Implementation and Evaluation of Fast Untyped Memory in a Java Virtual Machine

---

Study Thesis in Computer Science

by

Isabella Thomm

born October 2, 1983, in Vaihingen a.d. Enz

Department of Computer Science

Distributed Systems and Operating Systems

University of Erlangen-Nuremberg

Tutors: Dipl. Inf. Andreas Gal
Dipl. Inf. Christian Wawersich
Prof. Dr.-Ing. W. Schröder-Preikschat
Prof. Michael Franz

Begin: April 1, 2006
Submission: July 15, 2006

Implementierung und Auswertung von schnellem untypisiertem Speicher in einer Java Virtual Machine

---

Studienarbeit im Fach Informatik

vorgelegt von

Isabella Thomm

geb. am 2. Oktober 1983 in Vaihingen a.d. Enz

Angefertigt am

Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Dipl. Inf. Andreas Gal
Dipl. Inf. Christian Wawersich
Prof. Dr.-Ing. W. Schröder-Preikschat
Prof. Michael Franz

Beginn der Arbeit: 1. April 2006
Abgabe der Arbeit: 15. Juli 2006

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäss übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 15. Juli 2006 _____

Abstract

The Java virtual machine (JVM) was developed to execute bytecode programs written in the object-oriented Java programming language and is part of the Java runtime environment (JRE). Virtual machines allow the execution of programs in a controlled environment. JVMs are available for all prevalent platforms.

JSim is an emulator, that allows to execute legacy applications on modern host architectures by simulating a legacy instruction set architecture (guest architecture) on top of the Java virtual machine. JSim currently supports ARM, MIPS and PowerPC as guest architectures.

As a type-safe programming language, Java does not directly support the notion of raw memory as it occurs in physical machines, which imposes a problem when trying to emulate a physical machine architecture in a JVM. Various ways exist to simulate physical memory using pure Java, however, each suffers from significant performance loss due to Java runtime checks.

This work presents a performant solution to emulating raw memory in a JVM by extending existing JVMs with methods to access a memory area of the host system. A number of possible implementation approaches were explored and evaluated against each other. The first implementation extends an existing Java bytecode interpreter to intercept certain magic methods, replacing them with extended bytecodes that provide efficient access to untyped memory. In a second implementation, a Java just-in-time (JIT) compiler was extended to generate code that facilitates untyped memory access. To explore the impact of different host platforms on the overall performance of both approaches, an Intel x86 architecture and a PowerPC platform with different implementation ideas were evaluated against each other.

The developed prototype shows a significant performance improvement compared to the pure Java-based memory emulation. Particularly for memory intensive applications the execution time was nearly halved.

Zusammenfassung

Die Java Virtual Machine (JVM) wurde entwickelt, um Programme, die in der objektorientierten Programmiersprache Java geschrieben und dann zu Java Bytecode kompiliert wurden, auszuführen und sie ist Teil des Java Runtime Environments (JRE). Virtuelle Maschinen erlauben die Ausfhrung von Programmen in einer sicheren Laufzeitumgebung und sind für alle gängigen Plattformen erhältlich.

JSim ist ein Emulator, der es ermöglicht, Altanwendungen auf modernen Wirtsystemen auszuführen, indem er eine Instruktionssatzarchitektur (Gastarchitektur) in einer JVM simuliert. Derzeit unterstützt JSim die Gastarchitekturen ARM, MIPS und PowerPC.

Aufgrund des typsicheren Konzepts unterstützt Java keinen untypisierten Speicher, so wie er in physischen Maschinen existiert. Insbesondere stellt dies ein Problem dar, wenn man eine physische Maschine in einer JVM emulieren möchte. Es existieren zwar verschiedene Ansätze zur Simulation von physischem Speicher in einer JVM unter Verwendung reiner Java Sprachmittel, aber aufgrund der Laufzeitverifikationen von Java leiden all diese Ansätze unter Leistungseinbußen.

Diese Arbeit stellt eine Lösung für den effizienten Zugriff auf untypisierten Speicher aus einer JVM vor. Hierfür wurden verschiedene Implementierungen entwickelt und miteinander verglichen. Zunächst wurde ein bereits existierender Java Bytecode Interpreter dahingehend erweitert, sogenannte magische Methoden abzufangen und sie durch erweiterte Bytecodes zu ersetzen. In einer zweiten Implementierung wurde ein Java Just-In-Time-Compiler (JIT) so verändert, dass er Code generiert, der den direkten Speicherzugriff möglich macht. Um nun die Auswirkungen verschiedener Wirtsarchitekturen auf die Laufzeit zu zeigen, wurden eine Intel x86 und eine PowerPC Architektur zum Vergleich herangezogen und verschiedene Konzepte implementiert. Die Ausführungszeit hat sich im Vergleich zu der Java basierten Speicheremulation erheblich verbessert und wurde für speicherintensive Applikationen sogar fast halbiert.

Contents

Chapter 1

Introduction

Virtual machines gain more and more relevance in the computer science field these days, since they make it possible to execute Java programs on every platform a VM was developed for.

Besides portability, the execution of programs within a JVM environment offers a remarkable level of security mechanisms such as type-safety and the elimination of buffer overflows, that often occur in unsafe languages like C and allow to execute foreign or malicious code.

Frequently, it is necessary to rely on established legacy applications, but sometimes obsolete hardware forms the basis of the execution of that software. Instead of rewriting these programs, which would entail immense costs and development time, a virtual execution environment for executing legacy binaries on top of a JVM was implemented [Sti05]. The Java Simulator JSim compiles the native code of the application to Java bytecode, which is then in turn recompiled to machine code of the host architecture by a JVM (figure 1.1). JSim is part of the VEELS (Virtual Execution Environment for Legacy Software) project.

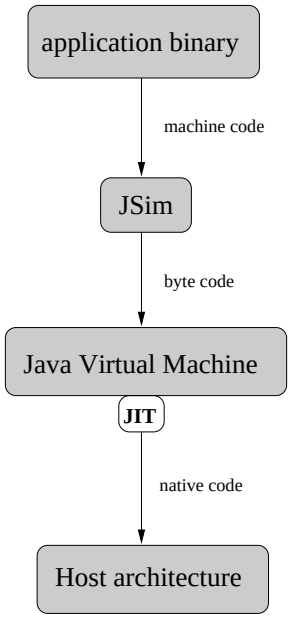


Figure 1.1: Legacy binary execution on top of a JVM

Unfortunately Java's type-safety implicates a problem when trying to emulate a physical machine on top of the virtual machine, since the concept of Java is not designed to support access to untyped memory. The performance of JSim is poor. Benchmarks have shown that the memory module of JSim is a major bottleneck. A promising approach for improving the performance of JSim thus is the development of a more efficient memory module. The idea is to extend an existing JVM with methods that

provide Java applications with efficient access to untyped memory [GPF05]. The implementation covers both the Intel x86 and the PowerPC architecture under Linux.

This thesis is structured as follows: In chapter 2 the existing memory module and its problems are presented and the basic concept of the developed untyped memory module is explained. Chapter 3 introduces the bytecode interpreter JamVM [Rob06] and describes the extensions made to support the untyped memory module. Chapter 4 gives an overview on the basic Jikes RVM [AAB+00] architecture components and illustrates the implementation of untyped memory in a just-in-time (JIT) compiler and related optimizations. The thesis concludes with the evaluation of the new storage construct and benchmarks that show the performance improvements in chapter 5.

Chapter 2

Conceptual Design

The next sections describe the hitherto existing memory module (*SafeMemory*) and the improved new approach to implement untyped memory more efficiently (*DirectMemory*).

2.1 Interface of a Memory Module

A memory module has to offer methods, that allow to access a memory area similar to the load and store instructions of a processor. The modules therefore provide methods to read and write the data types byte, short, integer and long in both little and big endian variants at specific addresses of emulated memory.

2.2 The SafeMemory Module of JSim

Physical memory can be considered as a sequence of bytes. The straightforward solution is to emulate the memory as a flat array of bytes. However, Java initializes arrays with zero, which causes that the underlying operating system allocates physical memory for the whole array, quickly consuming the entire virtual memory. Therefore a flat array does not pose a feasible solution.

To avoid this problem, SafeMemory introduces an additional level of indirection. The memory is divided into pages of 4 kB size, which are managed in a page table. Each page is implemented as an array of 1024 integers, because word access is the most frequent access type. The page table is an array of 512k references to pages, resulting in a total size of 2 MB for the page table and emulated address space of 2 GB. Limiting the address space to 2 GB allows JSim to calculate addresses with signed Java data types.

A virtual address is transformed to a page index and an offset within the page by logically shifting right the virtual address (division by 1024) to determine the page index and clearing all but the least significant 12 bits to determine the offset.

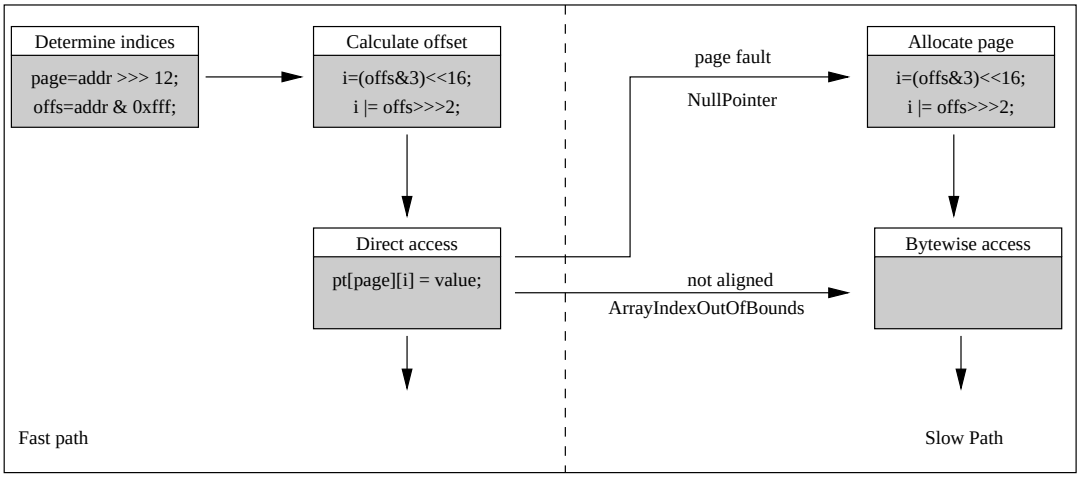


Figure 2.1: Address translation in the SafeMemory module as described in [Sti05]. This example shows the procedure for the memory access after an invocation to `putInt(addr, val)` to write an integer to a specific location.

