THE JEWEL VIRTUAL MACHINE

Comprehensive Reference

Rodrigo Augusto Barbato Ferreira
rodrigo.ferreira@jewelvm.com

march 2001

Contents

Chapter 1

Overview

Just-In-Time compilation, also known as *JIT*, is a well-known technique used to improve the execution time in the *Java*[1] *Virtual Machine* (JVM). However, the amount of time, and sometimes memory, used by the JIT internals, in many cases, degrades the application execution time. Some techniques have been used to decrease the JIT overhead while keeping its effectiveness[MMBC97, KG97]. Some of these techniques make use of heuristics to detect execution *hot spots* and produce quality code for them[Sun98]. Other techniques try to focus on providing faster algorithms only for gain-proven compiler tasks[YMP+99]. However, the trade-off between the JIT running time and its object code execution time will always exist. There is no silver bullet.

1.1   The Scenario

From our observation, an end-user Java Virtual Machine deals with the same code most of its time. Users always launch the same applications which are typically composed of the same set of classes. That repetitive situation does not change until the user buys an upgrade of the software he has been using. Also, the *Java Platform API*[GYT96a, GYT96b] is a huge slice of the code being executed by every application, and it only changes on a *JVM* release basis. So, why does the JIT need to recompile all those classes, over and over, on every JVM start up? Surely, the dynamics of the JVM linking model — with *class loaders* and a fine-grained linking unit — makes it difficult for the JIT to catch and cache complex contexts.

On the other hand, in big companies, dozens, or even hundreds of employees share the same application or application suite. Usually, they are connected under the same fast and secure *Intranet*. Why not hoist the JIT to a server, building a shared repository of compiled code? Thus not only users will get the common case fast, but they will also make the common case fast for other users.

In this scenario, depicted on Figure 1.1, the company will not have to buy powerful hardware for each employee. Instead, the company makes a rational investment on the JIT server machine. Even if the employee runs a poor machine, he will get optimized native code — at

---

[1]Java is a registered trademark of Sun Microsystems, Inc.

User Machines

Secure Channels

JIT Server

context

binary code
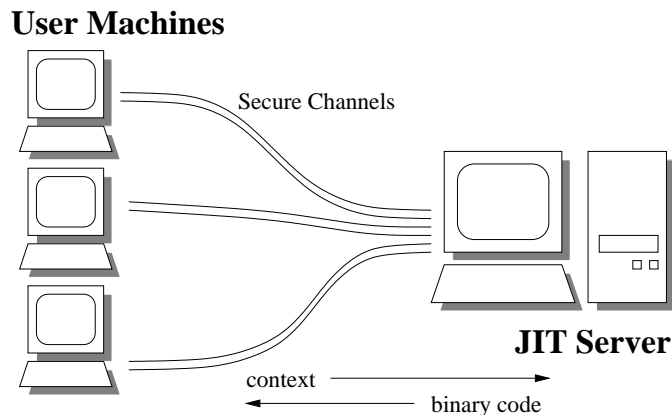
Figure 1.1: A shared JIT server.

the expense of waiting for its compilation on a fast server in the worst case. This is *true*[2] *Java* technology being delivered on standard hardware.

This approach can be extended to the whole *Internet* once JIT servers are trusted and connections made over secure sockets.

1.2   Our Goal

It is our believe that the Java platform shines on its portability. However, its machine-independent *bytecode* is not well suited for execution on most real-world processors. That fact, in conjunction with the security model and philosophy adopted by the Java platform, makes the JVM a heavy piece of software. Our effort is not to criticize the Java platform design, but to ask ourselves how we can achieve a straight-on-business light-weight implementation of it.

Our goal is to detach linking activities from the JVM to a shared server, on a distributed fashion. The client JVM turns to be a very simple piece of software that runs Java code natively, not requiring a JIT or interpreter. All complex linking activities — like link-time error checking, *class file* verification and JIT compilation — are done by the server, which caches its responses. The software is designed in a way that both *standalone* (built-in server) and distributed models could be achieved.

1.3   Road Map

The work herein described addresses all the issues discussed so far. Each chapter provides insight and details of every construction stage of a *Java 2* JVM implementation.

This document is organized as follows: In Chapter 2, the *state-of-the-art* on JVM implementation is presented. In Chapter 3, the design possibilities of a distributed JVM architecture are discussed. In Chapter 4, we give details about the context identification techniques we

---

[2]In the sense of a JVM, not a native compiler.

have adopted, as well as the solutions to problems arisen during their implementation. In Chapter 5, an efficient bytecode verification algorithm is described as a replacement for the standard algorithm. In Chapter 6, we introduce the *intermediate representation* and describe how to convert bytecode to it. In Chapter 7, we describe the platform-independent back-end for the *Intel Architecture 32-bit* family of processors. In Chapter 8, we give implementation details about the *runtime environment*, including data layout on heap and stack. In Chapter 9, we describe the *garbage collection* algorithm we have implemented. In Chapter 10, the mechanics of automatic virtual machine generation is exposed. Finally, we present the conclusions in Chapter 11.

Two appendices are provided as additional information to the reader. In Appendix A, we give a complete specification of the intermediate language. In Appendix B, we describe the tree rewriting tool that we developed targeting the *Java Programming Language*[AG00].

Chapter 2

Related Work

This chapter presents the *state-of-the-art* on *Java Runtime* implementation. First, we describe the most used techniques to implement the *Java Language Runtime*. They range from interpreters to native compilers, passing through mixed execution engines and JIT tricks. Second, we present bleeding edge *Java Runtime* implementations from various vendors. Finally, we discuss the goals of our work and how they fit into the whole picture.

2.1   Approaches to the Java Runtime Environment

There are basically two implementation approaches to the *Java Language Runtime Environment*: *native "static" compilers*[FKR$^+$99, PTB$^+$97] and *virtual machines*[KG97, YMP$^+$99].

A native compiler performs platform-dependent translation from *Java bytecodes* to machine language. Since the translation occurs before execution, native compilers cannot take advantage of run-time information. That is why native compilation is also referenced to as *static compilation*[1]. Native compilation is a Java adaptation of the standard compilation approach present in the C/C++ world. Native compilers differ from virtual machines in the sense that they are unable to efficiently handle bytecode loaded *on-the-fly*. While virtual machines usually have an interpreter or *Just-In-Time* (JIT) compiler to do that job, most native compilers don't. Therefore, native compilers only provide limited support for *class loaders*[2]. Native compilers usually lack some other Java features like reflective programming and built-in object serialization, but that is not a must. Although some virtual machines provide native embedding of core class libraries, the difference from native compilers is the fact that on native compilers user classes are included on preliminary compilation. The intense usage of Java native compilers is due to the belief that they provide faster execution. This is true nowadays, since native compilation time is not a constraint, but will not hold on a near future, once virtual machine technology matures. Vendors claim that native compilers protects intellectual property since reverse engineering compiled code is considerably harder. However, bytecode obfuscators can address this matter while still keeping its portability. Usually, native

---

[1]The technique is also known as *Way Ahead of Time (WAT)* compilation.
[2]Some implementations have no support for dynamic class loading at all. Others let applications load classes known at compile time.

compilation is used for deployment of large scale or mission-critical Java systems.

A virtual machine can be seen as a feature-unconstrained *Java Language Runtime* implementation. Virtual machines provide a bytecode execution-engine, which is usually an interpreter or JIT compiler[Yel96]. Some virtual machines use a mixed execution model, where bytecode known at virtual machine compile time is translated to machine language and coupled with the bytecode execution-engine provided by the implementation. High-end virtual machines use advanced JIT techniques and other methods to achieve performance.

Preliminary implementations of the Java Virtual Machine made use of bytecode interpretation as the only execution mechanism. Literal bytecode interpreters are slow, since bytecode semantics require link-time and run-time checks. The first effort to remove extra bytecode checks during interpretation resulted in a simple rewriting technique[LY96, §9.1]. This technique replaces bytecode opcodes by *quick opcodes*[LY96, §9.2] after first execution. Quick opcodes are free of link-time checks, since they will only execute when linkage actions are guaranteed to take place. Interpretation has been proven to constrain performance horizons, and it is a barrier to the Java technology.

Just-In-Time compilation is a technique incorporated by the Java Virtual Machine to address performance issues. JIT compilers have been used in language runtime implementations of symbolic systems for many years. In the Java Virtual Machine, the JIT is used to translate bytecodes to machine language in a method basis. After the first interpreted execution, subsequent method calls are faster since they executed natively. The main task of a Java JIT compiler is to remove redundant run-time checks[Ste96, MMS98]. Also the JIT must produce as good as possible machine code. However, JIT compilation is constrained on time because it occurs during application execution. So there is a *trade-off* between JIT compilation time and object code execution time.

A technique used to decrease JIT compilation time, while not sacrificing object code execution time, is *Aware JIT*[Aze99] compilation. Aware JIT compilation consists of an off-line bytecode preprocessing, which performs expensive analyses and optimizations. The information gathered is annotated in the bytecode. By doing that, the Aware JIT wastes less time during translation, since most of the information required to generate code is already present in the annotated bytecode. The Aware JIT technique has been proven to be effective when doing register allocation for RISC architectures and removing extra null pointer and array bounds checks. The major problem with the Aware JIT approach is security. The information annotated on the bytecode is usually trusted by the Aware JIT. This represents a real security hole since malicious modifications on the annotated information can produce serious hazard. Verification of annotated information could be a possibility if it were not as expensive as the computation itself.

Another technique present in the current generation of virtual machines[Sun98] is *adaptative optimization*. This technique has considerably improved performance in dynamic typed languages, like *Smalltalk*[GRI83] and *Self*[CUL91]. The idea behind adaptative optimization is based on the fact that a computer program spends most of its execution time in small portions of its code, the so called *hot-spots*. JIT compilers should give special attention to hot-spots and let the other portions of the program be interpreted. An adaptative optimization execution engine must have profiler support embedded in its interpreter, and a hot-spot detection

algorithm. The major problem with the adaptative optimization approach is the performance penalty for short-lived applications. Unfortunately, hot-spots are only noticeable after the application is running for a while, and starts to repeat itself. This technique is appropriate for server applications and extensions.

Another attempt to improve the performance of JIT compilers is code caching. The idea behind code caching is to identify repetitive situations and use code generated by the JIT compiler in previous executions. This saves JIT time on cache hits, and encourages the JIT to apply aggressive optimizations since the output code is likely to be reused. To the best of our knowledge no production JVM supports persistent caching of JIT produced code (by December/2000). Some systems provide one time JVM loading, meaning that the code produced by the JIT is compiled once at least for the API classes. However, this cannot be considered a true caching scheme since the JVM must stay loaded, wasting primary memory, and it cannot save the context to be used in a subsequent loading. There are many pratical problems when trying to identify and cache a repetitive context in the Java Virtual Machine. The main source of problems is the runtime typing model of the JVM.

2.2   High-End Machines and Native Compilers

This section provides an overview about major performance-aware implementations of the Java Runtime Environment: JVMs, native compilers and JITs. Each software description is a summary based on technical information available online and publications.

Sun Microsystems' Java 2 SDK

The *Java 2 Platform SDK, Standard Edition* (J2SE) is a feature-complete development and deployment platform. J2SE is the standard Java implementation from *Sun Microsystems*, which includes a Java Virtual Machine and related libraries. J2SE JVM incorporates Sun's proprietary *adaptive optimization* technology known as *HotSpot*[Sun98]. HotSpot provides an optimizing JIT compiler, profiler based hot-spot detection heuristics, dynamic deoptimization capabilities (to handle earlier optimizations invalidated by code loaded on-the-fly), and an accurate generational garbage collection algorithm[Ung84]. Currently, it is available on Windows, Solaris and Linux platforms.

URL: *http://www.javasoft.com/*

Kaffe

Kaffe is a complete, fully compliant open source Java environment. It comes with its own standard class libraries, native libraries, and a highly configurable virtual machine with a just-in-time and native compiler. Kaffe was designed with portability and scalability in mind. Its threading model allows the choice between an internal, Java-specific user-level threading system, and a native kernel-level threading system for platforms where this is available. Kaffe has its own heap management system with a mark-and-sweep garbage collector. It provides

the ability to replace the garbage collection algorithm with one that may be more appropriate to the application: reference counting, generational or copying. An interesting feature of Kaffe is the execution engine, which comes in three different flavors: interpreter, just-in-time compiler and native compiler. The interpreter is smaller and easier to port, but is significantly slower when executing code. The JIT requires a layer of macros to be written, containing the actual assembler instructions. This allows bytecodes to be translated to native code "on-demand". The native compiler allows java code to be compiled ahead of time directly to native code.

URL: *http://www.kaffe.org/*

LaTTe

LaTTe[YMP+99] is a Java Virtual Machine created by the *MASS (Microprocessor Architecture and System Software) Laboratory of the School of Electrical Engineering at Seoul National University*, as a joint work with the *VLIW Research Group at IBM T.J. Watson Research Center*. It includes a novel JIT compiler targeted to *RISC* machines, specifically the *UltraSPARC*. The JIT compiler generates quality RISC code through a clever mapping of Java stack operands to registers with a negligible overhead. Additionally, the runtime components of LaTTe, including thread synchronization, exception handling, and garbage collection, are optimized. As a result, the performance of LaTTe is competitive with that of other production JVMs. LaTTe was initially developed based on *Kaffe* virtual machine, but most parts of the JVM, including the JIT compiler, the garbage collector, monitor lock handling, and exception handling, have been replaced by clean room implementations.

URL: *http://latte.snu.ac.kr/*

JRockit

JRockit claims to be the fastest and most scalable JVM for server applications. Similar to Sun Microsystems' HotSpot technology, JRockit uses adaptative optimization to improve execution performance. It provides several different garbage collection policies: stop and copy, generational, "train based" incremental. JRockit uses *Thin Threads*, a better implementation of Java threads which allocates no more native threads than available machine processors. Each native thread executes one or more Java threads. Context switch and scheduling are done internally by the runtime. The Thin Threads model takes up less memory and is much faster.

URL: *http://www.jrockit.com/*

TowerJ

TowerJ consists of a native compiler and a runtime that optionally includes a dynamic linker and a bytecode interpreter. TowerJ takes bytecode as input and produces an optimized functionally equivalent self-contained executable program. It delivers a Java application deployment solution that provides native compilation benefits while preserving the flexibility of

Java's dynamic capabilities. TowerJ is based on *Tower*'s proprietary *TRIPLE CROWN* technology, which has been evolving since the early 1990s. TRIPLE CROWN technology was developed while looking for ways to significantly improve the performance of advanced object-oriented programming languages. It was originally implemented for *Eiffel*[Mey91] and due to the similarities of Eiffel and Java, the migration of TRIPLE CROWN was straightforward. The TRIPLE CROWN Runtime/VM is currently supported on Hewlett-Packard HP-UX, Compaq Tru64 Unix and NT/Alpha, Microsoft Windows NT, Sun Solaris, IBM AIX, Silicon Graphics IRIX, and Linux.

URL: *http://www.towerj.com/*

JET

JET[Les99] compiles Java applications into native *Win32* executables. JET is the first *Excelsior* project that employs their *XNJ* technology. XNJ (XDS Native Java) is based on the *XDS multi-target optimizing compiler construction framework*. The XDS framework supports generation of highly optimized native code for several widely used CPU architectures, such as Intel x86, Motorola M680x0, PowerPC, and Sun SPARC (and also generation of C/C++ source code). The core of the framework defines classes for internal representation (IR) of the program. As usual, the XDS framework has three major components: front-end, IR transformer (or middle-end) and back-end. The middle-end and back-end components support various optimization techniques commonly used in "classical" compilers[ASU86, §10]. Among them are: global inline substitution of methods[Muc97, §15], common subexpressions elimination[Coc70], constant propagation, loop unrolling[Mor98, §9.7], redundant run time checks removal[Ste96, MMS98], advanced register allocation algorithms[CAC$^+$81] and instruction scheduling[Mor98, §12]. Some object-oriented optimizations are also available: type inference[CU90] and stack allocation[GS99]. *Static Single Assignment*[CFR$^+$91] (SSA) is used in internal program representation greatly improving the quality of these optimizations. JET has a non-concurrent mark-compact garbage collection algorithm that possesses advantages of both mark-and-sweep and copying garbage collection algorithms. It is accurate and causes less memory fragmentation than traditional mark-and-sweep algorithms.

URL: *http://www.excelsior-usa.com/*

JOVE

JOVE is an optimizing native compiler for large-scale Java applications. JOVE combines sophisticated whole-program and object-oriented optimization technologies, native compilation, and a scalable runtime architecture. The runtime system includes precise multi-threaded multi-generational garbage collection, native threading, low overhead polymorphism, and a great number of minor optimizations. Object oriented optimizations include selective method inclusion, type analysis, polymorphic call-site reduction, and selective generation of reflective metadata. At this time, JOVE only targets the *Intel 32-bit* family of processors running *Windows*.

URL: *http://www.instantiations.com/*

IBM High Performance Compiler

The *IBM High Performance Compiler*[IBM98] (IBM-HPC) is an optimizing native code compiler for Java. Currently, there are beta-level versions for both *AIX* and *Windows*. Bytecodes are processed by a translator to produce an internal compiler intermediate language (IL) representation of each class. The common back-end from *IBM's XL* family of compilers for the *RS/6000* is then used to turn this intermediate representation into an object module (.o file) which is linked with other object modules from the application and libraries to produce an executable program. The libraries implement garbage collection, the Java Platform API, and various system routines to support object creation, threads, exception handling, application startup and termination. The use of a common back-end grants high-quality, robust code optimization capabilities. However, it also dictates the use a conservative garbage collector since the back-end provides no special support for garbage collection. It uses the publicly available Boehm[BW88] conservative garbage collector, which has been ported to many platforms. Code optimizations include instruction scheduling, common subexpression elimination, intra-module inlining, constant propagation, global register allocation. Java specific optimizations, like run time checking removal, are done during IL translation.

URL: *http://www.alphaworks.ibm.com/*

BulletTrain

BulletTrain is a system for statically compiling and linking JVM bytecode applications for *Windows* platforms. It includes an optimizing native compiler, a linker and recompilation manager, an advanced thread-hot runtime, and a core set of Java 2 optimized libraries. The crafting of the runtime system to the needs of the Java language provides high-speed locking, class casts, memory allocation, plus support for thousands of threads and smooth scaling onto multiprocessors. In addition, BulletTrain provides tools for observing the behavior of programs. Heap use can be categorized by object type and thread deadlocks can be automatically detected. JNI (*Java Native Interface*) operations are always checked for correct parameters and stack traces always contain line numbers.

URL: *http://www.naturalbridge.com/*

GNU Compiler for Java

The *GNU Compiler for Java*[Bot97] (GCJ) is a portable, optimizing, native compiler for the Java Programming Language. GCJ is part of the widely known *GNU Compiler Collection* (GCC). It compiles Java, in both source code and bytecode forms, to machine code. GCJ provides a set of auxiliary libraries which consist of the Java Platform API core classes, garbage collector, threading and optional bytecode interpreter. The presence of a bytecode interpreter means that GCJ compiled applications can dynamically load and interpret class files, resulting in a mixed execution model.

URL: *http://sources.redhat.com/java/*

Marmot

Marmot[FKR⁺99] is a performance competitive research native compiler developed at *Microsoft Research*. It was aimed to study the potential performance of large applications written in object-oriented languages. Marmot does static program analysis and transformation, including data flow and type-based local and whole-program analyses. Transformations include elimination of runtime safety checks and synchronization operations, allocating objects on the stack, elimination of unnecessary memory references, and profile-based method specialization and inlining. Garbage collection is supported by three garbage collection schemes: conservative, copying and generational.

URL: *http://www.research.microsoft.com/*

OpenJIT

The *OpenJIT*[OSM⁺00] project is an ongoing Java Programming Language JIT compiler project as a collaborative effort between *Tokyo Institute of Technology* and *Fujitsu Laboratory*, partly sponsored by the *Information Promotion Agency of Japan*. OpenJIT is a "reflective" JIT compiler in that it is almost entirely written in Java; it bootstraps and compiles itself during execution of the user program. Compiler components coexist as first-class objects in the user heap space; thus, users can tailor and customize the compilation of classes at runtime for a variety of purposes: application-specific optimization, partial evaluation, dynamic environment adaptation of programs, debugging, language extension, etc. OpenJIT allows full dynamic update of itself by loading the compiler classes on-the-fly. It is fully JDK compliant, and plugs into standard JVMs on several Unix platforms such as Solaris (Sparc), Linux (x86), and FreeBSD (x86).

URL: *http://www.openjit.org/*

2.3   Best of All Worlds

The JVM described by this document was designed to comprise the advantages, and supercede the ambitions, of all implementations available so far. In an early stage of development, the following features were defined as the set of goals we would like to achieve.

Virtual Machine The system must be a virtual machine, not a native compiler, in the sense that applications should be deployed on standard bytecode.

Full Functional End-User It must be a deployment JVM, supporting all end-user features including JNI[Sun97] and stack traces. Standard debugging and profiling interfaces were dropped on behalf of coherence. They are not required by end-users.

100% Native Execution Execution must be done natively, no interpreter bundled. Most implementations use mixed execution instead of 100% native execution. On JVMs this happens to avoid the overhead of compiling class initializers which will run only once. On native compilers, execution is 100% native for code compiled ahead of time. Some native compilers provide an interpreter to execute code loaded on-the-fly, but most don't.

