work for porting.

The optimizing compiler also performs some optimizations on the MIR. The main MIR peephole optimizations, branch simplification and null check folding, do not depend on the architecture and worked immediately on IA32. We have not yet ported the instruction scheduler to IA32.

One difficulty in generating IA32 code compared to PowerPC is dealing with register restrictions imposed by the non-orthogonal instruction set architecture. Figure 1 shows some examples.

For some IA32 instructions, a particular operand *must* reside in a distinguished register. For example, in Figure 1b, the value of `t2` must reside in `ecx` at the shift instruction (`shl_acc`). We represent this information in the IR by explicitly assigning `t2` to `ecx` before the shift instruction during instruction selection for the shift (Figure 1c). The register allocator respects liveness for physical registers, and will further attempt to allocate `t2` to `ecx` to remove the copy operation. Similarly, in our calling convention `eax` holds the return value; we enforces this by inserting a copy from the symbolic return value register to `eax`, as shown in Figure 1c.

When BURS inserts a memory operand, the register allocator must respect further restrictions. For example, if the allocator were to spill `t0` in Figure 1c, this would force a later pass to move `t0` to a scratch register before its use in the `movsx` memory operand. For this reason, the linear scan spill heuristics consider a spill of a symbolic register used in a memory operand to be more expensive than a spill of a register operand that could be replaced by a memory operand representing the spill location.

Furthermore, for many instructions, the IA32 architecture dictates that only four of the eight general-purpose registers can hold 8-bit values. For example, in Figure 1c, `t3` can reside in the low word of `eax, ebx, ecx`, or `edx`; but not, for example in `ebp` or `edi`. The register allocator handles these types of restrictions with special case code, computing the restrictions as a pre-pass to register allocation.

3.4 Other VM Subsystems

The other VM subsystems — memory management, thread scheduling, locking, class loading, dynamic type checking, etc. — ported without change to Linux/IA32. Our use of Java as an implementation language shielded this code (some of it very low level) from any target dependencies. The only exceptions occur in about half a dozen sequences which use `VM_Magic` methods to perform direct loads or stores of byte quantities; the address computation needed to be parameterized on whether the target platform was big or little endian.

3.5 VM Conventions

Details of IA32's `CALL` and `RET` instructions forced major differences in stack and calling conventions. `CALL` pushes the return address on the stack and then branches to an indicated address. `RET` pops a return address off the stack and branches to it (discarding an indicated number of parameter bytes in the process).

On AIX the return address is saved at a fixed address in the *caller*'s stackframe. Using `CALL` effectively prevents this since the relative address of the stacktop off the frame pointer varies from call-site to call-site. To be conveniently accessible at all, the return address must be at a fixed address in the *callee*'s stackframe. Thus, on IA32 the return address starts a new stackframe.

This introduces a number of complications some of them minor.

First, the header is at the bottom of an AIX stackframe but the top of an IA32 stackframe (stacks growing down from high memory in both cases). This does not present a stack addressing problem: stack offsets are positive on AIX, negative on IA32.

Second, the ordering of fields in the headers of stackframes differs on the two architectures. This did not cause a problem since the header fields are always accessed with `static final` constants off the frame pointer. These constants differ on the two architectures.

Third, the size of stackframes, which is fixed (per method) and known *a priori* on AIX, varies from call-site to call-site on IA32. On

```
byte foo(byte[] a,
int x, int y){
 int i=x<<y;
 return a[i];
}
```

a)

```
t1 = shl_acc t2
t3 = byte_aload t0,t1
return t3
```

b)

```
ecx = t2
t1 = shl_acc ecx
t3 = movsx [t0+t1]
eax = t3
return eax
```

c)

```
ecx = [esp + 12]
edx = shl_acc ecx
eax = movsx [eax+edx]
return eax
```

d)

Figure 1: Examples of architectural register restrictions in the optimizing compiler IR. a) Java source code; b) IR fragment before instruction selection; c) IR fragment before register allocation; d) IR fragment after register allocation.