JSim utilizes Java exceptions to benefit from JVM internal checks to handle page faults and unaligned access. Though Java using Java exceptions is expensive, it only affects the rarely taken paths of a page fault and unaligned access (slow path). A page fault does only occur once per page. Unaligned word access is not supported on many architectures. On Intel x86 it is possible, but much slower than aligned access and compilers therefore generate code with correct alignment.

While the most frequent path of aligned access to an already allocated page is not slowed down by additional verifications (fast path). Figure 2.1 illustrates the possible code paths of a memory access.

Pages are allocated upon access. Initially the page table is filled with `null` references. By accessing an unallocated page the JVM generates a `NullPointerException`, that designates a page fault. This exception is caught in the SafeMemory module and the page is allocated.

Words at unaligned addresses span two integers in a page. To detect unaligned access using JVM internal checks a trick is applied. The two least significant bits are shifted to position 16 and 17 in the generated offset, creating an offset greater than 1024. Using this offset causes an `ArrayIndexOutOfBoundsException`, that is caught and the slow path is entered, where the word is constructed using byte-wise access.

Compared to memory access on a physical machine, an emulated memory access through the SafeMemory module is multiple times slower. This is caused by the runtime checks of the JVM and the overhead of the Java method invocation. A more efficient implementation requires support by the JVM.

2.3 DirectMemory

The basic idea of the DirectMemory module is to provide the Java applications with access to a block of untyped memory of the host system. In the following, only the conceptual design of the DirectMemory module for Intel x86 architecture is described, because it allows for a high-performance solution. A PowerPC approach was only implemented for Jikes RVM for comparison and is described in section 4.7.

In the initial boot phase of the JVM a memory block with a particular size is allocated from the heap of the JVM process and the base address is stored in the FS register. This register is not utilized by applications and compilers do not generate code, that uses it in any way. Though the kernel uses this register, it is the task of operating system to save and restore the register state. The methods of the memory module interface (section 2.1) are provided as *magic methods* that allow JSim to access the allocated block of untyped memory. The memory is used via segment-relative addressing and the MMU (Memory Management Unit) is responsible to assure that these accesses do not violate any segment limits.

2.3.1 Magic Methods

The basic means to implement DirectMemory is the existence of so-called magic methods. These are necessary in order to perform actions, that are impossible in a type-safe high-level-language such as Java, e.g. the invocation of operating system calls or garbage collection performing unsafe casts in Jikes RVM [use05].

Magic methods were first applied in the programming language Oberon [WG05] developed by Niklas Wirth and Jürg Gutknecht and look like ordinary methods, but the compiler intercepts calls to those methods, ignores the method body and generates specific code for them at runtime. Thus it is not a surprise that these methods are empty. A disadvantage is that the deployed JVM has to be modified to intercept magic methods, but the result is much more efficient than other approaches.

Another possible solution is using native code invoked through the *Java Native Interface* (JNI) [Sun03]. These native method calls are significantly more expensive than invocations to Java methods due to encapsulation of parameters and return values accordant to the JNI specification. Code for a magic method can directly be produced at the call-side replacing the call to the magic method, which saves the overhead for the method invocation. This is most important with short code sequences.

2.3.2 The Intel x86 Architecture

The Intel x86 processor belongs to the CISC (Complex Instruction Set Computer) family. Compared to a RISC (Reduced Instruction Set Computer) its opcodes are more complicated and the instructions can perform more complex actions. Since the decoding of

instruction prefix	opcode	ModR/M byte
0x64 1 byte	0x89 1 byte	00 src dst 1 byte

Figure 2.2: Encoding of the `movfs` instruction with register-indirect register addressing

CISC instructions is more expensive, modern x86 processors are equipped with another functional unit, that translates complex opcodes to RISC instructions and passes them on to the CPU in order to exploit advantages of the RISC technology [Fog06]. Intel x86 machines feature a narrower and more specialized register set than PowerPC.

Memory management [Int99] on the Intel x86 architecture consists of two main mechanisms: segmentation and paging. While segmentation is used for protection and isolation, i.e. division of the memory into segments of different size such as code, stack and data segment, paging is essential, because it allows to retrieve memory on per page basis and thus allows to have a greater addressable memory space than physical memory is available.

2.3.3 Format of an Intel Machine Code Instruction

In order to understand the implementation of untyped memory via the FS selector register, a closer at the Intel's instruction layout has to be taken [Int97]. X86 instructions do not have a fixed opcode length. Generally all instructions share a common encoding scheme: The first part of that consists of up to 4 prefix bytes, followed by an opcode number of 1–3 bytes length. Optionally a ModR/M and a scale-index-base (SIB) byte can be appended and the end is formed by a displacement and an immediate value if applicable.

Figure 2.2 depicts the encoding of a specific variant of the `movfs` machine instruction for the register-indirect register addressing mode. This mnemonic does not directly exist in the Intel instruction set, rather it is a `mov` opcode, that uses the FS instead of the DS segment. It is introduced to facilitate the memory access in the JIT compiler, which is explained in section 4.4. In the ModR/M byte both the src holding the value and the dst holding the address stand for three bits, respectively, encoding one of the eight possible registers, that contain the appropriate information. The "00" in the ModR/M byte denotes that the memory address resides in the dst register. Further encoding values are described in [Int97].

Instruction prefixes can be prepended to an opcode to obtain a special execution mode and can be divided into four categories:

- *Lock* and *Repeat*: While Lock can set exclusive memory access in multiprocessor

systems, Repeat is only responsible for string operations.

- *Segment override* to employ another segment than the default DS segment as used in figure 2.2
- *Operand-size override* to switch between 16-bit and 32-bit operands
- *Address-size override* to switch between 16-bit and 32-bit addresses

The opcode field contains the encoding of an instruction and can—since the SSE extension of Intel Pentium 3 processors [sse06]—have a maximum length of three bytes. It can be extended with further three bits through the ModR/M byte. The *displacement* determines the offset of a memory address, whereas *immediate* is a fixed numerical value of one, two or four bytes in length, in case an opcode uses an immediate operand.

If the operand of an instruction resides in memory, the ModR/M and SIB bytes become relevant, which define the addressing mode in detail: The ModR/M byte is partitioned into three fields. The first two of them, the mod and the R/M part together can accept 32 values, which cover eight registers and 24 possible addressing modes. The third reg/opcode field can, as already alluded, extend opcodes and encode both a register and an additional instruction information according to requirements. How these bits are used, is defined by the primary opcode. In case that the mod and R/M parts are not combined, the R/M bits can select a register.

Depending on what addressing mode is required, another so-called SIB byte is necessary to represent it, that contains the scale factor, the index and the base register. This is the case, if the mod field contains a memory location and bits 0-2 encode the stack pointer register (esp).

2.3.4 Segments, Selectors and Descriptors

Access to the untyped memory block is provided by a set of magic methods. Figure 2.3 shows an example of such a magic method. The magic method is provided with an address of the virtual address space of the emulated program. This address can be considered as an offset into the untyped memory block and can be converted to an address of the virtual address space of the JVM process by adding it base address of the untyped memory block.

```
public static int
mr4(int addr, int value) {
   return 0;
}
```

Figure 2.3: Example for the magic method `mr4` for reading an integer from untyped memory

To calculate the address as efficient as possible the operation is performed by the MMU. The beginning of the FS data segment is set to the beginning of the untyped