Persistent Shared JIT Code Code generated by JIT must be cached on secondary memory for future reuse. The cache system must support sharing by multiple JVM instances running on the same computer. As far as we know, no JVM attempts to save JIT code on secondary memory. Some JVMs support one time loading which keeps JIT code on primary memory. Persistent caching enables 100% native execution.

Distributed Capability The JIT compiler may reside outside the computer that runs the JVM. Multiple JVMs may share a single JIT server. This extends the idea of sharing JIT code from a computer to a computer network.

Aggressive Compilation Aggressive compiler optimization techniques may apply, since the cache system dilutes compilation time. The JIT server has the ability to reoptimize frequently requested code on its idle time. We foresee not only "classical" optimizations, but also expensive whole-program optimizations (including object-oriented) only available on native compilers.

Mostly Written in Java The vast majority of the virtual machine is written in the Java Programming Language[AG00], including bytecode parser and verifier, and compiler data structures and algorithms. This takes advantage of object-oriented clarity and reuse and sets up a performance compromise: the virtual machine will run as fast as the code it generates. OpenJIT[OSM+00] have been successful on that.

Methods as First-Class Objects JIT compiled methods are represented inside the JVM as first-class objects. Therefore, methods may be garbage collected and handled from Java code.

Adaptative Optimization The JVM runtime must be constructed so as to support adaptative optimization. Profiling is done by inspecting stack frames from runtime callbacks. Recompilation occur conveniently for popular methods and classes. Replacement makes old version of methods eligible for garbage collection.

Accurate Garbage Collection The runtime must be able to compute accurately, at any time, the set of reachable objects. The accurate GC is clear, reliable, and flexible.

Small and Flexible Runtime The runtime must be small (simply JNI, threading and garbage collected heap). Easy replacement of threading and garbage collection strategies is a must. Portions of the runtime that can be written in Java are (less dependent on the correctness and performance of third party compilers).

Highly Portable The virtual machine is highly portable, being written almost totally in Java. The runtime is written in standard C[KR88] with well defined OS interfaces for native

threading and memory allocation. Portability is achieved rapidly for platforms based on the same processor. When porting to a new processor, a new compiler back-end must be written. Most optimizations occur in the intermediate representation, and do not require to be rewritten.

Precompiled Bootstrap Classes Bootstrap classes, including core library classes and JIT compiler classes, are precompiled during virtual machine generation. Those classes do not need to be changed by users, they only change when a new JVM version is released. This provides instant performance for system classes and avoid chicken-egg compilation problems (because the JIT is written in Java).

Selective Metadata The runtime does not need to waste primary memory with metadata for all loaded classes. It should be retrieved on demand from the cache system. Metadata are required by applications that use some special Java APIs (e.g. reflection) and for printing stack traces.

Chapter 3

Virtual Machine Design

This chapter describes the overall design of our JVM implementation. Rather than a Java Virtual Machine, the system is best qualified as a *Java Virtual Machine architecture*. Currently, this architecture is composed of three software components: a client, a server and a client-generator. The components, their roles, and how they interact are described here.

3.1 A JVM Architecture

Our implementation of the Java Runtime is best qualified as a Java Virtual Machine architecture. That is because it is not a single software that implements the JVM. There are software components that work independently and do specific tasks. Part of the system is generated by itself and the comunication is made using a simple service-oriented protocol. Currently, the JVM architecture is composed of three software components:

Client JVM The virtual machine itself, it comes in two flavors: *Standalone* and *Thin-Client*.

Client JVM Generator The virtual machine generator, automatically produces a Client JVM based on a given configuration.

Server JVM The virtual machine server, comprises basically the class file parser, bytecode verifier and JIT compiler. It has a secondary memory cache system.

3.2 Software Components

This section provides information about the software components that make up the JVM architecture.

3.2.1 Client JVM

The Client JVM component is the software that implements the user level Java Virtual Machine. Most of it is generated from Java bytecodes and coupled with a C runtime. Figure 3.1 shows the structural decomposition of the Client JVM in subcomponents.

As usual, the JVM is implemented as a platform dependent library that is loaded from an application, the *Launcher*, using JNI[Sun97]. It reads core and user classes from the file system (using the *CLASSPATH*[1]) or another application defined source. Native methods are implemented externally in *Native Libraries* which are loaded on demand by the JVM.
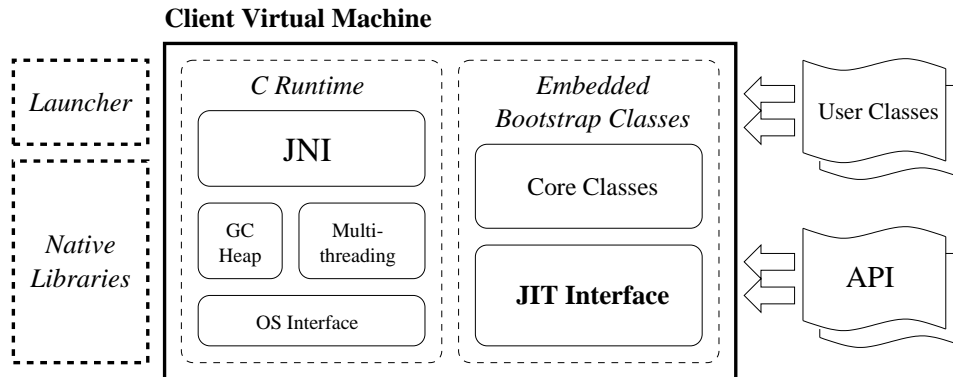
Figure 3.1: Client JVM subcomponents.

Internally, the Client JVM has a C runtime which comprises the implementation of JNI, heap structures and garbage collector algorithms, multithreading, exported extra JVM calls, and an OS interface (including assembly required calls). Most of the C runtime is written in standard C on a portable manner. There are special interfaces to processor and platform dependent operations. Both the garbage collector and multithreading subsystems have a well defined interface, which makes strategies replacement easy. The heap structures, however, are meant to be fixed. The extra JVM calls are calls required by native libraries that cannot be implemented from JNI calls (e.g. `Object.clone()`), they exist to decrease JVM internals exposure to core native libraries. The C runtime implementation details are described in Chapter 8. The garbage collector is detailed in Chapter 9.

As said before, part of the Client JVM is generated from Java bytecodes. These Java bytecodes comprise the core API classes[CLK98, CL97, CLK99] and some JVM implementation internal classes, including JIT compiler and Java language type support[LB98]. During generation, those classes are translated to assembly and placed on a segment following the heap layout. When the JVM is created, the C runtime does the bootstrap using those embedded classes. This enables the implementation of most of the Client JVM using the Java language.

The *JIT Interface*, as depicted on Figure 3.1, is part of the Java code embedded on the Client JVM. This interface defines a set of methods used by the Client JVM to handle dynamically loaded code, as well as other link-time activities. During generation, a JIT Interface implementation must be chosen to be embedded on the Client JVM. Currently, two implementations of the JIT Interface are provided, as shown in Figure 3.2.

The *Standalone* JIT Interface implementation (Figure 3.2 (a)) provides standard behavior to the Client JVM. As other JVM implementations, the complete functionality of the Client

---

[1]The CLASSPATH is specified by the launcher, which reads it from the command line or from an environment variable.

Standalone JIT (x86)

Execution Engine

Verifier

Embedded x86 Back-End

Optimizer

Cache Subsystem

Cache

(a)

Thin-Client JIT

Cache Subsystem

Connection Handler

Optional Local Cache

Server Connection

(b)
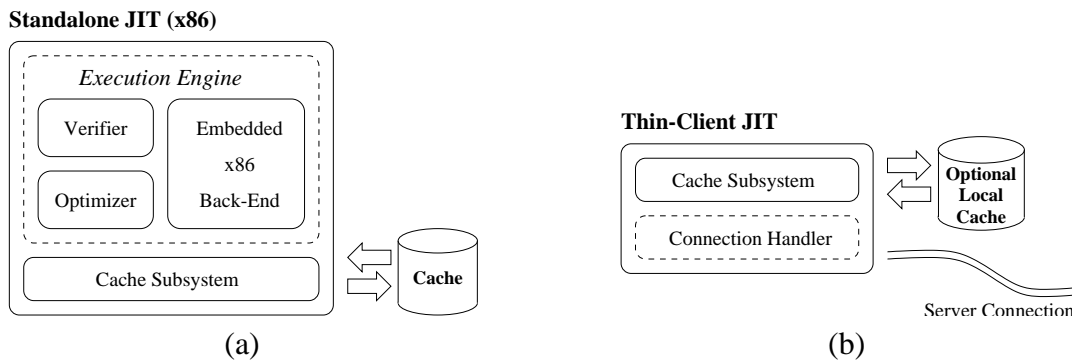
Figure 3.2: JIT interface implementations.

JVM is bundled in a single monolithic piece of software. In this case, all code — including class file parsing, bytecode verifier, JIT compiler and cache subsystem — will be embedded on the Client JVM. A specialized target-specific compiler back-end is also provided. The Standalone implementation allows compiled code caching and sharing on a single computer.

The other JIT Interface implementation available is the *Thin-Client* (Figure 3.2 (b)). A Client JVM that uses the Thin-Client implementation will require a Server JVM to execute, which receives requests delegated through a secure network connection. Therefore, the Thin-Client implementation is light-weight and is not required to run on a computer with secondary memory. Optionally it may have a local cache to minimize network activity.

3.2.2   Server JVM

The Server JVM component implements an external JIT compiler engine that receives one or more connections from Thin-Client JVMs. It is entirely written in the Java language and its core is composed of the same set of classes as the Standalone Client JVM.

Server Virtual Machine

Client Connections

Connection Handler

Execution Engine

Verifier

Optimizer

Back-End Plug-in Interface

x86 Back-End

Other Back-Ends

Cache Subsystem
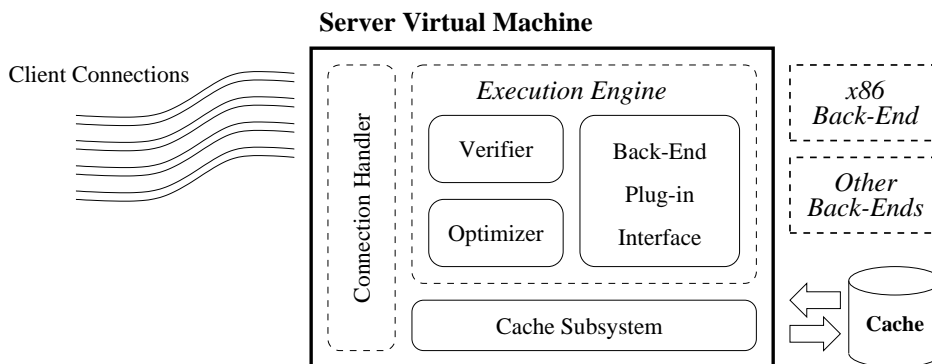
Cache

Figure 3.3: Server JVM subcomponents.

Figure 3.3 shows the structure and subcomponents of the Server JVM. Similar to Standalone Client JVM, it has a class file parser, bytecode verifier, JIT compiler, and cache sub-

system. The major difference is the *Back-End Plug-in Interface* which targets multiple simultaneous back-end implementations instead of a single embedded target-specific back-end. A Server JVM may generate machine code for many processors, as well as provide methods in raw intermediate representation. This raw IR mode can be used by *Interpreted Client JVMs*[2] which interpret IR, rather than bytecode, on unsupported processors.

Since it is written in the Java Programming Language, the Server JVM requires a Java Virtual Machine to execute. Instead of running the Server JVM on a third-party JVM, which could have a negative impact on performance, or run it on a Thin-Client JVM, which would require another Server JVM, we run it on a Standalone Client JVM. Doing that makes possible the sharing of common components of both the Server JVM and the underlying Standalone Client JVM, ideally.

3.2.3   Client JVM Generator

The *Client JVM Generator* generates Client JVMs based on a configuration. The Client JVM Generator configuration specifies the set of classes, and their linkage state, upon bootstrap. It simulates the loading and linking of core classes on a virtual heap — connecting to a Server JVM — and outputs an assembly file. The assembly file contains a segment declaration and the virtual heap transcript to the target platform layout. Automatic virtual machine generation is covered on Chapter 10.
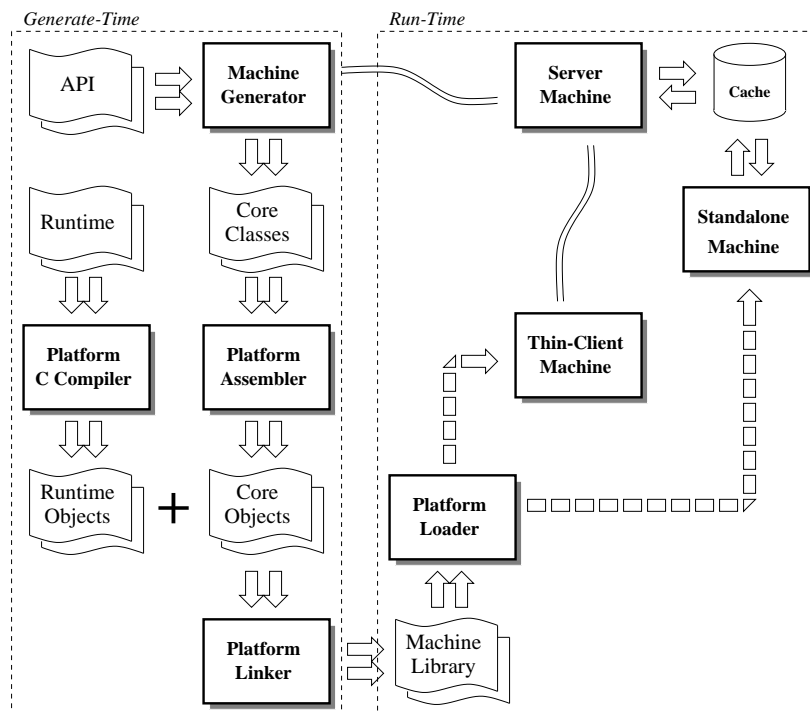
Figure 3.4: System functional diagram.

---

[2]Not implemented on the architecture.

3.3 Functional Overview

The Figure 3.4 depicts a functional diagram of the whole system.

At generate-time, a Client JVM is produced by linking objects resulting from compiling the C runtime and assembling the core classes output by the Client JVM Generator. For both Standalone and Thin-Client JIT Interface implementations, the C runtime is the same, but the core classes vary according to the configuration. During generation, the Client JVM Generator connects to the server to link classes and compile their associated methods. After linkage, a platform dependent shared library is output, and can be loaded using the standard JNI primitives.

At run-time, the Launcher application loads the Client JVM shared library using the platform loader. Depending on the configuration chosen during Client JVM generation, two runtime possibilities exist. In the Thin-Client implementation, the Client JVM connects to the server and delegates linkage requests to it. In the Standalone implementation, the Client JVM has embedded linkage functionality, it directly accesses the local cache system.

Aside from the rest of the system, the Server JVM is used by both the Thin-Client JVM and the Client JVM Generator to compile and link classes. It has a local cache system which may be shared with other instances of the JVM.

Chapter 4

Server-Side Context Identification

This chapter covers context identification. In order to implement the cache system, the Server JVM has to effectively detect repetitive situations in Client JVMs and respond using stored data. Details about how context identification was implemented and the problems that came up during its implementation are discussed.

4.1 States & Phases

The communication between Client JVMs and the Server JVM was designed using a very simple scheme. This communication is service-oriented and the special term *phase* is used to name services made available by the Server JVM. A phase has parameters that must be provided by the Client JVM, and a response that is computed by the Server JVM. Each phase starts a task for which the same parameters always yield the same response. This fact is the key point of the cache system.

The execution of the phase task always handles information regarding a particular class in a particular *state*. A class may be in one of three states: *registered*, *loaded* or *linked*, as depicted in Figure 4.1. Each one of these states is associated to a context required to compute the phase response.
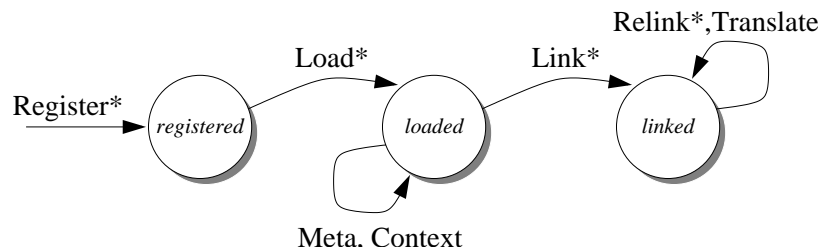
Figure 4.1: Diagram showing class states and phases.

A brief description of each phase is given bellow. As some important concepts are made clear in the following sections, an insight of each phase is provided to clarify the idea.

REGISTER Phase In this phase, a class image — as extracted from the class file — is registered in the system.

LOAD Phase In this phase, a registered class is hierarchicalized; information about its ancestors classes becomes available.

META Phase This phase provides meta information about a class.

CONTEXT Phase This phase provides class names for classes involved on the linkage of a loaded class.

LINK Phase In this phase, a loaded class is linked. This means it was fully verified and its methods have been optimized based on context information.

RELINK Phase In this phase, the methods of a linked class are reoptimized based on more context information.

TRANSLATE Phase This phase provides the binary native translation of the methods of a linked class.

The RELINK phase is not available on our first implementation. However, its purpose is exposed since a limited, but still useful, dynamic recompilation feature can be achieved based on it.

4.2   Computing Class Versions

In order to identify classes and class contexts off-line in the server side, we compute class versions. The class version is a number that in conjunction with the class name identifies a class in a particular context. There are three types of class versions: registered, loaded and linked (the states in the Figure 4.1). Some phases (REGISTER, LOAD, LINK, RELINK) change the version of a class.

The *registered version* is the version associated to a particular class image outside a context. A registered pair $<name\#rg\text{-}version>$ identifies the attempt to associate a particular class image, valid or not, with a class name.

The *loaded version* is the version of a class when it is placed in a type hierarchy. A loaded pair $<name\#ld\text{-}version>$ provides not only information about a class but also information about its ancestors.

The *linked version* is the version of a class when the information about classes and its *neighborhood* is known. The term neighborhood is used here to define all classes directly referenced by a particular class. Those classes comprises classes from which fields are accessed, methods are called, etc. It also includes classes handled by the JVM for some implicit operations (e.g. `ArrayIndexOutOfBoundsException` class may be instantiated by the `iaload` bytecode).

Not limited to class versions, the pair $<name\#version>$ is also used to identify phases (the arrows in the Figure 4.1). It is used as key to cache system entries.

In our implementation, the class version is coded as a 64-bit integer obtained from the 160-bit integer result of the application of the *SHA-1 hashing algorithm*[Nat95] to some parameters. The parameters depend on the information being requested. For instance, during class registration the hashing algorithm is applied over the class file image; on loading the algorithm is applied over the loaded versions of the ancestors of the class being loaded as well as its registered version and some other specific data. The mapping from 160-bit integer to the 64-bit is done by applying 4 xor operations over each 32-bit chunk shifted by 8. This map is done solely to provide shorter file names when writing the response to the cache system, since the file name is derived from the pair `<name#version>`.

Since this is the first implementation of our system, we did not handle clashes in class versions. We believe that the validation of the caching idea was important enough to ignore this fact. Moreover, the SHA-1 algorithm was designed to avoid clash occurrence. It is even computationally infeasible to simulate it[1]. However, its identification is an important issue since it is related to the system predictability. We intend to handle this issue in future implementations.

4.3   Dealing with Class Loaders

Class loaders offer a great challenge when trying to effectively identify contexts, which is crucial to exploit the cache system. The main problems which regard the existence of class loaders are related to safe type manipulation, the java security (package private members accesses) and the mapping of its dynamic nature.

Types inside the JVM are identified by a pair *(class name, class loader)*. However, since class loaders are instances of class `ClassLoader`, they may be instantiated at run-time. Therefore, new types can be introduced in the system during its execution. This feature gives some dynamic typing flavor to the Java language. As class loaders have a dynamic nature, it is difficult to identify and detect contexts inside the JVM. This is explained by the example, showed in Figure 4.2.
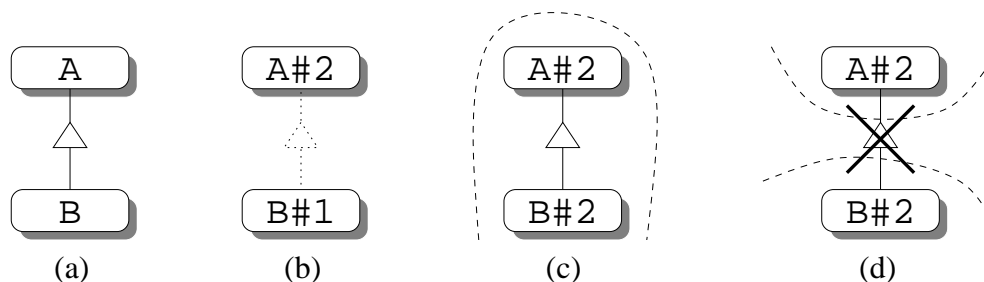
Figure 4.2: Extending non-public class, context: (a) Static; (b) Before loading; (c) Both classes defined by the same class loader; (d) Each class defined in a different class loader.

Suppose we are loading class *B* which symbolically extends a non-public class *A* (Fig-

---

[1]It did not happen during testing and benchmarking of the system.

ure 4.2 (a)). At load time, we have already registered class *B* so we have its registered version, say *1*; class *A* is also loaded and has its loaded version, say *2* (Figure 4.2 (b)).

