Jikes Research Virtual Machine Design and Implementation of a 64-bit PowerPC Port

by

Sergiy Kyrylkov

Bachelor's degree in Electronic Engineering,
Technological University of Podillya, 1999

Specialist's degree in Electronic Engineering,
Technological University of Podillya, 2000

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2003

Dedication

To my parents, grandparents, and sister

Acknowledgments

I would like to express my deep appreciation to Darko Stefanović and Eliot Moss for their expert assistance, guidance, criticisms, and comments throughout my study.

Jikes Research Virtual Machine

Design and Implementation of a 64-bit PowerPC Port

by

Sergiy Kyrylkov

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2003

Jikes Research Virtual Machine

Design and Implementation of a 64-bit PowerPC Port

by

Sergiy Kyrylkov

Bachelor's degree in Electronic Engineering,
Technological University of Podillya, 1999

Specialist's degree in Electronic Engineering,
Technological University of Podillya, 2000

M.S., Computer Science, University of New Mexico, 2003

Abstract

This work describes the design and implementation of a 64-bit PowerPC port of the Jikes Research Virtual Machine (Jikes RVM). It explains general design decisions for 64-bit implementations of Java virtual machines, specifically Jikes RVM, along with details of the specific 64-bit PowerPC implementation.

Contents

List of Figures

List of Tables

Glossary

GC	Garbage Collection
GCTk	Garbage Collection Toolkit
IMT	Interface Method Table
JTOC	Jikes RVM Table of Contents
JVM	Java Virtual Machine
RVM	Research Virtual Machine
TIB	Type Information Block
VM	Virtual Machine

Chapter 1

Introduction

64-bit computing, contrasted with 32-bit computing, can in general be characterized by very large memory support, very large application virtual address spaces, and 64-bit integer computation, using 64-bit general-purpose registers. In such 64-bit systems, an application's virtual address space is measured in terabytes and an increasing number of programs can exploit this opportunity. For example, some database servers use a large address space for scalability and maintain very large data buffers in memory, thus reducing the amount of I/O they perform. Some computationally intensive programs can benefit from keeping much larger arrays of data to be computed on entirely in memory. Both of these types of applications exhibit performance gains only when the physical memory available is large enough. On the other hand, other applications, including virtual machines with new memory management algorithms, can potentially benefit merely from having a very large virtual address space without additional physical memory.

As a result, 64-bit computing introduces a new set of research opportunities related both to evaluating previously existing 32-bit solutions in the 64-bit world and to inventing brand-new approaches specifically exploiting the benefits of 64-bit architectures. In the field of memory management, for instance, we can talk about the feasibility of reference-

counting algorithms for large address spaces, as an example of evaluating an existing solution in the 64-bit world, or we can focus on new, more flexible, heap layouts that provide for more efficient write barriers.

Taking into account these possibilities, we observed that there was no 64-bit, free, open-source virtual machine available that could serve as a useful test-bed for prototyping new 64-bit virtual machine technologies. This fact and the reasons stated above prompted us to design and implement a 64-bit PowerPC port of Jikes RVM, documented in this paper.

Chapter 2 offers overviews of some relevant aspects of Java, Java Virtual Machines, Jikes Research Virtual Machine, Garbage Collection Toolkit (GCTk), and of the 64-bit PowerPC architecture. Chapter 3 discusses design decisions relevant to any 64-bit implementation of Jikes Research Virtual Machine. Chapter 4 covers the details of our 64-bit PowerPC port of the virtual machine. Chapter 5 talks about differences between Jikes Research Virtual Machine versions 2.0.3 and 2.3.0.1, the latest version of the Research Virtual Machine to date. Chapter 6 concludes.

Chapter 2

Background

2.1 The Java Programming Language

The Java programming language [5] is a general-purpose object-oriented, robust, architecture-neutral, portable, secure, multithreaded language, with implicit memory management. The object-oriented features of Java are mostly the same as in C++, with addition of interfaces and extensions for more dynamic method resolution. At the same time, unlike C++, Java does not support operator overloading, multiple inheritance, or automatic type coercion. Robustness is mostly achieved by extensive dynamic (runtime) checking and a built-in exception handling mechanism. The Java compiler generates *bytecode* instructions that are independent of any specific architecture, and thus provide architecture neutrality. Additional portability is achieved by specifying the sizes of the primitive data types and the behavior of arithmetic operators on these types. For example, int always means a signed two's complement 32-bit integer, and float always means a 32-bit IEEE 754 floating point number. Java also has a set of synchronization primitives that are based on the widely used monitor and condition variable paradigm. Automatic *garbage collection* (GC) simplifies the task of Java programming and dramatically decreases the

number of bugs, but makes the system somewhat more complicated.

2.2 The Java Virtual Machine

The *Java Virtual Machine* (JVM) is a specification [8] for software responsible for running Java programs compiled into a special instruction set—Java bytecode. JVM is an abstract computing machine and is responsible for Java hardware and operating system independence, the small size of its compiled code, and its ability to prevent malicious programs from executing. The Java Virtual Machine does not assume any particular implementation technology, hardware, or operating system.

2.3 Jikes Research Virtual Machine

Jikes Research Virtual Machine (RVM) is a virtual machine developed at IBM T.J. Watson Research Center and capable of running a wide variety of Java programs[1]. The *virtual machine* (VM) is implemented in the Java programming language [5]. It uses two bytecode-to-native compilers: baseline and optimizing, but no interpreter. Jikes RVM also includes a family of modern garbage collectors and an adaptive compilation infrastructure.

Jikes RVM is an open-source project and runs on Linux/IA-32 [2], Linux/PowerPC, and AIX/PowerPC[2]. Although almost all of the VM is written in Java, its low-level functionality is implemented without breaking the Java language specification. This is achieved using “magic”, a mechanism to implement low-level functionality that is not possible in pure Java.

Jikes RVM can be divided into the following major subsystems: core runtime (class

[1]This section refers to Jikes Research Virtual Machine v2.0.3, released on March 11, 2002.
[2]Jalapeño, the precursor to Jikes RVM, ran only on PowerPC/AIX.

loader, library support, profiler, scheduler, verifier, etc.); compilers (baseline, optimizing); memory managers, which include a variety of copying and reference-counting garbage collection algorithms; and an adaptive optimization system. Below we provide some details of runtime implementation.

2.3.1 Object Model

The Java programming language [5] and the Java virtual machine [8] operate on two kinds of types: primitive types and reference types. Correspondingly, there are two types of values: primitive values and reference values. The Java virtual machine also supports objects. An object is either a class instance having fields or an array having elements. An object consists of two major parts: the *object header* and the *body* (instance fields or array elements).

The following requirements determine the Jikes RVM object model: instance field and array accesses should be as fast as possible, null-pointer checks should be performed by the hardware if possible, virtual method dispatch should be fast, other less frequent Java operations should not be prohibitively slow, and object header size should be as small as possible to minimize heap space overhead.

If a reference to an object is stored in a register, the fields of the object can be accessed at a fixed offset with a single RISC instruction. In the case of arrays, the reference to an array points to the first element and the remaining elements are laid out in ascending order. The array length is stored just before the first element. Thus, array elements can also be accessed at a fixed offset with a single instruction (not counting the array bounds check).

According to the Java Language Specification [5] and the Java Virtual Machine Specification [8], an attempt to use a `null` reference in a case where an object reference is required results in a `NullPointerException`. In Jikes RVM, references are machine addresses and `null` is represented by machine address `0x0`. Array objects grow up from

the object reference and scalar objects grow down (Figure 2.1). The AIX operating system permits loads from low memory, but accesses to very high memory, at small negative offsets from a null pointer, normally cause hardware interrupts. In this way, array object accesses are trapped by the hardware, because we need to load the array length, which has an offset -4. In the case of scalar objects, the hardware will trap field accesses at negative offsets from the object reference, if the object reference is null.

