

A Space-Aware AMD64 Port of Jikes RVM

Arun Thomas

at4a@cs.virginia.edu

Dan Williams

dww4s@cs.virginia.edu

Abstract

As computers attempt to work with larger and more complex datasets, the need for 64-bit computing is becoming more acute. Many applications, such as video editing and large scale databases, are reaching the limits of addressable memory in 32-bit computers. These new 64-bit architectures are dependent on compilers to produce equivalent or better code. Here we look at the issues in porting the Jikes RVM to the AMD64 architecture, the apparent 64-bit successor to the 32-bit x86 architecture. We explore various design decisions made in porting to AMD64 and provide recommendations for those who wish to port Jikes RVM to subsequent architectures.

One of the primary challenges in increasing processor word size is the corresponding increase in code size and memory usage due to the larger pointers and data. Techniques to save memory space in the JVM are hampered by the JVM spec, which favors 32-bit implementations. By bending the JVM spec, we have implemented a memory saving optimization to the AMD64 port Jikes RVM. We have also reduced memory space by using equivalent 32-bit instructions when possible. These two memory optimizations have saved as much as 7% on the SpecJvm98 and Java Grande benchmarks.

Keywords Porting, Code Size, Stack Size, Compilers, 64-bit, AMD64, x86-64, JVM

1. Introduction

64-bit computing is becoming increasingly prevalent. A number of application domains have been driving the push for 64-bit computing. For example, large databases and video editing applications require large address spaces, so that more data can be stored in memory, thereby improving performance. A 32-bit architectures can only address

4 gigabytes of memory directly. Any application needing more than 4G of addressable space must rely on operating system hacks or other tricks to achieve their goal. Data warehousing applications require large data movements to improve throughput. A 64-bit architecture allows for fewer data movement operations, since a full 64-bits can be accessed. Scientific computing applications perform operations on large, high precision numbers. A 64-bit architecture has native support for such operations, thereby increasing performance of computationally intensive simulations.

However, with the advantages of 64-bit systems comes a number of costs. Application size and data size can increase dramatically when moving from 32-bit systems to 64-bit systems. The code size increase can also have effects on performance; larger code size means more instruction cache pressure, and more conflict misses.

In terms of practical computing environments used today, AMD64 is the heir-apparent to the popular IA32 (x86) architecture. While Intel's IPF (Itanium) processor is still being manufactured, it looks as though it will soon be voted off the island. There are reports that Intel is abandoning Itanium for the AMD64 (Intel rebranded EM64T) architecture. Additionally, AMD is producing 64-bit desktop CPUs in the Athlon 64. There is a fair chance that a 64-bit CPU could be *America's Next Top Desktop CPU Model*.

Consequently, it is desirable to port systems to the AMD64 architecture. We chose the Jikes RVM as our porting target, as it is a popular, freely available, open-source platform for compiler, virtual machine, and memory management research. As a result, the Jikes RVM system provides an ideal foundation for research into all manner of 64-bit computing issues. We, of course, research memory space optimizations in this work.

The focus of this work is on the baseline compiler, which translates Java bytecode instructions into native machine code. Jikes RVM also includes an optimizing compiler, that can dynamically optimizing routines. However, the optimizing compiler is complex and would require more effort to port. The baseline compiler provides enough functionality for us to examine memory space issues. Even though the baseline compiler does not use any performance optimizations, space optimizations are still important to it, since all

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

code used by the RVM is initially compiled by the baseline compiler. Future work will focus on the optimizing compiler.

A port of the Jikes RVM baseline compiler is particularly interesting for a number of reasons. First, the Jikes RVM has been ported incrementally. Each port of the Jikes RVM has been done by a separate team, possibly with different goal, which makes the porting process more challenging than a system that has been build from the ground up for portability (e.g. GCC). Secondly, the interactions between Jikes RVM and AMD64 are complicated by the requirements of the JVM spec, which seems to favor 32-bit platforms. Finally, Jikes RVM is an interesting platform because of its significant space considerations. The baseline compiler compiles all classes before the optimizing compiler. This means that space is of concern to the baseline compiler, even if performance is not its highest priority.

Our contributions in this work are as follows:

- Design and implementation of the AMD64 port of the baseline compiler, including verification of bytecodes emitted by the compiler.
- Analysis of porting and debugging the Jikes RVM, including recommendations to make future ports easier and more effective.
- Size optimizations of the baseline compiler, reducing code and stack space compared to traditional implementations of the JVM.

The rest of the paper is organized as follows. Section 2 covers work that provided a foundation for ours. Section 3 gives an overview of the AMD64 architecture, as well as details of the Jikes Bootstrapping process. Section 4 describes our porting techniques, and recommendations for easing the porting of Jikes RVM to new architectures. Section 5 details the implementation of our memory space improvements for Jikes RVM on AMD64. Section 6 describes our experimental setup and results. Section 7 describes future work for the project. Finally, Section 8 concludes the paper.

2. Related Work

Much work has been done on the correct design and implementation of portable compilers [6]. However, there is relatively little literature on porting compilers for architectures that are essential extensions to currently implemented architecture. There are unique challenges when working with a compiler that was built for a single system, then ported to more and more systems as its popularity grows.

Our porting efforts targets the Jikes RVM architecture, based on the Jalapeno adaptively optimizing virtual machine [3]. Jikes RVM currently supports PowerPC on AIX, PowerPC64 on AIX and Linux on IA32. It was initially developed to run on the PowerPC architecture. Alpern et al. subsequently added support for the Intel IA32 architecture on Linux [1]. Then, Kyrlykov further extended Jikes RVM

to support the 64-bit extensions to the PowerPC architecture [12]. Each of these ports represents a significant effort, and of course as more architectures are added, the base has become fundamentally more portable.