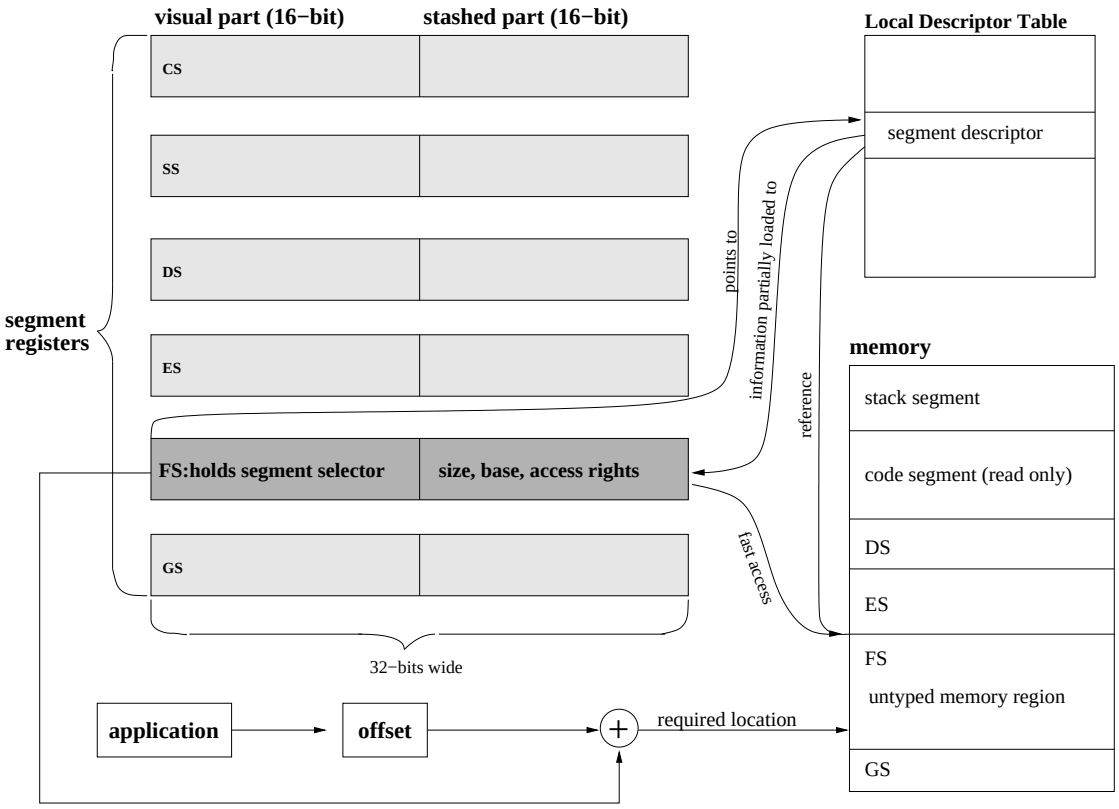


Figure 2.4: Address translation process via segment registers

memory block. The MMU calculates the *linear* address from a *logical* address, that consists of a segment selector and a displacement. A selector is a 16-bit wide identifier for a segment and a reference to one of 8192 entries in a descriptor table. On Intel x86 exist two tables of that kind, the global (GDT) and the local descriptor table (LDT). The latter is a per process memory management table and relevant for the provision of untyped memory. By means of this table a logical address is translated into a linear address, by which a byte is located in the linear address space. In order to map the beginning of the untyped memory block to the beginning of the FS segment, the relevant entry in the LDT has to be modified. This comprises a descriptor ID, to identify an entry in the LDT, the granularity flag, the base address of the FS segment and thus the beginning of the untyped memory region and the segment limit. The granularity bit constitutes whether the segment limit is interpreted in byte or 4096 byte blocks. This information is then passed to the `modify_ldt()` Linux system call to modify the LDT entry. Finally a FS selector is provided with the Table Indicator Flag, which specifies whether GDT or LDT is requested, and a descriptor ID to reference the table entry. The selector is loaded into to the visible part of the segment register, which can only be done by assembler instructions.

Figure 2.4 shows the correlation between segments, LDT and resulting memory location: The six available 32-bit segment registers of an Intel x86 processor are depicted here. These registers consist of a visual part holding the segment selector and a stashed part (shadow register), that saves size, base address and access rights of a once accessed segment to accelerate address translation. If the segment has not been utilized yet, the information has to be retrieved from the LDT and then stored into the shadow register. The suitable LDT entry has a reference to the required memory segment. The JSim application delivers the offset to calculate a logical and linear address.

It is sufficient to load the selector only once at the virtual machine start-up. The shadow register is once filled with access rights, base and limit address information, with the ambition to speed up the address translation process.

Chapter 3

Interpreter JamVM

The JamVM [Rob06] is a relatively small, fully functional JVM and was excellent for testing the idea of untyped memory by means of the FS selector register and if it causes a remarkable performance increase. It supports the Intel x86, AMD Athlon 64, PowerPC and ARM architecture at the moment and it is mainly written in C, except for certain target machine dependent assembler code. JamVM features only an interpreter mode, i.e. it does not compile high-level language instructions to machine code, but deconstructs and executes the read bytecode at runtime. This makes it simple to port JamVM to different architectures. Running JSim on an interpreter is still very slow, but with the DirectMemory extension faster than the SafeMemory mode.

The JamVM implementation covers POSIX threading, synchronization routines, signals, class loaders, JNI and has support for a special native interface for JVM specific functions to save the overhead for JNI invocations. A mark and sweep garbage collector is responsible for memory management and it supports object finalization.

3.1 Outline over the JamVM Components

An illustration of the components described in the following is given in figure 3.1. In the beginning the `jam` module is responsible for initializing the JamVM, that is to constitute the upper and lower heap boundaries, platform dependent preferences such as setting the standard 80-bit Intel x86 Linux precision to 64-bit for the interpreter to avoid roundoff errors [RH01] and the dimension of the stack that is 64KB by default. Furthermore a class loader is created, which determines the main Java class with its main method to find the execution entry point. When the static main method is executed a so-called `ExecutionEnvironment` is constructed and a stack frame is attached to the stack and the parameters of the Java main method are pushed onto it. Here the `executeJava()` method is called, which marks the entry point to the `interp` component, that is the core element of the JVM and comprises the main interpreter loop.

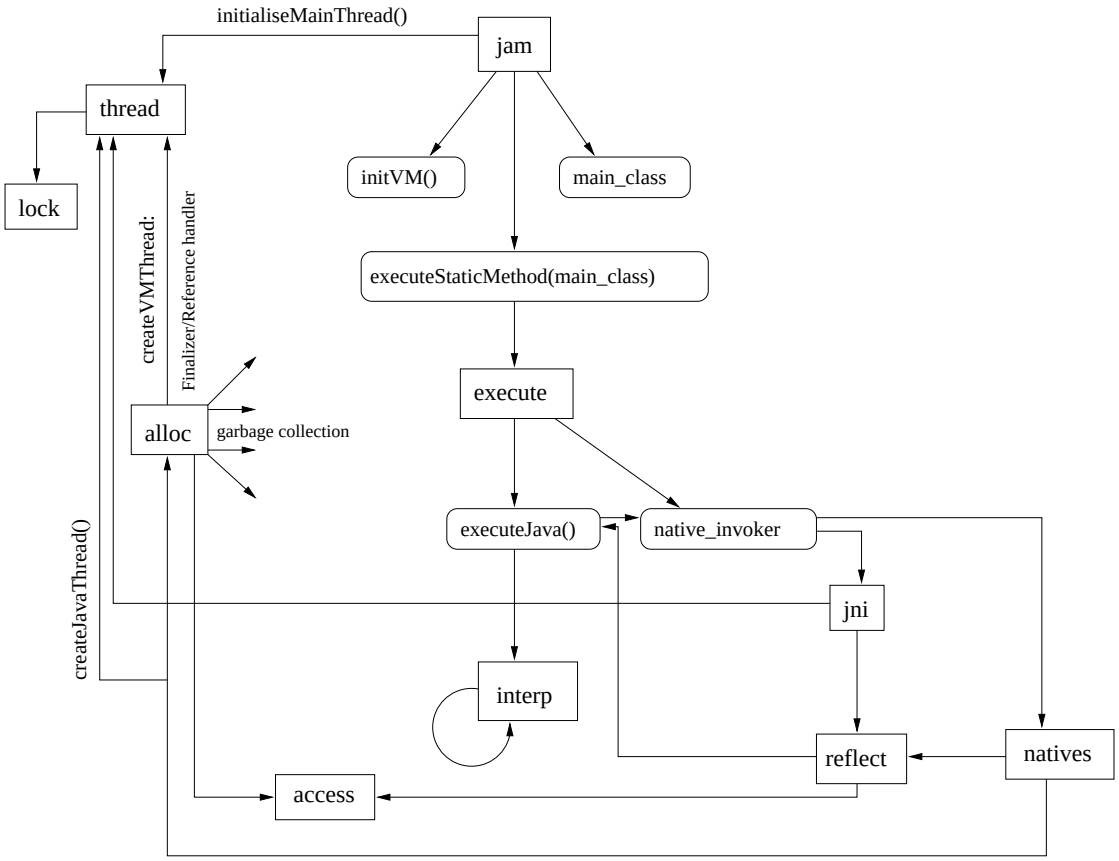


Figure 3.1: Architecture components of JamVM

Moreover, it contains three opcode handler tables, that mark used and unused opcode numbers. Two of the tables are necessary for *stack caching* described in section 3.1.1. In the `interp` module the Java bytecodes and magic methods are intercepted, converted to the JVM's internal representation and interpreted.

JamVM utilizes threads to implement concurrency and uses the monitor concept to protect critical areas. The virtual machine works usually in a direct threaded manner, but can also be used as an indirect threaded interpreter by request. While an indirect threaded interpreter takes the unmodified bytecode stream and the appropriate handler location has to be determined by means of the bytecode itself, a direct threaded interpreter saves this overhead by putting the handler reference into the rewritten opcode series.

The `access` module comprises functions to check access rights among components, that is whether classes, fields of classes and methods are `public`, `protected` or `private`, which is used by the `reflect` part of the VM, that implements the Java Reflection mechanism [Sun06]. It is concerned with how to create an instance of an object, whose type is not known until runtime, for example, and manages reflection access from the JNI.

Another important part is represented by the `class` implementation, that offers functionality to read classfiles containing bytecode streams and further processes this extracted information to define a class. Loaded classes and internally built arrays are managed within a hash table. Besides this, all primitive data types such as integer are arranged in an extensive array.

In JamVM a native interface `natives` for VM internal functionality is provided in order to circumvent expensive JNI method calls. This includes, besides another tasks such as setting or retrieving class fields, an interface to services of the `alloc` module, that is responsible for memory allocation and garbage collection. In case of native methods the `native_invoker` can be set to the accordant method of the `natives` module or the `jni` can be used. `Natives` is also responsible for creating Java threads.