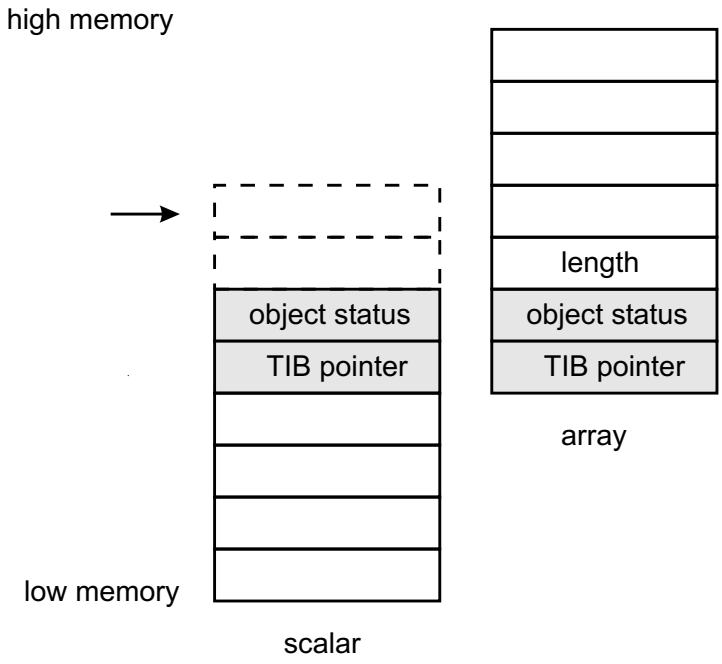

Figure 2.1: Object Layout

2.3.2 Object Headers

The default object model uses a two-word header, which supports virtual method dispatch, dynamic type checking, memory management, synchronization (each Java object has an associated lock state, which can be a pointer to a lock object or a direct representation of the lock), and hashing (each Java object has a default hash code). The object header is

located 3 words below the value of a reference to the object, leaving room for the length field in case of an array. Other object models include three-word header when Jikes RVM uses concurrent garbage collection or Brooks-style read barrier [4], and five-word header for Jikes RVM garbage collection tracing [6].

One word of the header is a *status* word, which is divided into three *bit fields*. The first bit field contains a pointer to a lock object or a direct representation of the lock. The second bit field is used for storing the default hash code for hashed objects. The third bit field is used by the memory manager and can include some combination of a reference count, forwarding pointer, etc.

Another word of the header contains a *Type Information Block* (TIB) pointer. The TIB holds information that applies to all objects of a particular class. It includes the virtual method table, a pointer to an object representing the type, and pointers to a few data structures to facilitate efficient interface invocation and dynamic type checking.

2.3.3 Methods and Fields

Compiled method bodies in Jikes RVM are represented as arrays of int and arrays of byte on PowerPC and IA-32. The *Jikes RVM Table of Contents* (JTOC) stores pointers to static fields and methods, literals, numeric constants, references to String constants, and references to the Type Information Blocks for each class in the system. These structures can have many types and the JTOC is declared to be an array of int, thus Jikes RVM uses a descriptor array, co-indexed with the JTOC, to mark the entries containing references.

Pointers to instance fields and virtual methods are stored in the Type Information Block of the class, which serves as a Jikes RVM virtual method table and insures simple dispatch of virtual methods invoked through the invokevirtual call. Regardless of whether a virtual method is overridden, the invokevirtual call dispatch is simple since the method occupies the same TIB offset in its defining class and in every subclass.

On the other hand, the invokeinterface call in Jikes RVM uses an *Interface Method Table* (IMT) [3], which resembles a virtual method table for interface methods. As with the TIB, any method that could be an interface method has a fixed offset into the IMT. However, unlike in the TIB, two different methods may have the same offset into the IMT, in which case a conflict resolution stub is inserted in the IMT.

2.3.4 VM Conventions

Dedicated Registers

General purpose registers (GPR) and *floating-point registers* (FPR) in Jikes RVM can be categorized into four types: scratch, dedicated, volatile, and nonvolatile. Dedicated registers are registers with known contents. There are four of them: JTOC (pointer to Jikes RVM Table Of Contents), FP (*Frame Pointer*, which addresses the top of the thread specific frame), TI (*Thread ID*, used to set and test the locking field of light-weight object locks) , PR (*Processor Register*, pointer to an object representing the current virtual processor).

Stacks

Stacks grow from high memory to low memory. A stack is an array of 4-byte slots, each containing either a primitive, an object pointer, a return address pointer, or a frame pointer (Figure 2.2). The interpretation of a slot's value depends on the value of the IP register.

Calling Conventions

All parameters are passed in volatile registers, if there are enough volatile registers available. Object references and int parameters (or results) consume one general-purpose

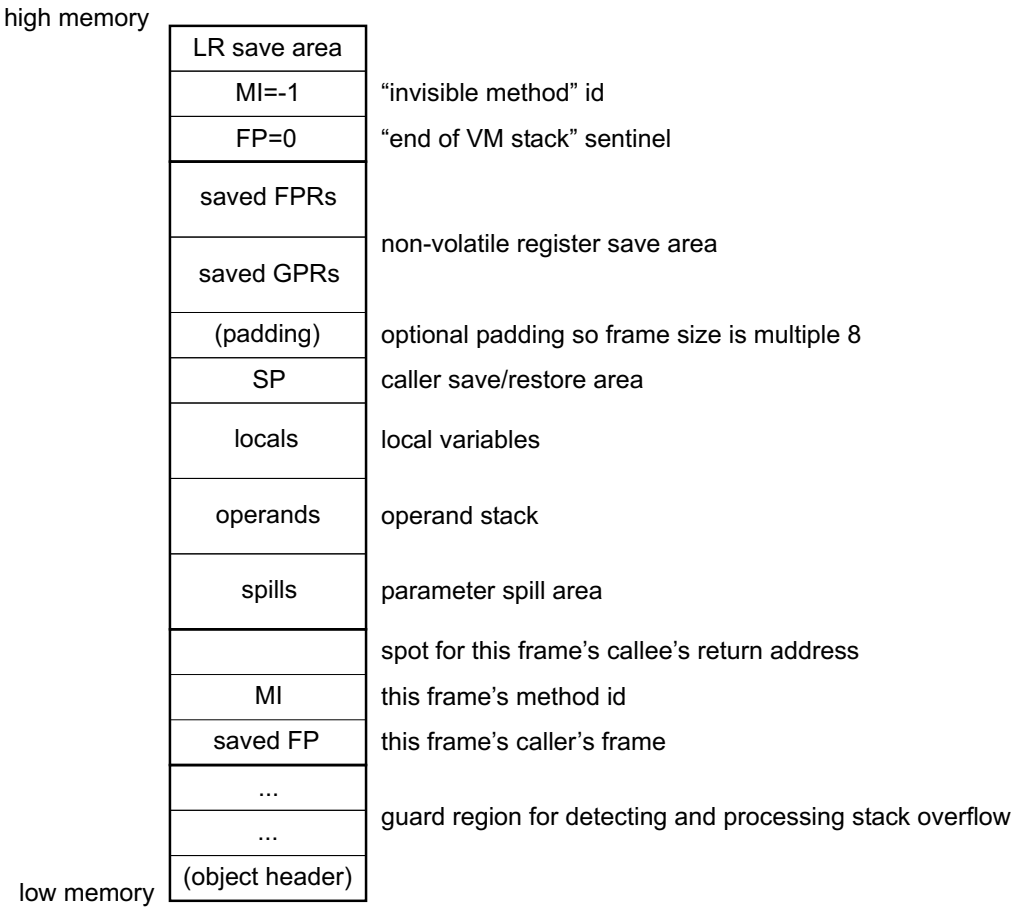

Figure 2.2: Stack Frame Layout

register, long parameters consume two general-purpose registers (low-order half in the first), and float and double parameters consume one floating-point register. Parameters are assigned to registers starting with the lowest volatile register through the highest volatile register of the required kind (GPR or FPR). Additional parameters are passed on the stack in a parameter spill area of the caller's stack frame. The first spilled parameter occupies the lowest memory slot. Slots are filled in the order that parameters are spilled. Similarly, an int or object reference result is returned in the first volatile general-purpose register, a float or double result is returned in the first volatile floating-point register, and a long

result is returned in the first two volatile general purpose registers (low-order half in the first).

Method prologue responsibilities in Jikes RVM include: executing a stack overflow check and growing the thread stack if necessary, saving the caller's next instruction pointer, the callee's return address, from the Link Register (LR), saving any nonvolatile floating-point registers used by the callee, saving any nonvolatile general-purpose registers used by the callee, storing and updating the Frame Pointer (FP), storing the callee's compiled method ID, checking to see if the Java thread must yield the virtual processor and yielding if a thread switch was requested.

Method epilogue responsibilities in Jikes RVM include: restoring the Frame Pointer (FP) to point to the caller's stack frame, restoring any nonvolatile general-purpose registers used by the callee, restoring any nonvolatile floating-point registers used by the callee, branching to the return address in the caller.