By now, lets forget about class loaders. The Client JVM sends to the Server JVM the parameters to the LOAD phase which are: superclass = *<A#2>*; interfaces = $\emptyset$; class = *<B#1>*. Note that we are trying to capture a context here (the detailed description of how this occurs will be given on Section 4.5 when we look at each phase). The Server JVM then analyses the context for the LOAD phase and sends to the Client JVM an error response or the loaded version of class *B*, say *2*. So, by ignoring class loaders, the server will send back to the client the response *<B#2>* (Figure 4.2 (c)).

However, when considering that class *A* was not defined in the same class loader as class *B*, then class *A* is not in the same *runtime package* of class *B*. Therefore, since class *A* is non-public, the access must be denied by the Server JVM. In the Client JVM, an instance of *IllegalAccessError* must be thrown (Figure 4.2 (d)).

How to capture this simple context when class loaders, and therefore types, cannot be trivially identified? The solution we found was not to identify class loaders at all. However, we associate to each class relation — where class loader existence matters (e.g. extended class, implemented interface, accessed field owner) — a boolean value indicating if both elements of the relation where defined by the same class loader.
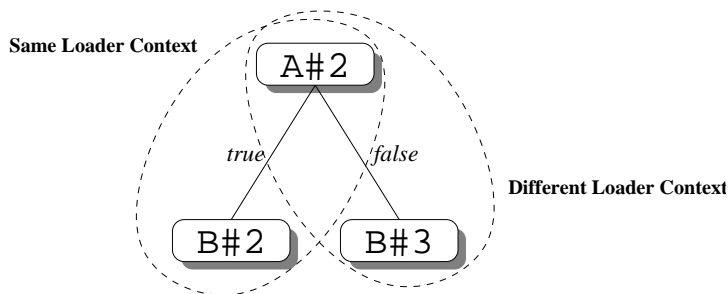
Figure 4.3: Mapped contexts.

Figure 4.3 shows the two contexts mapped using class loader information (In the figure, dashed lines group together classes defined by the same class loader). Note that the version of class *B* is different on each context, since the class loader flag is considered in version calculation.

4.3.1 Extended Loader-Based Class Names

Given that class loaders are not identifiable by our context mapping scheme, we have to provide an alternate form of identification for types. This is required if we intend to handle classes from outside the *namespace*[2] of the class in which a particular context is based.

We have extended the *fully-qualified internal class name form*[LY99, §2.7.4] in order to address this issue. The new syntax is shown in Figure 4.4.

---

[2]The association of namespaces to classes is an overloaded usage of the term, when we refer to the namespace of a class we are actually refering to its defining class loader namespace.

ExtendedClassName → *FullyQualifiedClassName*
| *ExtendedClassName* "^" *FullyQualifiedClassName*

Figure 4.4: Extended class name syntax.

The extended class name is always interpreted in the sense of a particular class, in the same way as class names are interpreted in the sense of a class loader. The first fully-qualified name (from left to right) identifies a class with that name in the current namespace. So on, each fully-qualified name identifies a class with that name component in the namespace of the previously identified class. For instance, the class name `B^C`, when interpreted in the context of class `A`, identifies the class `C` obtained from the namespace of class B which is obtained from the class loader of class A.

Obviously, each class may be identified by multiple extended class names. Sometimes, it is possible to tell if two extended class names refer to the same class; if they surely refer to distincts classes; as well as not being able to state any of that at all.

Here are some facts about extended class names:

- Two extended class names only refer to the same class if the rightmost fully-qualified name is the same for both.

- The syntax and semantics of extended class names in the context of a class is upwards compatible with the syntax and semantics of the fully-qualified class names of classes referenced by that class.

- Every subsequence of fully-qualified class names — where each name identifies a class defined by the same class loader as others — may be omitted, but the first, from the extended class name.

- No special syntax is used to identify bootstrap classes. Since the hierarchy root is the bootstrap `Object` class, any bootstrap class may be identified by first identifying the hierarchy root and then refering to a class in its namespace.

4.3.2 Type Uncertainty and Interfaces

Although it enables context mapping and caching, the mechanism for dealing with class loaders has serious problems regarding type inference. In some situations, the amount of context information available about class relations is not enough to tell if a versioned class represents one or many distinct classes. This is explained by an example.

Suppose we have two runtime contexts. In the first context, depicted in Figure 4.5 (a), a subclass `B` reimplements an interface `I` already implemented by its superclass `A`. In the second context, depicted in Figure 4.5 (b), the subclass `B` implements another interface binary compatible with interface `I` but defined in another class loader, so it represents another type.

When mapping these two contexts we get the same class relation, as depicted in Figure 4.6. This occurs because there is not enough context information to let us distinguish interfaces `I`
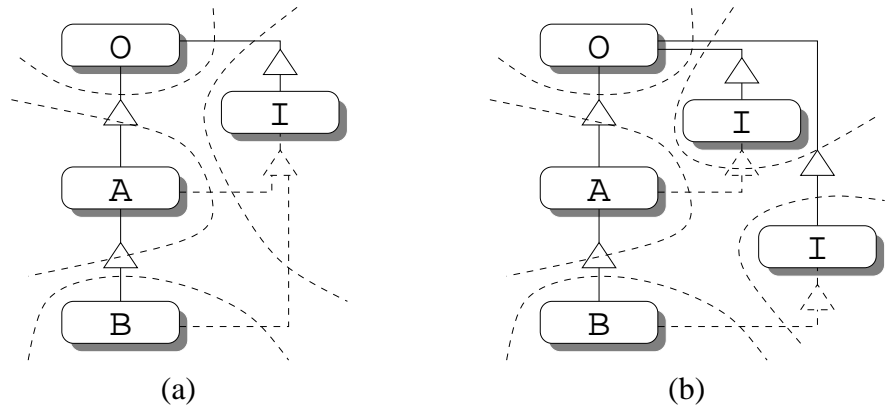
Figure 4.5: Two contexts: (a) Single interface; (b) Multiple interface.

in the second context. Both interfaces are subclasses of O not defined in its class loader. Therefore two distinct contexts have the same mapping; avoiding this cannot be done effectively.

The type uncertainty occurs whenever at least two class relation edges in the path from one class to another are false. This means that there is no way to tell if those classes are defined by the same class loader or not. In that case, both contexts have the same mapping.

Type uncertainty limits the use of type inference in optimizations. For instance, in the example shown, we cannot tell from class B if class $A \hat{} I$ equals class I.

Luckly, when constructing method tables, there is no need to reserve distinct method areas inside method tables for each path leading to a versioned ancestor class[3] passing through two or more false edges. Since all classes represented by the versioned class have binary compatibility, they can share the same method area.
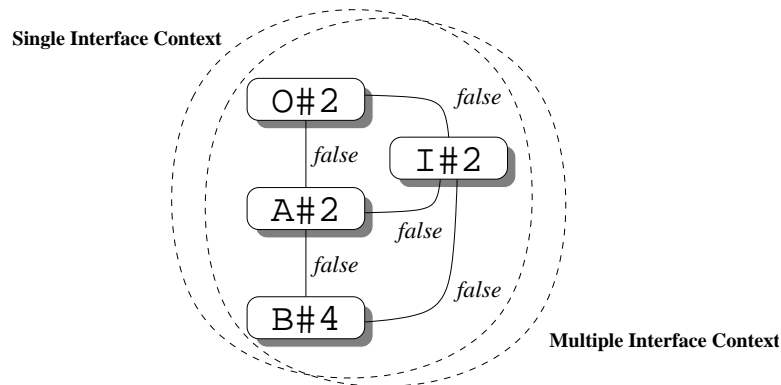
Figure 4.6: Different contexts that are identified equally.

---

[3] Actually, it only occurs to interfaces since the Java language does not support true multiple inheritance.

4.4 A Portable Way of Describing Sizes and Offsets

Since we did not want to tailor the Server JVM to a particular architecture or platform, we had to provide a portable way of describing sizes and offsets of associated to heap information. That was achieved by representing sizes or offsets by a pair *(references, bytes)*. The first element of the pair is the number of references that contribute to the measure. The second element of the pair is the number of bytes that contribute to the measure. To calculate the actual value in bytes represented by the pair, the Client JVM must scale the first element, using the size in bytes of references in its implementation, and add to the second element. For instance, the value of the pair *(3,4)* in a 32-bit system is $3 * 4 + 4 = 16$ bytes.

4.5 Describing Each Phase

In this section, each one of the phases introduced in Section 4.1 is described. We focus on the main features of each phase rather than providing a precise specification or listing all implementation details. However, whenever necessary, issues regarding precise understanding of these features are detailed.

4.5.1 REGISTER Phase

In the REGISTER phase, the Client JVM sends to the Server JVM the expected name and binary contents of the class file associated with the class being processed.

The Server JVM applies the hashing algorithm to the class file contents; the result is used as the *register version number* for that class. Then, the Server JVM looks for an entry `<name#version>` in the cache system. On a miss, it tries to parse the class file contents (Pass 1 on the verification process[LY99, §4.9.1]), caches it and returns a response.

```
REGISTER (java/util/Stack)

     class-id:   <java/util/Stack#8ee05ed6bda9bce1>

 access flags:   0x21
   superclass:   java/util/Vector
                 <no interfaces>
```

Figure 4.7: REGISTER phase result information for class `Stack`.

Figure 4.7 shows the result of applying the REGISTER phase to class `Stack`.

The response depends on successful parsing of the class file contents. If the class file contents are valid, the Server JVM sends back to the Client JVM the registered version number, the access flags, the superclass name (if any) and the implemented interfaces names (if any). As the result of an error during class registration, the Server JVM may provide three possible

causes: unsupported class file version number, malformed class file contents or unexpected class name.

Upon the successful response, the Client JVM will proceed to the LOAD phase which in conjunction with the REGISTER phase makes up the JVM class creation and loading process ([LY99, §5.3]). In the case of an error response, the Client JVM will throw an instance of the following classes, respectively: `UnsupportedClassVersionError`, `ClassFormatError` or `NoClassDefFoundError`.

4.5.2   LOAD Phase

As said before, to complete the JVM class loading process the LOAD phase takes place just after the REGISTER phase. In the LOAD phase the associated class is *hierarchicalized*, meaning that all its ancestors in the type hierarchy must be known. Once hierarchicalized, more information can be gathered about that class, including instance and static sizes, field offsets, dynamic dispatch method table length, method dispatch indices, etc. Also, after the LOAD phase has successfully being completed, a new version number is associated to that class.

In order to request a LOAD phase, the Client JVM should provide some parameters which are: the class name and registered version of the class being processed; a boolean value indicating if it is a bootstrap class; the superclass name, the superclass loaded version and a boolean value indicating if it was defined by the same class loader as the processed class; for each direct implemented interface, its name, its loaded version and a boolean value indicating if the interface was defined by the same class loader as the processed class.

The Server JVM then computes the loaded version number for the class being processed, by applying the hash algorithm to these parameters. A $<name\#version>$ pair is obtained and used as key in the cache system. On a miss, the Server JVM has to compute the response to the LOAD phase, otherwise the response is ready to be sent to the Client JVM.

As occurred in the REGISTER phase, the Server JVM processes the parameters and provide some useful client data or return an error message on failure. The data provided by the Server JVM in the LOAD response contains mostly information about the size of the areas used by the associated class, it comprises: the loaded version, the static field area size, the offset and length of the reference table inside the static field area, the instance field area size, the offset and length of the reference table inside the instance field area, the dynamic dispatch method table length, the direct interfaces base offsets in the dynamic dispatch method table and the native pointer table length. The meaning and use of each of these informations will become clear when we describe the client runtime in Chapter 8. As result of an error in this phase, the Server JVM may provide two possible causes: its superclass is an interface (or any of its direct implemented interfaces is a class), or the class is unable to access its direct ancestors.

In the client side, the information provided by the Server JVM is used to change the version number and allocate storage for the class being processed. On error, the Client JVM should throw an instance of the following classes respectively: `IllegalAccessError` and `IncompatibleClassChangeError`. Also, the Client JVM is responsible for keeping track of hierarchy circularity, throwing a `ClassCircularityError` instance whenever necessary.

Figure 4.8 shows the result of applying the LOAD phase to class `Stack`.

```
LOAD (java/util/Stack)

          class-id:   <java/util/Stack#5c5614ee3f7ba14>

     static fields:   size (0,8) refs 0 offset (0,8)
   instance fields:   size (1,12) refs 0 offset (1,12)
    static methods:   table length (2)
  instance methods:   table length (91)
    native methods:   pointers (0)
```

Figure 4.8: LOAD phase result information for class *Stack*.

The size and offset for the field areas are represented, as described in Section 4.4, using a pair *(references,bytes)*. The Server JVM tries to place the fields assuming that every field area will be padded based on the target architecture word size (as commonly implemented in the memory allocator). For instance, for field areas, it first tries to fill in holes left blank by the padding on the superclasses areas, then it places the remaining fields on a decreasing size order assuming references are 32-bit wide. Using this policy, the Server JVM generates the best alignment for 32-bit systems. Also the Server JVM tries to generate the best alignment for 64-bit systems, by leaving a 4-byte hole before placing 8-byte fields on an unaligned area; but this is done only if the hole is filled after the remaining fields are placed. We prefer having some misaligned references on 64-bit systems rather than wasting a word space on 32-bit systems. For static field areas, we place fields on a decreasing size order, getting the best alignment for all systems.

The calculation of dispatch method table size and placement is simpler. There is no alignment problem since all method pointers have the same size for all systems. The dispatch method table placement for the current class uses the superclass dispatch table as start point. Then, for each interface implementation not yet implemented by the superclass its dispatch table is appended, its base index is associated to the interface being implemented. At last, for every new method declared in the current class, an entry is appended to the method table. This provides the size of the dispatch method table for the current class. The initialization and override patches for the dispatch method table are done in the LINK and TRANSLATE phases. Static methods are placed on a separate table, since they need not to appear in subclasses dispatch tables.

4.5.3 META Phase

In the META phase, the Server JVM sends back to the Client JVM meta information about a particular class. The meta information is required basically by the JNI, the reflection API and when printing stack traces. The META phase does not change the linking state of the class nor cause any link-time error to be thrown.

The major advantage of having the META phase is the fact that the Client JVM does not

need to store meta information for most of its loaded classes. It is an effort to decrease the memory footprint on the client side.

In order to request a META phase, the Client JVM should provide the loaded version or linked version[4] of the related class. The Server JVM then sends its response, which is the compilation of all meta information currently required by the runtime.

```
META (java/util/Stack)


        class-id:   <java/util/Stack#5c5614ee3f7ba14>


  access flags:   0x21
      field[ 0]:   0x1a serialVersionUID J offset (0,0)


   method[ 0]:   0x1 empty()Z index 86
   method[ 1]:   0x1 push(Ljava/lang/Object;)V; index 87
   method[ 2]:   0x21 peek()Ljava/lang/Object; index 88
   method[ 3]:   0x21 pop()Ljava/lang/Object; index 89
   method[ 4]:   0x21 search(Ljava/lang/Object;)I index 90
   method[ 5]:   0x1 <init>()V index 92


declaring class:   <top level>

                   <no inner classes>

   source file:   Stack.java
```

Figure 4.9: META phase result information for class `Stack`.

Figure 4.9 shows the result of applying the META phase to class `Stack`. Metadata comprises the following information:

- Access flags (from source declaration).

- Owned fields metadata:

    – Access flags.

    – Name and descriptor.

    – Offset.

- Owned methods metadata:

    – Access flags.

---

[4]The loaded version is always available from the linked version on the server side.

- – Name and descriptor.

- – Exceptions thrown (from source declaration).

- – Dispatch index.

- • The declaring class if this class is an inner class.

- • Owned inner classes metadata:

  - – Access flags.

  - – Name.

- • Source file, if available.

4.5.4   CONTEXT Phase

The CONTEXT phase provides the class names of all classes required to proceed with the LINK phase of a particular class. The class names present in the CONTEXT response are in the extended fully-qualified form (Section 4.3.1). The CONTEXT phase does not change the linking state of the class nor cause any link-time error to be thrown.

```
CONTEXT (java/util/Stack)

    class-id:   <java/util/Stack#5c5614ee3f7ba14>

 context[ 0]:   java/lang/IllegalMonitorStateException
 context[ 1]:   java/util/EmptyStackException
 context[ 2]:   java/lang/NullPointerException
 context[ 3]:   java/util/Vector
```

Figure 4.10: CONTEXT phase result information for class *Stack*.

Figure 4.10 shows the result of applying the CONTEXT phase to class *Stack*.

4.5.5   LINK Phase

During the LINK phase, a particular class is linked, verified (see Chapter 5) and its methods are converted to the *intermediate representation* (see Chapter 6).

In order to request a LINK phase, the Client JVM should provide some parameters which are: the class name and loaded version of the class being processed; for each class in the neighborhood: its loaded or linked version, a boolean indicating if that class has been initialized, and a boolean for each of its superclasses (including itself) indicating if they were defined in the same class loader as the class being processed. This last parameter is used to

correctly implement the resolution and overriding of package private members only accessible by classes in the same runtime package.

The Server JVM then computes the linked version number for the class being processed, by applying the hash algorithm to these parameters. A $<name\#version>$ pair is obtained and used as key in the cache system. On a miss, the Server JVM has to compute the response to the LINK phase, otherwise the response is ready to be sent to the Client JVM.

As occurred in other phases, the Server JVM processes the parameters and provide some useful client data or return an error message on failure. The data provided by the Server JVM in the LINK response comprises: verification constraints, loading constraints, interface implementation dispatch table patches, and a flag indicating if instances of the processed class will require finalization. The *verification constraints* are a set of pairs that symbolically encode subtype tests that must hold to complete the class verification process. Each pair is composed of an extended class name representing the supertype and a set of extended class names representing the subtypes. To validate a verification constraint, the Client JVM must check if the supertype class can be assigned from the first common superclass of the subtype classes (Details are given in Chapter 5). If any of the verification constraints fail, verification also fails. Similarly, the *loading constraints* are a set of pairs that symbolically encode type equality tests that must be imposed whenever execution crosses class loader boundaries ([LY99, §5.3.4]). Both elements of the loading constraint are extended class names.

The *interface implementation dispatch table patch set* is a set of pairs indicating which method table entries must be copied from the superclass method table to cover new interface method areas whose method implementation was inherited. The construction of the dispatch method table is partially done during the LINK phase and completed in the TRANSLATE phase. In the LINK phase, the dispatch table is initialized with a snapshot of the superclass dispatch table, also some entries in the new interface method areas are initialized using the interface implementation patches. In the TRANSLATE phase, each method is sent back to the client with a set of patches that completes the dispatch table construction process. In case of an error during this phase, the Server JVM may provide seven possible causes: a access to class or member is denied, class did not pass verification, field does not exists, method does not exists, static method or interface is used where a non-static method or class is expected (or vice-versa), method to be invoked is abstract, or class to instantiated is abstract.

Figure 4.11 shows the result of applying the LINK phase to class `Stack`.

In the client side, the information provided by the Server JVM is used to change the version number, check verification constraints, impose loading constraints, perform the first step of dispatch table initialization, and optimize garbage collection. When an error occurs, the Client JVM should throw an instance of the following classes, respectively: `IllegalAccessError`, `VerifyError`, `NoSuchFieldError`, `NoSuchMethodError`, `IncompatibleClassChangeError`, `AbstractMethodError`, and `InstantiationError`. Also, the Client JVM is responsible for throwing a `VerifyError` or `LinkageError`, respectively, if the checking of verification constraints or the imposing of loading constraints fails.

```
LINK (java/util/Stack)

        class-id:   <java/util/Stack#9287908c86cf3330>

verify target[ 0]:   java/util/Vector
      source[ 0]:   java/util/Stack
verify target[ 1]:   java/lang/Throwable
      source[ 0]:   java/util/EmptyStackException

                     <no loading constraints>

                     <no implementation patches>

        finalizes:   false
```

Figure 4.11: LINK phase result information for class `Stack`.

4.5.6 RELINK Phase (Not Implemented)

The RELINK phase allows the Client JVM to update the context information for a particular class. Once a class has been relinked, usually, the Server JVM will have more information about classes on its neighborhood. At the same time, more information about that class will become available to classes whose neighborhood contains it. More information means that the JIT will have a greater opportunity to optimize the code during a subsequent TRANSLATE phase. The optimization is applied to all methods of a particular class, and the results are kept associated with the new context in the system cache.

Although the RELINK phase has not been fully implemented in our system, it has been foreseen in our design. The RELINK phase allows a limited but still valuable form of *dynamic adaptative reoptimization*. It allows the Client JVM to implement heuristics for detecting critical classes and replace their methods by improved versions. Although the Client JVM does not provide explicit runtime profiling data, those heuristics may be constructed by analysing each thread stack during callbacks to the runtime. As described on Chapter 8, this support for discovering the method call chain — and their declaring classes — in a thread stack, is a requirement to print stack traces and implement caller class inspection security checks in the API.

Similarly to the LINK phase, each RELINK phase receives as parameters the versions of the classes in a given neighborhood; it responds by just modifying the related class version. That version update reflects on the neighborhoods in which that class takes part.