The ground work for the AMD64 port of Jikes RVM was done as part of Frederik Deschrijver's Master's thesis work [4]. His work was primarily in the assembler and bootImageWriter, described in section 3. Deschrijver did some work on the baseline compiler, though few of the implementations of the Java bytecodes were tested.

Currently the Sun JVM, the IBM JVM, and the Kaffe JVM have been ported to AMD64. However little public information is available on the design and implementation of these ports. GCC, the GNU compiler collection, has also been ported to the AMD64 architecture. The GCC porting effort is detailed in [9]. Our stack space optimization detailed in Section 5 is built on Vensterman et. al.'s work [14]. They initially proposed the optimization, but did not implement it.

3. Background

Many of the challenges presented in this paper relate to the complicated interactions between the AMD64 architecture and Jikes RVM. This section summarizes the features of these systems that are most relevant to a successful port of Jikes RVM to AMD64.

3.1 AMD64 Architecture

The AMD64 architecture [2] is a 64-bit ISA, based heavily on Intel's IA32 (x86) architecture. This architecture extends the x86 architecture with a new "long mode." (Legacy mode is also provided for strict compatibility with IA32). Long mode is designed to run with a 64-bit operating system. Long mode is itself split into two sub-modes: 64-bit mode and compatibility mode. Compatibility mode enables 32-bit applications to run under a 64-bit operating system. 64-bit mode enables access to the added registers and increases the address space (i.e. allows for 64-bit pointers). This work adds Jikes RVM for long (64-bit execution) mode.

AMD64 adds a number of extensions to the traditional x86 architecture. Of course, the architecture supports 64-bit integers, since the general purpose registers have been extended to 64-bit. Also, the architecture can address a larger address space due to the 64-bit word size. In order to maintain as much transparency as possible to x86, AMD64 adds a REX prefix to the 64-bit long mode. The AMD designers co-opted the `inc` and `dec` instructions for this purpose. Consequently, there is no longer a single byte `inc` or `dec` encoding in the AMD64 ISA. Without the REX prefix the instruction operands default to 32-bits, and the additional registers offered by AMD64 are inaccessible. The AMD64 ISA also adds 8 general purpose registers to the relatively register poor x86 ISA. Additionally, the new architecture also adds 8 XMM streaming SIMD registers. The default operand size remains 32-bit, but push and pop instructions default to the

64-bit operand size. Also, a number of little used features, such as segmented addressing, were removed in long mode.

3.2 Jikes RVM Bootstrapping Process

Jikes RVM is a Java Virtual Machine and JIT compiler, written in Java. A robustness goal of any such system is for the system to be able to compile and run itself. Jikes RVM achieves this goal. In fact this is the standard way Jikes RVM is built. This creates a *bootstrapping* problem, that is if Jikes is to compile and run itself, how is the first instance created? Figure 1 shows the basic process of building Jikes RVM.

To solve the bootstrapping problem Jikes RVM uses two tools, the BootImageWriter and BootImageRunner. The BootImageWriter is a Java program that consists of the Jikes RVM core classes, most importantly the compiler and assembler. The BootImageWriter instantiates a baseline compiler instance running on a host JVM, and uses that instance to write out the machine code and basic Java data structures needed for a functional JVM. This is stored in the RVM.image file, which is a file that contains the memory layout of the Jikes RVM. Then, the BootImageRunner is responsible for loading the RVM.image file into memory, and transferring control to the RVM. It is interesting to note that this all happens *before* Jikes RVM is run. This means that the baseline compiler is assumed to be correct as part of the boot process.

4. Porting Design and Implementation

Porting a large and complex code base such as Jikes RVM presents a number of software engineering challenges. When Jikes RVM was ported to the IA32 architecture, it was only designed to support 32-bit architectures. PowerPC 64-bit support was only added later, and much of this support was not adapted for the Intel architecture. Of course, certain high-level architectural changes were made for 64-bit support, but much of the IA32 baseline compiler was not designed for 64-bits. The challenge was to add 64-bit support to the IA32 baseline compiler, maximizing code reuse while generating an efficient port.

Jikes RVM is an open-source project, and many of the original developers have moved on to other projects. Large scale, distributed projects such as this have a tendency to become unmaintainable unless care is taken. A port to a new machine can sometimes require significant changes in the assumptions and design of a system. These changes should be considered carefully, so that work by future collaborators and porters is not hindered. Here, we describe in detail our porting and debugging process, and explain our design decisions. We also give porting recommendations for making future ports of Jikes RVM easier to manage.

4.1 Porting Design Decisions

Perhaps, the most important decision in this port is how to represent the various data types in a 64-bit architecture.

Table 1. Type Sizes for 64-bit programming models (bits)

Type	Java Spec	LLP64	LP64	ILP64
byte	8	8	8	8
char	16	16	16	16
short	16	16	16	16
int	32	32	32	64
long	64	32	64	64
float	32	32	32	32
double	64	64	64	64
reference	und.	64	64	64
returnAddress	und.	64	64	64

There are three main 64-bit programming models: LP64, ILP64, and LLP64 [9]. The corresponding type sizes of these models (as well as JVM specification) are shown in Figure 4.1. The naming scheme for each model specifies which types of variables are 64-bit: int (I), long (L), reference (P), or long long (LL). In the LP64 model, which is used by GCC on AMD64, the int type is 32-bit and the long and reference types are 64-bit. In the LLP64 model, the int type is 32-bit, the long is also 32-bit, and the reference is 64-bit. If Java had a long long type, it would be 64-bit under this model. Finally, the ILP64 model sets the int, long, and reference types all to 64-bit. In order to evaluate which of these models to use, one must consult the Java Virtual Machine specification [13].