2.3.5 Magic

Magic is a mechanism that allows us to implement certain Jikes RVM functionality that cannot be expressed in pure Java. There are two types of magical operators. The first are static methods of VM_Magic class, while the second are mechanisms to declare code *uninterruptible*.

Some methods in class VM_Magic are treated specially by the compilers. These methods make operating system calls, implement access to raw memory, perform unsafe casts, etc. They cannot be implemented in Java code, thus the bodies of VM_Magic methods are undefined. For each of these methods the Java instructions to generate assembly code are stored in VM_MagicCompiler. When a compiler encounters a call to a magic method, it inlines appropriate code for the magic method into the caller method.

An uninterruptible method is compiled without the insertion of hidden thread switch points. This code can be written assuming that it cannot lose control while executing due to a timer-driven thread switch. Neither yield points (places where current thread may yield to another thread) nor stack overflow checks will be generated for uninterruptible methods.

2.4 Garbage Collection Toolkit

Garbage Collection Toolkit (GCTk) is software developed for Jikes RVM at the Department of Computer Science of the University of Massachusetts to greatly simplify development and evaluation of garbage collection algorithms inside the virtual machine. Like the rest of Jikes RVM, GCTk is written in Java.

Features of the toolkit include: a strong focus on code reuse, a set of built-in collectors (including semi-space, fixed-nursery generational, “Appel-style” generational, and “null” collectors), “brute-force” GC tracing (for getting a complete trace of object graph mutations over the execution of a Java program), and support for “direct” and “block-indirect” address mappings.

Limitations of GCTk include the following: GCTk is not supported software; documentation is limited to javadoc-generated API documentation; error messages and failure modes are not uniformly helpful; finalization is not supported; no SMP support; includes only copying collectors (i.e., no mark-sweep, reference counting, etc.)

Starting from Jikes RVM 2.2.0, the *Java Memory management Toolkit* (JMTk), which grew out of GCTk, replaced so-called “Watson” collectors in Jikes RVM 2.0.3. This marked the end of GCTk develoment.

2.5 64-bit PowerPC architecture

The PowerPC instruction set architecture was designed from the very beginning with both 32-bit and 64-bit computation modes in mind [9]. Thus, the 64-bit PowerPC architecture is a superset of the 32-bit architecture, providing binary compatibility for 32-bit applications. It extends addressing and fixed-point computation to 64 bits. It also supports dynamic switching between the 64-bit mode and the 32-bit mode (not likely to be supported within a single process by the operating system, but rather to allow 32-bit and 64-bit processes to exist simultaneously).

The 64-bit PowerPC execution environment is summarized by a number of properties. The C `long` type (and types derived from it) and all pointer types in 64-bit mode are 64 bits in size. 64-bit applications can make use of 64-bit PowerPC instructions, including 64-bit instructions for loading and storing 64-bit data operands, and for performing 64-bit arithmetic and logical operations. The size of a general machine register is 64 bits in 64-bit mode. The maximum theoretical limit for the size of 64-bit applications, their heaps, stacks, shared libraries, and loaded object files is millions of gigabytes, although the practical limits are dependent on the file system limits, paging space sizes, processor implementation limits on virtual and physical address space, and system resources available. New instructions for 64-bit implementations of the PowerPC architecture are summarized in Table 2.1.

2.6 Motivation for a 64-bit PowerPC port of Jikes RVM

The main motivation for implementating the 64-bit PowerPC port of Jikes RVM comes from the idea of address-order write barriers for the Older-First garbage collector [10, 11].

The Older-First garbage collector is a copying stop-the-world garbage collector that

collects older objects before the younger ones and halts program threads during garbage collection. It notably reduces the amount of copying in comparison with generational collectors with both fixed and variable-size nurseries. On the other hand, the 32-bit Older-First garbage collector suffers from the increased cost of the write-barrier (up to 10 times more expensive). Nevertheless, the savings in copying can prevail over the increased cost of the write-barrier to provide overall performance, which is comparable and in many cases better than performance of generational collectors.

The aforementioned situation leads to a possibility of improving the implementation of the Older-First garbage collector even further by decreasing the cost of the write-barrier. An intuitive solution to this problem comes with the introduction of a 64-bit address space, which may be used to significantly reduce pointer tracking cost.

Newly available inexpensive 64-bit machines, including recently introduced Apple Power Mac G5, make it even more attractive to have an open-source 64-bit VM available.

cntlzd	Count Leading Zeros Doubleword
divd	Divide Doubleword
divdu	Divide Doubleword Unsigned
extsw	Extend Sign Word
fcfid	Floating Convert From Integer Doubleword
fctid	Floating Convert To Integer Doubleword
fctidz	Floating Convert To Integer Doubleword with round toward Zero
lwa	Load Word Algebraic
lwaux	Load Word Algebraic with Update Indexed
lwax	Load Word Algebraic Indexed
ld	Load Doubleword
ldarx	Load Doubleword And Reserve Indexed
ldu	Load Doubleword with Update
ldux	Load Doubleword with Update Indexed
ldx	Load Doubleword Indexed
mulhd	Multiply High Doubleword
mulhdu	Multiply High Doubleword Unsigned
mulld	Multiply Low Doubleword
rldcl	Rotate Left Doubleword then Clear Left
rldcr	Rotate Left Doubleword then Clear Right
rldic	Rotate Left Doubleword Immediate then Clear
rldicl	Rotate Left Doubleword Immediate then Clear Left
rldicr	Rotate Left Doubleword Immediate then Clear Right
rldimi	Rotate Left Doubleword Immediate then Mask Insert
slbia	SLB Invalidate All
slbie	SLB Invalidate Entry
sld	Shift Left Doubleword
srad	Shift Right Algebraic Doubleword
sradi	Shift Right Algebraic Doubleword Immediate
srd	Shift Right Doubleword
std	Store Doubleword
stdcx.	Store Doubleword Conditional Indexed
stdu	Store Doubleword with Update
stdux	Store Doubleword with Update Indexed
stdx	Store Doubleword Indexed
td	Trap Doubleword
tdi	Trap Doubleword Immediate

Table 2.1: New instructions for 64-bit implementations of the PowerPC architecture

Chapter 3

Design

We now describe general design decisions for 64-bit implementations of Jikes RVM. These decisions are relevant not only to the 64-bit PowerPC architecture, but to most of the 64-bit architectures available today. We first consider the layout of various data structures (static data, heap objects, and thread stacks) and then consider effects on signatures and code.

3.1 JTOC Layout

As we mentioned before, the Jikes RVM Table of Contents (JTOC) stores pointers to static fields and methods. In addition to this, the JTOC includes all pointers to the global data structures of the VM, such as classes and virtual method tables (called TIBs, for Type Information Blocks), as well as literals, numeric constants, and references to String constants. Since these data can be of many different types, the JTOC is declared as an int array. Hence, Jikes RVM uses a descriptor array, co-indexed with the JTOC, to identify which entries contain references (so the garbage collector can trace them).

For 64-bit implementations of Jikes RVM, the JTOC allocation unit remains 32-bits,

4-byte aligned. Statics of reference type take 2 array slots, with the second one marked "EMPTY" (but actually containing half of the 64-bit pointer value).

Alignment is an additional consideration that comes up in 64-bit implementations. On the PowerPC, the 8-byte load and store instructions provide only for offsets that are a multiple of 4 (i.e., 4-byte aligned), and the atomic access instructions (used for volatiles and in most updates of object Status words (see below)) require 8-byte alignment. Further, even though most accesses will *work* 4-byte aligned, performance is better when they they are 8-byte aligned. So we 8-byte align all items that are 8 bytes long. Figure 3.1 contrasts the 32-bit and 64-bit versions of the JTOC layout corresponding to these Java declarations:

```
static double a = 3.14D;
static int b = 123;
static Object o = null;
static void foo () { ...  }
```

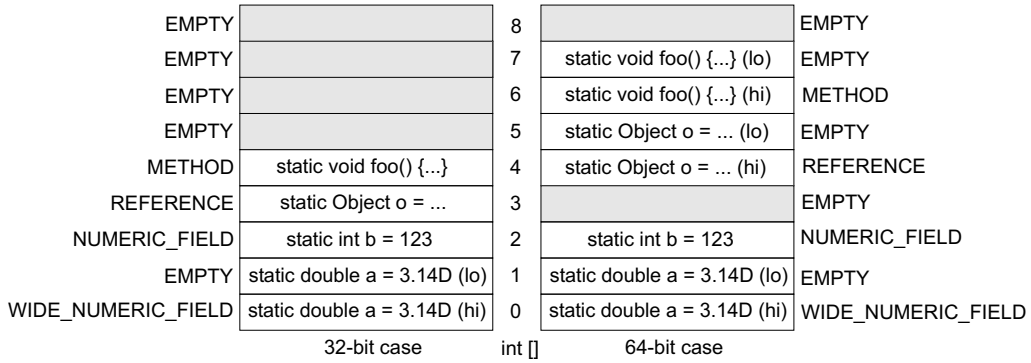

Figure 3.1: Jikes RVM Table of Contents (JTOC) Layout

3.2 Object Layout

Jikes RVM associates a two-word object header with each class instance object. One word of the header is the *status word*. The other word is a reference to the TIB of the object's

class, which describes the object's class, its superclass, the interfaces it implements, and has pointers to the class's virtual methods.