If a class takes part of another class neighborhood ($\mathcal{N}$), and vice-versa, the RELINK request in one will modify the other class context. When the latter is relinked the former context will also be affected. This should happen only until no more change on contexts happens, but versions could keep changing indefinately. The convergence criteria which avoids this is that a class C can only be relinked until all classes on its *iterated neighborhood* ($\mathcal{N}^*$) are in a

linked state. The iterated neighborhood of a class is the union of the set of classes in its neighborhood, and the classes in their iterated neighborhoods.

$$\mathcal{N}_c^* = \mathcal{N}_c \cup \bigcup_{d \in \mathcal{N}_c} \mathcal{N}_d^*$$

This criteria prevents the system from applying the RELINK phase indefinately for a particular class. However, this criteria does not help the system from applying useless RELINK phases — and version changes — until all classes are in linked state. A RELINK phase is useless if the cardinality of the iterated neighborhood subset of classes in loaded state does not change upon its application. This can be detected in the server side and, in this case, the Server JVM must not apply a class version change. It does not prevent the useless RELINK phase from taking place, but prevents a class version change which increases the probability of a cache hit.

4.5.7 TRANSLATE Phase

In the TRANSLATE phase, the Server JVM sends to the Client JVM the native binary images of its methods, as well as the entries in the dynamic dispatch method table that should point to each of them. The parameters to the TRANSLATE phase are the *linked class version* and a *back-end name*. Upon a TRANSLATE request, the Server JVM applies the hashing algorithm over its parameters and check for an entry in the cache system. On a miss, the Server JVM will uses the back-end name to dynamically load an implementation of its *back-end interface*. If such back-end implementation exists, each method is translated, otherwise an error is returned to the Client JVM. At this time, our implementation supports two back-end implementations: *raw* and *x86*.

The *raw* back-end returns to the Client JVM the methods without translating then to any machine language, they are kept in internal *intermediate representation* IR form. The *raw* back-end is intended to be used by client implementations based on interpretation. The interpretation of the IR is simpler and faster than the interpretation of the Java bytecodes. In the IR form, complex operations are broken into simpler ones, and execution takes advantage of mid-level optimizations.

The *x86* back-end, described on Chapter 7, translates the methods from IR form to the Intel Architecture 32-bit machine language, targeting the 80386 processor. The response sent to the client is an array of bytes, as well as some relocation tables that should be used to patch it in the client side.

Chapter 5

Efficient Bytecode Verification

This chapter provides details about the *bytecode verification* procedure. The verification procedure herein described is mostly symbolic which means it can be done off-line with some checks at run-time. The approach we use for the data flow analysis is different from the standard procedure[LY99, §4.9] in the sense that it iterates over basic blocks instead of instructions, moreover subroutines restrictions are relaxed for the sake of generality. Bytecode conversion to intermediate representation, covered in Chapter 6, relies on information provided by the verification procedure to discover the actual types of untyped operations (e.g. `dup` bytecode).

5.1   Symbolic Bytecode Verification

Symbolic bytecode verification provides the same results of standard bytecode verification but working with class names instead of using actual class and hierarchy information. It was designed to be performed off-line, when actual type information is not available. It generates data to be used to complete the verification process once type information is available. Actual type information is required at two times during verification:

1. To discover the type of two or more references that share the storage after a path merge in the data flow analysis. The type after the merge is considered to be the first common superclass of the types prior the path merge.

2. To test subtyping whenever a type is used where another type is expected. The former must be a subtype of the latter.

We have implemented the verification procedure in a way that both situations can be handled later, when type information becomes available. First, to handle types resulting of a path merge, we encode types not as bare class names but as *class name sets*. So, each set represents the first common superclass of its elements. Trivially, a set of cardinality 1 represents its unique element. During a path merge, a union of the sets is performed. Second, to handle subtype tests, we generate a set of *verification constraints*. A verification constraint is a pair of class name sets that encodes a subtype test. The first element of the pair is the target type

which must be a supertype of the second element of the pair. The symbolic verification can thus take place off-line. If it does not fail, it generates a set of verification constraints that must be checked using actual type information. If any of the verification constraints fail, the verification procedure also fails.

5.2   Parsing the Class File

The parsing of the class file is performed in the REGISTER phase and it ensures that the format is not corrupted. During parsing the constant pool, class, fields, methods and attributes are extracted from the class file and checked for their basic layout and contents. Names and descriptors have their syntax checked. The bytecode array for each method is read from the class file but it is not checked, the check is postponed until the LINK phase when static and structural constraints are checked.

5.3   Checking Static Constraints

Checking the static constraints of the bytecode guarantees that instructions and their corresponding placements obey certain simple rules. It can also be used to gather some important information that will be required later in the verification process. Failures during static constraint check should throw an instance of `VerifyError`. The procedure for checking static constraints is the following:

1. Check if the bytecode array size is greater than zero.

2. Set *leader* flag, used to identify basic blocks.

3. Starting at offset 0, for each instruction do:

   (a) Check if the opcode at the offset is legal.

   (b) Mark the offset as *valid*.

   (c) If *leader* is set, mark the offset as *leader* and reset *leader*.

   (d) If the instruction has size greater than 1, check if its size exceeds the bytecode array (special care should be taken to handle variable sized instructions `tableswitch` and `lookupswitch`).

   (e) If the instruction accesses local variables, check if the frame index is less than the frame capacity (special care should be taken to handle instructions that accesses long and double data).

   (f) If the instruction makes references to the constant pool, check if the constant pool entries are legal according to the instruction semantics.

   (g) If the instruction makes an explicit branch, check if the branch target offset is inside the bytecode array and mark it as *leader* and *target*.

(h) If the instruction may throw an exception, and is enclosed by at least one exception handler then set *leader* flag.

(i) If the instruction does not fall through then set *leader* flag.

(j) If the instruction is a return instruction, check if it is the one required by the return type of the associated method.

(k) If the instruction is a invoke instruction, check if the number of parameter words required by it does not exceed 255. Also check if the method being called is not `<init>`, except for `invokespecial`.

(l) If the instruction allocates an array, check if the array dimensions does not exceed 255 and is legal according to the instruction semantics.

(m) Increment offset by the instruction size.

4. For each exception handler do:

   (a) Check if the start pc offset is *valid*.

   (b) Check if the end pc offset is *valid* or equals the bytecode array length.

   (c) Mark the handler pc offset as *leader* and *target*.

5. Starting at the offset 0, for every offset do:

   (a) Check if the offset is not *valid* and is *target*.

   (b) If the offset is *leader* or equals the bytecode array length, mark the previous visited offset as *trailer*.

If none of the checks have failed, the bytecode has been successfully checked against the static constraints.

5.4   Checking Structural Constraints

Checking structural constraints requires computing operand stack sizes and accurate type information. This is done by applying data flow analysis over the bytecode array since that information depends on the execution flow.

In order to provide an efficient — faster and consuming less memory — implementation of the data flow analysis, we extract the control flow graph from the bytecode array, differently from the specification proposal[LY99, §4.9] which suggests doing the data flow on a instruction basis. Working with basic blocks instead of instructions is a well known technique to speedup data flow analysis. We have successfully adapted this technique to the bytecode verification.

The control flow graph is built using information gathered during the static constraints check. Each basic block encloses instructions from a *leader* offset to a *trailer* offset inclusive. By looking at the instruction in each *trailer* offset, it is possible to add the edges to the graph. Edges are classified in two types: normal edges and exception edges. Normal edges are those

edges generated from explicit control actions in the code, i.e. branching and falling through. Exception edges are those edges generated implicitly by assuming that an exception is thrown and catched by a handler. There will be an exception edge from each instruction that may throw an exception, to each handler that encloses that instruction. Each exception edge is labeled with the class name of the exception being handled.

In our implementation, the `ret` instruction does not generate edges during control flow graph building nor during the data flow analysis (it is treated specially as described a further ahead).

Once the control flow graph has been built, for each basic block we generate useful information required to semantically provide the effects of the verification on that block. This information comprises:

Operand Stack Delta The increment or decrement of the operand stack size after the execution of this basic block. Used during flow analysis to compute operand stack sizes at each basic block entry.

Maximum Operand Stack Decrement Used to check if this basic block will underflow the operand stack.

Maximum Operand Stack Increment Used to check if this basic block will overflow the operand stack.

Written Frame Indexes Set A set of indices written by this basic block when executed. Used to flow through `ret` instructions.

Flow Function Pseudo code modeling the effects of the verifier on the basic block as a whole.

The set of pseudo code instructions used by the verifier to encode the semantics of basic blocks is show in Table 5.1. During the construction of the flow function, some simplifications may be applied to the sequence of pseudo code instructions. For instance, if an IPUSH is followed by an IPOP then both pseudo code instructions can be removed from the instruction sequence without affecting its semantics.

In order to compute the data flow analysis, we have to define the data flow item that reflects execution state for each point in the bytecode. The data flow item comprises the current operand stack size and types, the current local frame types, and a boolean flag indicating if the `this` parameter has been initialized so far. The data flow item types used by the verifier during the data flow analysis are:

Integer The data flow item represents an integer.

Float The data flow item represents a float.

Long First Word The data flow item represents the first word of a long.

Long Second Word The data flow item represents the second word of a long.

Double First Word The data flow item represents the first word of a double.

PSEUDO CODE	ATTRIBUTE	SEMANTICS
IPUSH		pushes integer
IPOP		pops integer
IENSURE	frame index	ensures local variable is integer
ISET	frame index	sets local variable to integer
LPUSH		pushes long
LPOP		pops long
LENSURE	frame index	ensures local variable is long first word
LENSURE2	frame index	ensures local variable is long second word
LSET	frame index	sets local variable to long first word
LSET2	frame index	sets local variable to long second word
FPUSH		pushes float
FPOP		pops float
FENSURE	frame index	ensures local variable is float
FSET	frame index	sets local variable to float
DPUSH		pushes double
DPOP		pops double
DENSURE	frame index	ensures local variable is double first word
DENSURE2	frame index	ensures local variable is double second word
DSET	frame index	sets local variable to double first word
DSET2	frame index	sets local variable to double second word
APUSH	class name	pushes reference
APOP		pops reference
APOPSUBTYPE	class name	pops reference ensuring its a subtype
APOPARRAY		pops array reference
APOPBARRAY		pops boolean or byte array reference
AGETCOMP		pops array reference and pushes component type reference
ASETCOMP		pops reference and array ensuring component subtyping
AULOAD	frame index	loads and pushes reference or uninitialized reference
AURSTORE	frame index	pops and stores reference, uninitialized reference or ret address
UPUSH	offset	pushes uninitialized reference
UPOPINIT	class name	pops uninitialized reference and records its initialization
RPUSH	offset	pushes ret address
RENSURE	frame index	ensures local variable is ret address
POP1		same semantics as bytecode *pop*
POP2		same semantics as bytecode *pop2*
DUP1		same semantics as bytecode *dup*
DUP1X1		same semantics as bytecode *dup_x1*
DUP1X2		same semantics as bytecode *dup_x2*
DUP2		same semantics as bytecode *dup2*
DUP2X1		same semantics as bytecode *dup2_x1*
DUP2X2		same semantics as bytecode *dup2_x2*
SWAP1X1		same semantics as bytecode *swap*
CHECKINIT		checks if *this* has been initialized
CHECKFALL		checks if execution falls through the bytecode array

Table 5.1: Pseudo code instruction set.

Double Second Word The data flow item represents the second word of a double.

Null The data flow item represents the `null` reference.

Reference The data flow item represents a reference to an instance of a known type. The type is encoded as a set of class names which symbolically refers to the first common superclass of those classes.

Uninitialized Reference The data flow item represents a reference to a newly created instance, not yet initialized (i.e. lacks class or superclass constructor call). It is encoded as the offset of the instantiation instruction or a negative flag if it is the `this` parameter of a constructor. When a `<init>` method is called using an uninitialized reference, the data flow analyser searches the local frame and operand stack for copies of that uninitialized references, using its offset, and replace them by its actual type as extracted from the instruction at that offset. Specially if the uninitialized reference offset is negative, the data flow item flag for `this` parameter initialization is set, and the copies are replaced the the type of the current class.

Ret Address The data flow item represents a subroutine ret address. It is encoded as a set of pairs. Each pair consists of the ret address actual offset and the set of frame indices written since the associated subroutine start. It is used when flowing through `ret` instructions.

The data flow analysis algorithm we have implemented uses a basic block *working list*. Initially the working list contains only the entry basic block; the algorithm iterates until the list is empty. Each iteration, a basic block is chosen and removed from the working list, its flow function is used to compute the data flow item at the exit point of the basic block (*output data flow item*) using the data flow item at its entry point (*input data flow item*). The data flow item in the exit point is merged and compared to the data flow item at the entry point of each of the successor basic blocks. If the comparation fails the data flow item at the entry point of the successor is overridden by the new value and it is inserted into the working list.

Performing the dat flow analysis through a basic block consists of the following simple steps:

1. Check for operand stack overflow. This is done by comparing the input data flow item operand stack size plus the current basic block maximum operand stack increment with the bytecode maximum operand stack size.

2. Check for operand stack underflow. This is done by comparing the input data flow item operand stack size minus the current basic block maximum operand stack decrement against zero.

3. For each ret address in the input data flow item do:

    (a) Replace the set of frame indices of each pair by its union with current basic block written frame indices.

4. For each pseudo code instruction do:

   (a) Modify the input data flow item operand stack and local frame according to the semantics of the pseudo code instruction.

   (b) Check if the available types in the input data flow item matches the pseudo code instruction operands required types.

   (c) If the pseudo code instruction requires a subtype test then a new verification constraint is added.

   (d) If the pseudo code instruction requires `this` initialization (CHECKINIT), check if the flow data `this` initialization flag is set.

   (e) Fails if the pseudo code instruction is CHECKFALL because the execution reaches the last basic block of the bytecode which falls through.

5. Replace the output data flow item for the current basic block by the current data flow item.

6. For each control edge leaving current basic block do:

   (a) If the control edge is a normal edge and the last pseudo opcode was a UPOPINIT, then broadcast in the frame the initialization of associated uninitialized reference.

   (b) If the control edge is a normal edge, merge and compare the current input data with the successor input data. If there is a change, override the input data and insert the successor in the working list.

   (c) If the control edge is an exception edge, clear the operand stack and push the associated reference type. Then, merge and compare the current input data with the successor input data. If there is a change, override the input data and insert the successor in the working list.

7. If the last bytecode of the basic block is a return from subroutine (`ret`), then for each ret address pair do:

   (a) Replace the local frame slots whose indices are not present in the associated frame indices set by its value in the output data flow item of the basic block preceding the target basic block.

   (b) If any of the frame slots replaced contains a ret address data flow item, replace the frame indices set by its union with the frame indices set associated to current ret address.

   (c) Merge and compare the current input data with the target basic block input data. If there is a change override the input data and insert the target basic block in the working list.

The procedure for merging data flow items is the following:

1. Check if the operand stack size is the same.

2. For each operand stack and local frame slot do:

   (a) If both types match and are integer, float, long first word, long second word, double first word, double second word or null, keep the type.

   (b) If one type is a reference and the other is null, keep the reference type.

   (c) If both types are references, the result type will be a reference type with a class name set obtained from the union of both classes name sets.

   (d) If both types are uninitialized reference and their offset is the same, keep the type.

   (e) If both types are ret address, the result type will be a ret address where the set of pairs is the union of both set of pairs. The union is done over the frame indices set for pairs with same ret address offset, so that at most one pair is associated to the same offset.

   (f) Otherwise, if the slot is a local frame slot, mark the slot as being invalid. If the slot is a operand stack slot, fail.

3. Apply an *and* operation using both `this` parameter initialization flags.

Failures during structural constraint check should throw an instance of `VerifyError`.

The verification procedure presented above allows extended semantics for subroutines, while still keeping bytecode secure. In our opinion, the subroutine semantics allowed by the JVM specification[LY99, §4.9.6] is tailored to the implementation of the verification algorithm provided by their authors. We believe that this is wrong since the implementation should be tailored to the semantics, and not the opposite. So we have generalized the semantics of subroutines, and provided an alternate verification procedure. In our generalized semantics, subroutines are allowed to recurse, no ret addresses are invalidated by any subroutine return (including itself), and subroutines may share or have more than one returning sites (`ret` bytecode). Those restrictions to the subroutine semantics where clearly imposed by limitations of the standard verification procedure, and not by real security threats.

5.5   Verification Example

The following method is used to illustrate the verification procedure. It is a constructor that invokes the superclass constructor, catching exceptions and then do some "spaghetti" subroutine use. The bytecode was hand written and does not pass the standard verification procedure. However, the code demonstrates that subroutine semantics can be extended without damage. Specially, it has out of order return from subroutines and a recursive subroutine call.

```
     .method public <init>()V
         .limit stack 2
         .limit locals 5
         aload_0
@1:  invokespecial java/lang/Object/<init>()V
@4:  iconst_0
```

```
        istore 4
        jsr @15
        iconst_1
        istore 4
        ret 2
@15:  astore_1
        jsr @28
        getstatic java/lang/System/out Ljava/io/PrintStream;
        iload 4
        invokevirtual java/io/PrintStream/println(I)V
        return
@28:  astore_2
        jsr @34
        ret 1
@34:  astore_3
        aload_0
        ifnonnull @45
        fconst_0
        fstore 4
        jsr @34
@45:  ret 3
@47:  pop
        new java/lang/RuntimeException
        dup
        invokespecial java/lang/RuntimeException/<init>()V
        athrow
        .catch java/lang/Exception from @1 to @4 using @47
  .end method
```

The first step of the verification procedure is to construct the control flow graph and the information required to do the data flow analysis in a basic block basis. That can be seen in Figure 5.1.

Once the control flow graph and basic block data flow information has been constructed, we start doing the data flow analysis by iterating over a basic block work list.

Iteration 1 Processing BB[0], the input data flow item is:

> `this` initialized: false
> local frame: U[-1]XXXX
> operand stack: *empty*

BB[4] is scheduled with input data flow item:

> `this` initialized: true
> local frame: LExample;XXXX
> operand stack: *empty*

BB[47] is scheduled with input data flow item:
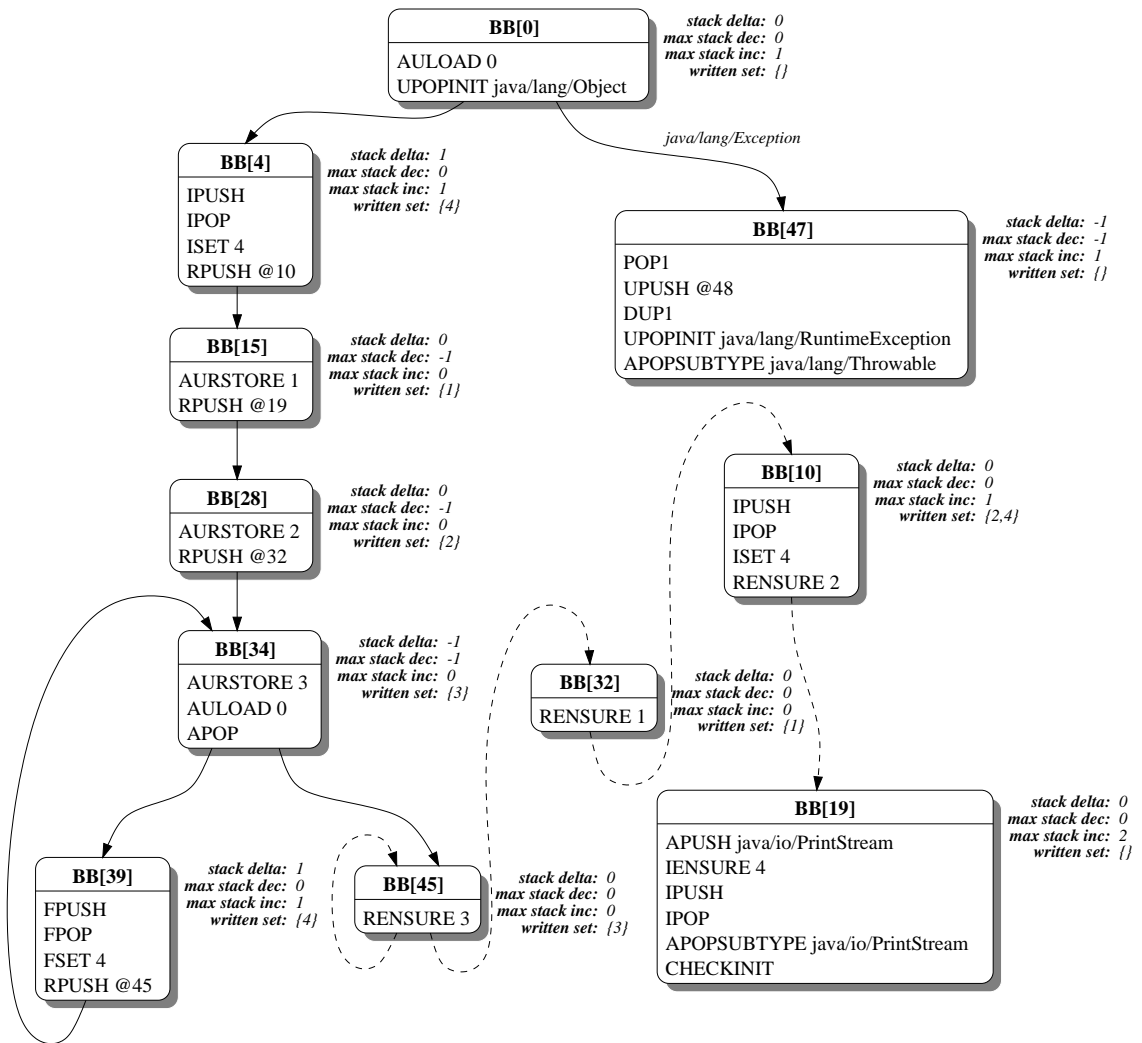
Figure 5.1: Verifier example control flow graph.