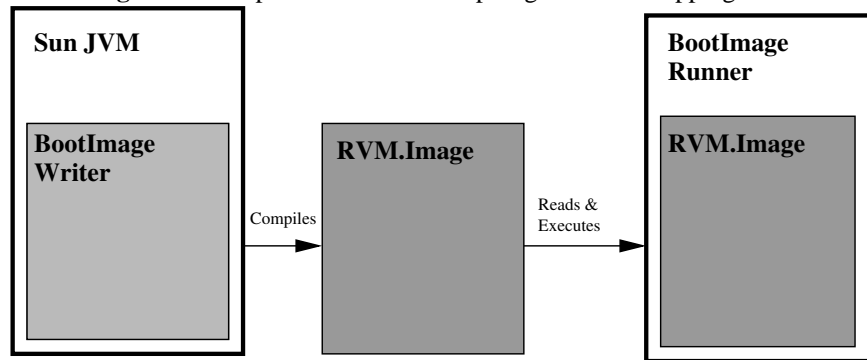
The JVM specification requires that certain types have certain lengths also in order to ensure the portability of the Java language. For example, an int in Java must be a 32-bit quantity; whereas, a long is specified as 64-bit. The Java specification says nothing about the size of reference. The specification states that the word size must be long enough to accommodate a byte, char, short, int, float, reference, or returnAddress. The spec also dictates two words must be long enough to hold a long or a double. The LP64 model is the only model that satisfies the requirement int be 32-bit and a long 64-bit. Additionally, references should be 64-bit, so that they can address large memories. A returnAddress should also be 64-bit to support large memory. LP64 is a natural fit for a 64-bit JVM.

It is important to differentiate the data type sizes and the size of data items as they are represented on the stack or in registers. All non-64 bit quantities will be represented on the stack as words (64-bit) in order to preserve alignment. The alignment on 8 byte boundaries will increase stack consumption, but increase performance. Stack issues are covered more thoroughly in Section 5.2.

4.2 Baseline Compiler Porting Process

As described in Section 2, Deschrijver had already implemented some of the low level tools needed to build a 64-bit version of the RVM.image, and to enable the assembler to emit 64-bit code. The next major hurdle to getting a running RVM is the baseline compiler. The baseline compiler implements a simple JVM compiler, which takes single byte-

Figure 1. The process of Jikes compiling and bootstrapping itself



code and emit the corresponding assembly. In this set up the standard system stack emulates the JVM stack. The baseline compile does not consider any adjacent instruction or potential inefficiencies. This substantially simplifies the implementation of the compiler because each bytecode can be implemented independently, by simply considering its effects on the stack. Figure 2 shows an example of one bytecode, `i2l` (integer to long), being converted to its corresponding assembly. Notice that the operation pushes and pops the stack, and uses the registers only as temporaries, in this case, for conversion.

We first began by verifying that simple bytecodes were being generated correctly. However, when we began our porting work, the Jikes RVM virtual machine could not boot completely, due to the fact that the RVM has compiled itself, complete with untested implementations of various bytecode instructions. This means that traditional methods of unit testing were ineffective because the base on which we would do the test was unreliable (A problem discussed in Section 4.3.2. We solved this problem by inserting our bytecode tests in the `VM.boot()` function, which is the first Java function that is executed by the `bootImageRunner` (i.e. the entry point of `RVM.image`). At the beginning of the `VM.boot()` function, we would insert Java code that corresponded to the bytecodes we wished to test. We began with simple test cases, such as integer addition. The changes to bytecodes like `i2l` are relatively simple, involving mostly additional instructions to preserve stack alignment. Later bytecodes caused more problems, like the `tableswitch` instruction. This is the bytecode used to implement switch statements. Here the code being generated assumed the offsets for the jump targets to be 32-bits, however when calculating the total size of the table, they used `BYTES_IN_WORD` as the multiplier, causing the jump instruction to jump far beyond the intended target. After enough bytecodes were working the VM was able to correctly call and return from methods, and it was possible to use `VM.Syscall.print`, which directly called to the underlining C support functions. This process took up the largest portion of our time on this project, due to the sub-

tle problems that arise, and the large number of bytecodes that needed to be implemented correctly.

4.3 Porting Recommendations

There are a number of opportunities to improve Jikes RVM to make future ports easier to manage. We make two specific recommendations to simplify the porting process. First are changes to the x86 assembler, to provide better handling of future extensions to the x86 and AMD64 architectures. Second is a recommendation for bytecode inlining, to simplify the testing process when porting to a new architecture.

4.3.1 Improved auto-generated assembler

Assemblers are obviously highly ISA specific, but when one ISA is the basis for another, it makes sense to try to extend the assembler to the new ISA. This is what has been done with x86 and AMD64. However, due to the very large number of instructions and addressing modes in the x86 architecture, the original authors of the assembler took the approach of “semi-automating” the creation of the assembler [1]. In practice, however this automation takes the form of a bash script that emits Java code. The desired code is parametrized and inlined (using “here documents”) into the script, which filled in the different addressing modes for each instruction. This certainly saves a lot of code duplication, but is still not a very clean implementation.