In our 64-bit implementation of Jikes RVM, the header size is 16 bytes, twice that of the 32-bit world. The object status is mostly kept in the low-order part of its 8-byte word. The status must be accessed and updated using 8-byte operations, however, since it can contain a GC forwarding pointer (taking 64 bits since it is a reference). We allocate objects aligned on 8-byte boundaries, so that atomic accesses to the header words are legal.[1] The system uses such accesses when performing monitor (locking) operations on the object, and also during garbage collection to store a forwarding pointer, mark bits, etc.

Part of the design of Jikes RVM is that the header and length word precede the elements of an array (see Figure 3.2), and object fields precede the header for scalar (non-array) objects (see Figure 3.3). The reason for the latter design decision is so that references to the header or fields of a scalar object, or to the header or length of an array, will go to very high memory locations when the object reference is null (zero), i.e., they will wrap around and become "negative". Thus references through null pointers turn into accesses to unmapped memory. This avoids explicit null-pointer checks in most cases.

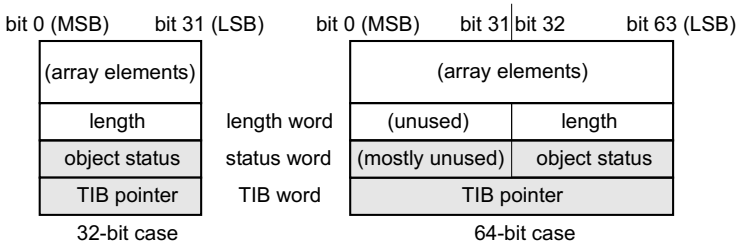

Figure 3.2: Array Object Layout

The figures contrast the 32-bit and 64-bit object layouts, showing that there are only two differences introduced in the 64-bit version. The status word usually uses only the

[1]The LDARX and STDCXr atomic access instructions on the 64-bit PowerPC architecture require 8-byte alignment.

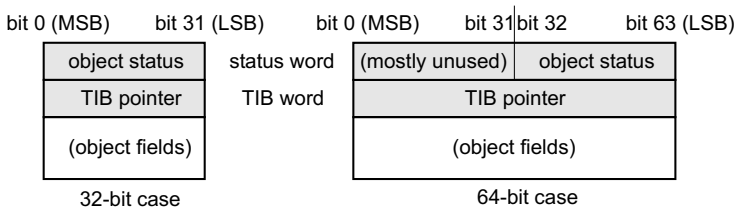

Figure 3.3: Scalar Object Layout

low order 32 bits of its 64-bit slot—but it uses all 64 bits in the case of a GC forwarding pointer, so we cannot pack array headers more tightly. The length word for arrays uses only the low 32 bits of its slot, because the Java language [5] does not permit arrays with more than $2^{31} - 1$ elements, i.e., array indices are always of type int). Given the alignment constraints, and the need for object header fields to lie at fixed offsets from object pointers, it does not appear possible to avoid the unused half-slot associated with array lengths, at least not without a more complex design or significant cost impact to some operations on objects that one would like to be cheap. Fortunately, the space impact in terms of percentage increase in storage needed for arrays is likely to be small in most cases.

3.3 Stack Frame Layout

We first describe generic properties of Java stack frames, mostly derived from the JVM specification. Each JVM thread has its own JVM stack. A JVM stack stores frames and is not manipulated directly except to push and pop frames. In Jikes RVM, these per-thread stacks are declared as arrays of int, merely reserving some memory to be used as a JVM stack. (In the 64-bit implementation we actually use the space as a collection of aligned 8-byte quantities.)

A frame stores a variety of data and partial results, participates in dynamic linking, etc. The JVM creates a frame for each method invocation, and deletes it when the invocation

completes. Each frame, among other things, has its own array of local variables and an operand stack. Figure 3.4 shows the stack layout used in Jikes RVM, contrasting the 32- and 64-bit versions. The 64-bit version simply widens each slot from 32 to 64 bits, as discussed further below. Here are explanations of the abbreviations in the figure. LR is the link register, which receives the return address when performing a call ("branch and link") instruction. MI is the *method identifer*. As it builds a frame, Jikes RVM stores the method identifier of the method being invoked in the frame. Every method in the system has a unique 32-bit identifier. The system uses the identifier to interpret stack frame contents when handling exceptions, during garbage collection, etc. FP stands for the frame pointer; during a method invocation the FP register points to the method invocation's stack frame. Calling and returning update the FP in LIFO fashion. FPR and GPR stand for floating point register and general purpose register. The SP is a stack pointer, used only in methods compiled by the baseline compiler, to refer to the top of the operand stack within the current stack frame (about which more below). The stack grows from high to low memory. The bottom frame (highest in memory) has a special "sentinel" marker, indicated by the MI being -1.

A single slot in the array of local variables or the operand stack of a frame holds a value of type boolean, byte, char, short, int, float, reference, or returnAddress. A pair of consecutive slots may hold a value of type long or double. VM instructions address local variables by indexing. One may access a value of type long or double only by using the lesser index of the pair of slots.

The operand stack starts empty when its frame is created. The JVM provides instructions to push constants, local variables, static fields, object instance fields, array elements, etc., onto the stack; to pop items from the stack and store them into variables, fields, array elements, etc.; to take operands from the operand stack, operate on them, and push the result back onto the operand stack; to prepare parameters to be passed to a method and to receive method invocation results; and other stack manipulations, such as

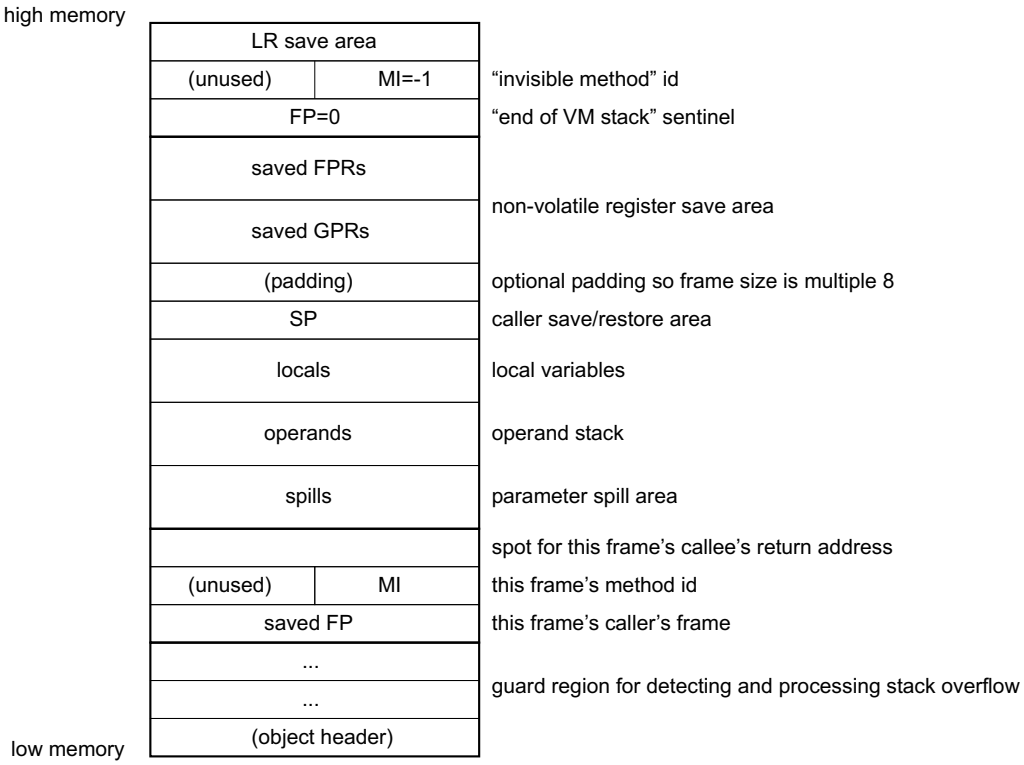

Figure 3.4: New Stack Frame Layout

duplicating and swapping around items near the top of the stack.