AIX it is natural to check for stack overflow when allocating a new stackframe; on IA32 this requires explicitly testing against a calculated upperbound of the eventual size of the stackframe.

Finally, stack-walking (e.g. during exception handling or garbage collection) was severely complicated by the fact that the return address was in the caller stackframe on AIX and the callee stackframe on IA32. The code difference was finally minimized by adopting the convention that the return address would be computed immediately before moving from callee to caller. On AIX this entails a redundant load off the contents of the callee's frame pointer. But, since stack-walking is not expected to be performance critical, we tolerate the pain in the interest of compatibility.

To facilitate the functional port, we initially added an extra word to the header of an IA32 stackframe. Whenever a method was called, the return address in the callee's stackframe header was copied into the new slot in the caller's header. This allowed immediate utilization of code that assumed the AIX convention. However, restructuring this common code so as to eliminate the need for this redundant header word was an ongoing porting headache for several months.

3.6 Synchronization

The PowerPC and IA32 architectures have different mechanisms for synchronizing multiprocessors.

The PowerPC architecture uses a weak memory consistency model. The PowerPC instruction set includes two synchronization instructions: `lwarx` and `stwcx`. The first of these loads a word from an address memory while setting a reservation for the executing processor on this address (any reservation another processor may have on the address is cleared as a side effect). The second stores a word at an address provide the executing processor holds a reservation on the address. The success or failure of this operation is recorded in a condition register.

The IA32 architecture enforces stronger memory consistency among the multiple processors. The IA32 instruction set has a compare-and-swap instruction (`cmpxchg`): the value at a specified address is compared to a second value, if the two are equal, a third value is stored at the address. The success or failure of the operations can be obtained from a machine register. A locking prefix byte to this instruction makes its behavior appear atomic to any other processors.

The initial PowerPC implementation of Jikes RVM used methods of the `VM_Magic` class to directly emit `lwarx` and `stwcx` instructions. Rather than create architecture-specific classes for all the methods that called these methods, we designed `VM_Magic` methods that could be used on either architecture but whose implementations were architecture specific. We developed a synchronization idiom whereby attempting to perform a synchronized write first requires obtaining the old value using a *prepare* operation and then issuing an *attempt* operation which takes both the old and a new value as well as the raw address. Higher level pseudo-primitives, such as `fetch-and-add`, were implemented in Java using this discipline and provided as runtime utilities.

The `int VM_Magic.prepare()` method takes a raw address as its only parameter. On the IA32 architecture this is implemented as an ordinary load instruction. On the PowerPC it is a `lwarx` instruction.

The `boolean VM_Magic.attempt()` method takes a raw address and two (32 bit) values as parameters. On the IA32, it causes the corresponding atomic compare-and-swap to be executed. On the PowerPC, the second parameter is ignored, while the other two are used by a `stwcx` instruction.[7] In either case, the success of the operation is returned as the result of the method.

3.7 Operating system issues

The thin layer of C/C++ code that interfaces between the RVM and the operating system was ported from AIX to Linux either without change or by replacing the invocation of a system function on AIX with its Linux equivalent. We rewrote for IA32 less than half a page of assembly code which transfers initial execution to the RVM image. The only system service which proved troublesome was the Linux POSIX thread (pthread) library where we had troubles both with the earlier implementation of the library and with differences between the AIX and later Linux implementations.

We started this work on the 2.2 Linux kernel and associated libraries. For this Linux release, the pthread library computes the identity of a thread as a function of the ESP (stack pointer) machine register. Since the RVM virtual processors multiplex several Java threads, each with its own stack (none of them beginning on the large power of two boundary expected by the library), and since our implementation used the ESP register to address the stack of the running thread, we cannot run with this pthread library. (On 2.2 Linux, the RVM only runs with a single virtual processor.)

Fortunately, the 2.4 Linux/IA32 release (and some earlier development releases) resolved this problem.[8] We also found some differences in the behavior of POSIX threads on AIX and Linux. On AIX, one process may have many pthreads. On Linux, each pthread looks and acts like a separate process. One area where this difference manifests is the behavior of the system with respect to signal handling. The RVM uses two signal handlers which are not reentrant and cannot execute simultaneously. On AIX, we specified that these signals were to be masked for the duration of the signal handler, and as expected pending signals wait until earlier executions of a signal handler finish before executing. Linux did not follow this behavior, so the Linux signal handlers need to provide their own explicit synchronization.