In the `alloc` component, unallocated memory is managed in a free memory list of so-called *chunks*, each of which consists of a header holding information about the size of the chunk and a pointer to the next chunk. Objects with a finalizer are kept within a separate `has_finaliser` list. As already mentioned JamVM uses the mark-and-sweep garbage collector algorithm. `Alloc` comprises a `doMark()` and `doSweep()` function. In the mark phase a living object reachability analysis is performed, i.e. the entire heap is scanned in order to determine objects that are still needed. All unmarked objects are garbage, however, objects with a finalizer are moved to a `run_finaliser` list, that wait for their handling. Finalizer and reference handlers run as dedicated threads. In the sweep phase the space of the unmarked present objects is freed and merged with the chunks of the free memory list. The size of the free heap is recomputed again. The size of the largest block is memorized to quickly determine which is the largest memory demand that can be accomplished.

3.1.1 Stack Caching

JamVM utilizes the stack caching technique, that two cache registers are provided for. When pushing return values onto a stack residing in RAM and retrieving them on demand a lot of accesses to the rather slow main memory are performed. Holding a certain amount of stack elements in registers improves execution time significantly and furthermore reduces adjustments to the stack pointer. In order to implement the stack caching optimization in the interpreter every opcode has three representations and that is also the reason why three opcode handler tables exist. The opcode versions comprise the following cases:

- both operands are kept in registers
- one operand of the opcode resides in memory and the other in a register
- both operands are kept in main memory.

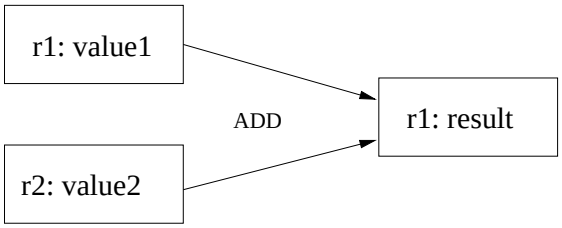
An illustration of these three cases, opcode versions have to deal with, is given in figure 3.2. It shows the addition of two values. In the `interp` module a handlers_0 dispatch table therefore means basically no use of the stack caching technique, while handlers_1 and handlers_2 involve this optimization, with the use of one and two registers, respectively.

Due to the smaller number of registers on Intel x86 architectures stack caching should be deactivated on these machines, because it results in a performance loss. Register contents have to be stored in memory multiple times, since there are only few registers. So writing 512 MB memory with integer values continuously with stack caching enabled on an Intel testing machine resulted in a 21% slow-down in comparison to a non-stack cached interpreter. However, on PowerPCs this optimization can result in a higher performance.

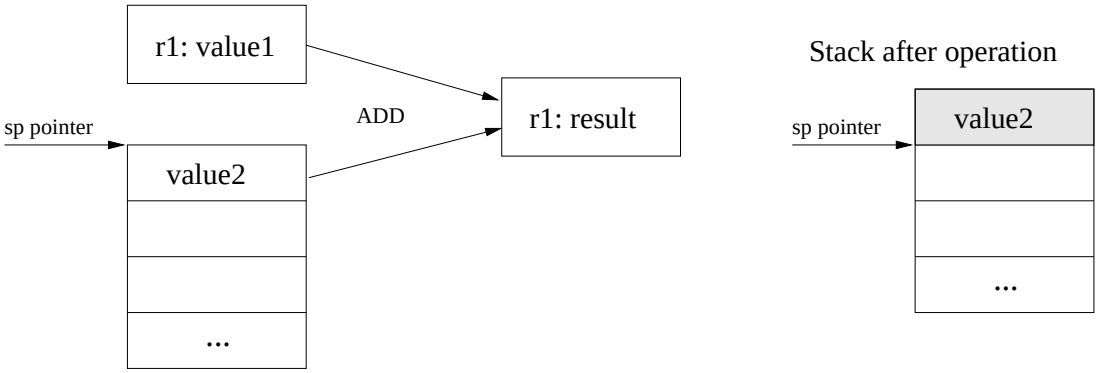
3.2 Implementation Details

The Java Virtual Machine Specification [LY99] currently defines 204 bytecode instructions that every JVM has to implement. The remaining 52 bytecodes are reserved for future extensions and can be used to define JVM internal operations. JamVM defines 26 so-called *quickened* bytecodes, which is explained further below in this section. Of the remaining 26 bytecodes 14 are used for the magic methods of the memory module interface. Their task is to read and write the primitive data types byte, short, int and long in little and big endian sequence.

a) Opcode operands are kept in registers exclusively



b) One operand resides in a register, the other in a memory location



c) Opcode operands are kept on a stack in main memory

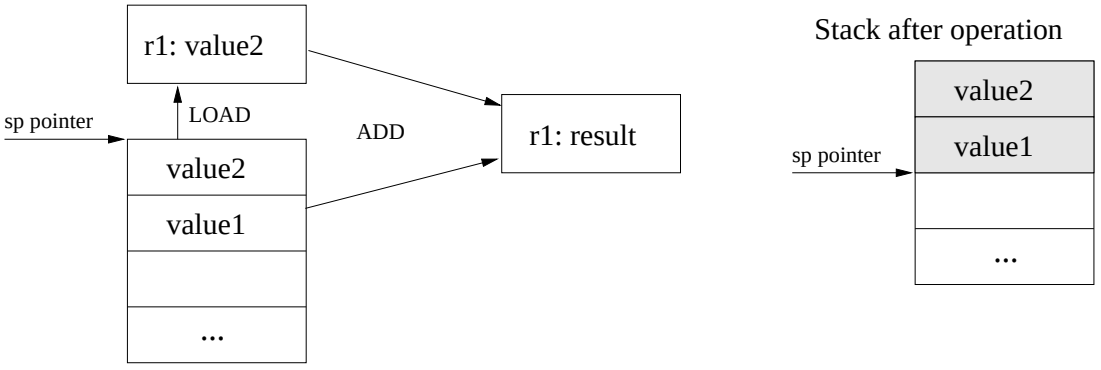


Figure 3.2: Opcode versions to implement stack caching

3.2.1 Providing an Untyped Memory Area

In the `natives` module, the interface between the operating system and the interpreter, the function `initDirectMemory()`, that prepares the FS register and allocates the untyped memory block as explained in section 2.3.4, is defined. It can either be invoked as a magic method by the Java application program or by the JVM in the start-up phase. For the last approach `initDirectMemory()` is called in the `initVM()` initializer function shown in figure 3.1, which is invoked by the `jam` module in the beginning of the JVM execution. The untyped memory region is then available to the virtual machine application.

3.2.2 Magic Bytecodes and Quickening

The magic methods are implemented as *static* calls, since otherwise the object reference would be pushed onto the stack and this is not needed and would just cause an overhead. In the next step, the `invokestatic` bytecodes, that call the magic methods of the DirectMemory module have to be intercepted by the interpreter and replaced by magic bytecodes. The magic bytecodes are added to the opcode handler table described in section 3.1. It determines whether a bytecode number is used or not.

JamVM has implemented the so-called instruction quickening technique, for which the 26 additional bytecodes are necessary. When a static method is invoked for the first time, the class it belongs to has to be loaded and initialized, if it has not been done so far. The verification whether a class has already been prepared must usually be performed on every static method invocation, which results in a significant performance loss. Bytecodes handled in that way are referred to as *slow bytecodes* in the following explanations. To avoid the checking overhead, JamVM utilizes *quick bytecodes*. One such quick bytecode is introduced for `invokestatic`. When JamVM interprets a regular `invokestatic` bytecode, it is replaced by the quick variant in the bytecode stream and on subsequent invocations, the check if the class was already loaded is omitted. The operand of the quick bytecode is determined by memorizing the result of the slow bytecode operand resolution.

This mechanism is modified to replace invocations of the DirectMemory methods with the appropriate introduced magic bytecode. For each of the magic bytecodes, the functionality of the corresponding DirectMemory method is implemented in assembler. The following example shows the simplified implementation of the method, that reads an integer.

```
asm volatile("movl %%fs:(%1), %0\n : "=r"(val) : "r"(addr));
```

In GNU C Assembler [gcc03], which is used here, *"=r"* is the output operand, *val* in this case, and *addr* is the input operand, popped from the VM's operand stack. Afterwards the *val* is pushed onto this stack, to provide the correct return value. Both

Test program	SafeMemory	DirectMemory
wsort-arm (EBBB)	1m19.865s	0m57.765s
wsort-arm (EBSB)	1m20.477s	0m59.587s
Write_Byte-arm	–	–
Write_Short-arm	–	–
Write_Integer-arm	91m+	64m17.914s
Write_Long-arm	39m18.866s	12m37.556s
Read_Byte-arm	67m43.536s	15m15.292s
Read_Short-arm	34m50.007s	10m34.742s
Read_Integer-arm	12m27.484s	3m54.961s
Read_Long-arm	7m49.739s	2m44.337s

Table 3.1: Benchmarks: Execution of ARM binaries within JSim in EBSB mode

variables reside in registers. The memory is addressed relatively to the modified FS selector, thus every address displacement calculation and boundary checking is performed in hardware.

3.3 Conclusion

The provision of new memory constructs in JamVM is not as time-consuming and complex as in Jikes RVM, which is explained in the next chapter, since JamVM only supports an interpreter. Even though the execution of JSim on top of this VM is still very slow, it can be asserted from trivial test programs, that read and write a logical continuous memory area, that the DirectMemory access improves performance.