Based on the information provided above, stack frame layout in any implementation must strictly conform to the JVM specification, i.e., somehow model it faithfully, so as to allow correct execution with the same semantics as the JVM bytecode instruction set.

The Jikes RVM Base compiler generates native code by producing naïve code for each bytecode, explicitly pushing and popping items on the operand stack and directly modeling the local variable array.[2]

[2]Later versions of the Jikes RVM Base compiler determine the stack height at each bytecode and thus avoid maintaining an explicit stack pointer (SP), but they still generate code that does all the data movement to/from the operand stack.

For this strategy to work, particularly for "untyped" bytecodes such as dup and dup2, which duplicate one or two slots on the top of the stack, we must use the same number of *slots* in the 64-bit implementation as in the 32-bit one. Because references and return addresses must fit in one slot (so as to satisfy the JVM specification), *all* slots need to be 64 bits in size. Thus, we simply "widen" all the stack frame slots in going from 32 to 64 bits, as shown in Figure 3.5. This uses more space than strictly necessary, but affects only thread stacks, which usually do not dominate space consumption. As shown in the figure, an int (or other non-64-bit numeric value) uses only the low-order half of its slot, whereas references and return addresses use the whole slot. Items of type long and double use one whole slot, but have two slots reserved. Again, this wastes space, but only in thread stacks, not objects or statics (globals). In order to offer best performance and to provide for atomic updates of stack frame slots if necessary slots are 8-byte aligned.

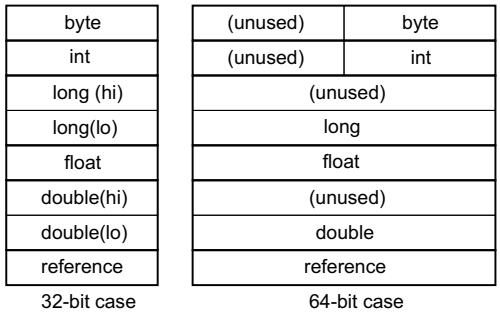

Figure 3.5: Operand Stack Layout

There are two ways to provide correct 32-bit integer arithmetic in these implementations: maintain a correct sign-extended 64-bit value in the operand stack, or store only the low-order 32 bits (leaving the high-order half of the slot undefined). The latter approach is better because it avoids having to sign-extend most integer arithmetic bytecode results so as to guarantee that the result is a proper sign-extended value. (For example, if we add two integers, the result can overflow into the 33rd bit, but Java int arithmetic is modulo 2^{32}.) We always sign-extend 32-bit integers as we load them from

the operand stack, and we store only the low-order 32 bits into the operand stack.

We observe that in an optimizing compiler, it is possible for intermediate results in registers, which are always 64 bits wide, to be correct in the low-order 32 bits but not properly sign extended. This may lead to problems with some operators (division comes to mind, as well as array indexing) so as to require explicit sign-extension in some cases (though many cases may be improved [7]). Comparison is a common case, but fortunately the PowerPC offers 32-bit as well as 64-bit comparison operators, so we simply use the appropriate one.

One final observation we have is that using 64-bit wide stack slots might benefit some 32-bit implementations that model the JVM specification directly. If the processor requires loads of 64-bit longs or doubles to be 64-bit aligned, the only way to guarantee that in the operand stack is to make the slots 64 bits wide. Using a single aligned load avoids using separate instructions to load the two halves of a 64-bit value, which is likely faster and certainly takes less code space. Also, as with the PowerPC, proper alignment in memory gives faster access, since it guarantees that a single access does not cross a cache line or page boundary.

3.4 Method Signatures and Code

There are two somewhat distinct ways in which we encountered the need to change or add methods as we moved from the 32-bit to the 64-bit world. One is support for system calls. In the 32-bit world, all arguments to system calls fit into 32 bits, whether they are integers or pointers; in the 64-bit world we need to distinguish integers and pointers since we prepare arguments and return results of these types differently. More fundamental is the semantic distinction between an int and an *address*.

An interesting property of the design of Jikes RVM is that most of it is written in Java,

including the garbage collectors, thread scheduler, and most of the exception handling mechanism, in addition to the compilers, class loader, etc. In order to support systems programming in Java, Jikes RVM includes a number of so-called *magic* routines [1]. There are magics for system calls to the C library, and magics for performing address arithmetic, among other things. In the version of Jikes RVM we used, a pre-processor rewrites the type ADDRESS to either int (in the 32-bit world) or long (in the 64-bit world). The magic operators that convert between object pointers and ADDRESS therefore have different signatures in the two worlds. We now consider implications of the new mapping of ADDRESS to long in the 64-bit world.

3.4.1 System Calls

To minimize the number of system call interface magic routines that the compiler needs to know about (each one is a special case in compilation), the Jikes RVM convention is to convert all object pointers to type ADDRESS using the objectAsAddress magic. Thus, in the 32-bit world, system calls that take one parameter always take an int, whether that is written as int (for an integer) or as ADDRESS (for a pointer).

In the 64-bit world we encounter the need for a system call magic for one int argument and a separate one for one long argument (what ADDRESS maps to). If we consider a system call that takes k arguments, then we may need up to 2^k different methods in the 64-bit world to deal with all possible combinations of int and ADDRESS.

We considered two approaches to make the necessary changes. One way was to keep the method *name* the same and distinguish the different versions using *signatures*, i.e., overloading the same method name. The other way was to introduce different method names, which encode the desired signature (including a distinction between ADDRESS and int). In the latter case we change a name like sysCall1 (a one argument call) to something like sysCall_I and sysCall_A. This better documents intent, and is therefore a

good software engineering practice. It also helps make sure that we adjust all the *calls* to sysCall1 (since we are making that name go away), and further, that if we change the name in a particular call to sysCall_A, that we ensure that the argument is of type ADDRESS. (If it is not, then compilation will fail in either the 32-bit world, the 64-bit world, or both.) Note that this comes up in part because we want to build the 32- and 64-bit versions from the same source code, with as few "if 32-bit" and "if 64-bit" conditionals as reasonably possible.

Our full system call naming convention is sysCall_, followed by one letter for each argument, then _r, and lastly a letter indicating the result type (X for void). We use I for int, A for ADDRESS, and L for long (which comes up a few times). For example, sysCall_A_rI denotes a call that takes one argument, an ADDRESS, and returns an int result. Table 3.1 summarizes system call changes.

System calls need to be "magic" because they involve a transition to the C world, which uses different calling conventions. All "magic" methods are static methods of the class VM_Magic. All compilers in Jikes RVM recognize calls of these static methods and generate code for them specially (similar to, but distinct from, the "native" mechanism offered at the Java language level).

3.4.2 Semantic Splitting

Substituting ADDRESS for int resulted in what we call *splitting* of system call methods, such as sysCall1 into multiple methods, such as sysCall_I_rX and sysCall_A_rX. This splitting happens because we introduce what amounts to a type distinction, between ADDRESS and int. (The point is even more clear with VM_Address and int, since the former is a reference type from the standpoint of the Java source compiler.)