4 Improving performance

The functional port of Jikes RVM to Linux/IA32 was mostly complete by the end of August 2001. The initial port achieved 35% of our performance target. Over the course of the next five months, Jikes RVM Linux/IA32 performance more than doubled to reach 95% of the IBM 1.3.0 DK. Figure 2 shows how the performance[9] increased over this time period, and Figure 3 shows relative performance for each of the SPECjvm98 benchmarks[10] as of February 2002. The results show that Jikes RVM performance is competitive overall, but lags behind the IBM DK on the two floating-point codes (`mpegaudio` and `mtrt`) and on `compress` (an array-based set of tight nested

[7]On the PowerPC architecture, the baseline compiler expends unnecessary extra work pushing the unused middle parameter onto the stack. However, the optimizing compiler identifies the computation as dead and eliminates it. Thus, the optimizing compiler produces code for the `prepare` and `attempt` primitives that is as efficient as architecture-specific primitives.

[8]It remains in the 2.4 Linux/PowerPC release.

[9]The Jikes RVM FastAdaptiveSemispace images, used for Figures 2 and 3, gives the best overall performance due to feedback-directed optimization and delayed compilation effects [4]. However, to reduce the impact of timer-driven non-deterministic actions, the remainder of this paper reports results using a (non adaptive) FastSemispace image which compiles each method the first time it is invoked at optimization level O2. In all cases, two virtual processors and a 400 MB heap were used. All runs are on a 4-way IBM Netfinity with 700MHz Pentium^{TM} III processors and 3 GB of memory. The Linux installation is a customized RedHat version with a 2.4.12 kernel and GNU Libc version 2.2.4; the libc and kernel versions support the version of LinuxThreads that uses the GS segment register for thread-local storage enabling it to support Jikes RVM's user-level multithreading mechanism.

[10]These benchmarks were developed by the Standard Performance Evaluation Corporation [6]. The performance numbers reported in this paper are the best run of 5 on each individual SPECjvm98 benchmark. These runs do *not* conform to the official SPEC run rules, so our results do not directly or indirectly represent a SPECjvm98 metric, and are not comparable with a SPECjvm98 metric.

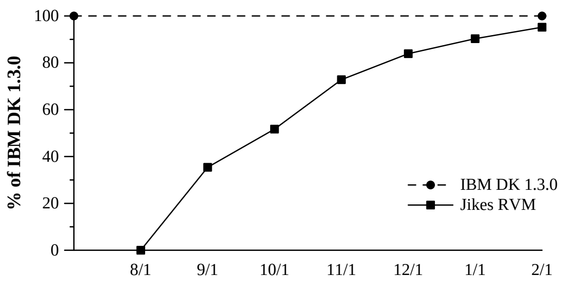


Figure 2: Monthly performance of Jikes RVM on the SPECjvm98 benchmarks as a percentage of the performance of the IBM 1.3.0 DK for Linux/IA32 from August 1, 2001 through February 1, 2002.

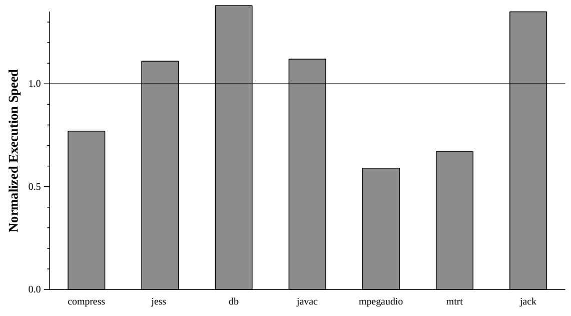


Figure 3: Performance of Jikes RVM on the individual SPECjvm98 benchmarks as a percentage of the performance of the IBM 1.3.0 DK for Linux/IA32 on February 1, 2002.

loops).