One alternative approach is to explicitly parametrize the operation and addressing mode, so that the method `emitMOV_Reg_RegDisp(...)` would be changed to a more generalized method like `emit(MOV,RegRegDisp,...)` (with appropriate enumerations of opcodes and addressing modes). Retrofitting an implementation like this would be a difficult task because hundreds of lines of code have already been written that assume the existence of method that encode the operation and addressing mode into the method name. Further this approach faces the problem that some of these instructions take different numbers of arguments, which is not allowed in the Java Language.

We propose two possible solutions to this problem. First is to create a core `Assembler` class that uses a more generalized `emit` method, and then create a script (bash or oth-

erwise) that would create an assembler that would create a wrapper class with all the desired functions, that would call `emit` in the core assembler, and perhaps box any parameters to solve the problem of variable argument count. A second approach would be to implement the wrapper class in a more dynamic language, like the Java implementation of python, jython. This would allow methods like `emitMOV_Reg_RegDisp(...)` to be dynamically changed to `emit(MOV,RegRegDisp,...)` without any explicit declarations.

4.3.2 Inline Bytecode Testing

The next feature that would simplify future porting efforts is a standardized way to automatically run the bytecode tests without a fully functional virtual machine. Before the memory manager is implemented, basic bytecodes must be working properly. That means that the bytecode tests cannot create new classes or call non-static methods. The easiest way to test these bytecodes is to have them inlined directly into the beginning of `VM.boot()`, the entry point of the virtual machine. However, doing this manual entails much tedious copy/paste work. One possible solution to this is to leverage the already existing preprocessor. The preprocessor would have to be extended to allow an `include` directive, to include a file. Then the bytecode tests could be programmatically inserting into the beginning of the VM process during the testing phase.

4.4 Debugging Methodology

There are a number of difficulties in debugging Jikes RVM code, especially when attempting to add support for another architecture. In this section, we describe utility scripts we developed and describe a methodology to debug Jikes RVM issues. Our debugging methodology is most useful to those porting Jikes RVM to a new architecture or Operating System or those needing to debug low-level Jikes RVM code (e.g. assembler, baseline compiler, virtual machine).

The Jikes RVM User's Guide [10] recommends The GNU debugger (Gdb) for RVM debugging. Gdb is freely available, widely used, and familiar to most systems programmers. Most important, Gdb understands AMD64 assembly. Unfortunately, Jikes RVM does not emit stabs symbol support for its Java code when it compiles its boot image. The stab format provides a standard way to describe a program to the debugger. It is used by many debuggers other than Gdb[7, 8]. The C portions of Jikes RVM already include stabs information, since they were compiled with `gcc's -g` option. Only a small fraction of Jikes RVM is written in C, however, so the vast majority of Jikes RVM has no stabs information and thus minimal debugging support. Our proposed debugging methodology addresses this limitation of Jikes.

As part of the boot image writing phase of the build process, Jikes RVM outputs a map file, `RVM.map`, which consists of a table of symbols, their associated address, and

other attributes. We created a bash utility script that parsed the map file and converted it into Gdb commands. The script allows us to break in specific functions and trace execution, even in the early virtual machine initialization. We found this script especially useful as we were able to break in the `VM.boot()` function, the first Jikes RVM Java function that is called. This script allowed us to trace through the Java code generation test cases we inserted in `VM.boot()`.

Additionally, we created a python script that would take a given instruction pointer and find the nearest symbol in the `RVM.map` file. This script proved especially useful when we were debugging segmentation faults in Jikes RVM. It is important to note that this script does not allow for source-level debugging, but it does provide symbol support.

We found Gdb helpful in our debugging, since it supports debugging on the AMD64 architecture. The disassembly functionality was especially useful, since it provides an independent means to verify that our assembler is generating correct machine code. Gdb is widely used on AMD64, so there is high likelihood that Gdb will disassemble instructions correctly.

In addition to Gdb, there is some intrinsic Jikes RVM debugging support that would be helpful to anyone porting the RVM. Jikes RVM includes a `VM_Lister` class that allows one to print out the bytecode operation the baseline compiler is generating, as well as the corresponding assembly instructions and machine code. All that is needed to make use of this functionality is to pass the option `-X:bc:mc=true` to Jikes RVM. By default, this option will print information for all compiled code. Thankfully, Jikes RVM also provides an option (`-X:bc:method_to_print`) to specify which methods need to use the `VM_Lister` functionality. We would often print the `VM.boot` method, where we injected bytecode tests, and other problem functions. This allows the bytecode implementer to see precisely what assembly is generated. We used `VM_Lister` functionality extensively in the bootstrapping phases to print out assembly as the boot image was being written. This process is fairly involved. As a result, we created yet another utility script, adapted from the official Jikes RVM build script, to simplify this process. The script will run the baseline compiler on the host virtual machine and output information on all the methods we specify.

Additionally, we used the time-honored debugging technique of interspersing print statements within our modified Jikes RVM code. The `VM_Lister` class provides the assembler with the `noteByteCode` method, which allows the compiler to annotate the output listed. We used it to output statements during boot image creation. We also used `VM_sys.print()` for "printf" debugging during actual RVM execution. We found all these debugging techniques useful and would recommend them to any who wished to work on porting Jikes RVM. The Gdb debugger was helpful in verifying our assembler implementation and tracing through Jikes RVM execution in general. The `VM_Lister`

Figure 2. Implementation of the i2l (int to long) bytecode

```
[2] i2l
00004C |      POP          rax          | 4058
00004E |      CDQE         | 4898
000050 |      PUSH         0           | 6A00
000052 |      PUSH         rax         | 4050
```

class was useful, since it allowed us to easily relate assembly and machine code to the corresponding bytecode operation; Gdb could not do this easily. The `VM_Lister` was our main method of debugging in the early stages of the port. The `println` method allowed us to pinpoint bugs by allowing us to test assumption about certain execution paths.