    this initialized: false
local frame: U[-1]XXXX
operand stack: Ljava/lang/Exception;

Iteration 2 Processing BB[4], the input data flow item is:

    this initialized: true
local frame: LExample;XXXX
operand stack: *empty*

    BB[15] is scheduled with input data flow item:

    this initialized: true
local frame: LExample;XXXI
operand stack: R{(@10,{})}

Iteration 3 Processing BB[15], the input data flow item is:

    this initialized: true
local frame: LExample;XXXI
operand stack: R{(@10,{})}

    BB[28] is scheduled with input data flow item:

    this initialized: true
local frame: LExample;R{(@10,{1})}XXI
operand stack: R{(@19,{})}

Iteration 4 Processing BB[28], the input data flow item is:

    this initialized: true
local frame: LExample;R{(@10,{1})}XXI
operand stack: R{(@19,{})}

    BB[34] is scheduled with input data flow item:

    this initialized: true
local frame: LExample;R{(@10,{1,2})}R{(@19,{2})}XI
operand stack: R{(@32,{})}

Iteration 5 Processing BB[34], the input data flow item is:

this initialized: true
local frame: LExample;R{(@10,{1,2})}R{(@19,{2})}XI
operand stack: R{(@32,{})}

BB[39] is scheduled with input data flow item:

this initialized: true
local frame: LExample;R{(@10,{1,2,3})}R{(@19,{2,3})}R{(@32,{3})}I
operand stack: *empty*

BB[45] is scheduled with input data flow item:

this initialized: true
local frame: LExample;R{(@10,{1,2,3})}R{(@19,{2,3})}R{(@32,{3})}I
operand stack: *empty*

Iteration 6 Processing BB[39], the input data flow item is:

this initialized: true
local frame: LExample;R{(@10,{1,2,3})}R{(@19,{2,3})}R{(@32,{3})}I
operand stack: *empty*

BB[34] is scheduled with input data flow item:

this initialized: true
local frame: LExample;R{(@10,{1,2,3,4})}R{(@19,{2,3,4})}XX
operand stack: R{(@32,{}),(@45,{})}

Iteration 7 Processing BB[34], the input data flow item is:

this initialized: true
local frame: LExample;R{(@10,{1,2,3,4})}R{(@19,{2,3,4})}XX
operand stack: R{(@32,{}),(@45,{})}

BB[39] is scheduled with input data flow item:

this initialized: true
local frame:
LExample;R{(@10,{1,2,3,4})}R{(@19,{2,3,4})}R{(@32,{3}),(@45,{3})}X
operand stack: *empty*

BB[45] is scheduled with input data flow item:

    this initialized: true
    local frame:
    LExample;R{(@10,{1,2,3,4})}R{(@19,{2,3,4})}R{(@32,{3}),(@45,{3})}X
    operand stack: *empty*

Iteration 8 Processing BB[39], the input data flow item is:

    this initialized: true
    local frame:
    LExample;R{(@10,{1,2,3,4})}R{(@19,{2,3,4})}R{(@32,{3}),(@45,{3})}X
    operand stack: *empty*

Iteration 9 Processing BB[45], the input data flow item is:

    this initialized: true
    local frame:
    LExample;R{(@10,{1,2,3,4})}R{(@19,{2,3,4})}R{(@32,{3}),(@45,{3})}X
    operand stack: *empty*

    BB[32] is scheduled with input data flow item:

    this initialized: true
    local frame: LExample;R{(@10,{1,2,3})}R{(@19,{2,3})}R{(@32,{3}),(@45,{3})}I
    operand stack: *empty*

Iteration 10 Processing BB[32], the input data flow item is:

    this initialized: true
    local frame: LExample;R{(@10,{1,2,3})}R{(@19,{2,3})}R{(@32,{3}),(@45,{3})}I
    operand stack: *empty*

    BB[10] is scheduled with input data flow item:

    this initialized: true
    local frame:
    LExample;R{(@10,{1,2,3})}R{(@19,{1,2,3})}R{(@32,{1,3}),(@45,{1,3})}I
    operand stack: *empty*

Iteration 11 Processing BB[10], the input data flow item is:

    this initialized: true
    local frame:
    LExample;R{(@10,{1,2,3})}R{(@19,{1,2,3})}R{(@32,{1,3}),(@45,{1,3})}I
    operand stack: *empty*

    BB[19] is scheduled with input data flow item:

    this initialized: true
    local frame: LExam-
    ple;R{(@10,{1,2,3,4})}R{(@19,{1,2,3,4})}R{(@32,{1,2,3,4}),(@45,{1,2,3,4})}I
    operand stack: *empty*

Iteration 12 Processing BB[19], the input data flow item is:

    this initialized: true
    local frame: LExam-
    ple;R{(@10,{1,2,3,4})}R{(@19,{1,2,3,4})}R{(@32,{1,2,3,4}),(@45,{1,2,3,4})}I
    operand stack: *empty*

Iteration 13 Processing BB[47], the input data flow item is:

    this initialized: false
    local frame: U[-1]XXXX
    operand stack: Ljava/lang/Exception;

5.6   What is Required to Go Further

As said before, the verification procedure gathers information that simplifies the bytecode conversion to the intermediate representation (Chapter 6). This information comprises, for each offset in the bytecode array, of:

1. A *valid* flag indicating if the offset is the start offset of an instruction.

2. A *target* flag indicating if the offset is the target of a branch or the entry point of an exception handler.

3. A *leader* flag indicating if the offset is the first instruction of its enclosing basic block.

4. A *trailer* flag indicating if the offset is the last instruction of its enclosing basic block.

5. A *unreachable* flag indicating if the instruction at this offset is not reachable, i.e. will never be executed.

BYTECODE	FIRST OPERAND	SECOND OPERAND
baload	boolean or byte array	
astore	reference or ret address	
astore_0	reference or ret address	
astore_1	reference or ret address	
astore_2	reference or ret address	
astore_3	reference or ret address	
bastore	boolean or byte array	
pop	integer, float, reference or ret address	
pop2	pair32, long or double	
dup	integer, float, reference or ret address	
dup_x1	integer, float, reference or ret address	integer, float, reference or ret address
dup_x2	integer, float, reference or ret address	pair32, long or double
dup2	pair32, long or double	
dup2_x1	pair32, long or double	integer, float, reference or ret address
dup2_x2	pair32, long or double	pair32, long or double
swap	integer, float, reference or ret address	integer, float, reference or ret address

Table 5.2: Untyped bytecodes and their possible operands.

6. A *stack size* integer indicating the bytecode operand stack size prior to executing of current instruction.

7. The type of the operands of untyped bytecodes, see Table 5.2.

   Using this attributes computed during verification the conversion procedure can occur without having to do any extra analysis. Table 5.2 displays the bytecodes that are untyped, i.e. are allowed to handle multiple types, and the possible types of their operands. In the table, pair32 represents all pair combinations of types: integer, float, reference and ret address.

Chapter 6

Bytecode Conversion

This chapter addresses issues regarding *bytecode to intermediate representation conversion*[1]. The conversion takes place after bytecode verification. The conversion algorithm can thus be simplified to rely on the checks already performed during verification. Extra information gathered by the verifier is also used to simplify the identification of basic blocks and the processing of untyped bytecodes (e.g. `dup`).

During the conversion, operations implicit to some bytecodes are made explicit. That can be observed specially for bytecodes that may throw *runtime exceptions*. The code for checking an exceptional situation — and throwing the exception — is placed right before the code that implements the bytecode. The same happens when dealing with bytecodes that can trigger class initialization. Breaking the bytecode into smaller and simpler operations increases the chances of removing redundant code by the intermediate representation optimizer. Care was taken not to discard semantic information when designing the intermediate language and its converter.

The approach used for exception windows and handlers deserves special attention. After conversion, exception windows are eliminated. Instead, the control information associated to exception handling is stored within each operation giving more freedom for rearranging the code. Exception handlers are expanded to explicitly check exception subtyping. The impact of this approach over asynchronous exceptions is discussed.

We describe the solution we gave for implementing subroutines (`jsr`/`ret`). Subroutines impose a lot of difficulties not only to the verifier, but also to the bytecode converter. The main problem of subroutines deals with liveness information required to do accurate *garbage collection*[ADM98]. It is a well-known problem related to the Java Virtual Machine. The solution here described is not only simple and elegant, but also effective. It does not require code duplication.

This chapter does not treat optimization. Although some of the examples herein were hand-tuned for the sake of clarity, optimization is not central to this chapter. However, we do expose the optimizations that could be done during conversion but were left out based on the compiler motto:

---

[1]The term *translation* would be a natural choice. However, the term *conversion* has been adopted to avoid confusion with the transformation from intermediate representation to machine language that occurs during the TRANSLATE phase.

> *Make each compilation stage as simple and clear as possible.*
> *Achieve a reliable and effective code by grouping them together.*

6.1   Intermediate Representation Presentation

The *intermediate representation* (IR) was crafted to be a fine-grained optimization-aware representation of Java programs. It is more flexive than bytecode, and also machine independent. This section gives a shallow presentation of the IR features. For a complete reference check Appendix A.

The IR is typed, it works with five types: integers (32-bit signed), long integers (64-bit signed), floats, doubles and references. For each type there is a set of virtually infinite registers. Table 6.1 describes valid register indices for each type.

TYPE	VALID INDICES
integer	$\{n \mid n\%5 = 0\}$
long integer	$\{n \mid n\%5 = 1\}$
float	$\{n \mid n\%5 = 2\}$
double	$\{n \mid n\%5 = 3\}$
reference	$\{n \mid n\%5 = 4\}$

Table 6.1: Valid register indices for each IR type.

Each method, when translated to the IR, consists of a sequence of *IR statements* forming an *IR program*. Each IR program has its own set of registers which is not shared with others. Communication between IR programs is done through heap memory and parameters passed on calls.

An *IR statement* is a tree of *IR opcodes* where the root is an untyped IR opcode. It may be control related or not, may define registers, write memory, call other IR programs, etc.

The *IR opcodes* are the building blocks of the intermediate language. Each IR opcode defines an operation which may have arguments (provided by other IR opcodes), a result and attributes. There are IR opcodes for doing arithmetics, reading/writing memory, using/defining registers, converting between types, allocating memory, synchronization, receiving/passing parameters, calling methods, among others.

Next section gives a broad idea of the intermediate language by showing various examples of conversion from bytecode.

6.2   Conversion Examples

The examples provided by this section were based on the examples used to explain source to bytecode compilation in [LY99, §7]. Those examples were borrowed not only because they are a good way to introduce the matter, but also because they cover most of the constructs we would like to show.

The translation from Java source code to bytecode is assumed to be known by the reader. Readers not familiar with Java source code to bytecode translation should check [LY99, §6 and §7].

6.2.1   Constants, Local Variables, and Control Constructs

We start looking at a very simple example, the `spin` method. It is a bounded loop with empty body.

```
void spin() {
  for (int i = 0; i < 100; i++)
    ;
}
```

The translation from Java source to bytecode is given.

```
.method spin()V
        .limit stack 2
        .limit locals 2
        iconst_0
        istore_1
        goto @8
@5:   iinc 1 1
@8:   iload_1
        bipush 100
        if_icmplt @5
        return
.end method
```

The conversion from bytecode to the intermediate language of method `spin` is the following.

```
1. areceive(%4,#Example)
2. idefine(%0,iconst($0))
3. jump(@8)
4. label(@5)
5. idefine(%0,iadd(iuse(%0),iconst($1)))
6. label(@8)
7. ijump(LT,iuse(%0),iconst($100),@5)
8. vreturn()
```

The first statement defines register `%4` with the `this` parameter, note that `spin` is not a static method. The type of the `this` parameter is the class where `spin` was declared, `Example`. Next variable `i`, binded to register `%0`, is initialized with value `0` in statement 2. Statement 3 transfers unconditionally the execution to statement 6. Statements 4 and 6 simply defines labels used as target of control statements. The loop body is basically two statements: 5 and 7. Statement 7 checks if register `%0` is less than constant `100` keeping the execution in

the loop. Statement 5 increments variable *i* by adding to register %0 the constant *1*. Finally, when leaving the loop, the *void* method returns in statement 8.

Some points can be highlighted when looking at the IR program for the *spin* method. First, stack based operations are converted to tree based expressions. Tree based expressions are easier to handle when rearranging code. Second, similar operations are represented equally. The bytecodes *iconst_0* and *bipush 100* do something similar, push an integer constant. They were unified in the IR opcode iconst. Finally, resources that were limited in the bytecode are not yet limited in the IR program.

Next example is the *double* version of the *spin* method.

```
void dspin() {
  for (double i = 0.0; i < 100.0; i++)
    ;
}
```

The translation of the *dspin* method from Java source code to bytecode is given below.

```
.method dspin()V
      .limit stack 4
      .limit locals 3
      dconst_0
      dstore_1
      goto @9
@5:   dload_1
      dconst_1
      dadd
      dstore_1
@9:   dload_1
      ldc2_w 100.0
      dcmpg
      iflt @5
      return
.end method
```

The conversion from Java bytecodes to IR is similar to that of the *spin* method. There are a few differences though. Variable *i* is binded to double register %3. IR operations are typed so the name of operations with doubles start with letter *d* instead of letter *i*.

```
1. areceive(%4,#Example)
2. ddefine(%3,dconst($0.0))
3. jump(@9)
4. label(@5)
5. ddefine(%3,dstrict(dadd(duse(%3),dconst($1.0))))
6. label(@9)
7. ijump(LT,dcmpg(duse(%3),dconst($100.0)),iconst($0),@5)
8. vreturn()
```

The greatest difference is the `dstrict` opcode. This opcode does *value set conversion*[LY99, §2.6.6]. Value set conversion remaps values with extended exponents — as allowed in the semantics of floats and doubles — to the standard encoding. This conversion may lead to underflow or overflow and is required at some points in the bytecode. In method *dspin*, extended exponents are allowed in intermediate stack values, but not in frame slots. So it is necessary to generate a `dstrict` opcode before writing back register `%3` in statement 5.

The next example is the `doubleLocals` method which receives two doubles as parameters and returns their sum.

```
double doubleLocals(double d1, double d2) {
  return d1+d2;
}
```

The bytecode generated for the `doubleLocals` is shown below. The main reason of this example is showing that double variables use two slots in the frame.

```
.method doubleLocals(DD)D
    .limit stack 4
    .limit locals 5
    dload_1
    dload_3
    dadd
    dreturn
.end method
```

The IR program for `doubleLocals` obtained from bytecode is very simple. It consists of four statements. The first three are parameter receiving statements. The fourth does all the job, sums the values passed as parameters and returns from the *double* method.

```
1. areceive(%4,#Example)
2. dreceive(%3)
3. dreceive(%8)
4. dreturn(dadd(duse(%3),duse(%8)))
```

The following example is the `sspin` method. It has exactly the same functionality as the `spin` method except that the loop counter was declared as short integer.

```
void sspin() {
  for (short i = 0; i < 100; i++)
    ;
}
```

The translation to bytecode of the `sspin` method is shown below. The `i2s` bytecode is used to keep variable `i` value within the short integer range.

```
.method sspin()V
      .limit stack 2
      .limit locals 2
      iconst_0
      istore_1
      goto @10
@5:  iload_1
      iconst_1
      iadd
      i2s
      istore_1
@10: iload_1
      bipush 100
      if_icmplt @5
      return
.end method
```

The conversion from Java bytecode to IR is similar to that of the `spin` method. The only difference appears in statement 5 where a `i2s` opcode is used.

```
1. areceive(%4,#Example)
2. idefine(%0,iconst($0))
3. jump(@10)
4. label(@5)
5. idefine(%0,i2s(iadd(iuse(%0),iconst($1))))
6. label(@10)
7. ijump(LT,iuse(%0),iconst($100),@5)
8. vreturn()
```

6.2.2   Arithmetic

The `align2grain` method is used to show how arithmetic expressions are handled in the IR.

```
int align2grain(int i, int grain) {
  return (i+grain-1)&~(grain-1);
}
```

The translation of method `align2grain` from Java source code to bytecode is straightforward. Note that the logical negation operation (`~` operator) is not supported on bytecode. It is implemented by applying an *exclusive or* operation over the operand and the integer constant `-1`.

```
.method align2grain(II)I
      .limit stack 3
      .limit locals 3
      iload_1
      iload_2
```

```
        iadd
        iconst_1
        isub
        iload_2
        iconst_1
        isub
        iconst_m1
        ixor
        iand
        ireturn
    .end method
```

The conversion of bytecode to IR is also straightforward. The expression, as appeared in the Java source, is extracted from the stack based bytecode and rewritten as an expression tree in statement 4.

```
1. areceive(%4,#Example)
2. ireceive(%0)
3. ireceive(%5)
4. ireturn(iand(isub(iadd(iuse(%0),iuse(%5)),iconst($1)),ixor(isub(
   iuse(%5),iconst($1)),iconst($-1))))
```

6.2.3   More Control Examples

The `whileInt` method implements the same functionality as the `spin` method using the *while* construct.

```
void whileInt() {
  int i = 0;
  while (i < 100)
    i++;
}
```

The bytecode generated from the `whileInt` method is actually the same bytecode as the one generated from the `spin` method. It is provided below.

```
.method whileInt()V
      .limit stack 2
      .limit locals 2
      iconst_0
      istore_1
      goto @8
@5:   iinc 1 1
@8:   iload_1
      bipush 100
      if_icmplt @5
      return
.end method
```

The conversion of the `whileInt` method from bytecode to IR also yields the same result.

```
1. areceive(%4,#Example)
2. idefine(%0,iconst($0))
3. jump(@8)
4. label(@5)
5. idefine(%0,iadd(iuse(%0),iconst($1)))
6. label(@8)
7. ijump(LT,iuse(%0),iconst($100),@5)
8. vreturn()
```

Next example is the `whileDouble` method that is a slightly modified version of the `dspin` method using the *while* construct.

```
void whileDouble() {
  double i = 0.0;
  while (i < 100.1)
    i++;
}
```

Its translation to bytecode is provided below.

```
.method whileDouble()V
      .limit stack 4
      .limit locals 3
      dconst_0
      dstore_1
      goto @9
@5:   dload_1
      dconst_1
      dadd
      dstore_1
@9:   dload_1
      ldc2_w 100.1
      dcmpg
      iflt @5
      return
.end method
```

As expected, the conversion of the `whileDouble` method provides a result similar to the one obtained from the conversion of the `dspin` method.

```
1. areceive(%4,#Example)
2. ddefine(%3,dconst($0.0))
3. jump(@9)
4. label(@5)
5. ddefine(%3,dstrict(dadd(duse(%3),dconst($1.0))))
6. label(@9)
7. ijump(LT,dcmpg(duse(%3),dconst($100.1)),iconst($0),@5)
8. vreturn()
```

The following example is the *lessThan100* method. This method receives a double parameter and returns *1* if it is less than *100.0*, or *−1* otherwise.

```
int lessThan100(double d) {
  if (d < 100.0)
    return 1;
  else
    return -1;
}
```

The translation of method *lessThan100* to bytecode is presented below. Note that the *dcmpg*/*ifge* bytecodes are used; if parameter *d* is *NaN*, the *lessThan100* method will return *−1*.

```
.method lessThan100(D)I
      .limit stack 4
      .limit locals 3
      dload_1
      ldc2_w 100.0
      dcmpg
      ifge @10
      iconst_1
      ireturn
@10: iconst_m1
      ireturn
.end method
```

The conversion of method *lessThan100* from bytecode to IR is straightforward.

```
1. areceive(%4,#Example)
2. dreceive(%3)
3. ijump(GE,dcmpg(duse(%3),dconst($100.0)),iconst($0),@10)
4. ireturn(iconst($1))
5. label(@10)
6. ireturn(iconst($-1))
```

The next example is method *greaterThan100*. It is similar to the *lessThan100* method.

```
int greaterThan100(double d) {
  if (d > 100.0)
    return 1;
  else
    return -1;
}
```

The difference in the translation to bytecode from the *lessThan100* method and the *greaterThan100* method is the *if* test which uses *dcmpl*/*ifle* instead of *dcmpg*/*ifge* bytecodes.

```
.method greaterThan100(D)I
      .limit stack 4
      .limit locals 3
      dload_1
      ldc2_w 100.0
      dcmpl
      ifle @10
      iconst_1
      ireturn
@10: iconst_m1
      ireturn
.end method
```

The conversion of method `greaterThan100` from bytecode to IR is also straightforward.

```
1. areceive(%4,#Example)
2. dreceive(%3)
3. ijump(LE,dcmpl(duse(%3),dconst($100.0)),iconst($0),@10)
4. ireturn(iconst($1))
5. label(@10)
6. ireturn(iconst($-1))
```

6.2.4   Receiving Arguments

Two examples are provided to show how arguments are received in the IR. It is not any different from the code already shown in the examples presented so far.