This section of the paper describes the main IA32 specific enhancements made to improve Linux/IA32 performance.[11] The first section discusses performance-motivated changes to the VM's register conventions. The next two sections describe enhancements to the optimizing compiler's instruction selection and register allocation phases. Generating even mediocre IA32 floating point code was challenging; section 4.4 describes some of the alternatives we explored.

[11]During the six month period shown in Figure 2 several new platform independent optimizations were added to the optimizing compiler and existing optimizations were enhanced. Although these contributed to the performance improvements shown in the graph, most of the gain was caused by IA32 specific improvements (during the same period AIX/PowerPC performance only improved by about 10%).

4.1 VM Register Conventions

The initial functional port followed the PowerPC implementation in dedicating four (of the eight IA32 "general purpose") registers: a pointer to the currently executing stack-frame (FP), a pointer to a region of static data (JTOC), an indirect pointer to thread-local storage for the current Java thread (TI), and a pointer to pthread-local storage associated with the current virtual processor (PR). In addition, it dedicated `esp` as a stack pointer, leaving only three registers for general use. It soon became apparent that good performance would hinge in part on freeing up some of these dedicated registers.

We first reclaimed the TI and JTOC registers. Instead of dedicating registers to hold these values, the system now caches these values in the pthread-local storage accessed by the PR register. This strategy adds an extra indirection to access Java thread-local and static storage.

Next, we reclaimed the frame pointer register. This change required more intrusive system modifications. As with TI and JTOC, the system caches the current frame pointer register in pthread-local storage. Each compiler was modified to maintain this frame pointer field in the prologue and epilogue sequences. The C trap handler that handles a hardware trap or software interrupt was modified to acquire the frame pointer indirectly through the PR register instead of from a register. Additionally, each compiler was modified to manage stack storage solely off the stack pointer (`esp`), with no reliance on the frame pointer. There were also some complications with IA32 baseline compiler assumptions for low-level details of stack resizing and GC map computation, which fall beyond the scope of this paper.

The current system has two dedicated IA32 registers: SP (`esp`) and PR (`esi`). The remaining six GPR registers are available for register allocation. We have considered reclaiming PR as a non-volatile by using an IA32 segment register as a pointer to pthread-local storage. Three considerations have so far deterred us from making the attempt. First, since the segment registers can't address arbitrary words in memory, it would be difficult to encode the structures that they point to a ordinary Java objects. Second, we do not feel certain that Linux does not, or, more importantly, will not in the future use any particular segment register for its own purposes. Third, we are concerned that on some IA32 implementations segment register access might be prohibitively slow.

4.2 Instruction Selection

The heart of the instruction selection phase relies on a BURS-based tree-pattern-matching system. We have extended `iburg` [7] to generate Java code (instead of C) and work on a general dependence graph. The key idea is to partition the dependence graphs into a forest of expression trees based on their register-true dependencies. The BURS pattern matching system then processes each tree in the forest to perform instruction selection and emit code, one tree at a time, in an order that respects the inter-tree ordering constraints encoded by the original dependency graph [5, 10].

In the initial port, we defined a "bare bones" grammar that described a straightforward translation of the 124 LIR operators into MIR operators using 142 rules and 7 non-terminals. As the performance work progressed, the grammar tripled in size. The current grammar includes additional patterns for optimizing floating point computations and conditional branching, exploiting memory operands, and other miscellaneous enhancements such as recognizing complex addressing modes, exploiting special instructions such as `LEA`, `TEST`, `INC`, etc., and avoiding needless sign extension of byte/short loads. Table 1 reports the contribution of each group of enhancements to the size of the rules.

In addition to extending the grammar as described above, we also enhanced our BURS driver with heuristics to reduce register pressure. We label each tree node with an estimate of the number of live values required to compute it, using the algorithm from section 9.10 of the Dragon book [1]. The BURS driver uses this numbering to choose which child node to emit first when generating code for a given tree and to select from the set of ready trees[12] which tree to emit next. In both cases, choosing the node/tree with the largest number of registers first tends to reduce the number of simultaneously live values and thus reduce register pressure. As reported in section 4.3, this heuristic made a small but measurable difference in overall performance and was extremely simple to implement.