4.5 Debugging Recommendations

The inability to use source level debuggers in systems that rely on dynamically created code is a serious usability concern. Kumar et al. [11] have developed a system for enabling Gdb for dynamic systems. Such a system can map debugging information to dynamic code addresses, thereby enabling practical source-level debugging.

Failing a full implementation of a Jikes-aware debugger, Jikes RVM would be significantly easier to debug if it at least natively emitted stabs information. Consequently, we recommend that a stabs file (`RVM.stabs`) be generated as part of the build process. This should not be too difficult, since a fair amount of the information is already outputted in the `RVM.map` file. The `gdbrvm` wrapper script, provided in the Jikes RVM, distribution could then be modified so that it would read the stabs information in the Gdb initialization sequence. This would allow for true source-level debugging, allowing for much easier debugging. GCJ, the GNU compiler collection's Java to machine code compiler, already supports source-level debugging with `gdb`, so it is feasible.

5. Space Optimizations

It is important to be aware of changes in code size when moving from a 32-bit to 64 bit architecture. This is a particular problem when dealing with the Java Virtual Machine which makes many assumptions that are difficult to reconcile when dealing with a non-32-bit platform [14]. This problem presents itself most directly when implementing the JVM stack on the system stack, as is done for the Jikes RVM baseline compiler. Furthermore, a compiler should emit 32-bit instructions, whenever possible, in order to minimize code size.

5.1 Java Stack Model

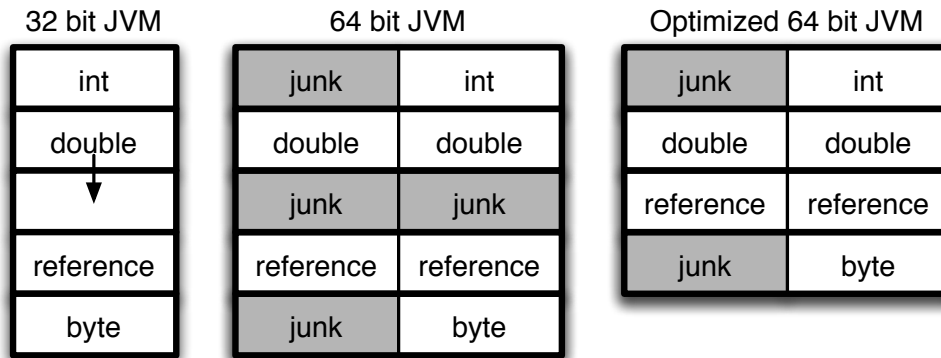
The Jikes RVM baseline compiler uses the system stack to model the JVM stack, as described in Section 4.2. This simplifies the implementation of the compiler, but forces the system stack to follow the semantics of the JVM stack, as

described in the JVM Specification [13]. The JVM specification describes the stack as acting on two data types, category I and category II. Category I data types include 32-bit integers, bytes, 32-bit floats, references, and returnAddresses. Category II data types include 64-bit long integers and 64-bit double precision floating points. The JVM Specification mandates that category I types must occupy one "stack slot," while category II must occupy two. Since all stack operations are 64-bit on the AMD64, category II types must therefore occupy 128 bits of space on the stack, even though they only contain 64 bits of usable data. Figure 3 illustrates the problem. The 32-bit stack is shown on the left and the corresponding 64-bit stack in the center. Essentially, the diagrams are the same, except the middle diagram is extended to 64-bits. The double type still encompasses two stack slots in the 64-bit stack, even though only one is needed.

5.2 Stack Size Optimization

In order to improve code size and stack usage, we have implemented a stack size optimization by bending the Java Virtual Machine specification. Although the JVM specification requires category II data types to occupy two stack slots, most bytecodes are type safe, that is, the bytecode expects a well-defined set of types on the top of the stack before the instruction executes, and the instruction leaves a well-defined set of types on the stack. This means that it would be possible to eliminate some of the padding due to the size differences between the required size of the stack slot, and the actual size of the data held in the stack slot. There are two prime candidates for this type of size optimization. First is the integer, which is defined as 32 bits, but since it is in the same category as a reference, it must reside in a 64-bit slot. We decided not to implement this operation because AMD64 defaults all stack operations to 64 bits, and stack misalignment could cause serious performance penalties. The second candidate is the category II data types. Since all category II types require only 64 bits of space, they are padded with an additional 64 bits of junk, which can be eliminated. Figure 3 (rightmost diagram) shows the optimized stack with the extra stack slot allocated to the double removed. We feel this is a practical and useful optimization, and have implemented it in our baseline compiler bytecode implementation.