Concluding this section a comparison of the untyped memory implementation and usual field access is given in table 3.1 and in figure 3.3, the difference of the execution times is depicted. The measurements were performed on an Intel machine with two Xeon 3.06 GHz processors on JamVM version 1.4.1. The first test program *wsort* is a program that sorts words in alphabetical order. The first benchmark was made in JSim's Emulator-By-Basic-Block (EBBB) and the second in Emulator-By-Super-Block (EBSB) mode. The execution modes of JSim are discussed in [Sti05]. The word list *wlist3* contains 45402 unsorted words and serves as input file to wsort. The *Write_* and *Read_* prefixed programs write and read 256 MB of a logical continuous memory space, respectively, and are executed in EBSB mode. Due to the simple structure of these programs, super blocks can quickly be determined and the DirectMemory extension causes a up to 77 % performance improvement in contrast to SafeMemory module. The wsort program is a more real-world application and DirectMemory shortens the execution time by about 20 %. The execution of the read byte, short and int programs were aborted. At this point, it can already be seen, that the interpreter is very slow, but

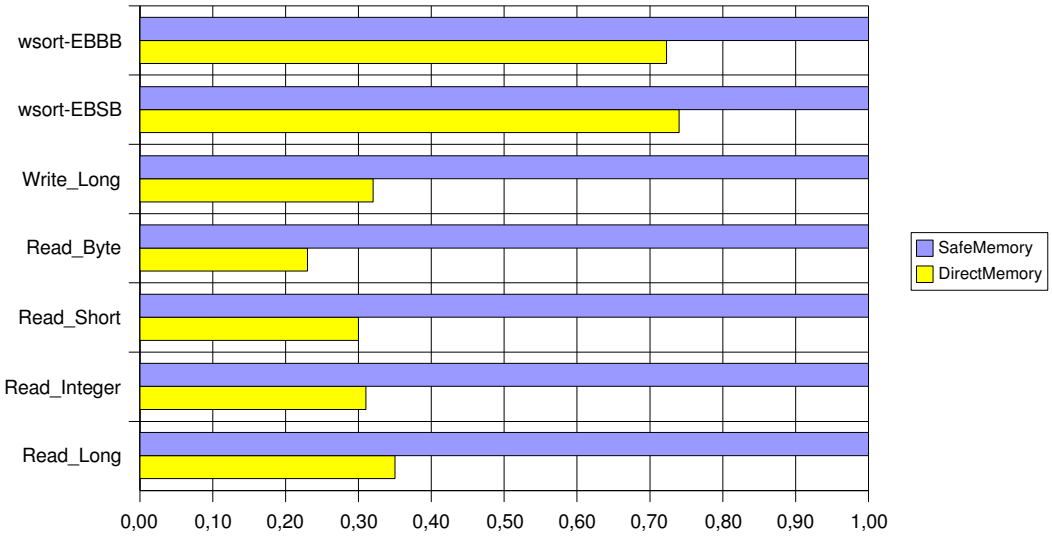


Figure 3.3: Performance improvement of DirectMemory in contrast to SafeMemory

that the untyped memory extension yields to a better performance of JSim. The same programs are utilized on a different testing machine in chapter 5. The performance of Jikes RVM—discussed in chapter 4—in contrast to JamVM is much higher.

Chapter 4

Jikes RVM

The Jikes Research Virtual Machine [AAB+00] (RVM), also named Jalapeño VM, is an open source project by IBM and it is written almost completely in pure Java. Jikes RVM requires another *host* JVM for its bootstrapping phase.

RVM was designed to meet the needs of a server environment, i.e. the execution of software over a long period of time. Jikes RVM currently supports the PowerPC and Intel x86 architecture on Linux and PowerPC on AIX, and an extension to Intel's 64-bit version is planned. Jikes RVM does not offer an advantage over other JVMs, when executing programs over a short time, it is even rather slower due to time-consuming dynamic compilation of code, which is only profitable, if the compiled code is executed frequently. When meeting a frequently traversed code block, a so called *hot spot*, Jikes RVM's internal JIT compiles it to architecture bound machine code. Subsequent executions of the hot spot will be really fast, because it can be executed directly on the machine and does not need to be interpreted anymore.

4.1 Architecture

The RVM consists of a JVM and a JIT compiler subsystem and is written in pure Java, which imposes a series of problems [AAB+00]. Usually the core elements of a JVM, a compiler and a class loader besides other services, are written in native code, whereas in this case a boot image containing these essential modules has to be executed before the RVM can be started. The boot-image writer, which is coded in Java, generates this image wherefore a host JVM is needed.

It instantiates a minimal set of needed objects, but it is necessary to transform them from the object model of the host JVM to the object model of the via the Java reflection mechanism [Sun06]. The transformed objects are packaged into the boot image. A C program, the boot-image runner, loads the boot image into the memory, defines some register states and kicks off the bootstrapping of the JVM. A general survey of the JVM's

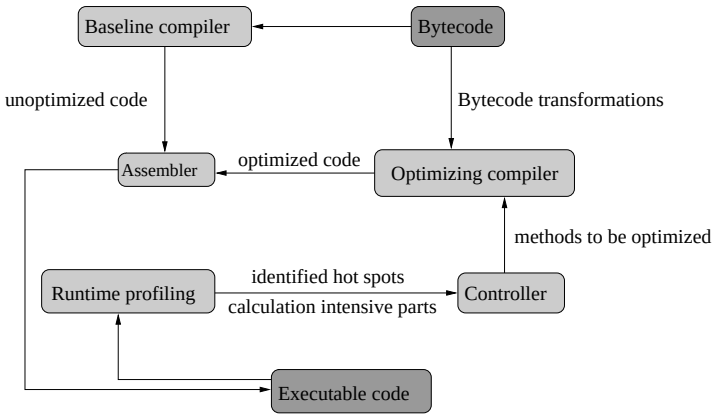


Figure 4.1: Architecture components of the JVM

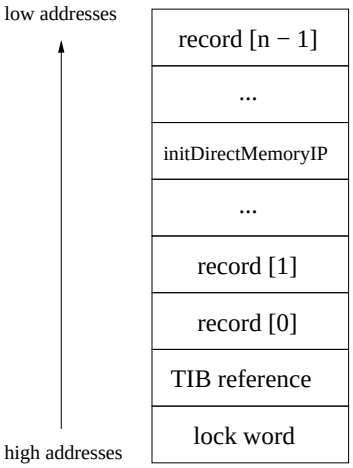


Figure 4.2: Structure of the boot record

architecture components is given in figure 4.1.

While the optimizing compiler usually is part of the RVM's adaptive system, it can also be used to compile code, when it is first executed and to build the boot image, but this is only an additional option. The runtime profiling component measures the performance of the methods, identifies hot spots and calculation intensive bytecode blocks. This information is then passed on to the controller, that designs an optimization plan. Due to this plan the optimizing compiler is informed about, which methods have to be compiled. Profiling is continued for the optimized code and can trigger further optimizations.

4.2 Native Module Interface

In order to invoke operating system services some native C routines are necessary, which reside in a single module, but the virtual machine image is set up on Java objects. Thus there has to be way that these objects are connected to the C code somehow. This is achieved by the boot record, which marks the first created Java object and constitutes the communication interface between the underlying operating system and the JVM, that runs on it. The boot record possesses both methods essential for booting the VM and function pointers to the system calls. As the method `initDirectMemory()`—defined in the native module—which allocates the required memory block and prepares registers for hardware address translation, has to be made available to the JVM, a reference to the native method has to be declared here. It is crucial, that this reference has exactly the same name, as the referenced C function suffixed by `IP`, which forms a field in the boot record. This field can be accessed by the `VM_SysCall` class, which

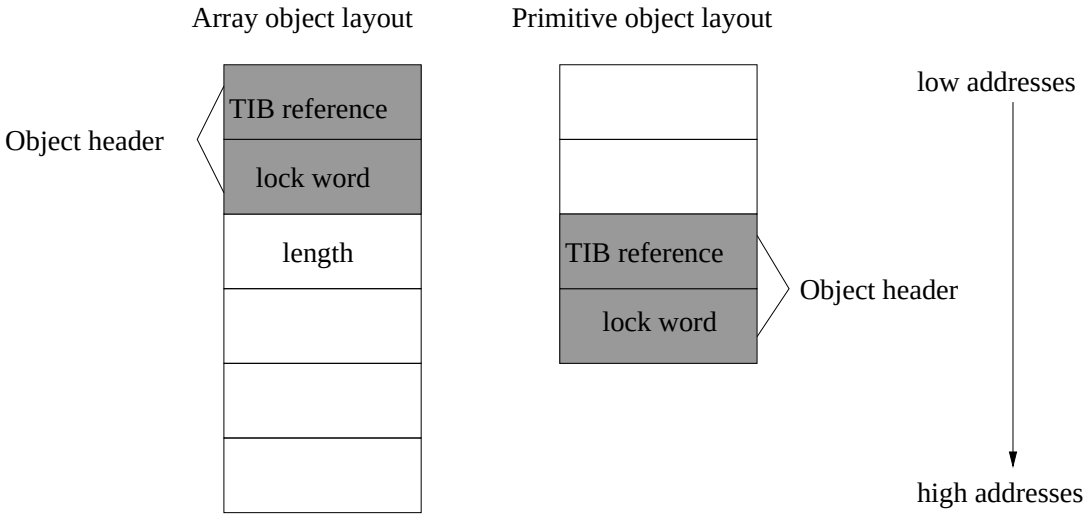


Figure 4.3: Object layout of array and primitive objects

consists of method signatures and empty function bodies similar to magic methods.

A simplified representation of the boot record structure is given in figure 4.2. Additionally to the depicted records it contains the JTOC (Jalapeño Table of Contents) record. The JTOC references a capacious array, which holds static variables and addresses to all static methods of the Jikes RVM.

4.2.1 Object Layout

As the goal was to implement array accesses and execute virtual method calls in an efficient way, two design decisions on the Jalapeno object layout had to be made:

- Arrays expand from low to high memory, while primitive objects grow down from the object address. Therefore array field access can be implemented by taking the base address information and then adding a certain offset.
- Introduction of a further data structure: TIB (Type Information Block)

The object layout of both array and primitive objects is shown in figure 4.3.