[12]A tree is ready if all of the trees on which it is dependent have already been emitted.

Benchmark	% Speedup
compress	1.9
jess	7.2
db	1.4
javac	-0.5
mpegaudio	52.3
mtrt	5.1
jack	8.0
geo. mean	9.7

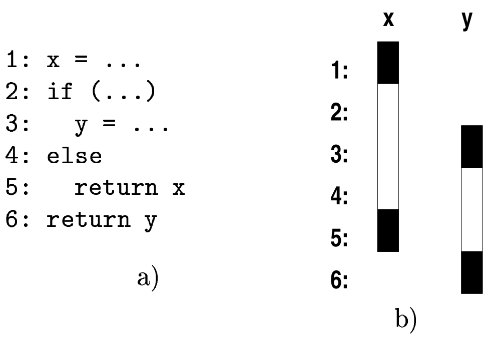
Table 2: Percentage speedup obtained by Complete rules over the initial Basic rules.

While the PowerPC architecture supports three-operand ALU operations (`a=b+c`), IA32 supports two-operand ALU operations(`x+=y`). The IA32 compiler converts the three-operand LIR to two-operand form in a pre-pass to BURS, which tripped some subtle issues. Initially, this pre-pass took obvious actions to avoid introducing useless move instructions. For example, it would transform `a=a+b` to `a+=b`. In other cases, it would use local liveness information to avoid inserting moves. For example, if `b` is dead after `a=b+c` and this is the only definition of `a`, then the pass would emit `b+=c` and the replace uses of `a` with uses of `b`. This second optimization tripped a subtle difficulty; it can hinder instruction selection of other expression trees within the basic block by introducing additional inter-tree anti and output dependencies and by extending the live range of `b` beyond the current basic block. Thus, in some cases it is actually better to translate `a=b+c` into `b+=c; a=b` and rely on the register allocator to coalesce away the move instruction.

Table 2 shows the performance improvements obtained by adding all of the enhancements to the basic grammar. The most important enhancement was adding rules to exploit the floating point stack (`mpegaudio`, `mtrt`), but the other enhancements also resulted in modest gains. The impact of instruction selection and register allocation on floating point performance is explored in more detail below.

Description	Number of Rules	Number of Non-terminals
Basic	142	7
Floating Point Stack	90	2
Conditional Branches	42	6
Memory Operands	170	8
Misc. Other Enhancements	144	2
Complete	588	25

Table 1: The Basic grammar was enhanced with 446 rules and 18 non-terminals to support optimizations in instruction selection. BURS must recognize 124 LIR operators.



```
1: x = ...
2: if (...)
3:   y = ...
4: else
5:   return x
6: return y
```

Figure 4: Example of linear scan live intervals with holes.

4.3 Register Allocation

The optimizing compiler relies on a variant of linear-scan [9] register allocation. To give the register allocator more freedom, we implemented a variant of linear scan that deals with holes in live ranges. Consider Figure 4. The basic linear-scan algorithm would not allocate x and y to the same physical register, as their live intervals (denoted by the rectangular bars) overlap. However, our enhanced algorithm represents the live intervals with holes, as represented by the black-shaded areas in the Figure. As a result, our allocator could allocate x and y to the same physical register.

Traub et al. [11] previously described a linear scan variant to deal with holes. Our approach differs in two ways.[13] First, we perform the intersection of two sparse live intervals, rather than insisting on perfect nesting of one interval within another. Although this introduces super-linear complexity to the algorithm, we do not believe this causes major problems in practice.

[13]We have not performed an apples-to-apples comparison of our algorithm compared to Traub et al.

Secondly, Traub et al.'s algorithm splits live ranges on the fly, with a post-pass clean-up phase to reconcile differences. In contrast, our algorithm marks certain symbolic registers as spilled. A post-pass clean-up phase deals with the spills. If an instruction uses a spilled register, the clean-up phase either introduces a memory operand referring to the spilled memory location, or moves the spilled value into a scratch register. On PowerPC, the original register allocator reserved three registers for use as scratch, so finding a free scratch register was almost always trivial. On IA32, we did not reserve any scratch registers, so the register allocator must create scratch registers upon demand.