The first example is the `addTwo` method. It receives two integer arguments and returns its sum. Note that the `addTwo` method was not declared as a static method.

```
int addTwo(int i, int j) {
   return i+j;
}
```

The translation of method `addTwo` from Java source code to bytecode is presented below.

```
.method addTwo(II)I
      .limit stack 2
      .limit locals 3
      iload_1
      iload_2
      iadd
      ireturn
.end method
```

The conversion of method `addTwo` is shown below. Statements 1 to 3 are *parameter receiving* statements. In statement 1, the `this` parameter is stored in reference register %4

with associated type `#Example`. In statements 2 and 3, parameters *i* and *j* are stored into integer registers `%0` and `%5`, respectively. Finally, statement 4 adds the two integers and returns the result.

```
1. areceive(%4,#Example)
2. ireceive(%0)
3. ireceive(%5)
4. ireturn(iadd(iuse(%0),iuse(%5)))
```

The second example is the *addTwoStatic* method. It is similar to the *addTwo* method except that is was declared as a static method.

```
static int addTwoStatic(int i, int j) {
  return i+j;
}
```

The translation of the *addTwoStatic* method to bytecode is shown below.

.method static addTwoStatic(II)I
      .limit stack 2
      .limit locals 2
      iload_0
      iload_1
      iadd
      ireturn
.end method

The IR program obtained from the conversion of method *addTwoStatic* is exactly the same IR program obtained from the *addTwo* method, except that statement 1 was removed. This occurred because the *addTwo* method is static and does not receive the *this* implicit parameter.

```
1. ireceive(%0)
2. ireceive(%5)
3. ireturn(iadd(iuse(%0),iuse(%5)))
```

6.2.5   Invoking Methods

The *add12and13* method is used to show how method invocation is handled in the IR. It simply invokes the *addTwo* method passing constants *12* and *13* as parameters.

```
int add12and13() {
  return addTwo(12, 13);
}
```

The bytecode result of the translation of method *add12and13* is shown below. It pushes onto the local operand stack the *this* reference and integer constants *12* and *13* respectively. In the sequence, the *addTwo* method is invoked and the result of its execution is kept on the top of the operand stack. Finally, it is used as the return value of method *add12and13*.

```
.method add12and13()I
      .limit stack 3
      .limit locals 1
      aload_0
      bipush 12
      bipush 13
      invokevirtual Example/addTwo(II)I
      ireturn
.end method
```

The conversion of method *add12and13* from bytecode to IR is the following.

```
1. areceive(%4,#Example)
2. ipass(iconst($13))
3. ipass(iconst($12))
4. apass(ause(%4))
5. call(mlookup(getclass(ause(%4)),[10]),[$Example,12,61])
6. iresult(%0)
7. ireturn(iuse(%0))
```

The IR program for method *add12and13* requires some explanation. Statement 1 defines reference register %4 with the *this* parameter because *add12and13* is not a static method. Statements 2 to 4 are parameter passing statements to the subsequent call that will occur in statement 5. The parameters are passed from *right-to-left* — instead of *left-to-right* as adopted in the Java bytecode — therefore integer constants *12*, *13* and the implicit *this* reference are passed, respectively.

Statement 5 implements a method invocation (IR program call). It uses the IR program at dispatch index *10* of a class object to transfer execution (mlookup opcode). This is a *virtual* invocation, and thus the invoked method depends on the actual type of the object pointed by the *this* reference (its class is retrieved using the getclass opcode). The index *10* is the dispatch index of method *addTwo* assigned during the LOAD phase of class *Example*. Finally, the triple that can be seen in statement 5 is a *stack trace* information. If a stack trace is requested during the *addTwo* call, the 12th declared method of class *Example* (the *add12and13* method actually) and line *61* of its source file will be included in the stack trace[2].

In statement 6, the result of the invocation of method *addTwo* is assigned to integer register %0. In statement 7 integer register %0 is used as the return value.

The next example is the static version of method *add12and13*. The *addStatic12and13* method simply invokes static method *addTwoStatic* passing as parameters the integer constants *12* and *13*.

---

[2]The stack trace information is not solely a triple, but a triple list, which can be empty or have more than one triple. This enables inlining optimizations to occur without damaging stack traces, for instance.

```
int addStatic12and13() {
  return addTwoStatic(12, 13);
}
```

The translation from Java source to bytecode of method *addStatic12and13* is shown below.

.method addStatic12and13()I
        .limit stack 2
        .limit locals 1
        bipush 12
        bipush 13
        invokestatic Example/addTwoStatic(II)I
        ireturn
.end method

The IR program obtained from the conversion of method *addStatic12and13* is similar to the IR program obtained from method *add12and13*. There is a difference though. Since the method being called is static, the class used by the `mlookup` opcode is known at conversion time, and its reference is provided using the `aclass` opcode.

1. `areceive(%4,#Example)`
2. `ipass(iconst($13))`
3. `ipass(iconst($12))`
4. `call(mlookup(aclass($Example),[19]),[$Example,13,65])`
5. `iresult(%0)`
6. `ireturn(iuse(%0))`

The following example shows a bit more about method invocation. Two methods are examined: *getItNear* and *getItFar*. Both methods use non-virtual instance method invocation to delegate execution.

```
class Near {

  int it;

  public int getItNear() {
    return getIt();
  }

  private int getIt() {
    return it;
  }

}

class Far extends Near {

  int getItFar() {
```

```
        return super.getItNear();
    }

}
```

The following bytecode is the translation of method `getItNear`. Since it delegates execution to a private method, `invokespecial` is used.

> *.method public getItNear()I*
>   *.limit stack 1*
>   *.limit locals 1*
>   *aload_0*
>   *invokespecial Near/getIt()I*
>   *ireturn*
> *.end method*

The following bytecode is the translation of method `getItFar`. Since it delegates execution to a superclass method, `invokespecial` is used.

> *.method getItFar()I*
>   *.limit stack 1*
>   *.limit locals 1*
>   *aload_0*
>   *invokespecial Near/getItNear()I*
>   *ireturn*
> *.end method*

The conversion of method `getItNear` is similar to the method invocation examples shown so far. Note that the `invokespecial` method is known at conversion time, thus `aclass` is used in statement 3.

```
    1. areceive(%4,#Near)
    2. apass(ause(%4))
    3. call(mlookup(aclass($Near),[8]),[$Near,1,75])
    4. iresult(%0)
    5. ireturn(iuse(%0))
```

The conversion of method `getItFar` is similar to the method invocation examples shown so far. Note that the `invokespecial` method is known at conversion time, thus `aclass` is used in statement 3.

```
    1. areceive(%4,#Far)
    2. apass(ause(%4))
    3. call(mlookup(aclass($Near),[5]),[$Far,1,86])
    4. iresult(%0)
    5. ireturn(iuse(%0))
```

6.2.6   Working with Class Instances

The next example is method `create` that instantiate class `Object` and returns the newly created instance.

```
Object create() {
  return new Object();
}
```

The translation from Java source code to bytecode is shown below. The instantiation consists of allocating heap space for the new object (`new` bytecode) and invoking a constructor over it (`invokespecial` bytecode).

.method create()Ljava/lang/Object;
  .limit stack 2
  .limit locals 1
  new java/lang/Object
  dup
  invokespecial java/lang/Object/<init>()V
  areturn
.end method

The conversion of method `create` to IR is the following. The usage of the `newinstance` opcode is shown in statement 2. The `newinstance` opcode has as arguments the class to be instantiated; here `Object` provided by the `aclass` opcode. Like invocation opcodes, the `newinstance` opcode also provides stack trace information. The newly created instance becomes available in statement 3 where register `%4` is defined with a reference to it.

```
1. areceive(%4,#Example)
2. newinstance(aclass($java/lang/Object),[$Example,14,69])
3. aresult(%4,#java/lang/Object)
4. apass(ause(%4))
5. call(mlookup(aclass($java/lang/Object),[12]),[$Example,14,69]
   )
6. areturn(ause(%4))
```

Next we present two more methods: `example` and `silly`. The `example` method instantiates class `MyObj` and returns by calling the `silly` method. The `silly` method does a useless null reference test, on the received reference, and returns it anyway.

```
MyObj example() {
  MyObj o = new MyObj();
  return silly(o);
}

MyObj silly(MyObj o) {
  if (o != null)
    return o;
```

```
      else
        return o;
    }
```

The translation of method `example` from Java source code to bytecode is shown below.

.method example()LMyObj;
     .limit stack 2
     .limit locals 2
     new MyObj
     dup
     invokespecial MyObj/<init>()V
     astore_1
     aload_0
     aload_1
     invokevirtual Example/silly(LMyObj;)LMyObj;
     areturn
.end method

The translation of method `silly` from Java source code to bytecode is shown below.

.method silly(LMyObj;)LMyObj;
     .limit stack 1
     .limit locals 2
     aload_1
     ifnull @6
     aload_1
     areturn
@6: *aload_1*
     areturn
.end method

The conversion of method `example` from bytecode to IR is shown below. The usage of the `init` opcode is shown in statement 2.

Instantiating a class requires its initialization prior to allocating the new object. As shown above in method `create`, the initialization of the class being instantiated, `Object`, is redundant. In that case, the initialization is redundant because the class that declared method `create`, `Example`, executes only after its initialization and is a subclass of the class to be initialized. Since class initialization is only successful after superclass initialization the `init` operation can be suppressed.

```
1. areceive(%4,#Example)
2. init(aclass($MyObj),[$Example,15,73])
3. newinstance(aclass($MyObj),[$Example,15,73])
4. aresult(%9,#MyObj)
5. apass(ause(%9))
6. call(mlookup(aclass($MyObj),[6]),[$Example,15,73])
7. apass(ause(%9))
```

```
 8. apass(ause(%4))
 9. call(mlookup(getclass(ause(%4)),[19]),[$Example,15,74])
10. aresult(%4,#MyObj)
11. areturn(ause(%4))
```

The conversion of method `silly` from bytecode to IR is straightforward.

```
1. areceive(%4,#Example)
2. areceive(%9,#MyObj)
3. ajump(EQ,ause(%9),anull(),@6)
4. areturn(ause(%9))
5. label(@6)
6. areturn(ause(%9))
```

The next example shows how an integer instance field is written and read in methods `setIt` and `getIt`, respectively.

```
int i;

void setIt(int value) {
  i = value;
}

int getIt() {
  return i;
}
```

The following bytecode is obtained from the translation of method `setIt`.

```
.method setIt(I)V
      .limit stack 2
      .limit locals 2
      aload_0
      iload_1
      putfield Example/i I
      return
.end method
```

The following bytecode is obtained from the translation of method `getIt`.

```
.method getIt()I
      .limit stack 1
      .limit locals 1
      aload_0
      getfield Example/i I
      ireturn
.end method
```

The conversion of method `setIt` is shown below. The `istore` is used in statement 3. Its first argument is the reference to the object being written, the `this` reference read from register `%4`. Its second argument is the integer value to be written, the parameter value read from register `%0`. Two attributes make up the `istore` opcode: the field offset in the instance (dynamic) field area (using the encoding described in Section 4.4); and a volatile write flag.

```
1. areceive(%4,#Example)
2. ireceive(%0)
3. istore(ause(%4),dynamic(0,0),false,iuse(%0))
4. vreturn()
```

The conversion of method `getIt` is shown below. The `iload` opcode is used similarly as in the `istore` opcode in `setIt` IR program.

```
1. areceive(%4,#Example)
2. ireturn(iload(ause(%4),dynamic(0,0),false))
```

6.2.7   Arrays

The `createBuffer` method is an example of how to instantiate integer arrays and access its elements.

```
void createBuffer() {
  int buffer[];
  int bufsz = 100;
  int value = 12;
  buffer = new int[bufsz];
  buffer[10] = value;
  value = buffer[11];
}
```

The translation of method `createBuffer` from Java source code to bytecode is shown below.

```
.method createBuffer()V
    .limit stack 3
    .limit locals 4
    bipush 100
    istore_2
    bipush 12
    istore_3
    iload_2
    newarray int
    astore_1
    aload_1
    bipush 10
    iload_3
    iastore
    aload_1
```

```
        bipush 11
        iaload
        istore_3
        return
.end method
```

The following IR program was obtained from the conversion of method `createBuffer`. Special attention should be given to statements 4, 6 and 7. In statement 4, a new integer array is being instantiated using the `newarray` opcode. The arguments of the `newarray` opcode are the array class to be instantiated and the desired length of the new array. The new instance becomes available in statement 5 when register `%9` is defined with its reference. In statements 6 and 7, IR opcodes `iastore` and `iaload` are used to write and read array elements, respectively. Both opcodes receive as arguments the array to be accessed and the element index. In the case of the `iastore` opcode, an extra argument is the value to be written. The `iaload` opcode provides the value read from the array.

```
1. areceive(%4,#Example)
2. idefine(%0,iconst($100))
3. idefine(%5,iconst($12))
4. newarray(aclass($[I),iuse(%0),[$Example,19,98])
5. aresult(%9,#[I)
6. iastore(ause(%9),iconst($10),iuse(%5))
7. idefine(%5,iaload(ause(%9),iconst($11)))
8. vreturn()
```

The next example is method `createThreadArray`. The `createThreadArray` method instantiates a `Thread` array and initializes its first element with a new instance.

```
void createThreadArray() {
  Thread threads[];
  int count = 10;
  threads = new Thread[count];
  threads[0] = new Thread();
}
```

The translation of method `createThreadArray` from Java source code to bytecode is shown below.

```
.method createThreadArray()V
        .limit stack 4
        .limit locals 3
        bipush 10
        istore_2
        iload_2
        anewarray java/lang/Thread
        astore_1
        aload_1
        iconst_0
```

> *new java/lang/Thread*
> *dup*
> *invokespecial java/lang/Thread/<init>()V*
> *aastore*
> *return*
> *.end method*

The following IR program is obtained from the conversion of method `createThreadArray`. The `aastore` is the only IR opcode not presented before. It writes a reference read from register `%14` to index `$0` of array read from register `%9`.

```
 1. areceive(%4,#Example)
 2. idefine(%10,iconst($10))
 3. newarray(aclass($[Ljava/lang/Thread;),iuse(%10),[$Example,20,
    106])
 4. aresult(%9,#[Ljava/lang/Thread;)
 5. init(aclass($java/lang/Thread),[$Example,20,107])
 6. newinstance(aclass($java/lang/Thread),[$Example,20,107])
 7. aresult(%14,#java/lang/Thread)
 8. apass(ause(%14))
 9. call(mlookup(aclass($java/lang/Thread),[44]),[$Example,20,107]
    )
10. aastore(ause(%9),iconst($0),ause(%14))
11. vreturn()
```

The `create3DArray` method is an example of how multidimensional arrays are created in the IR. It instantiates two dimensions of a three dimensional integer array and returns the newly created instance.

```
int[][][] create3DArray() {
  int grid[][][];
  grid = new int[10][5][];
  return grid;
}
```

The translation of method `create3DArray` from Java source to bytecode is shown below. The multidimensional array is created using the `multianewarray` bytecode.

> *.method create3DArray()[[[I*
> *.limit stack 2*
> *.limit locals 2*
> *bipush 10*
> *iconst_5*
> *multianewarray [[[I 2*
> *astore_1*
> *aload_1*
> *areturn*
> *.end method*

The IR program for the `create3DArray` method is shown below. Since the IR does not support a multidimensional opcode, the instantiation is implemented by nesting loops and instantiating each array at its time. Statement 2 instantiates the first dimension array with length $10. Statements 4 to 10 is the loop that executes $10 times instantiating the second dimension arrays with length $5.

```
1. areceive(%4,#Example)
2. newarray(aclass($[[[I),iconst($10),[$Example,21,112])
3. aresult(%9,#[[[I)
4. idefine(%0,iconst($0))
5. label(@6)
6. newarray(aclass($[[I),iconst($5),[$Example,21,112])
7. aresult(%14,#[[I)
8. aastore(ause(%9),iuse(%0),ause(%14))
9. idefine(%0,iadd(iuse(%0),iconst($1)))
10. ijump(LT,iuse(%0),iconst($10),@6)
11. areturn(ause(%9))
```

6.2.8 Compiling Switches

This section presents two examples of compiling *switch* statements. The first is implemented using a `tableswitch` bytecode, and the second using a `lookupswitch` bytecode. As shown, both table driven control transfer bytecodes are mapped to a single IR opcode.

The first example is the `chooseNear` method shown below.

```
int chooseNear(int i) {
  switch (i) {
  case 0:  return 0;
  case 1:  return 1;
  case 2:  return 2;
  default:  return -1;
  }
}
```

The translation of method `chooseNear` from Java source code to bytecode is the following. The Java compiler detects that the *case* values are sequential and generates a `tableswitch` bytecode.

```
.method chooseNear(I)I
     .limit stack 1
     .limit locals 2
     iload_1
     tableswitch 0
          @28
          @30
          @32
          default: @34
@28: iconst_0
     ireturn
```

@30: iconst_1
         ireturn
@32: iconst_2
         ireturn
@34: iconst_m1
         ireturn
.end method

The conversion of method `chooseNear` to IR program is shown below. The `iswitch` is used to implement the table driven control transfer in statement 3. The `iswitch` opcode does not define how the control transfer is done, it only defines a map between integer values and labels. Also the `iswitch` does not define a label for default (not mapped) values; if a not mapped value is used as argument the `iswitch` opcode will simply falls through. The default transfer should then be done in the next statement (the `jump` opcode in statement 4 transfers execution to default label `@34`).

```
 1. areceive(%4,#Example)
 2. ireceive(%0)
 3. iswitch(iuse(%0),[$0,@28][$1,@30][$2,@32])
 4. jump(@34)
 5. label(@28)
 6. ireturn(iconst($0))
 7. label(@30)
 8. ireturn(iconst($1))
 9. label(@32)
10. ireturn(iconst($2))
11. label(@34)
12. ireturn(iconst($-1))
```

The second *switch* example is method `chooseFar` shown below.

```
int chooseFar(int i) {
  switch (i) {
  case -100:  return -1;
  case 0:  return 0;
  case 100:  return 1;
  default:  return -1;
  }
}
```

The translation of method `chooseFar` from Java source code to bytecode is below. Since switch key values are not sequential the Java compiler generates a `lookupswitch` bytecode.

.method chooseFar(I)I
     .limit stack 1
     .limit locals 2
     iload_1
     lookupswitch
         -100: @36

```
            0: @38
            100: @40
            default: @42
@36: iconst_m1
      ireturn
@38: iconst_0
      ireturn
@40: iconst_1
      ireturn
@42: iconst_m1
      ireturn
.end method
```

The conversion of method *chooseFar* is shown below. It is similar to the conversion of method *chooseNear*, also using the iswitch opcode.

```
 1. areceive(%4,#Example)
 2. ireceive(%0)
 3. iswitch(iuse(%0),[$-100,@36][$0,@38][$100,@40])
 4. jump(@42)
 5. label(@36)
 6. ireturn(iconst($-1))
 7. label(@38)
 8. ireturn(iconst($0))
 9. label(@40)
10. ireturn(iconst($1))
11. label(@42)
12. ireturn(iconst($-1))
```

6.2.9   Operations on the Operand Stack

The next example is method *nextIndex* shown below. It provides a read and increment index generation procedure.

```
private long index = 0;

public long nextIndex() {
  return index++;
}
```

The translation of method *nextIndex* from Java source code to bytecode is the following.

```
.method public nextIndex()J
      .limit stack 7
      .limit locals 1
      aload_0
      dup
      getfield Example/index J
```

dup2_x1
lconst_1
ladd
putfield Example/index J
lreturn
.end method

The IR program obtained from the conversion of method `nextIndex` is shown below.

```
1. areceive(%4,#Example)
2. ldefine(%1,lload(ause(%4),dynamic(0,0),false))
3. lstore(ause(%4),dynamic(0,0),false,ladd(luse(%1),lconst($1)))
4. lreturn(luse(%1))
```

6.2.10 Throwing and Handling Exceptions

This section presents four examples of throwing and handling exceptions. They provide an overview of the IR support for exception handling. The presentation is completed in the next section when we describe the conversion of *try/finally* constructs.

The first example is method `canBeZero`. It instantiates and throws an exception if the integer parameter is zero.

```
void cantBeZero(int i) throws TestExc {
  if (i == 0)
    throw new TestExc();
}
```

The translation of method `canBeZero` from Java source code to bytecode is shown below.