Arrays need in comparison to other objects an auxiliary length information, which is left empty with other data types. For efficiency reasons references in this virtual machine are programmed as machine addresses, and the value 0 is assigned to a `null` reference. But the Java specification requires the generation of a `NullPointerException` by accessing a `null` object reference. On Linux OS, a hardware trap is triggered upon dereferencing of a `null` address. This allows `null` reference checks to be handled by the operating system. This is similar to the idea of allowing the hardware

to perform `null` and array boundary checks, to provide an accelerated DirectMemory solution.

4.2.2 Object Header Design

The header consists of a lock word followed by the already mentioned TIB reference. The lock word contains besides some bits used for synchronization, information for the garbage collector and a hash code for the object.

The TIB reference points to the TIB, which contains data about the appropriate class and compiled code for its virtual methods. At this point it becomes clearer, why the magic methods are implemented as static functions. An `invokevirtual` bytecode would require looking up the TIB field slot in the object header, finding the suitable method body in the TIB array, loading this address in a dedicated register and finally executing the code. The static method dispatch within the RVM is accomplished in a less expensive way: Static data and references to static methods reside in the JTOC and the address of the JTOC array is once loaded into a register at JVM start-up.

4.3 Magic Methods

The Jikes RVM explicitly offers the `VM_Magic` class to escape the limitations Java imposes in order to perform unsafe casts and raw memory accesses for garbage collection and here the methods for direct memory manipulation can be defined. Furthermore declarations for them have to be made in the `VM_MagicNames` class, in which so-called `VM_Atoms` of the magic methods are created, that are byte strings and represent the names of the untyped memory functions in the constant pool of a class.

4.4 Assembler

The assembler of the RVM translates assembler code, generated by one of the compilers, to machine specific code. The low-level class for the IA32 (32-bit Intel Architecture) ISA (Instruction Set Architecture) is composed of a part, that deals with particular cases of Intel opcodes, and machine generated functions that emit similar structured machine codes like `mov`. For each possible addressing mode a method is assigned to a machine instruction. The method name is composed of three parts: The *emit* keyword is a prefix to every embedded method in this module, the name of the desired opcode is directly appended and the last part describes the addressing mode. As an example the newly defined mnemonic `movfs`, which uses the FS instead of the DS data segment, is depicted in figure 4.4. The instruction takes two byte encoded registers as parameters. The `setMachineCodes()` function first sets a segment prefix override, 0x64 for the

```
public final void emitMOVFS_Reg_RegInd(byte dstReg,byte srcReg)
{
  int miStart=mi;
  setMachineCodes(mi++,(byte)0x64);
  setMachineCodes(mi++,(byte)0x8B);
  emitRegIndirectRegOperands(srcReg, dstReg);
}
```

Figure 4.4: An Intel `mov` instruction via the FS segment: `movfs`

```
if (methodName == VM_MagicNames.mr4) {
  asm.emitPOP_Reg(T0); // address
  asm.emitMOVFS_Reg_RegInd(T0, T0);
  asm.emitPUSH_Reg(T0);
  return true;
}
```

Figure 4.5: Compilation of the magic method mr4

FS segment. 0x8B is the encoded hexadecimal opcode of the `mov` instruction, that expects a word (32-bit) or a double-world (64-bit). In order to write bytes another `mov` instruction has to be utilized. The `emitRegIndirectRegOperands()` function sets the appropriate SIB and Mod/RM bytes as described in section 2.3.3. Both is written into the machine code buffer. The same assembler class is also used by the optimizing compiler, but another optimizing instance is set up on top of it, which is discussed later in this chapter.

4.5 The Baseline Compiler

The baseline compiler is divided into two parts: An architecture independent compiler frontend and machine specific backend, that generates assembler code. The functionality of the baseline compiler is straightforward. It compiles very fast, because does not use complex optimizations, but the performance of the generated code is poor. However, it is a core element of the bootstrapping process and it is a crucial part of the compiler subsystem, because generating optimized code is expensive and only profitable at hot spots.

The `VM_BaselineCompiler` class is the target architecture independent part and the core of the baseline compiler for the Intel x86 target machine is the `VM_Compiler` class. In the `VM_Compiler` Java bytecodes and magic methods are compiled to an assembler-like code, as shown in figure 4.5. When the baseline compiler is first instan-

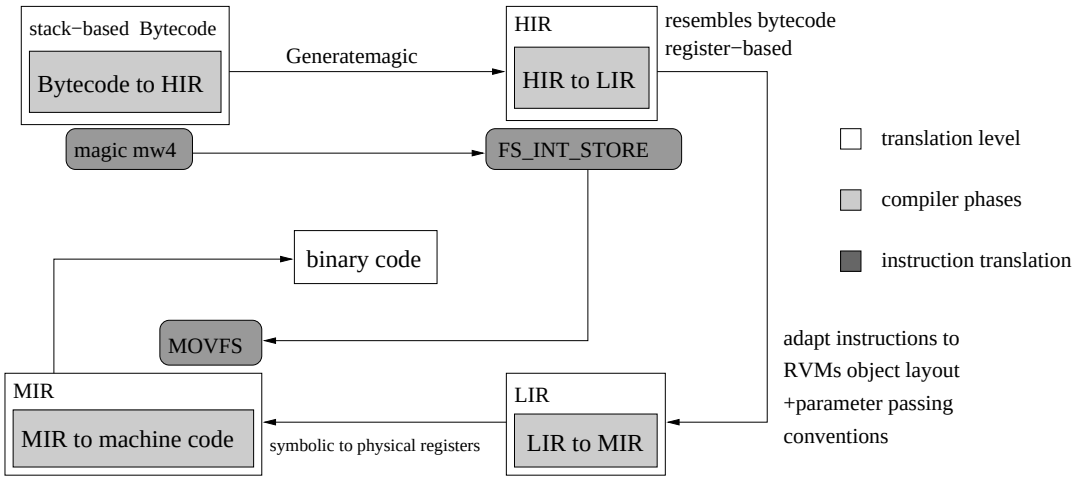


Figure 4.6: Generation of layer-specific instructions during compilation of a magic method

tiated, a new `VM_Assembler` object `asm` is created, on which certain *emit* methods can be invoked. The detected method name determines, which code has to be generated and first the desired parameter, in this case the address, is retrieved from the operand stack and saved into T0, which is the EAX register on an Intel machine. Afterwards a register-indirect register `movfs` instruction is performed on register T0, which is part of the assembler. In the first parameter of this function the value fetched from memory is saved, and the second one contains the address. The method sets a segment prefix override for the FS segment as described above and writes the `mov` Intel opcode into the machine code array. After this instruction has been executed by the CPU, the result is put into the T0 Register and pushed onto the operand stack to provide the return value.

4.6 Compiler Optimizations

4.6.1 Machine-independent Layer

While the baseline compiler implementation allows to use the DirectMemory manipulating functions, it is important to let the Jikes RVM internal JIT optimize the newly added machine constructs. For this a closer look at the RVM's compilation and optimization process has to be taken, which can be divided into several parts and translation levels to generate highly efficient code at given time limitations [AAB+00].

```
FS_INT_LOAD          {identifier}
Load                 {instruction format}
load                 {traits}
<empty>              {defined registers}
<empty>              {used registers}
```

Figure 4.7: Machine independent IR instruction for loading an integer relative to the FS segment

High-level Intermediate Representation (HIR)

First of all Java, bytecode is translated to the high-level intermediate representation (HIR). In this phase the stack-based bytecode is transformed to register-based three address representation.

In the compilation phase the code is still on a high abstraction level, i.e. is very similar to bytecode. The three address representation facilitates the code motion process. In order to make these optimizations available for the magic untyped memory constructs, new intermediate representation operators have to be implemented. One aspect to facilitate the work of the optimizing compiler is the notion of *instruction formats*. When the new instructions for the machine independent layer are defined, the JIT is informed about the kind of instruction. The compiler has predefined methods of how to optimize some types of instructions. For example, there are several bytecode store instructions, that can be optimized in the same way.

An instruction format in the *InstructionFormatList* describes the needed operand types in detail. However, the existing instruction formats meet the needs of all magic bytecodes, since *Load* and *Store* schemes have already been implemented. The structure of the `fs_int_load` operator, is shown in figure 4.7. It resides in the machine independent *OperatorList*, which defines besides bytecode similar HIR also LIR (low-level intermediate representation) operators, which are used for Jikes RVM specific issues. The added operator is identified by an unique denotation and fits into the instruction format *Load*. Furthermore it can have certain traits, telling whether it is a load or a conditional instruction, for example, to further specify the operator. Implicitly used and defined registers could also be set here, but due to the fact that the auxiliary operator is machine independent, these lines are blank.

To transform Java bytecode to the HIR form, magic methods have to be intercepted by the `OPT_GenerateMagic` class. Correct parameters are passed through the operand stack and symbolic registers are assigned for return values, while the previously added proper IR operator is appended to the current basic block for a specified called magic method. The construction of basic blocks is crucial to make further optimizations possible.

First of all a look at optimizations, that do not exceed the basic block, is taken.

Among them is common sub-expression elimination, that is, if the calculation of a result for different variables is the same and the components of this computation expression cannot have changed their values in the meantime, the result can be evaluated once, memorized and can then replace the calculation term.