We found other splitting necessary because of *semantic* distinctions we needed to make. One such change has to do with the size of quantities, specifically of fields. In

sysCall1	
sysCall_A_rX	sysFree, sysMalloc
sysCall2	
sysCall_II_rX	sysNetSocketListen, sysNetSocketNoBlock, sysNetSocketNoDelay, sysWrite, sysWriteByte
sysCall_IA_rI	sysNetSocketAccept
sysCall_AI_rX	sysSyncCache, sysZero, sysZeroPages
sysCall_AI_rI	sysOpen, sysStat
sysCall_AA_rI	sysMUnmap, sysRename
sysCall_AA_rA	sysMMapDemandZeroFixed
sysCall3	
sysCall_III_rX	sysWriteLong
sysCall_III_rI	sysNetSocketLinger, sysSeek
sysCall_IAI_rI	sysArg, sysReadBytes, sysWriteBytes
sysCall_AII_rX	sysFill
sysCall_AAI_rX	sysCopy
sysCall_AAI_rI	sysList, sysMAdvise, sysMProtect, sysMSync
sysCall4	
sysCall_IIII_rI	sysNetSocketBind, sysNetSocketConnect
sysCall_AAII_rI	sysNetSelect
sysCall_AAII_rA	sysMMapGeneralFile, sysMMapNonFile
sysCall_AAAA_rI	sysVirtualProcessorCreate

Table 3.1: New system call naming convention

the heap (instance or static field), an int takes 4 bytes, a long takes 8, and a reference takes 4 or 8 depending on whether the implementation is 32- or 64-bit. In the stack, an int and a reference each take one *slot* and a long takes two, with the slot size varying between the implementations. The upshot is that we needed to distinguish between size in the *heap* and size on the *stack*. It was easy to introduce appropriate methods to do that. The harder part is changing all the callers of those methods appropriately.

A second place where we encountered the need for semantic splitting is making a distinction between int and a machine word. These two were more or less interchangeable in the 32-bit implementation, but clearly not the same in the 64-bit one. A specific example

is the internal routine aligned32Copy, for fast copying of aligned 32-bit arrays. Its original signature was (int dst, int src, int numBytes). We changed this to (ADDRESS dst, ADDRESS src, int numBytes), and also added a method alignedWordCopy with the same signature. The new method can copy faster on the 64-bit platform by doing 64-bit loads and stores; it also assumes that the source and destination are 64-bit aligned (as will naturally happen in the 64-bit heap). In the 32-bit world, alignedWordCopy copies 32-bit quantities and assumes only 32-bit alignment, which is appropriate to that world.

Here are two additional places of note where our port introduced semantic splits. One is the magics having to do with accessing "words" in memory. The original system offered int getMemoryWord(int address); the 64-bit port offers these routines instead:

```
int     getIntAtAddress     (ADDRESS address)
ADDRESS getAddressAtAddress(ADDRESS address)
WORD    getWordAtAddress    (ADDRESS address)
```

We made analogous changes to a number of similar magics.

A second splitting of note has to do with atomic access operations. The 64-bit PowerPC offers load-with-reservation and store-conditionally instruction for both 32 and 64 bit quantities. The magic interface presented these instructions using these magics:

```
int     prepare(Object obj, int offset)
boolean attempt(Object obj, int offset, int oldValue,
                int newValue)
```

These we split, and obtained:

```
int           prepareInt       (Object obj, int offset)
ADDRESS       prepareAddress   (Object obj, int offset)
boolean       attemptInt       (Object obj, int offset,
                                int oldValue, int newValue)
boolean       attemptAddress   (Object obj, int offset,
                                ADDRESS oldValue,
                                ADDRESS newValue)
```

We needed only int and ADDRESS variants because those are the only two kinds of things for which the system needed atomic accesses.

Old Method	**New Methods**
getMemoryWord	getIntAtAddress, getAddressAtAddress, getWordAtAddress
setMemoryWord	setIntAtAddress, setAddressAtAddress, setWordAtAddress
prepare	prepareInt, prepareAddress
attempt	attemptInt, attemptAddress
getSize	getStackSize, getHeapSize
aligned32Copy	aligned32Copy, alignedWordCopy
internalAligned32Copy	internalAligned32Copy, internalAlignedWordCopy

Table 3.2: Summary of method splitting

Chapter 4

Implementation

Having described the overall design decisions we now take a look at specifics of the implementation. In our implementation of the 64-bit PowerPC port of Jikes RVM, we excluded the Opt compiler, adaptive optimization system, JNI (Java Native Interface), and so-called "Watson" (IBM) garbage collectors. The Opt compiler is a relatively large and complicated part of Jikes RVM. We felt that the initial implementation of the 64-bit PowerPC port would be complete enough with only the Base compiler, and would provide us with enough functionality to explore a number of interesting questions specific to the 64-bit world.[1] We comment later on what would be involved in porting the Opt compiler.

The adaptive optimization system in Jikes RVM provides functionality to identify and recompile "hot" methods. It also enables context-insensitive online profile-directed inlining. Thus, it requires both Base and Opt compilers to be present in order to be useful. Since we decided to omit the Opt compiler from our initial porting effort, porting the adaptive system was irrelevant, though we estimate that it would be relatively simple to do (so far as we know, it has no word-length dependencies).

[1] As a practical matter, the Opt compiler would have required many more student-months of effort, in a project assigned to a single graduate student.

JNI, the Java Native Interface, is the native programming interface for Java. It enables programmers to take advantage of platform-specific functionality outside of the Java Virtual Machine. JNI was not required for Jikes RVM itself to be fully functional—it was needed only for a small subset of the Java programs that run on Jikes RVM and that use JNI in some way or another. The vast majority of the programs in which we were interested for our research purposes did not rely on JNI. In addition to these observations, JNI contained a fairly large amount of very platform-specific code, which would require a lot of rewriting. Guided by these facts, we decided to exclude JNI from our initial porting effort as well.

In our choice of garbage collectors, we favored the GCTk (Garbage Collection Toolkit), which provides a unified framework for developing various kinds of garbage collection algorithms. Hence, we ported most GCTk collectors but we left out the older "Watson" collectors written by the run-time team of the Jikes RVM project.

Jikes RVM source code can be divided into an architecture-dependent part, containing the low-level compiler implementation, and a mostly architecture-independent part implementing the run-time system, including the class loader, memory managers, scheduler, etc. We first describe our implementation of the 64-bit PowerPC Base compiler port. We then describe address disambiguation and introduction of new system-wide constants. Later we consider implementation issues related to method splitting, including system calls. Finally we talk about other changes to the Jikes RVM including the bootimage writer, C code, and the configuration and build system.

4.1 Base Compiler

The Base compiler is the simpler of the two compilers of Jikes RVM. Its goal is to generate code that is clearly correct, and to do so simply and quickly. Base compiler

code generation is a straightforward, single pass, bytecode-by-bytecode translation of Java virtual machine instructions into PowerPC machine code.

Our goal was to have a pure 64-bit implementation of the Base compiler, which fully benefits from the new features of the 64-bit PowerPC architecture and the new 64-bit instructions. In passing, we updated the mnemonics used in both the 32- and 64-bit versions of the Base compiler from the old POWER mnemonics to the current PowerPC ones.

The Base compiler in Jikes RVM consists of about a dozen files, with most of the code contained in VM_Assembler.java and VM_Compiler.java. VM_Assembler.java mostly contains methods that emit corresponding PowerPC instructions by generating the necessary bit patterns encoding different instructions and their arguments.

The PowerPC 64-bit mode instruction set is a superset of the 32-bit instruction set. Instructions are 32-bit values, so the type INSTRUCTION remained the same (mapped to int), offsets into compiled code remained divisible by 4, and so on.

We implemented a number of new 64-bit instructions in VM_Assembler to enable their use in the 64-bit PowerPC Base compiler (Figure 4.1). These look much like the instructions previously supported, and are pure additions in the same style.

In addition to new 64-bit instructions, we introduced a number of "macro" instructions to load and store 32-bit integer and word-sized quantities to and from the operand stack (Figure 4.2). These introduce a higher degree of abstraction in code generation and reduce the number of 32- versus 64-bit conditionals elsewhere in the code.

In some cases, we changed the implementation of bytecodes only slightly to guarantee correct 32/64-bit code generation, using the new, more abstract, "macro" instructions (Figure 4.3). In many other cases, Java code implementing the bytecodes was split into two separate versions and/or significantly changed in other ways so as to generate machine code to implement both 32- and 64-bit cases, using new 64-bit opcodes and features of the

64-bit PowerPC architecture (Figure 4.4). (In the example, the 32-bit version calls an out-of-line support routine for dividing long values, whereas the 64-bit version uses the 64-bit hardware instructions.)

Register usage conventions for the 64-bit implementation mostly remained the same, except that values of type long now take only one general-purpose register (or one spill slot). This change additionally affected the interpretation of saved registers in dynamic bridge frames, the loading of registers as a call is generated, and the saving of registers in the method prologue.

Upgrading VM_Assembler and VM_Compiler, and a few other files that directly produce machine code was mostly straightforward and localized. It was time-consuming because of the sheer number of cases one must consider carefully and get right. Also, we ended up doing two or three iterations of changes. For example, we added the "macro" instructions later in the process, when we saw that we were introducing many more 32/64-bit conditionals into the code than we would like.