.method cantBeZero(I)V
.limit stack 2
.limit locals 2
iload_1
ifne @12
new TestExc
dup
invokespecial TestExc/<init>()V
athrow
@12: return
.end method

The conversion of method `canBeZero` from bytecode to IR is below. The `athrow` is used to throw the newly created exception into the caller method frame (statement 9).

```
1. areceive(%4,#Example)
2. ireceive(%0)
```

```
 3. ijump(NE,iuse(%0),iconst($0),@12)
 4. init(aclass($TestExc),[$Example,25,141])
 5. newinstance(aclass($TestExc),[$Example,25,141])
 6. aresult(%9,#TestExc)
 7. apass(ause(%9))
 8. call(mlookup(aclass($TestExc),[12]),[$Example,25,141])
 9. athrow(ause(%9))
10. label(@12)
11. vreturn()
```

The next example is method *catchOne*. It calls method *tryItOut* protected by an exception handler that calls method *handleExc* if a *TextExc* exception occur.

```
void catchOne() {
  try {
    tryItOut();
  } catch (TestExc e) {
    handleExc(e);
  }
}
```

The translation of method *catchOne* from Java source to bytecode is below. An exception window is defined from label *@0* inclusive to label *@4* exclusive with the handler at label *@7*.

```
.method catchOne()V
      .limit stack 2
      .limit locals 2
@0:   aload_0
      invokevirtual Example/tryItOut()V
@4:   goto @13
@7:   astore_1
      aload_0
      aload_1
      invokevirtual Example/handleExc(LTestExc;)V
@13:  return
      .catch TestExc from @0 to @4 using @7
.end method
```

The conversion of method *catchOne* from bytecode to IR is shown below. In statement 3, the `callx` opcode is used instead of a `call` opcode once exceptions thrown during the call are handled by the current IR program and not delegated to the caller IR program. The `callx` opcode has exactly the same semantics as the `call` opcode except that it defines a label as *exception catching entry point* (@15 is the exception catching entry point for the exception window that encloses the *tryItOut* call). Statements 11 to 14 implements explicit exception catching, subtype testing, handler delegation or rethrowing. Statement 12 defines register %9 with the reference of the exception thrown during the *tryItOut* call. Statement 13 tests if the catched exception is a subtype of class *TestExc*. If true the control is transfered to the

exception handler at label @7. Otherwise, the exception is rethrown in the frame of the caller method in statement 14.

```
 1. areceive(%4,#Example)
 2. apass(ause(%4))
 3. callx(mlookup(getclass(ause(%4)),[29]),[$Example,28,152],@15)
 4. jump(@13)
 5. label(@7)
 6. apass(ause(%9))
 7. apass(ause(%4))
 8. call(mlookup(getclass(ause(%4)),[30]),[$Example,28,154])
 9. label(@13)
10. vreturn()
11. label(@15)
12. acatch(%9)
13. ijump(NE,subtypeof(getclass(ause(%9)),aclass($TestExc)),iconst(
    $0),@7)
14. athrow(ause(%9))
```

The next example, method `catchTwo`, is a variant of the previous example that defines two exception handlers for method `tryItOut`.

```
void catchTwo() {
  try {
    tryItOut();
  } catch (TestExc1 e) {
    handleExc(e);
  } catch (TestExc2 e) {
    handleExc(e);
  }
}
```

The translation of method `catchTwo` from Java source code to bytecode is shown below. Two exception windows are defined for the same code segment (@0 inclusive to @4 exclusive), however the test for `TestExc1` exception occurrence is done before the test for `TestExc2` exception occurrence.

```
.method catchTwo()V
     .limit stack 2
     .limit locals 3
@0:  aload_0
     invokevirtual Example/tryItOut()V
@4:  goto @22
@7:  astore_1
     aload_0
     aload_1
     invokevirtual Example/handleExc(LTestExc;)V
     goto @22
@16: astore_2
     aload_0
```

> *aload_2*
> *invokevirtual Example/handleExc(LTestExc;)V*
> @*22: return*
> *.catch TestExc1 from @0 to @4 using @7*
> *.catch TestExc2 from @0 to @4 using @16*
> *.end method*

The conversion of method `catchTwo` is similar to the conversion of method `catchOne`. In statement 3, a `callx` is also used defining label @24 as exception catching entry point. The major difference is on the code that catches the exception in statements 16 through 22. Statements 18 and 19 implement the test for `TestExc1` subtyping. On success, they transfer the execution to its handler at label @7. On subtyping failure, the exception catching is delegated to the enclosing exception window, which tests for `TestExc2` subtyping in a similar manner (statements 20 to 22).

```
 1. areceive(%4,#Example)
 2. apass(ause(%4))
 3. callx(mlookup(getclass(ause(%4)),[29]),[$Example,29,160],@24)
 4. jump(@22)
 5. label(@7)
 6. apass(ause(%9))
 7. apass(ause(%4))
 8. call(mlookup(getclass(ause(%4)),[30]),[$Example,29,162])
 9. jump(@22)
10. label(@16)
11. apass(ause(%9))
12. apass(ause(%4))
13. call(mlookup(getclass(ause(%4)),[30]),[$Example,29,164])
14. label(@22)
15. vreturn()
16. label(@24)
17. acatch(%9)
18. ijump(EQ,subtypeof(getclass(ause(%9)),aclass($TestExc1)),iconst(
    $0),@25)
19. jump(@7)
20. label(@25)
21. ijump(NE,subtypeof(getclass(ause(%9)),aclass($TestExc2)),iconst(
    $0),@16)
22. athrow(ause(%9))
```

As probably noted by the reader, the exception catching scheme — implicit on bytecode — is made explicit in the IR program. Protected calls define their exception catching labels as the entry points associated to its first enclosing exception window. The exception catching code for each exception window tests for exception subtyping, transfering the control to its handler; otherwise it delegates the catching to its enclosing exception window or, if it is a top level exception window, throwing the exception in the caller method frame. This scheme only works if every exception window is fully enclosed by another (or not enclosed at all); sometimes it is necessary to rewrite exception windows to achieve that. The rewriting technique is discussed in Section 6.3.

The next exception handling example is the *nestedCatch* method shown below. It has similar semantics to the *catchTwo* method (it tests first for *TestExc1* subtyping and then for *TestExc2* subtyping), but written in a different syntax. The main difference is subtle, the handler for the *TestExc1* exception is also protected by the *TestExc2* exception window.

```
void nestedCatch() {
  try {
    try {
      tryItOut();
    } catch (TestExc1 e) {
      handleExc(e);
    }
  } catch (TestExc2 e) {
    handleExc(e);
  }
}
```

The translation of method *nestedCatch* from Java source code to bytecode is below. Note that it is clear that one exception window encloses another.

```
.method nestedCatch()V
      .limit stack 2
      .limit locals 2
@0:   aload_0
      invokevirtual Example/tryItOut()V
@4:   goto @13
@7:   astore_1
      aload_0
      aload_1
      invokevirtual Example/handleExc(LTestExc;)V
@13:  goto @22
@16:  astore_1
      aload_0
      aload_1
      invokevirtual Example/handleExc(LTestExc;)V
@22:  return
      .catch TestExc1 from @0 to @4 using @7
      .catch TestExc2 from @0 to @13 using @16
.end method
```

The conversion of method *nestedCatch* from bytecode to IR is shown below. In statement 3, the *tryItOut* method is invoked using a `callx` opcode with exception catching label @24. In statement 8, the *handlerExc* method is invoked from an exception handler also using a `callx` opcode with exception catching label @25. Statements 17 through 25 implements the exception catching code. Statements 17 and 18 catch exceptions for the innermost exception window. Statements 19 and 20 test for *TestExc1* subtyping, transfering control to the handler label @7 on success, or delegating exception catching to its enclosing

exception window in label @26. Statements 21 and 22 catch exception for the outermost exception window. Statements 23 to 25 test for *TestExc2* subtyping, transfering control to the handler label @16 on success, or throwing the exception into the caller method frame.

```
 1. areceive(%4,#Example)
 2. apass(ause(%4))
 3. callx(mlookup(getclass(ause(%4)),[29]),[$Example,30,171],@24)
 4. jump(@13)
 5. label(@7)
 6. apass(ause(%9))
 7. apass(ause(%4))
 8. callx(mlookup(getclass(ause(%4)),[30]),[$Example,30,173],@25)
 9. label(@13)
10. jump(@22)
11. label(@16)
12. apass(ause(%9))
13. apass(ause(%4))
14. call(mlookup(getclass(ause(%4)),[30]),[$Example,30,176])
15. label(@22)
16. vreturn()
17. label(@24)
18. acatch(%9)
19. ijump(EQ,subtypeof(getclass(ause(%9)),aclass($TestExc1)),iconst(
    $0),@26)
20. jump(@7)
21. label(@25)
22. acatch(%9)
23. label(@26)
24. ijump(NE,subtypeof(getclass(ause(%9)),aclass($TestExc2)),iconst(
    $0),@16)
25. athrow(ause(%9))
```

6.2.11   Compiling Finally

This section presents two examples of the *try/finally* construct. These examples demonstrate how subroutines are translated to the IR.

The first example is the *tryFinally* method shown below. It calls method *tryItOut* protected by a finally block that calls method *wrapItUp*.

```
void tryFinally() throws TestExc {
  try {
    tryItOut();
  } finally {
    wrapItUp();
  }
}
```

The translation of method *tryFinally* from Java source code to bytecode is shown below. The *jsr/ret* bytecodes are used to implement a shared subroutine that is executed in both normal and exceptional cases.

```
.method tryFinally()V
      .limit stack 1
      .limit locals 3
@0:   aload_0
      invokevirtual Example/tryItOut()V
      jsr @16
      goto @23
@10: astore_1
      jsr @16
      aload_1
      athrow
@16: astore_2
      aload_0
      invokevirtual Example/wrapItUp()V
      ret 2
@23: return
      .finally from @0 to @10 using @10
.end method
```

The conversion of method `tryFinally` from bytecode to IR is shown below. The exception catching and handling implementation for finally exception windows is exactly the same as the implementation of the exception windows presented so far, except that there is no subtyping test and the handler is always executed (that can be seen in statements 22 to 24).

The conversion of the subroutine requires special attention. There is no construct similar to subroutines in the IR, however subroutines are implemented using available opcodes. The `jsr` bytecode is converted to a call-site identification integer assignment, followed by an unconditional branch. That can be seen in statements 5 to 6 and 10 to 11. The `ret` bytecode is converted to a `iswitch` opcode that maps each call-site identification integer to the label after each `jsr`, followed by an unreachable infinite loop. Statements 17 to 19 show that.

```
 1. areceive(%4,#Example)
 2. adefine(%9,anull())
 3. apass(ause(%4))
 4. callx(mlookup(getclass(ause(%4)),[29]),[$Example,32,182],@25)
 5. idefine(%0,iconst($0))
 6. jump(@16)
 7. label(@7)
 8. jump(@23)
 9. label(@10)
10. idefine(%0,iconst($1))
11. jump(@16)
12. label(@14)
13. athrow(ause(%9))
14. label(@16)
15. apass(ause(%4))
16. call(mlookup(getclass(ause(%4)),[31]),[$Example,32,184])
17. iswitch(iuse(%0),[$0,@7][$1,@14])
18. label(@26)
19. jump(@26)
20. label(@23)
```

```
21. vreturn()
22. label(@25)
23. acatch(%9)
24. jump(@10)
```

By doing so, we transfer the semantics of the subroutine from a powerful and complex control structure to a simple data driven control structure, with the advantage of no code duplication. However, as a result of the semantic translation from control to data we face a liveness problem. The reference register %9 is now live during the *finally* code execution (statements 14 to 16) because it is used by reachable statement 13. But %9 may not be initialized at that time, since it is only initialized through the exceptional path (statement 23). Therefore we need to generate a reference nullifying statement for each uninitialized live reference register at the top of the IR program (statement 2 in this example). This guarantees that all live reference registers always hold legal values, avoiding problems with garbage collection. The detailed description of this subroutine conversion procedure is given in Section 6.4.

The second example is the method `tryCatchFinally` that mixes the exception handling and finally constructs.

```
void tryCatchFinally() {
  try {
    tryItOut();
  } catch (TestExc e) {
    handleExc(e);
  } finally {
    wrapItUp();
  }
}
```

The translation of the method `tryCatchFinally` from Java source code to bytecode is the following.

```
.method tryCatchFinally()V
      .limit stack 2
      .limit locals 4
@0:   aload_0
      invokevirtual Example/tryItOut()V
@4:   jsr @28
      goto @35
@10:  astore_1
      aload_0
      aload_1
      invokevirtual Example/handleExc(LTestExc;)V
      jsr @28
      goto @35
@22:  astore_2
      jsr @28
      aload_2
      athrow
@28:  astore_3
```

```
        aload_0
        invokevirtual Example/wrapItUp()V
        ret 3
@35: return
        .catch TestExc from @0 to @4 using @10
        .finally from @0 to @22 using @22
.end method
```

The conversion of method `tryCatchFinally` is shown below. This is a bit longer example that consolidates the idea behind explicit exception catching and subroutine implementation, coupled together. Only features discussed so far are presented.

```
 1. areceive(%4,#Example)
 2. adefine(%9,anull())
 3. apass(ause(%4))
 4. callx(mlookup(getclass(ause(%4)),[29]),[$Example,33,190],@37)
 5. idefine(%0,iconst($0))
 6. jump(@28)
 7. label(@7)
 8. jump(@35)
 9. label(@10)
10. apass(ause(%9))
11. apass(ause(%4))
12. callx(mlookup(getclass(ause(%4)),[30]),[$Example,33,192],@38)
13. idefine(%0,iconst($1))
14. jump(@28)
15. label(@19)
16. jump(@35)
17. label(@22)
18. idefine(%0,iconst($2))
19. jump(@28)
20. label(@26)
21. athrow(ause(%9))
22. label(@28)
23. apass(ause(%4))
24. call(mlookup(getclass(ause(%4)),[31]),[$Example,33,194])
25. iswitch(iuse(%0),[$0,@7][$1,@19][$2,@26])
26. label(@36)
27. jump(@36)
28. label(@35)
29. vreturn()
30. label(@37)
31. acatch(%9)
32. ijump(EQ,subtypeof(getclass(ause(%9)),aclass($TestExc)),iconst(
    $0),@22)
33. jump(@10)
34. label(@38)
35. acatch(%9)
36. jump(@22)
```

6.2.12 Synchronization

This last example demonstrates how synchronization primitives are implemented in the IR. The method *onlyMe* receives a reference parameter which is synchronized during the call to method *doSomething*.

```
void onlyMe(Foo f) {
  synchronized (f) {
    doSomething();
  }
}
```

The translation of the method *onlyMe* from Java source code to bytecode is shown below. The *monitorenter* and *monitorexit* bytecodes are used to *lock*/*unlock* the parameter object monitor, respectively. Implicitly the Java compiler generates code similar to a *try/finally* construct enclosing the synchronized code. It does that to guarantee that the object monitor will always be exited before the method termination.

```
.method onlyMe(LFoo;)V
      .limit stack 1
      .limit locals 4
      aload_1
      astore_2
      aload_2
      monitorenter
@4:   aload_0
      invokevirtual Example/doSomething()V
      aload_2
      monitorexit
      goto @18
@13: astore_3
      aload_2
      monitorexit
      aload_3
      athrow
@18: return
      .finally from @4 to @13 using @13
.end method
```

The IR program obtained from the conversion of method *onlyMe* is shown below. Statements 2 to 8 is code generated to test the parameter object *f* for a null value and throwing a *NullPointerException* if the case. Statements 10 to 11 implement the *monitorenter* operation, it is broken in two opcodes: lock to enter the object monitor, and readbarrier to invalidate cached memory reads. Statements 14 to 15 (and 18 to 19) implement the *monitorexit* operations, it is broken in two opcodes: unlock to exit the object monitor, and writebarrier to writeback cached memory writes. Both readbarrier and writebarrier does nothing but limiting the way that the IR program could be rearranged.

```
1.  areceive(%4,#Example)
2.  areceive(%9,#Foo)
3.  ajump(NE,ause(%9),anull(),@3)
4.  init(aclass($java/lang/NullPointerException),[$Example,35,201]
    )
5.  newinstance(aclass($java/lang/NullPointerException),[$Example,
    35,201])
6.  aresult(%14,#java/lang/NullPointerException)
7.  apass(ause(%14))
8.  call(mlookup(aclass($java/lang/NullPointerException),[12]),[$Example,
    35,201])
9.  athrow(ause(%14))
10. label(@3)
11. lock(ause(%9),[$Example,35,201])
12. readbarrier()
13. apass(ause(%9))
14. callx(mlookup(getclass(ause(%9)),[37]),[$Example,35,202],@20)
15. writebarrier()
16. unlock(ause(%9))
17. jump(@18)
18. label(@13)
19. writebarrier()
20. unlock(ause(%9))
21. athrow(ause(%14))
22. label(@18)
23. vreturn()
24. label(@20)
25. acatch(%14)
26. jump(@13)
```

6.3   Exception Windows Conversion

As described in the examples presented on Section 6.2, during the conversion of bytecode to intermediate representation, exception windows are replaced by explicit control code. Exceptional operations will have a direct reference to their exception catching entry point in the current IR program. For nested exception windows, each window will be associated to an exception catching entry point where subtyping test and handler delegation is implemented. Non-nested exception windows must be transformed to nested ones.

Exception windows in the bytecode are encoded as a *per method* array. The exception windows at the beginning of the array are the *innermost*. The exception windows at the end of the array are the *outermost*. When an exception is thrown, the JVM must search linearly the array, from the innermost to the outermost, for the first exception window that encloses the exception PC. If the exception thrown is caught by that particular window (the subtyping test does not fail) the control is transfered to its associated handler, otherwise the search continues. If no window catches the exception, the JVM rethrows it in the caller method frame.

The exception windows conversion procedure we have implemented requires nested exception windows. Exception windows usually are nested (Figure 6.1 (a)). This is specially true for bytecode generated from Java sources. Although its occurrence is very rare, non-nested

exceptions can occur[LY99, §4.9.5] and must be treated correctly.

The algorithm for transforming non-nested exception windows to nested ones is very simple. For each exception window, from the outermost to the innermost, break it into the minimum number of equivalent exception windows, so that each one of them is either fully enclosed by an outermost exception window or not enclosed at all. Figure 6.1 (b) and Figure 6.1 (c) shows a non-nested exception window and its equivalent nested exception window.
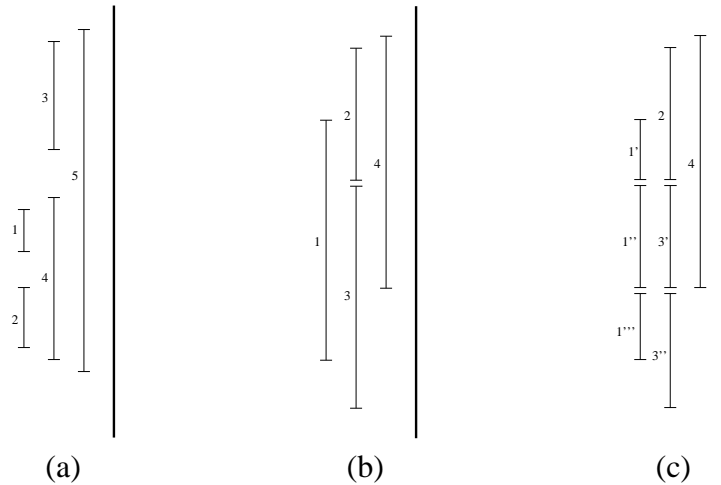
Figure 6.1: Exception windows: (a) Nested; (b) Non-nested; (c) Nested after transformation.

In the worst case, the number of exception windows after applying the algorithm increases exponentially. However, non-nested exception windows are rare enough to prevent choosing this implementation. Most important is that it provides correct behavior for all cases, although in some rare cases it lacks efficiency.

The procedure for converting exception windows is the following:

1. Adjust exception windows to have a nested structure. Each exception window must be top-level or fully enclosed in another exception window.

2. For each exception window do:

   (a) Create an exception catching label and entry point (`acatch` opcode).

   (b) Create a label for catching delegation.

   (c) If the exception window is not a *finally* exception window, generate a test for exception subtyping delegating control — if the test fails — to its directly enclosing exception window (using its catching delegation label), or rethrowing the exception in the caller frame (using `athrow` opcode) if it is a top-level exception window.

   (d) Transfer control unconditionally to the exception handler.

3. For each implicit exceptional operation do:

(a) If the operation is not enclosed by an exception window, generate the operation using its usual opcode.

(b) Otherwise, generate the operation using the *exception-prone* opcode variant and set its exception catching label to be the exception catching label of its directly enclosing exception window.

4. For each explicit exception throwing operation do:

(a) If the operation is not enclosed by an exception window, throw the exception into the caller frame (using `athrow` opcode).

(b) Otherwise, define the appropriate register with the reference to the exception being thrown and transfer control unconditionally to its directly enclosing exception window (using its catching delegation label).

Examples of exception windows conversion are shown in Section 6.2.10 and Section 6.2.11.

6.4   Subroutine Conversion

We choose not to include in the IR special opcodes for implementing subroutines (implemented using `jsr` and `ret` bytecodes). Typically, subroutines would be implemented using an intra-procedural *call*/*return* construct. However, since all *subroutine call-sites* are known at conversion time, we decided to implement it by indexing call-sites and switching at return points. This can be implemented without having to extend the IR, and no code is duplicated.

The subroutine conversion procedure is straightforward:

1. Initialize the *call-site index* to zero.

2. For each subroutine call-site (`jsr` bytecode) do:

(a) Define an integer register with current *call-site index*. The integer register must be the register bounded to the top of the bytecode operand stack after the subroutine call.

(b) Unconditional transfer control to the subroutine entry point.

(c) Declare a unique *return label* and associate it with the current *call-site index*.

(d) Increment the *call-site index*.

3. For each subroutine return point (`ret` bytecode) do:

(a) Switch on the integer register bounded to the top of the bytecode operand stack before the subroutine return (using `iswitch` opcode). Include in the switch a case entry for all call-site indices and their associated labels.

(b) For the fall through path, declare a label and an unconditional transfer to it, defining an unreachable infinite loop.

4. For each register live at the IR program entry point do:

   (a) Displace a default value register defining statement just after all *parameter-receiving* opcodes.

In principle, the subroutine conversion procedure could take advantage of precise control flow information (gathered during the bytecode verification) to suppress useless unreachable case labels. In our implementation, we choose not to use that control flow information in order to minimize the interface between the bytecode verification and the bytecode conversion modules. It is not a big overhead to keep those useless unreachable case labels, because rarely a method has more than one subroutine. Also, the data flow analysis to detect useless unreachable case labels can be formulated easily.