Figure 5 details the algorithm for dealing with spilled values. The algorithm makes one pass over the statements. As it progresses, the algorithm keeps track of which symbolic register values are cached in each scratch register. Before each statement, the algorithm vacates any scratch registers which are needed for the next statement. Then, the algorithm processes a statement. It attempts to replace spilled symbolic registers with memory operands representing the spill location. If this is infeasible, the algorithm chooses a physical register as a victim to be vacated, so the victim can be used as a scratch register. The victim will continue to cache the symbolic value until either the physical register is needed for a future statement, or the victim is chosen to cache a different symbolic register. Scratch register mappings are not maintained across basic block boundaries; all victims are vacated at block exits.

As a side effect of vacating and introducing scratch registers, the code updates stack maps required for type-exact GC.

```
for each statement $s$ do
    if $s$ can leave the basic block via a call, jump, fall-through, or exception then
        vacate cached value and restore original value for each scratch register before $s$
    vacate cached value and restore original value for any scratch register used by $s$
    for each spilled symbolic register $r$ in $s$ do
        if $r$ is currently cached in scratch register $p$ then
            replace $r$ with $p$ in statement $s$
        else if $s$ needs a scratch register for $r$ then
            choose a scratch register victim $p$ to hold $r$ in $s$
            vacate current value of $p$
            cache value of $r$ in $p$
            replace $r$ with $p$ in statement $s$
        else replace $r$ with a memory operand representing $r$'s spill location
    done
done
```

Figure 5: Algorithm for dealing with spilled values after linear scan.

With the IA32 limited register set, spills are common and have a huge impact on performance. Subsequent to the initial functional port, we introduced several heuristics and optimizations to improve performance.

We now evaluate the following heuristics to reduce register pressure. We provide only a high-level overview of each heuristic; consult the open-source code for more details.

Intelligent scratch victim selection The initial implementation of the algorithm in Figure 5 chooses a victim arbitrarily. Furthermore, the initial implementation does not use liveness to determine whether the victim needs to be vacated and restored. This heuristic uses liveness computed during linear scan and attempts to choose a victim that does not currently hold a live value. Furthermore, this optimization uses the same information to avoid vacating and restoring dead values.

Smarter linear scan spill heuristic When facing a spill situation, the original linear scan implementation chooses a symbolic register to spill at random. This heuristic estimates the cost of spilling based on appearances of the register, weighted by loop depth, and chooses spill candidates accordingly.

Register-pressure-aware BURS This optimization introduces a heuristic into instruction selection to attempt to generate code in an order to minimize overlapping live ranges. When faced with multiple expression trees which can be emitted in any order, this heuristic chooses the tree that consumes the most register values.

Global Coalescing We implemented a separate pass to coalesce registers; basically, if there is a `MOV x = y`, if the live ranges of `x` and `y` do not overlap, then we replace all appearances of `y` with `x`.

Figure 6 shows the marginal performance improvement gained by enabling each of these four optimizations cumulatively, compared to performance with none enabled. Thus, the bar labeled "Scratch Victim" enables intelligent scratch victim selection; the next bar "LS Spill" further enables the linear scan spill heuristic, etc.

Altogether, the register pressure heuristics improve performance on average by 15%. The results show that the spill heuristics and scratch register selection heuristics are most effective across the board. Anomalously, the scratch victim heuristic hurts `mpegaudio`; we currently have no explanation for this. Instruction selection re-ordering has a minimal impact, slightly improving `compress` and `db`. Coalescing is usually insignificant; except it causes a substantial improvement for `mpegaudio`. `Mtrt` degrades as we add more optimizations; however, the floating-point results here should be taken with a grain of salt, since