4.2 Address Disambiguation and System-wide Constants

Address disambiguation and introduction of new system-wide constants constituted the next big step of our porting effort. From the very beginning, the Jalapeño virtual machine, the predecessor of Jikes RVM, was designed to run on only a single combination of architecture and operating system—PowerPC/AIX. This influenced the initial design and implementation of the system in several ways, including the use of the Java int type to represent logically diverse types such as machine addresses, machine words, distances between machine addresses, field offsets, and ordinary integers. At the same time, the system contained numeric literals representing quantities such as "number of bytes in an address", "number of bytes in an int", "log base 2 of the number of bytes in an address",

```
static final int LWAtemplate = 58<<26 | 2;

static final INSTRUCTION LWA (int RT, int DS, int RA) {
  return 58<<26 | RT<<21 | RA<<16 | (DS&0xFFFC) | 2;
}

final void emitLWA (int RT, int DS, int RA) {
  if (VM.VerifyAssertions) {
    VM.assert(fits(DS, 16));
    VM.assert(correctds(DS));
  }
  INSTRUCTION mi =
    LWAtemplate | RT<<21 | RA<<16 | (DS&0xFFFC);
  if (VM.TraceAssembler)
    asm(mIP, mi, "lwa", RT, signedHex(DS), RA);
  mIP++;
  mc.addInstruction(mi);
}
```

Figure 4.1: Load Word Algebraic (LWA) implementation in VM_Assembler.java

and “log base 2 of the number of bytes in an int”, throughout its source code as numeric literals (4, 4, 2, and 2, for these constants).

Implementation of a 64-bit port of Jikes RVM required both system-wide

```
final void emitLint (int RT, int D, int RA) {
  //-#if RVM_FOR_POWERPC32
  emitLWZ(RT, D, RA);
  //-#endif
  //-#if RVM_FOR_POWERPC64
  emitLWA(RT, D, RA);
  //-#endif
}
```

Figure 4.2: Load int implementation in VM_Assembler.java

```
case 0x68: /* --- imul --- */ {
  if (VM.TraceAssembler) asm.noteBytecode("imul");
  //asm.emitLWZ  (T0, 4, SP);
  //asm.emitLWZ  (T1, 0, SP);
  //asm.emitMULLW(T1,T0, T1);
  //asm.emitSTWU(T1, 4, SP);
  asm.emitLint (T0, BYTES_IN_WORD, SP);
  asm.emitLint (T1,             0, SP);
  asm.emitMULLW(T1, T0, T1);
  asm.emitSTWU (T1, BYTES_IN_WORD, SP);
  break;
}
```

Figure 4.3: imul bytecode translation in VM_Compiler.java (commented-out code corresponds to the initial 32-bit only implementation)

disambiguation of addresses from other values and replacement of numerous numeric literals in the source code with system-wide symbolic constants that could be set once, during the build process.

For the purpose of address disambiguation, we introduced a new name, ADDRESS, which we used throughout the system to represent constants or variables storing machine addresses or distances between them.[2] Another name, WORD, was used to represent constants or variables storing machine words. We relied on a feature of the Jikes compiler[3] that allows mapping a name to a type keyword. For the 32-bit implementation, both ADDRESS and WORD are mapped to Java int, which is a 32-bit integer type. For the 64-bit implementation, we map ADDRESS and WORD to Java long, a 64-bit integer type. The last class of values, offsets within objects or single tables, remained of type Java int, since they will always fit into a 32-bit variable even in 64-bit environments.

At the same time, we introduced a set of system-wide constants into VM_Constants

[2]This substitution had already begun within the GCTk code, but had not propagated further and had never been tested for the 64-bit target.

[3]This is a Java source code to bytecode compiler that is entirely separate from the Jikes RVM.

(Figure 4.5), and replaced all uses of numeric literals in the source code with these new constants. These changes produced a 32/64-bit clean implementation of the system; we give a code comparison example in Figure 4.6. In practice, going through all the code in the system to find uses of numeric constants and replacing them with appropriate symbolic names was one of the most time-consuming aspects of the port. There are two reasons for this: the volume of code we needed to peruse, and the level of understanding we needed of each bit of code in order to substitute the proper symbolic name.

4.3 Method Splitting

Changes in method signatures and code, described in Section 3, involved two steps in terms of implementation. The first step was to implement new methods, conforming to new naming conventions. The second step was to identify the parts of the system that used those methods with changed signatures and to rewrite the calls using the new conventions. The fact that many places that needed adjustment to use new conventions coincided with places that we had to address-disambiguate in the previous step of the port helped to identify parts of the code that used methods with changed signatures. It was helpful that we had changed the signatures, since we thus got Java source compilation errors at those places that needed updating.

4.4 Miscellaneous Changes

In order for Jikes RVM to run and be able to load and compile additional classes, a certain set of basic functionality (a classloader, an object allocator, a compiler, etc.) must be compiled into an executable *boot image*. The boot image is a copy of Jikes RVM in memory, written to a file. A short program, called the *boot-image runner*, written in C, loads a boot image back to memory and executes it, branching to a designated startup

method. A boot image is created by a special program, the *boot-image writer*. The boot-image writer is an ordinary Java program and can run on any JVM.

Extending the boot-image writer to generate a 64-bit boot image involved most of the previously described modifications we made to Jikes RVM. In other words, it included address disambiguation and introduction of new system-wide constants, changing method naming conventions and method signatures, and implementing separate 64-bit method cases in some parts of the code, using preprocessor directives.

Only a very small part of Jikes RVM is written in C. This is the most low-level part of the code responsible for access to the file system, the network, and processor resources, using the standard C library. Half of this code includes simple functions that convert parameters and return values between Java and C formats. The other half of the C code consists of the boot-image runner and two signal handlers, one for handling hardware traps and trap instructions and one for passing timer interrupts to Jikes RVM.

In C code we use the __64BIT__ preprocessor flag to implement 64-bit cases of certain functions along with ADDRESS, WORD, and UWORD typedef identifiers, which map to appropriate signed or unsigned values in the 32-bit and 64-bit environments.

The Jikes RVM configuration and build system consists of the main jconfigure script, a number of additional helper scripts, a set of predefined configurations for different architecture/OS configurations, and a set of predefined configurations of the Jikes RVM (including compiler combinations and the type of garbage collector and allocator used).

The required changes in the configuration and build system for 64-bit PowerPC configurations included creating a new architecture/OS configuration powerpc-ibm-aix4.3.3.0-64, defining a new RVM_FOR_POWERPC64 preprocessor variable in the jconfigure script, and modifying jconfigure to include script cases for the new preprocessor variable.

```
case 0x6d: /* --- ldiv --- */ {
  if (VM.TraceAssembler) asm.noteBytecode("ldiv");
  //asm.emitLWtoc(T0, VM_Entrypoints.longDivideOffset);
  //asm.emitMTLR(T0);
  //asm.emitLWZ   (T1, 12, SP);
  //asm.emitLWZ   (T0,  8, SP);
  //asm.emitLWZ   (T3,  4, SP);
  //asm.emitLWZ   (T2,  0, SP);
  //asm.emitCall(spSaveAreaOffset);
  //asm.emitSTW  (T1, 12, SP);
  //asm.emitSTWU (T0,  8, SP);
  //-#if RVM_FOR_POWERPC32
  asm.emitLwordtoc(T0, VM_Entrypoints.longDivideOffset);
  asm.emitMTLR(T0);
  asm.emitLWZ (T1, 3 * BYTES_IN_WORD, SP);
  asm.emitLWZ (T0, 2 * BYTES_IN_WORD, SP);
  asm.emitLWZ (T3,     BYTES_IN_WORD, SP);
  asm.emitLWZ (T2,                 0, SP);
  asm.emitCall(spSaveAreaOffset);
  asm.emitSTW (T1, 3 * BYTES_IN_WORD, SP);
  asm.emitSTWU(T0, 2 * BYTES_IN_WORD, SP);
  //-#endif
  //-#if RVM_FOR_POWERPC64
  asm.emitLD    (T0, 2 * BYTES_IN_WORD, SP);
  asm.emitLD    (T1,                 0, SP);
  asm.emitTDIEQ0(T1);
  asm.emitDIVD  (T0, T0, T1);
  asm.emitSTDU  (T0, 2 * BYTES_IN_WORD, SP);
  //-#endif
  break;
}
```