Figure 3. JVM Stack



5.3 Implementation of Stack Size Optimization

In order to correctly implement this optimization an number of assumptions made by Jikes RVM had to be changed. The most obvious of these changes is in how the compiler emits long and double bytecodes. For example, Figure 2 is the implementation of the `i2l` bytecode, the first push instruction (`PUSH 0`) exists simply to maintain stack consistency. These instructions were guarded such that if our optimization is turned on, they will not be emitted. Other necessary changes were more subtle. Figure 4 shows the original (Jikes RVM 2.4.0 x86) implementation of the `getSize()` method in the class `VM_TypeReference`. This method is used to calculate the space needed to store data on the stack, which is used to determine how much stack space to reserve for locals during a method prologue. The difficulty this implementation is that it makes a number of assumptions that are problematic for our optimized implementation. For example, is the size that is being returned the size the data takes up on the stack, or is it the actual size of the data? The final return is for “all int like types.” which includes bytes and chars, so the intent seems to be for this function to return the size on the stack, instead of the actual size of the value (as the comment implies). The poses a problem for our 64-bit implementation. For it to give the correct stack-size value for the unoptimized version, `BYTES_IN_LONG` must be set to 16. However, `BYTES_IN_LONG` is a system-wide constant, and it would be dangerous to set it to 16 when clearly longs still are only 8 bytes. Our solution to this is to split the `getSize` function into two functions, `getSize` and `getStackSize`. Using these two function it is easy to differentiate between the intended use of the two functions.

5.4 Problem Bytecodes

Changing the implementation of the stack offers potential bytecode saving, however some bytecode instructions present a problem for this technique. For example, the `*load_<n>` instruction loads a local variable from the stack frame, indexed by the opcode. It takes the value at that index

and pushes it onto the top of the stack. This index corresponds to stack slot, therefore all category II data on the frame must take two slots so the indices will remain correct.

This problem can be solved by making the distinction between the Operand Stack and its surrounding frame. Local variables are stored in the frame, before the operand stack, not as part of the stack [13]. Therefore, our size optimization maintains the layout of the local variables as described in the specification, and allow the operand stack to use the optimization.

Another more serious problem is the `dup_2x` bytecode instruction. The `dup_2x` instruction duplicates the category I operand on the top of the stack, two stack slots lower on the stack. It explicitly allows these two stack slots to contain either two category I data values, or one category II data value. Under our optimization, if the data value is one category II, the `dup_2x` would put the data value too far down the stack.

This problem can be solved with some amount of context-sensitive information. The instruction must know to emit only one pop if there is a category II data value in the second position. We have not implemented this fix yet, as we believe it rare to use the `dup_2x` instruction in this way.

5.5 REX prefix optimization

The REX prefix extension enables 64-bit operations, as described in Section 3.1. The REX prefix encodes four flags: `REX.W`, `REX.R`, `REX.X`, `REX.B`. If `REX.W` is set, the default word size is changed from 32 bits to 64 bits. The `REX.R`, `REX.X`, `REX.B` flags provide register extensions to the Mod/RM byte, SIB index, and SIB base. These are x86 instruction bytes that are used for various addressing modes. Most instructions default to 32-bit operands. A REX prefix is only needed if 64-bit operands/operations or the extended AMD64 register set are required. Furthermore, certain instructions imply a 64-bit operation, so a REX prefix is unnecessary. These instructions include `PUSH`, `POP`, `near CALL`, `near JMP`, `LOOPcc` among others. Unfortunately, the initial AMD64 assembler [4] emits a REX prefix

Figure 4. Original Jikes RVM implementation of the `getSize` function for non local stack calculations

```
/**
 * How many bytes of memory words do value of this type take?
 */
public final int getSize() throws UninterruptiblePragma {
    if (isReferenceType() || isWordType()) return BYTES_IN_ADDRESS;
    if (this == Long || this == Double) return BYTES_IN_LONG;
    if (this == Void) return 0;
    if (this == Code) return VM.BuildForIA32 ? BYTES_IN_BYTE : BYTES_IN_INT;
    return BYTES_IN_INT; //all int like types
}
```

for all instructions that could possibly require one. This simplifies the assembler implementation, but causes to assembler to emit REX prefixes when they are unnecessary. To solve this problem, we have modified the assembler to emit prefixes only as necessary. First, the REX prefix usage was removed from all the instructions that are defined to be 64-bit. Then, the assembler’s methods were made to check for 64-bit operands using the existing `fits()` utility method. 64-bit versions of certain assembly instructions were also added. Finally, a check for extended register usage was also added. The assembler modification was complicated by the Jikes RVM bash assembler macro scheme [1]. Certain assembler macros had to be refactored into separate macros, and new utility functions were added. We have not finished implementing the REX prefix optimization in all cases, but our results (see 6) indicate a significant savings.

6. Results

All experiments were performed on a dual AMD Opteron 244 system with 2GB of RAM running Fedora Core Linux. For the bootstrapping process we used the Sun JVM 1.4.2, and GCC 3.4.4. To extract the size of the emitted code we used the `BootImageWriter`’s `-demographic` switch. The benchmark class files were added to the `RVM.primordials` file, and the total code size was subtracted from the base RVM code size so that the measurement would only include the benchmark classes, and not the RVM classes.

6.1 Code Size measurements

We performed size experiments on two standard benchmark suites as well as our own micro-benchmark. The results of these experiments are described below.

6.1.1 SPECjvm98 Benchmarks

Table 2 gives a summary of the SPECjvm98 benchmark suite. The code size optimizations result in relatively small reduction in total code size, due to the small number amount of double and long arithmetic in the benchmarks. The best performing spec benchmarks are `mtrt` and `check`. `mtrt` is somewhat misleading in that the 1.16% saving actually only represents a code size decrease of 16 bytes. This is due to the

very small size of `mtrt`, which only has one function (that returns a long), during which it creates an instance of the `raytrace` benchmark. The `check` benchmark has a savings of 0.70%, since it verifies double-precision computations. The `raytrace` benchmark has the smallest decrease in code size. To get a better idea of why this is the case, we examined the `raytrace` source, and discovered that all the floating point computation is done with floats, which are defined to be 32 bits. The other benchmarks vary in their code savings based on how much each relies on doubles and longs.