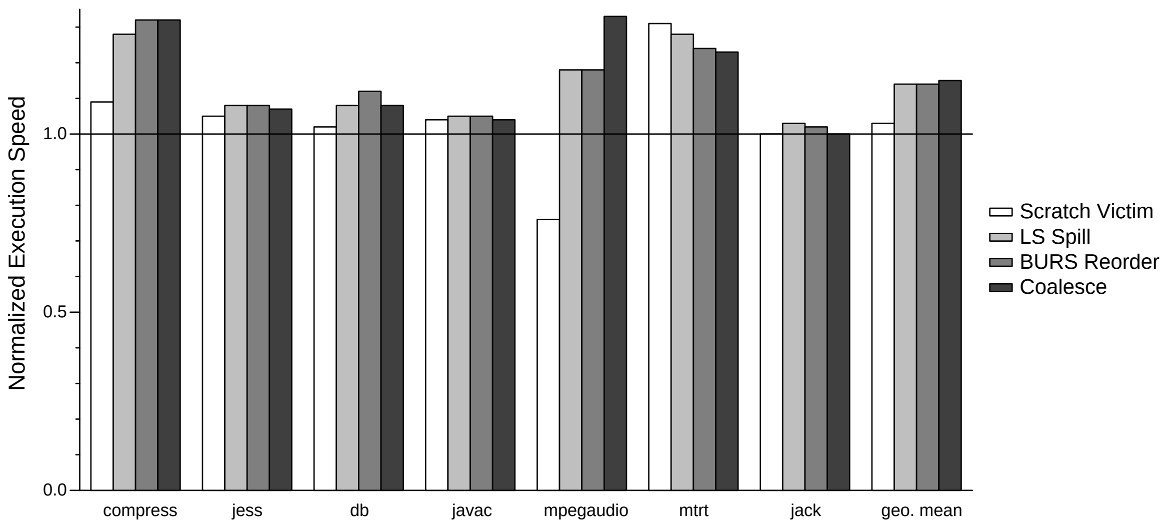


Figure 6: Relative performance gains by enabling, cumulatively, four optimizations designed to reduce register pressure.

Figure 3 shows that Jikes RVM IA32 floating-point performance is mediocre, even with all optimizations enabled.

4.4 Floating point

The IA32 architecture provides an abstraction of a floating-point stack, a sharp difference from the flat floating-point register set of the PowerPC.

In the original functional port, in order to minimize changes to the system, we treated the floating-point stack locations as independent physical floating-point registers. During instruction selection, BURS treated symbolic floating-point registers just like symbolic integer registers. The linear scan register allocator allocated the symbolic floating-point registers to seven floating-point stack locations, as if these were seven physical registers. The eighth stack location was reserved for use as a scratch register downstream, in order to generate code that moves values between stack locations and to memory.

This original scheme had the advantage that the linear scan allocator had full freedom to allocate the stack locations using global analysis. However, this scheme has a severe drawback. Since the BURS instruction selection saw only orthogonal symbolic floating-point registers, it could not generate code to exploit the stack operations available in the IA32 instruction set.

An alternative scheme could allow BURS to generate floating-point stack code freely within a basic block. With this scheme, instruction selection could use the floating-point stack resources freely within a basic block. However, since the linear scan algorithm does not understand stack locations, it could not allocate values to stack locations across basic blocks. In effect, all register allocation would be constrained to a single basic block, spilling values to memory across basic blocks.

We chose a hybrid scheme. We give instruction selection the freedom to place a floating-point value either on the floating-point stack or in a symbolic floating-point register. The register allocator allocates symbolic registers to free stack locations. Note that if BURS allocates a value to a floating-point stack location, that stack location is not available for use by the register allocator. We model this by inserting dummy def and use instructions for physical stack locations reserved by instruction selection.