Figure 4.4: ldiv bytecode translation in VM_Compiler.java (commented-out code corresponds to the initial 32-bit only implementation)

```
//-#if RVM_FOR_POWERPC32
static final int BYTES_IN_ADDRESS_LOG = 2;
//-#endif
//-#if RVM_FOR_POWERPC64
static final int BYTES_IN_ADDRESS_LOG = 3;
//-#endif
static final int BYTES_IN_INT_LOG = 2; // defined by Java
static final int BYTES_IN_INT = 1<<BYTES_IN_INT_LOG;
static final int BITS_IN_BYTE_LOG = 3; // defined by Java
static final int BITS_IN_BYTE = 1<<BITS_IN_BYTE_LOG;
static final int BITS_IN_INT_LOG = 5;  // defined by Java
static final int BITS_IN_INT = 1<<BITS_IN_INT_LOG;
static final int BITS_IN_ADDRESS_LOG =
  BYTES_IN_ADDRESS_LOG + BITS_IN_BYTE_LOG;
static final int BITS_IN_ADDRESS = 1<<BITS_IN_ADDRESS_LOG;
static final int BITS_IN_WORD_LOG = BITS_IN_ADDRESS_LOG;
static final int BITS_IN_WORD = BITS_IN_ADDRESS;

static final int MAX_INT = 0x7fffffff;
static final int BYTES_IN_SHORT_LOG = 1;
static final int BYTES_IN_SHORT = 1<<BYTES_IN_SHORT_LOG;
static final int BYTES_IN_LONG_LOG = 3;
static final int BYTES_IN_LONG = 1<<BYTES_IN_LONG_LOG;
static final int BYTES_IN_DOUBLE = 8;
```

Figure 4.5: System-wide constant declaration in VM_Constants.java

```
int beg = VM.objectAsAddress(instructions);
private static int obsoleteMethodCount;
                  versus
ADDRESS beg = VM.objectAsAddress(instructions);
private static int obsoleteMethodCount;

int tibIndex = method.getOffset<<2;
                  versus
int tibIndex = method.getOffset<<BYTES_IN_ADDRESS_LOG;
```

Figure 4.6: Address disambiguation and new system-wide constants in practice

Chapter 5

Historical

This chapter describes the differences between Jikes RVM versions 2.0.3 and 2.3.0.1

5.1 “Magic”

Jikes RVM 2.3.0.1 implements a set of new magical classes: VM_Address, VM_Word, VM_Offset, and VM_Extent, which are used in parts of the runtime and the garbage collector.

VM_Address in particular is used to represent a raw machine address type, which is naturally machine-dependent. In our 64-bit port of Jikes RVM 2.0.3, we used ADDRESS for this purpose, which was replaced with either Java int or long by the Jikes source-to-bytecode compiler. This approach was simple to implement, but it had several disadvantages. First, it lacks appropriate abstraction. Second, Java int and long are signed, whereas machines addresses are unsigned values. This difference limited usable address space in 32-bit implementation of Jikes RVM 2.0.3 to 2GB.

VM_Address supports a number of necessary methods such as methods for adding an integer offset to an address to calculate another address, computing the difference of two addresses, and comparison operations. At the same time, it does not implement operations like address multiplication, which make sense for Java int, but no sense whatsoever for addresses.

To use a Java object to represent a raw machine address type would have been extremely inefficient. Instead, when the Jikes compiler encounters creation of a VM_Address object, it returns a primitive value that represents an address for a particular platform. This means that currently address maps to a 32-bit or 64-bit unsigned integer.

5.2 Memory Management

Starting from Jikes RVM 2.2.0, the *Java Memory management Toolkit* (JMTk) became the default memory management system of Jikes RVM. It replaced so called "Watson" collectors in Jikes RVM 2.0.3 and phased out the development of GCTk.

JMTk is more portable, modular, and object-oriented toolkit than GCTk. It consists of three major components: plans (SemiSpace, MarkSweep, GenCopy, GenMS, CopyMS, and NoGC); policies (mark-sweep collection, free-list allocation, bump-pointer allocation, etc.); and utilities (load-balancing parallel queues, sequential store buffers, etc.).

5.3 Baseline Compiler

Since Jikes RVM 2.0.3, the baseline compiler has undergone a major restructuring to increase the amount of platform-independent code to about 2/3. Principal changes include: a rewrite of PowerPC baseline compiler to enable it to generate code for both 32 and 64-bit PowerPC without preprocessor conditionals, elimination of explicit use of the

SP register by computing offsets from the FP at compile time, and uniform use of PowerPC mnemonics (instead of mixed POWER and PowerPC) for the instructions in VM_Assembler.

5.4 Miscellaneous

Other important changes since Jikes RVM 2.0.3 include: introduction of packages in Jikes RVM source code, switching entirely to the GNU Classpath libraries, implemention of support for separate name spaces for classloaders, and implementation of JNI for Linux/PowerPC.

Chapter 6

Conclusions

We have described the design and implementation of a port of Jikes RVM to the 64-bit PowerPC architecture. In the port, we addressed a number of issues that came up because of the differences between modern 32-bit and 64-bit architectures. We introduced new types ADDRESS and WORD and a number of system-wide constants to disambiguate variables and code dealing with machine addresses. In parallel, we updated a number of method signatures and respective code to allow Jikes RVM to treat correctly both four-byte and eight-byte addresses. We believe that major design decisions we made and implemented during our effort will make porting of Jikes RVM to other 64-bit architectures substantially easier. In fact, we are currently porting Jikes to the IA-64 architecture.

Our effort, described in this thesis, provides a number of useful insights to people involved in virtual machine research and development and will help them in design and implementation of future robust and portable software. Several design factors are important for this task. The most important observation is that one should avoid making simplifying assumptions about the system. If certain values are the same in terms of size and can be stored in a variable of the same type, it is still very important to preserve semantic distinctions between different types of values (int, ADDRESS, and WORD

versus int), because on other architectures their size may be different. This will also in many cases resolve the problem of method signature changes and will help to come up with appropriate method naming conventions (prepareInt and prepareAddress versus prepare). In any case, method naming conventions must be selected carefully without being closely related to a particular architecture to avoid unnecessary disambiguation later (setEightBytes versus setDoubleWord). System-wide constants should be used in as many places as possible to further enhance the portability of the system.

Porting Jikes RVM to the 64-bit PowerPC architecture created a number of interesting research opportunities and introduced the first 64-bit, freely available, open-source virtual machine to the academic and research community. It is also worth mentioning that portability was not an initial design goal for Jikes RVM. Thus, our experience probably covers a large set of major issues that will come up during the porting of Java virtual machines to different architectures.

Chapter 7

Future Work

We contributed the source code of our 64-bit PowerPC port of Jikes RVM 2.0.3 to IBM. Currently there is an effort to create a 64-bit PowerPC port of CVS head (version 2.3.0.1+) of Jikes RVM at IBM. As of November 2003, this port is still not functional mostly due to the problems with JNI, which is required for GNU Classpath. Thus, future work will include debugging of the 64-bit PowerPC port of CVS head of Jikes RVM to provide the same functionality as we achieved in Jikes RVM 2.0.3 plus JNI and JMTk (without optimizing compiler and adaptive infrastructure). Further improvements will include adding a 64-bit pure PowerPC optimizing compiler and adaptive compilation functionality.

References

[1] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Mark Mergen, Ton Ngo, Janice Shepherd, and Stephen Smith. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, 1999.

[2] Bowen Alpern, Maria Butrico, Anthony Cocchi, Julian Dolby, Stephen Fink, David Grove, and Ton Ngo. Experiences porting the Jikes RVM to Linux/IA32. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium (Java VM '02)*, 2002.

[3] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *Proceedings of the 2001 ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, 2001.

[4] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, 1984.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[6] Matthew Hertz, Stephen M Blackburn, J Eliot B Moss, Kathryn S McKinley, and Darko Stefanović. Error free garbage collection traces: How to cheat and not get caught. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, volume 30(1) of *ACM SIGMETRICS Performance Evaluation Review*, pages 140–151. ACM, June 2002.

[7] Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. Effective sign extension elimination. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI 2002)*, 2002.

[8] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[9] Cathy May, Ed Sikha, Rick Simpson, and Hank Warren. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann, 2nd edition, 1994.

[10] Darko Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts Amherst, 1999.

[11] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In *Proceedings of the 1999 ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, 1999.