The REX prefix optimization performed significantly better on SPECjvm98. The results are summarized in Table 3. Some benchmarks saw as much as a 7% decrease in code size due to the elimination of unnecessary REX prefixes, with an average of 5.83% overall. All benchmarks receive a significant code size reduction. The results of the SPECjvm98 benchmark with both optimizations are listed in Table 4. As expected, the use of both optimization performs slightly worse than the sum of the two individual optimizations, since the stack optimization will eliminate `PUSH` and `POP` operations that the REX prefix optimization could optimize.

6.1.2 Java Grande Benchmarks

The Java Grande Benchmark Suite were run to augment our SPECjvm98 analysis. The Java Grande Benchmarks are selected to “use large amounts of processing, I/O, network bandwidth or memory.” [5]. This better models the types of applications we expect to be of interest to 64-bit users. The Java Grande Benchmarks perform slightly better than SPECjvm98, with an average code size savings of .5% as shown in Table 5. Interestingly, the Java Grande Benchmark suite also contains a `raytracer`, which responds significantly better to our code size optimizations. This is because the Java Grande `raytracer` is implemented using doubles, instead of floats. Certain benchmarks, such as `heapsort` and `sparse matrix multiplication`, still don’t see a large code savings since they rely on the 32-bit computations.

As seen in the SPECjvm98 benchmarks, the REX prefix optimization gave much more significant improvements compared to the stack optimization. Table 6 shows a code

Table 2. Code size reductions using single words for longs and doubles on SPECjvm98.

Benchmark	Original Code Size	Reduced Code size	% reduction
check	58624	58216	0.70%
compress	23960	23936	0.10%
jess	320864	320664	0.06%
raytrace	108480	108464	0.01%
db	24720	24696	0.10%
javac	653424	651352	0.32%
mpegaudio	184744	184648	0.05%
mtrt	1400	1384	1.16%
jack	252896	251808	0.43%
checkit	5400	5392	0.15%

Table 3. Code size reductions using REX prefix optimization on SPECjvm98.

Benchmark	Original Code Size	Reduced Code size	% reduction
check	58624	55136	6.33%
compress	23960	22416	6.89%
jess	320864	304880	5.24%
raytrace	108480	103344	4.97%
db	24720	23360	5.82%
javac	653424	618896	5.58%
mpegaudio	184744	174424	5.92%
mtrt	1400	1328	5.42%
jack	252896	238232	6.16%
checkit	5400	5096	5.97%

Table 4. Code size reductions using REX prefix and stack optimization on SPECjvm98.

Benchmark	Original Code Size	Reduced Code size	% reduction
check	58624	54760	7.06%
compress	23960	22408	6.93%
jess	320864	304720	5.30%
raytrace	108480	103344	4.97%
db	24720	23352	5.86%
javac	653424	617328	5.85%
mpegaudio	184744	174376	5.95%
mtrt	1400	1328	5.42%
jack	252896	237424	6.52%
checkit	5400	5096	5.97%

savings of up to 6.66%. The average savings for the Java Grande Benchmark is 5.31%. Table 7 gives results for both optimizations on the Java Grande Benchmark Suite. Here also, the combined optimization did not do as well as the sum of the two optimizations' code savings.

6.1.3 Synthetic Benchmark

In order to test the limits of the stack savings technique, we created a synthetic micro-benchmark. This benchmark consists of a simple class that performs multiplication of a matrix and a scalar, exclusively using long and double data types. This does not represent the standard style of Java programming currently, however, as 64-bit computing becomes more prevalent, we believe that Java programmers will move to using more longs and double types. Additionally the multiplication loop of the benchmark has been manually unrolled five times. Again, this is not normal style, however, the purpose of this is to give an idea of the dynamic effects that this optimization will have on the system. Compiling the microbenchmark with our system resulted in a 10.7% improvement, reducing the code size from 2232 bytes to 2016 bytes. This benchmark shows that the stack size optimizations can achieve significant savings.

6.2 Stack Size Improvements

Due to the fact that there are still vital portions of the Jikes RVM infrastructure that have not yet been ported to AMD64, we are unable to obtain specific stack size savings measurements. However, our intuition is that the stack size improvements will be more significant than the code size savings. This is because the additional pushes and pops of doubles and longs will be more noticeable in a dynamic instruction count, where the benchmark is expected to loop many times over certain calculations. Also, it's important to remember that the PUSH and POP instructions are generally one byte in code size, but represent an 8 byte change to the stack pointer.

7. Future Work

This work represents an important part of an ongoing effort to create a working implementation of Jikes RVM for AMD64. Our first order of business is to finish the baseline compiler port. This will require debugging issues in `VM_Magic`, which provides low-level virtual machine constructs, and the memory manager. The memory manager also requires some study, not only for correctness, but also for space saving opportunities specific to the AMD64 architecture. Finally the optimizing compiler is a large source of future work, specifically the register allocator, which should be modified to use AMD64's additional registers. The optimizing compiler will provide another avenue for exploring 64-bit computing issues.

In addition to the "structural" future work involved with creating a fully functional Jikes RVM (incl. optimizing compiler) working under AMD64, we believe there is also fruit-

ful future work in continuing to look at the space issues associated with the baseline compiler. There are a number of edge cases that represent potential problems with bytecodes such as `dup_2x`. We believe these can be solved using data structures already built into Jikes RVM.