Table 3 compares performance on the two floating-point SPECjvm98 codes. The Table shows that each technique helps `mpegaudio`, but shows an anomaly where inter-block register allocation hurts `mtrt`. Our initial functional port used only the “RA” register allocation strategy, as this option most closely matches the extant PowerPC port. Later we also added the BURS floating-point stack code

	None	RA only	BURS only	Both
`mpegaudio`	1	1.548	1.544	1.957
`mtrt`	1	0.668	1.251	1.181

Table 3: Performance comparison of alternative floating-point code generation strategies (speed normalized to "None"). "RA" allows inter-block register allocation, while "BURS" allows intra-block generation of floating-point stack code for expressions.

generation. We didn't seriously consider the other two possibilities, but include them to enable comparisons.

Although we have improved RVM floating point performance compared to the initial functional port, performance still lags behind the IBM product DK. We still face the two anomalies reported for floating-point performance: recall that the smart scratch victim selection hurts `mpegaudio` and floating point register allocation degrades `mtrt`. We have not yet investigated these anomalies, and we hope to improve Jikes RVM floating-point performance in the future.

5 Conclusions

We have described our experiences porting a high-performance virtual machine to its second architecture. In the process, we have endeavored to enforce a clean separation between architecture-dependent and architecture-independent code. As a result, we expect that a port to a third 32-bit architecture would be much easier.

A major issue only partially addressed is migration to a 64-bit architecture. Recently, a `VM_Address` type has been introduced to statically isolate code that manipulates raw addresses in the VM implementation (originally Jikes RVM used the type `int` to represent raw addresses, making it impossible to statically isolate such code). However, work still remains to find and update all code in the VM implementation that assumes that reference/address values are four byte quantities.

Substantial work remains to be done in the optimizing compiler. Clearly, opportunities remain to improve performance on both PowerPC and IA32. On IA32 in particular, floating-point performance still lags behind the IBM DK 1.3.0. We have recently implemented live range splitting to help further reduce register pressure, but have not yet seen substantial performance improvements. Further optimization passes may also help; one optimization not yet enabled is instruction scheduling, which could help reduce register pressure and/or increase instruction-level parallelism. Also, it may be an interesting research topic to determine how to constrain HIR optimizations, such as SSA conversion and redundancy elimination, to reduce register pressure.

Since the open-source release in October 2001, we are aware of several efforts by academic and other researchers to help address some of these concerns. We understand that significant progress has been made porting to IA32 on Win32, and to 64-bit PowerPC. We hope these and other enhancements will make their way into the open-source code base, so that Jikes RVM will further mature as a platform for programming language implementation research.

Acknowledgments

We thank all the members of the Jalapeño team who have contributed to the code base over the past four years. We especially thank Vivek Sarkar and Mauricio Serrano, who contributed key ideas in the portable design of the optimizing compiler. (Mauricio also warned us that we would get in trouble in a port to IA32 if we associated the return address with the caller's stackframe. He was right.) We additionally extend special thanks to Mike Hind and Mark Mergen for their efforts in support of this work. Also thanks to Dave Hansen for making the `iburg` tool available, and to Steve Blackburn for his help and feedback.

Jikes RVM is available under the Common Public License (CPL) from `www.ibm.com/developerworks/oss/jikesrvm`.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. Flynn Hummel, D. Lieber, V. Litvinov, T. Ngo, M. Mergen, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalepeno virtual machine. *IBM Systems Journal special issue on Java performance*, 39(1), 2000. (see also http://www.research.ibm.com/jalapeno).

[3] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. Flynn Hummel, D. Lieber, T. Ngo, M. Mergen, J. Shepherd, and S. Smith. Implementation of Jalepeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

[4] M. Arnold, D. Grove, S. Fink, M. Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN* Conference on Object-Oriented Programming Systems, Languages, and Applications *(OOPSLA 2000)*, Minneapolis, MN, Oct. 2000. Also published as ACM SIGPLAN Notices, volume 35, number 10.

[5] M.G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M.J. Serrano, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, June 1999.

[6] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/, 1998.

[7] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering, a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[8] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996.

[9] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999.

[10] Vivek Sarkar, Mauricio Serrano, and Barbara Simons. Register-sensitive selection, duplication, and sequencing of instructions. In *ACM International Conference on Supercomputing*, June 2001.

[11] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *SIGPLAN '98 Conf. on Programming Language Design and Implementation*, pages 142–151, May 1998.