A further improvement can be achieved by identifying checks that have to be performed and the possibility that an exception can occur. If one entry in an array has to be manipulated, one verification, that the to be accessed index is within bounds, is necessary. It is apparent that for the actual change of the value, the array check can be omitted. In this example there are two array accesses, but only one exception can be thrown, because it is about one operation. Such cases can be detected in the HIR and unnecessary checks can be deleted, thus saving a lot of time. In addition the JVM possesses the `OPT_Simplifier` to reduce the complexity of instructions, if possible. This module does only take the instruction itself into consideration and does not analyse the relation to the overall program process. For example, an `int_bswap` instruction to change the byte order can be simplified: if the value, that has to be swapped, does not change over a period of time, constant folding can be applied, i.e. the evaluation of constant terms. The `int_bswap` is moved to an `int_move` instruction, which is of the *Move* instruction format and transforms the operand into an register operand. In the HIR a control-flow graph, in which the single basic blocks are ordered, is constructed.

Up to now only local optimizations were taken into account. In order to further improve the code formation, it has to be tweaked across basic blocks. Assignments that define unused variables can be omitted. This procedure is called passive code elimination and is closely related to copy propagation: If a value is assigned to a variable and the variable is in turn assigned to anything else, the value can directly be used. The JIT utilizes the SSA (Static Single Assignment) form [FKS00], that improves both local and global optimizations and beyond that simplifies register allocation. Additionally aggressive inlining of methods is performed to reduce the call overhead.

Low-level Intermediate Representation (LIR)

At the LIR level the JVM independent object layout is adapted to the RVM's object model. A bytecode is split up into several LIR instructions in order to operate with the TIB pointer, for example. Since the generated LIR is much more extensive, only few optimizations are done here. A dependence graph, that is used for a Bottom-Up-Rewrite-System (BURS)—described in the next section—is constructed for every basic block.

4.6.2 Machine-dependent layer

Machine-specific Intermediate Representation (MIR)

The machine-dependent layer is represented by the MIR, that decides which machine specific instructions have to be generated and on which further code simplifications can be performed. The previously created dependence graph for each basic block is split up into trees, that serve as input to a BURS.

BURS

BURS is used by the optimizing compiler for instruction selection by taking a complex expression tree from the LIR and translating it to MIR code, while applying dynamic programming [AAB+00] and it is a pattern matcher based on trees and is modelled on *Iburg* [FHP92].

A BURS code-generator generator processes a series of rules and each of them comprises four items:

- a *production*, which indicates the tree pattern to be replaced by a specified symbol
- a cost function, that predicts whether the application of the rule is profitable at all
- a set of flags, which specify the actions to be taken
- a template for Java code emission

The symbol substituting the dependence tree is a non-terminal located on the left side of the rule, and the pattern to be detected is on the right. Both are separated by a colon and the collection of all rules forms the grammar. Besides non-terminals a grammar can have operators with a varying number of operands.

```
r: FS_INT_STORE(riv, OTHER_OPERAND(riv, INT_CONSTANT))
15
EMIT_INSTRUCTION
EMIT(MIR_Move.mutate(P(p), IA32_MOVFS, MO_S(P(p), DW),
     Store.getValue(P(p))));
```

Figure 4.8 illustrates the dependence tree, which is represented by the above production specification, where *riv* can be resolved either to a register, a subtree that can be mapped to a register or an immediate operand. OTHER_OPERAND is an operator and is merely applied to construct a binary tree data structure. In order to evaluate the cost function or to facilitate code generation a set of methods is provided in Jikes RVM and a symbol p is assigned to refer to the current root tree node, whereas P(p) in the code example gets the instruction associated with that node. If required, the left and the right child, in

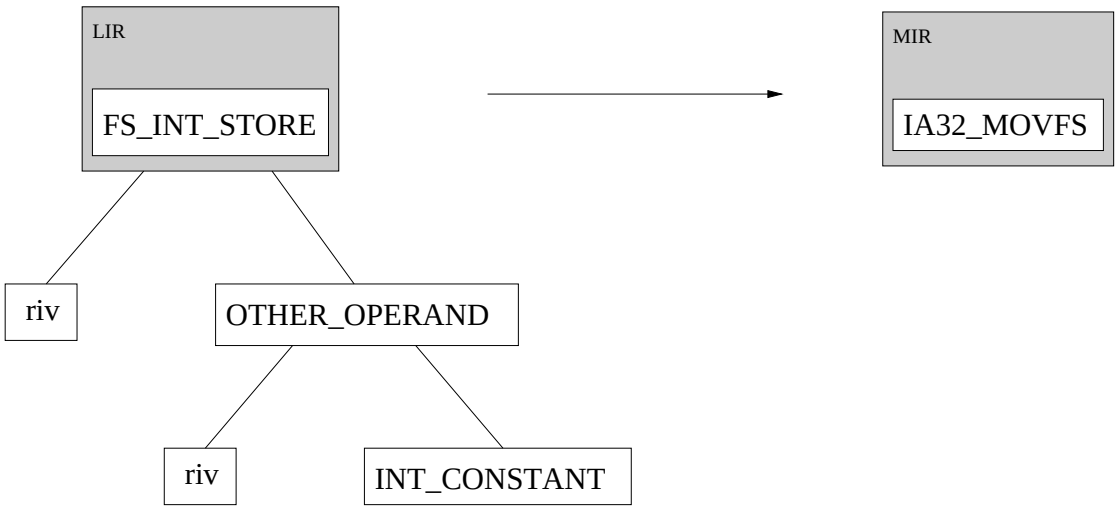


Figure 4.8: Application of a BURS rule and resulting MIR form

case they exist, can be accessed via PL(p) and PR(p), and their successors by means of concatenating L's or R's with respect to the root.

Whenever a tree defined in the production is matched and the code generation costs do not exceed 15, the code defined by EMIT is emitted and substitutes the fs_int_store for a IA32_MOVFS target architecture dependent instruction with the appropriate operands.

The rules file is used to create a state machine, that can apply dynamic programming at compile time, which then converts the IR to the next level.

The BURS theory has established for a longer time and a lot of implementations have been made, while the new approach of Hanson [FHP92] offers a remarkable performance enhancement. In comparison to approved BURS programs, which emit ideal code in a constant period of time per tree node, it is additionally possible to have non-constant costs, as implemented in Jikes RVM. This is achieved by postponing the dynamic programming process until compilation time.

The output of BURS then forms the final MIR, which is the last level before emitting the actual code.

4.6.3 Machine Code Generation

Jikes RVM has implemented the *linear scan register allocation algorithm* as described in [PS99] to map the arbitrarily sized number of virtual registers to physically existing ones. The RVM offers an option to print the final machine code, so it is possible to analyze the effect of the untyped memory implementation on the extent of generated code. For this PowerPC application binary *wsort*, that sorts words alphabetically, was executed within JSim. As a result the number of instructions was halved.

4.6.4 Survey of the Overall Optimized Compilation Process

Concluding the section about optimizations, an overview of how magic methods are treated by the JIT compiler is depicted in figure 4.6 and presented as follows.

The stack-based bytecode is transformed into HIR code, that operates on registers to facilitate code transformations, that operate on three-address code. The magic bytecode `mw4` is intercepted by the modified compiler, thus generating a `fs_int_store` instruction, that is used both at the HIR and LIR. Appended to a basic block, several optimizations can be applied on and across the block and in the LIR the instruction is adapted to the Jikes RVM's object design. Furthermore a dependence graph, used in the MIR, is constructed for each basic block. In the transformation from LIR to MIR, symbolic registers are mapped to physical ones. Finally in the MIR the dependence graphs are used for BURS to select the machine instruction to be emitted, as illustrated in figure 4.8.

4.7 PowerPC Baseline and Quick Compiler Implementation

4.7.1 The PowerPC Architecture

The G4 is a 32-bit general purpose processor and belongs to the RISC family [IBM00]. It has a simpler instruction set than Intel x86 processors. The operational part of the opcodes has the same length for all opcodes. Unlike the Intel x86 architecture PowerPC features 32 GPRs (General Purpose Registers).

4.7.2 Implementation Details

The PowerPC architecture on the AIX operating system was the first supported architecture in IBM's research JVM. For PowerPC, Jikes RVM additionally features a third compiler, the so-called *quick* or *quickline* compiler and will probably replace the baseline compiler in the future, since baseline implementation on contemporary PowerPCs is extremely slow. The quick compiler tries to keep stack variables in registers and not at memory locations, thus causing a remarkable performance speed-up. While it does several optimizations, e.g. memorizes, which variables have already been loaded into registers, it rather resembles the baseline than the optimizing compiler structure, i.e. it has no notion of any intermediate representations [use05].

As the PowerPC architecture has a completely different machine design and does not possess an unused register, another approach has to be made. The easiest way in the baseline compiler implementation is to select a register, which is acquainted with the `VM_Compiler` class, and to manipulate it in that way, that it holds the base

address to the virtual memory block, that has to be performed for each interception of a magic method. For this the base address, returned in the native code module has to be provided to the compiler class, that does not have a notion of the underlying layer. For this, similar to make the memory initializer function `initDirectMemory()` available in the JVM, two methods returning both the two high and to low bytes of the base address, have to be embedded in the `VM_BootRecord` and `VM_SysCall` class. Since the PowerPC instruction set only allows to load 16-bit words, but the virtual base address usually is a 32-bit value, the two high bytes and consecutively the last bytes are loaded in the accordant position into the register. For every magic method, the baseline compiler encounters, the base address has to be loaded again, which requires two machine instructions. Another problem is, that the PowerPC architecture does not provide a swap instruction, so it had to be implemented in software. Unfortunately this results in a lot more machine instructions, that in turn need more cycles. Altogether the implementation on PowerPC was not as efficient as on an Intel x86, but nevertheless faster than the SafeMemory module. The implementation in the optimizing compiler resembles Intel x86 implementation and will therefore be not discussed here.

Chapter 5

Benchmarks

In this final chapter the DirectMemory implementation and execution on a guest architecture within JSim is evaluated against the SafeMemory module and native execution of ARM and PowerPC binaries.