Once we are finished with the baseline compiler, we would also like to do a more thorough analysis of the runtime performance of the size optimization we have implemented, specifically with regard to stack size. Because push and pop instructions are only one byte long but affect the stack by 4 bytes, we expect the relatively small change in bytecode size to have a much larger effect on stack usage. We would also like to apply our REX prefix optimization more aggressively. Finally, we would like to have our work incorporated into the official Jikes RVM project, so that others may explore 64-bit computing issues as well.

8. Conclusion

This work is part of a larger ongoing project to get a usable port of the Jikes RVM on AMD64. The most up-to-date status of the project is available through our wiki, found at <http://www.osheim.org/moin.cgi/Jikes>. This work critically examined the process of porting Jikes RVM to AMD64. We offered a number of recommendations for changes to the virtual machine and the assembler to make future ports easier. We implemented a number of scripts to ease the debugging process, available on the wiki. Additionally we investigated memory space improvements to the code emitted by the baseline compiler. Ultimately, we hope to submit a full working version of the Jikes RVM back to the sourceforge community.

Acknowledgments

We would like to thank Kris Venstermans, Kim Hazelwood, Fredrick Deschrijver, David Groves, and Elliot Moss for their input and guidance on this project. We would also like to thank all those on the Jikes RVM mailing list for their prompt and instructive answers to our questions. We would finally like to thank Greg Humphreys for graphing, figure and \LaTeX assistance. We think he is delightful. Additionally, we would like to thank the nice gentlemen from Buck's pizza for delivering delicious late-night pizza to Olsson.

References

- [1] Bowen Alpern, Maria Butrico, Anthony Cocchi, Julian Dolby, Stephen Fink, David Grove, and Ton Ngo. Experiences porting the jikes rvm to linux/ia32. In *JVM '02: The 2nd Java(TM) Virtual Machine Research and Technology Symposium*, 2002.
- [2] AMD. *AMD64 Architecture Programmer's Manual*. AMD, 2005.
- [3] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeno jvm. In *OOPSLA '00: Proceedings of the 15th*

Table 5. Code size reductions using single words for longs and doubles on Java Grande Benchmarks.

Benchmark	Original Code Size	Reduced Code size	% reduction
crypt	24992	24888	0.42%
fft	21344	21152	0.91%
heapsort	16622	16664	0.14%
lufact	24304	24176	0.53%
series	16744	16816	0.76%
sor	15704	15672	0.20%
sparsematmult	15136	15120	0.11%
euler	122168	121176	0.82%
moldyn	37208	36672	1.46%
montecarlo	69616	69376	0.35%
raytracer	42872	42504	0.87%
search	37264	37168	0.26%

Table 6. Code size reductions using REX prefix optimization on Java Grande Benchmarks.

Benchmark	Original Code Size	Reduced Code size	% reduction
crypt	24992	23432	6.66%
fft	21344	20168	5.83%
heapsort	16688	15944	4.67%
lufact	24304	22872	6.26%
series	16944	16176	4.75%
sor	15704	14960	4.97%
sparsematmult	15136	14448	4.76%
euler	122168	115840	5.46%
moldyn	37208	35432	5.01%
montecarlo	69616	66680	4.40%
raytracer	42872	40880	4.87%
search	37264	35096	6.18%

Table 7. Code size reductions using REX prefix optimization and stack optimization on Java Grande Benchmarks.

Benchmark	Original Code Size	Reduced Code size	% reduction
crypt	24992	23336	7.10%
fft	21344	20024	6.59%
heapsort	16688	15920	4.82%
lufact	24304	22760	6.78%
series	16944	16072	5.43%
sor	15704	14936	5.14%
sparsematmult	15136	14440	4.82%
euler	122168	114968	6.26%
moldyn	37208	34936	6.50%
montecarlo	69616	66336	4.94%
raytracer	42872	40576	5.66%
search	37264	35056	6.30%

ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 47–65, New York, NY, USA, 2000. ACM Press.

- [4] Frederik Deschrijver. Jikesrvm voor een 64-bit x86 platform. Technical report, Ghent University, June 2005.
- [5] epcc. The java grande benchmark suite. <http://www.epcc.ed.ac.uk/javagrande/javag.html>.
- [6] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.
- [7] GNU. Gdb internals manual. <http://sources.redhat.com/gdb/current/onlinedocs/gdbint.html>.
- [8] GNU. Stabs debug format. <http://sources.redhat.com/gdb/current/onlinedocs/stabs.html>.
- [9] Jan Hubicka. Porting gcc to the amd64 architecture, May 2003.
- [10] IBM. The jikes research virtual machine user's guide. <http://jikesrvm.sourceforge.net/userguide/HTML/userguide.html>.
- [11] Naveen Kumar, Bruce R. Childers, and Mary Lou Soffa. Tdb: a source-level debugger for dynamically translated programs. In *AADEBUG'05: Proceedings of the Sixth sixth international symposium on Automated analysis-driven debugging*, pages 123–132, New York, NY, USA, 2005. ACM Press.
- [12] Sergiy Kyrylkov. Jikes research virtual machine: Design and implementation of a 64-bit powerpc port. Technical Report TR-CS-2003-41, Department of Computer Science, University of New Mexico, December 2003.
- [13] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, 1999.
- [14] Kris Venstermans and Koen De Bosschere. Jvm spec favours 32-bit platforms. In *Workshop on Signal Processing, Integrated Systems and Circuits*, 2003.