5.1 Testing Environment

An Intel Pentium M with 1.6 GHz serves as guest architecture for JSim for both the use of the SafeMemory module and the implementation via magic methods. For measurements of the native execution time a PowerPC G4 with 400 MHz and an Intel StrongARM-110 are conferred with the emulated runtime environment. Jsim is executed on top of Jikes RVM version 2.4.2 using the optimizing compiler.

5.2 Microbenchmarks

Table 5.1 shows the execution times for the benchmarks programs used in section 3.3 to illustrate the performance improvement of the DirectMemory extension of the interpreter JamVM: *wsort* sorts a list of words and the *Write_* and *Read_* programs write and read 256 MB from memory with the appropriate data types. A comparison of DirectMemory and SafeMemory is given in figure 5.1. All test programs utilize the EBSB mode of JSim, which was not possible with the SPEC benchmarks described in section 5.3.

5.3 SPEC Benchmarks

The Standard Performance Evaluation Corporation (SPEC) group offers a series of benchmark programs [spe03], that can be run to give an impression of the approximate

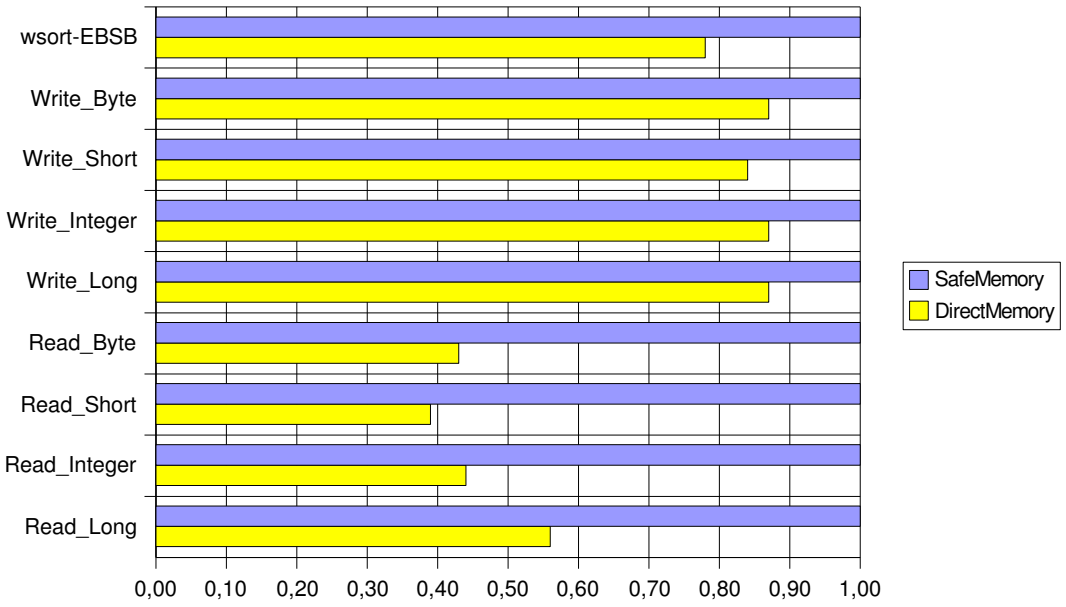


Figure 5.1: Microbenchmarks

Test program	SafeMemory	DirectMemory
wsort-arm	1m16.469s	0m59.788s
Write_Byte-arm	7m51.353s	6m47.921s
Write_Short-arm	4m30.953s	3m48.906s
Write_Integer-arm	2m15.448s	1m57.227s
Write_Long-arm	1m15.901s	1m5.824s
Read_Byte-arm	1m28.642s	0m37.802s
Read_Short-arm	1m13.257s	0m28.930s
Read_Integer-arm	0m27.954s	0m12.365s
Read_Long-arm	0m18.089s	0m10.277s

Table 5.1: Microbenchmarks: Execution of ARM binaries within JSim in EBSB mode

performance of computers. Each of the benchmarks charges hardware components in a different way.

Unfortunately only a subset of the these programs can be run on JSim, which is in some extent due to the incomplete kernel emulation described in [Sti05] and is secondly caused by the still partial implementation of Jikes RVM.

The CPU2000 benchmark suite is the successor version of SPEC CPU95 and now provides more CPU intensive calculations. Besides this, some memory intensive compression programs are also available, on which the difference between the SafeMemory and DirectMemory module is most notable.

In the following a short overview of the tested programs will be given.

5.3.1 Description of the SPEC CINT2000 Programs

gzip

gzip is a compression program and holds the balance between execution time and dimension of the resulting packed file. It implements the Deflate-Algorithm, that is based on LZ77 (Lempel-Ziv) and Huffman encoding.

bzip2

While the time overhead of *bzip2* is significantly higher compared with *gzip*, the resulting compression is much more efficient and works with the BWT (Burrows-Wheeler-Transformation) algorithm. Since all compression and decompression of the input files for both *gzip* and *bzip* are performed in memory, the execution on JSim with direct memory access provides particularly good performance.

mcf

mcf uses a lot of integer and pointer arithmetics and there are also frequent accesses to the RAM. It implements the network simplex algorithm and solves the problem of having as minimal traffic in a network as possible, while serving each node nevertheless.

5.4 Results

As shown in table 5.2 and figure 5.2 the discrepancy between the performance of memory and CPU intensive programs is notable. Since only the memory performance is improved by the DirectMemory module in the JIT compiler and the compression programs perform almost their complete work in memory, the performance advantage is greater than with the CPU intensive *mcf* benchmark.

Figure 5.2: Native vs JSim with DirectMemory access vs. JSim with SafeMemory

The execution time was measured from the beginning of the Jikes RVM bootstrapping phase, but this is negligible, because the benchmarks run over a relatively long time. The mode of JSim was emulation by basic block. Available code generation strategies are explained in [Sti05]. The super block mode in combination with magic methods was not executable. This could be caused by the incomplete implementation of the RVM, since the DirectMemory access with the JSim super block mode was possible on the Java bytecode interpreter JamVM.

Test program	bzip2	gzip	mcf
input file	input.program	input.program	inp.in
JSim-SafeMemory	115m6.560s (ARM)	77m19.646s (PPC)	55m57.410s (PPC)
JSim-DirectMemory	58m48.293s (ARM)	47m12.165s (PPC)	50m21.781s (PPC)
ARM	45m15.780s	64m51.620s	70m58.070s
PowerPC	7m6.978s	14m45.805s	20m36.291s

Table 5.2: Benchmarks

5.5 Does Untyped Memory violate Type-Safety?

The SafeMemory module of JSim shows that untyped memory can be implemented using pure Java methods. Since the methods of the memory module interface do not allow to load and store references, the presented DirectMemory implementation does not violate the type-safety of Java.

5.6 Conclusion and Future Prospects

Although performance of the obsolete PowerPC G4 and Intel StrongARM-110 cannot be compared to newer technology processors, the time measurements of the binaries show that an execution of legacy applications within a type-safe emulation environment is feasible in a reasonable period of time.

It is not claimed that the runtime of an application within JSim is as fast as native performance on the same machine, but legacy software is going to be emulated as efficient on modern hardware as it used to execute on obsolete hardware. The computer technology will develop consistently, so that the overhead, that the simulation of an entire architecture causes, will hardly be noticeable and established applications will not have to be reinvented anymore.

The next generation of computers have a 64-bit wide address bus. Thus the 32-bit virtual address space of the guest architectures guarantees not to violate any limitations.

Bibliography

[AAB+00] Bowen Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM System Journal*, (VOL 30, NO 1), 2000.

[FHP92] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a Simple, Efficient Code Generator Generator. *ACM Letters on Programming Languages and Systems 1*, September 1992.

[FKS00] S. Fink, K. Knobe, and V. Sarkar. Unified analysis of array and object references in strongly typed languages. In *Proceedings of the 7th International Symposium on Static Analysis*, 2000.

[Fog06] Agner Fog. The microarchitecture of Intel and AMD CPUs. Technical report, 2006.

[gcc03] GCC Inline Assembly, 2003. Web: `http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html`.

[GPF05] Andreas Gal, Christian W. Probst, and Michael Franz. Untyped memory in the Java virtual machine. In *2nd ECOOP Workshop on Programming Languages and Operating Systems*, 2005.

[IBM00] IBM. *PowerPC Microprocessor Family: The Programming Environment for 32-Bit Microprocessors*, 2000.

[Int97] Intel Corporation. *Intel Software Developer's Manual, Volume 2: Instruction Set Reference*, 1997.

[Int99] Intel Corporation. *Intel Software Developer's Manual, Volume 3: System Programming*, 1999.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., second edition, 1999.

[PS99] M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM TOPLAS*, 1999.

[RH01] Randall and Hyde. The art of assembly language. Technical report, 2001.

[Rob06] Robert Lougher. Jam Virtual Machine interpreter, 2006. Web: `http://jamvm.sourceforge.net`.

[spe03] Standard Performance Evaluation Corporation, 2003. Web: `http://www.spec.org/cpu2000/CINT2000`.

[sse06] Introduction to the Streaming SIMD Extensions in the Pentium 3, 2006. Web: `http://www.x86.org/articles/sse_pt1/simd1.htm`.

[Sti05] Michael Stilkerich. Portable Ausführung von Altanwendungen durch Laufzeitkompilierung zu Java Bytecode. Study Thesis, July 2005.

[Sun03] Sun Microsystems. JNI 5.0 specification, 2003.

[Sun06] Sun Microsystems. The Reflection API, 2006.

[use05] *The Jikes Research Virtual Machine User's Guide*, May 2005. Web: `http://jikesrvm.sourceforge.net`.

[WG05] Niklaus Wirth and Jürg Gutknecht. Project Oberon - The design of an operating system and compiler. Technical report, 2005.

List of Figures