The infinite loop generated for the default path of the `iswitch` opcode is a simple workaround. An alternate implementation may choose to elect one of the call-site indices and use it as the default path.

This implementation of subroutines is efficient, even though it requires an integer switch during the subroutine return. Since switch case labels are actually call-site indices — which were generated sequentially — the switch can have a table-based translation.

The last step of the subroutine conversion procedure is a repair in the IR program to guarantee that all registers are initialized before their use. Although the use of initialized variables is a property of verified bytecode, the control structure simplification, that occurs during the subroutine conversion, inserts previously non-existing paths in the control flow graph, in which variables could be used without being initialized. We decided to keep this bytecode property also in IR programs. This is specially important when we discuss our implementation of *garbage collection* in Chapter 9. The accurate GC algorithm requires some live reference tables to run. Although uninitialized reference registers are not used in the IR program, it can possibly be included in a reference table for a portion of the IR program where it is considered live and is not initialized. This happens because the liveness analysis is a conservative approximation, and unfortunately semantic control information to detect this situation is not available anymore after conversion. In the best case, an uninitialized reference included in a live reference table makes the GC algorithm crash.

Subroutines makes GC hard, it is no news. This approach is simple and elegant. The price we pay is the cost of executing the initializing statements on method entry (for Java programs they are at most one per subroutine), and the increase of IR registers lifetime which may impact *register allocation*.

Examples of subroutine conversion are shown in Section 6.2.11 and Section 6.2.12.

6.5 Post Conversion Optimizations

This section describes optimizations that must be applied to IR programs after conversion. In principle, the conversion algorithm could produce IR programs that would not require those optimizations. However, we decided to simplify the conversion algorithm, for the sake of correctness, and apply these optimizations only afterwards.

6.5.1   Building Expression Trees

During conversion, bytecode local variables and operand stack values are bounded to IR registers. However, most operand stack values are intermediate values of complex expressions being computed. Therefore, representing them as *expressions trees*, rather than as a sequence of three operand statements, is a better choice. The greater the expression tree, the more effective is its translation to machine code. There are optimal *instruction selection* and *register allocation* algorithms for expression trees[AJ76].

   The transformation of three operand statements to a expression tree is simple. We look for register definitions that are used only once in the IR program; that use must be inside the same *basic block* where the definition is. Then we replace the use of the associated register by its defining expression.

   Care should be taken when the defining expression contains a memory read or register use operation. If the memory read is *volatile*, or there is a *memory read barrier* (`readbarrier` opcode) after the *defining statement* and before the *use statement*, the optimization must not occur. Also, it must not occur if, between those statements, there is a memory write operation possibly aliased to the same memory location as the memory read present in the defining expression (a similar restriction applies to registers).

   The following IR program was obtained from the conversion of method `align2grain` before applying the expression tree optimization. The optimized IR program is shown in Section 6.2.2.

```
 1. areceive(%4,#Example)
 2. ireceive(%0)
 3. ireceive(%5)
 4. idefine(%10,iuse(%0))
 5. idefine(%15,iuse(%5))
 6. idefine(%10,iadd(iuse(%10),iuse(%15)))
 7. idefine(%15,iconst($1))
 8. idefine(%10,isub(iuse(%10),iuse(%15)))
 9. idefine(%15,iuse(%5))
10. idefine(%20,iconst($1))
11. idefine(%15,isub(iuse(%15),iuse(%20)))
12. idefine(%20,iconst($-1))
13. idefine(%15,ixor(iuse(%15),iuse(%20)))
14. idefine(%10,iand(iuse(%10),iuse(%15)))
15. ireturn(iuse(%10))
```

6.5.2   Eliminating Null Checks

Null check elimination is an important optimization for converted IR programs. The *null check* is the most common runtime check present in bytecodes. Performing a null check for each operation that semantically requires it is expensive and usually redundant. By performing data flow analysis, we can discover that many checks can be safely suppressed and substituted by an equivalent check performed previously in the execution. Specifically, the `this` implicit parameter is never null, and its storage is never redefined in the bytecode generated from Java sources.

This optimization consists in removing IR statements of the form

```
ajump(EQ,ause(%4),anull(),@5)
```

where `%4` is never null, or replacing by an unconditional control transfer (`jump` opcode) if `%4` is always null. The similar idea can be applied to statements of the form

```
ajump(NE,ause(%4),anull(),@5)
```

they can be removed if `%4` is always null, or replaced by an unconditional control transfer if `%4` is never null.

The data flow analysis that provides information required by null check elimination is a *forward* analysis based on reference registers. The analysis works with three sets of reference registers: *null registers* (NL), *non-null register* (NN) and *unknown registers* (UK). As usual, the implementation of these sets is done using bit vectors, each reference register is associated to two bit positions ($c_1 c_2$). The first bit position indicates if the register is null (0) or non-null (1). The second bit position supercedes the value of the first bit position if the register contents cannot be known (1). Table 6.2 is the *truth table* for the *confluence operator* $\sqcup$.

$\sqcup$	NL (00)	NN (10)	UK (X1)
NL (00)	NL (00)	UK (X1)	UK (X1)
NN (10)	UK (X1)	NN (10)	UK (X1)
UK (X1)	UK (X1)	UK (X1)	UK (X1)

Table 6.2: Truth table for the confluence operator $\sqcup$.

The logical equations obtained from the above truth table are show below. Both can be implemented efficiently for bit vectors.

$$C = A \sqcup B \Rightarrow \left\{ \begin{array}{l} c_1 = a_1 + b_1 \\ c_2 = a_2 + b_2 + (a_1 \otimes b_1) \end{array} \right.$$

In order to efficiently implement the data flow analysis, we associate to each basic block a *flow function*. The flow function consists of a data flow item (NL, NN, UK) that each each reference register assumes at the basic block exit point, or a source register index from where the item must be copied at the basic block entry point. To compute the flow function we use the following procedure:

1. Initialize the flow function, each register is associated with its index.

2. For each statement basic block statement in forward direction do:

   (a) If the current statement is a reference copy statement, replace the item or index of the defined register in the flow function by the current value or index of the used register in the flow function.

(b) If the current statement defines a reference register and is not a copy statement, replace the item or index of the defined register by the value associated to the defining expression (see Table 6.3).

(c) Otherwise, do nothing.

Table 6.3 defines the flow value associated to each opcode that provides a reference as result.

OPCODE	FLOW ITEM
getclass	NN
aload	UK
aaload	UK
mlookup	NN
imlookup	UK
anull	NL
aclass	NN
astring	NN

Table 6.3: Flow item for each opcode that provides reference result.

The data flow is computed iteratively. Starting at the entry basic block the flow functions are applied until convergence. During computation, edges leaving basic blocks that end with a null check are treated specially; the flow item of the reference register being checked must be modified to reflect the test result on each path. Once the data flow analysis is over, the flow items at exit point of basic blocks are used to determine if the transformation may apply.

6.5.3 Factoring Exception Throwing Code

The IR program is generated by the conversion algorithm with a small section of code, for throwing internal exceptions, after each check for a violated property (e.g. null pointer access, division by zero). However, the conversion algorithm is naive enough to miss the fact that the same code may be replicated many times inside the same IR program.

The extra code generated to instantiate and throw an internal exception can usually be shared by many checks of the same property that occurs in the same line of code. This can be seen in the example below.

```
Object getNextNext() {
  return next.next;
}
```

Assuming that no null checks were eliminated yet, two null pointer checks are present in the IR program generated from the `getNextNext` method. The first check is done for the `this` parameter when reading field `next`. The second check is done when reading field `next` from the possibly null reference just read from field `next` of `this` object. The IR program obtained from the conversion of method `getNextNext` is shown below.

```
 1. areceive(%4,#Example)
 2. adefine(%9,ause(%4))
 3. ajump(NE,ause(%9),anull(),@6)
 4. init(aclass($java/lang/NullPointerException),[$Example,1,6])
 5. newinstance(aclass($java/lang/NullPointerException),[$Example,
    1,6])
 6. aresult(%19,#java/lang/NullPointerException)
 7. apass(ause(%19))
 8. call(mlookup(aclass($java/lang/NullPointerException),[12]),[$Example,
    1,6])
 9. athrow(ause(%19))
10. label(@6)
11. adefine(%9,aload(ause(%9),dynamic(0,0),false,#Example))
12. ajump(NE,ause(%9),anull(),@21)
13. init(aclass($java/lang/NullPointerException),[$Example,1,6])
14. newinstance(aclass($java/lang/NullPointerException),[$Example,
    1,6])
15. aresult(%19,#java/lang/NullPointerException)
16. apass(ause(%19))
17. call(mlookup(aclass($java/lang/NullPointerException),[12]),[$Example,
    1,6])
18. athrow(ause(%19))
19. label(@21)
20. adefine(%9,aload(ause(%9),dynamic(0,0),false,#Example))
21. areturn(ause(%9))
```

It is easy to see that the code instantiating and throwing the `NullPointerException` for both checks is the same (statements 4 to 9 and 13 to 18). The IR program can thus be rewritten to share that code.

```
 1. areceive(%4,#Example)
 2. adefine(%9,ause(%4))
 3. ajump(NE,ause(%9),anull(),@6)
 4. label(@21)
 5. init(aclass($java/lang/NullPointerException),[$Example,1,6])
 6. newinstance(aclass($java/lang/NullPointerException),[$Example,
    1,6])
 7. aresult(%19,#java/lang/NullPointerException)
 8. apass(ause(%19))
 9. call(mlookup(aclass($java/lang/NullPointerException),[12]),[$Example,
    1,6])
10. athrow(ause(%19))
11. label(@6)
12. adefine(%9,aload(ause(%9),dynamic(0,0),false,#Example))
13. ajump(EQ,ause(%9),anull(),@21)
14. adefine(%9,aload(ause(%9),dynamic(0,0),false,#Example))
15. areturn(ause(%9))
```

The idea we have used to implement this *factoring exception throwing code* optimization is the same used in *Language Theory* to minimize states in a *Deterministic Finite Automaton* (DFA) [Ner58, Hop71]. We divide the IR statements in sets, each set containing initially

statements that are equal in syntax, including their attributes. Then we start partitioning each set that contains statements leading execution to statements in different sets. When no more sets can be created, the sets with more than one statement indicates the statements that are replicated and can be removed from the IR program. Actually the replicated code is not removed but the control structure of the IR program is modified to shared a single copy of replicated code. The other copies are eliminated by *unreachable code elimination* described on Section 6.5.4.

This algorithm is generic and do the factoring for any replicated code, including the exception throwing code as well as user code.

6.5.4   Control Optimizations

The control optimizations to be applied to IR programs after conversion are basically two: *unreachable code elimination* and *jump optimization*.

Unreachable code elimination is a trivial optimization, it eliminates from the *control flow graph* of the IR program basic blocks not reachable by any path. Code may become unreachable after applying the optimizations described in Section 6.5.2 and Section 6.5.3.

The jump optimization changes the linear placement of the control flow graph of a IR program — sometimes merging basic blocks — in order to remove useless unconditional control transfers. To merge two basic blocks one of them must be the only *predecessor* of the other, which must also be its only *successor*. The following IR program was obtained from applying the jump optimization to the method `tryFinally` shown in Section 6.2.11.

```
 1. areceive(%4,#Example)
 2. adefine(%9,anull())
 3. apass(ause(%4))
 4. callx(mlookup(getclass(ause(%4)),[29]),[$Example,32,182],@25)
 5. idefine(%0,iconst($0))
 6. label(@16)
 7. apass(ause(%4))
 8. call(mlookup(getclass(ause(%4)),[31]),[$Example,32,184])
 9. iswitch(iuse(%0),[$0,@7][$1,@14])
10. label(@26)
11. jump(@26)
12. label(@7)
13. vreturn()
14. label(@14)
15. athrow(ause(%9))
16. label(@25)
17. acatch(%9)
18. idefine(%0,iconst($1))
19. jump(@16)
```

6.6   Discussion about Assynchronous Exceptions

An important issue to be discussed in the shade of the bytecode conversion is the *assynchronous exception* support. An assynchronous exception is an exception thrown by one Java

thread in the context of another Java thread.

Although user level assynchronous exception throwing has been *deprecated*[3] from the Java platform (`stop` method), it is also required in some implementations of the runtime to implement safely part of its internal operations (e.g. destroying the JVM).

Assynchronous exceptions are precise and may be detected after a small but bounded amount of time [LY99, §2.16.2]. The assynchronous exception detection by the JVM must be designed and implemented with care — not to sacrifice performance — since its occurrence is rare.

The intermediate representation does not have explicit opcodes for checking assynchronous exceptions. However, it was designed to check for assynchronous exceptions, at its convenience, by some selected opcodes. Those opcodes are actually the opcodes that are expensive and usually require runtime callbacks. They must also have support for stack trace information and have a variant form that defines exception catching labels. Those opcodes comprise the opcodes for calling native methods, allocating memory, initializing classes and synchronizing. We believe most multithreaded programs will inevitably use one of these operations from time to time.

With this scheme, it is true that some Java code may never check for assynchronous exceptions. If the code does not do any of the special operations listed above, this will happen. For instance the following *loopForever* method will never check for assynchronous exceptions in our implementation.

```
void loopForever() {
  for (;;)
    ;
}
```

However, code like this does not follow the Java multithreading guidelines. Since thread scheduling is not strictly defined in the Java platform, the *loopForever* method may stuck; not even given the chance to another thread execute and post an assynchronous exception. Portable well-written multithread Java programs will sometimes yield on tied loops (by calling native method *yield*), giving the runtime the possibility for throwing any pending assynchronous exceptions. The *loopForever* method could be rewritten this way.

```
void loopForever() {
  for (;;)
    Thread.yield();
}
```

---

[3]Deprecation means it should not be used by new software, but must be still available for backward compatibility.

Chapter 7

The x86 Back-End

This chapter describes the back-end for the *Intel Architecture 32-bit* family of processors. The *x86 back-end* is a simple and naive platform-independent code generator implementation. As a first implementation, reliability and simplicity where the main goals.

We describe the code generation strategy; the data structures required by the runtime to implement garbage collection, stack tracing and exception handling; the relocation and patch tables used to update the method text once in the client-side; and improvements that should appear in an enhanced version of this x86 back-end.

The reader is assumed to be familiar with the popular Intel 32-bit architecture features and instruction set [Int97a, Int97b]. Code samples are displayed using AT&T assembly syntax.

7.1   Code Generation

The x86 back-end uses a simple and naive code generation strategy. In a first moment, the back-end does the binding of IR registers on the stack frame by doing liveness analysis and building an interference graph. Afterwards, in a second moment, it does instruction selection by pattern matching using the tool described in Appendix B. Register allocation is done only for expression trees, using a well-known technique applied during instruction selection [AJ76]. The x86 back-end lacks local or global register allocation, instruction scheduling and peephole optimizations. A discussion about its improvement is left to Section 7.4.

7.1.1   Stack Frame and Registers Usage Protocol

The code generated by the x86 back-end obeys the following protocol:

- All parameters are passed in the stack.

- All general purpose registers are caller-saved.

- Return values are kept in registers (see Table 7.1).

- Callees pop parameters on return.

95

TYPE	REGISTER
integer	`%eax`
long integer	`%eax` (low word) `%edx` (high word)
float	`%st(0)`
double	`%st(0)`
reference	`%eax`

Table 7.1: Registers used to store return values.

The stack frame organization is depicted in Figure 7.1. During calls, callers push parameters and the return address. Callees create a new frame by saving caller frame pointer and making room for local variables. Local variables are actually IR registers, the term *local variable* is used to avoid confusion with processor physical registers (e.g. `%eax`). Local variables are bounded to the new frame but parameters storage is reused whenever possible (see Section 7.1.2 for details). The general prologue for methods is the following:

```
SELF:
  ...                        ;object fields
ENTRY:
  pushl  %ebp                ;save caller frame pointer
  movl   %esp,%ebp           ;set frame pointer
  pushl  $SELF               ;push text reference
  subl   $8,%esp             ;make room for 2 local variables
  ...
```

The *text reference* is a reference to the method implementation itself. Since method texts are first-class objects, which may become eligible for garbage collection, this reference prevents the method from becoming unreachable during its execution (see Chapter 9 for details about garbage collection). For the same reason, the area between the SELF label and the ENTRY label is reserved for instance fields.

During method return, the caller stack frame must be restored and the control transfered back to the instruction immediately following the call instruction. Also the callee is responsible for popping parameters from the stack. The general epilogue for methods is the following:

```
  ...
EXIT:
  leave                      ;restore previous frame
  ret    $4                  ;return and pop 1 parameter
```

7.1.2 Local Variable Binding

The first step in code generation is *local variable binding*. Local variable binding consists of associating stack frame indices — and thus reserving storage — to each local variable (IR register).
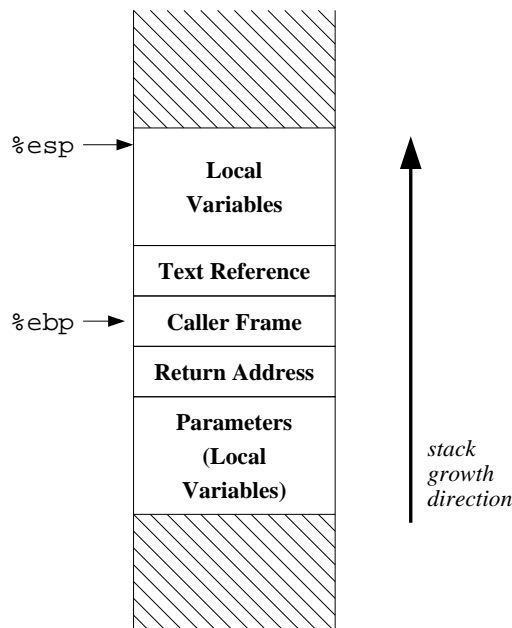
Figure 7.1: Stack frame organization.

In order to efficiently allocate storage to each local variable, we need to compute a data flow analysis called *liveness analysis*. Liveness analysis provides information about the liveness of each local variable for every point of the IR program. A local variable is *live* at a particular point if there is at least one definition of that local variable that reaches an use of the same variable passing through that point[1].

The characteristics of the liveness data flow analysis are:

- It is a *backward* analysis.

- In a path merge, the set of live variables is obtained by the *union* of the set of live variables on each path.

- A local variable use inserts that local variable into the set of live variables.

- A local variable definition removes that variable from the set of live variables.

Using liveness information we can compute the *interference graph*, a graph where nodes represent local variables and edges represents liveness interferences. A *liveness interference* encodes a pair of local variables that are both live at least in one point in the IR program. Using the interference graph we can discover the storage required by a particular IR program. The number of words required to do the local variable binding is at most one plus the degree of the node with greatest degree.

---

[1]A local variable use without a definition also makes that local variable live, but that should never occur in IR programs.

After building the interference graph, we assign to each node a stack frame index. For local variables defined by a parameter-receiving IR statement, e.g. `ireceive`, the frame index is positive and obtained from the parameter order. For other local variables, the stack frame index is usually negative and its assignment is done greedily — until all indices are assigned do: select a node not yet assigned, and assign an index different from the already assigned adjacent nodes indices.

Care must be taken when handling two word local variables (i.e. long integers and doubles). In our interference graph implementation, they are split into two separate nodes. Since two word local variables must be allocated contiguously, during stack frame index assignment we have to consider that the high word index must be subsequent to the low word index.

By doing local variable binding using an interference graph we obtain a very good allocation, which does not waste storage. Therefore, ordinary local variables are sometimes bound to the same storage as parameter local variables, because their liveness does not interfere (and thus it has positive index).

7.1.3   Instruction Selection

Instruction selection is done by tree pattern matching using the tree rewriting tool described in Appendix B. During instruction selection, we do register allocation for expression trees, based on a well-known technique [AJ76].

The tree pattern matcher is generated by the tree rewriting tool based on a grammar specification. The grammar specification contains the set of tree patterns and associated actions to generate code. Each tree pattern, plays one of two roles:

1. Matches a tree pattern that is a straight map of one instruction format available on the underlying instruction set. Tries to capture all addressing modes and minimize size/cycles.

2. Matches a tree pattern that cannot be mapped to a single instruction format. Usually a small pattern covering a special IR opcode. Generates a code segment with fixed addressing and instructions. In general, the number of pattern variants to cover all possible instruction formats and order combinations is impracticable.

Figure 7.2 shows sample tree pattern rules extracted from the x86 back-end grammar specification (see Appendix B for a full reference about the specification syntax). These are rules for matching the signed integer division (`idiv` opcode). The signed division in the x86 architecture requires the dividend to be in the `%eax` register, being the divisor in any other general purpose register. Before executing the division instruction, the issue of a `cltd` instruction is mandatory in order to sign extend `%eax` to the 64-bit pair `eax:edx`. Once the division takes place (`idivl`), the result is kept in `%eax` register while the associated remainder is written to the `%edx` register.

Both rules shown in Figure 7.2 matches the signed integer division `idivl` instruction. The difference between the two rules is in the scheduling of the code generated from subexpressions, which tries to accommodate the register pressure of the whole expression. This is exactly the implementation technique for expression tree register allocation described in

[AJ76] adapted to a *CISC* machine. The general idea — employed for orthogonal registers sets architectures — is that the minimum number of registers required by an expression tree r will be the $\max(r_1, r_2)$, if $r_1 \neq r_2$, or $r_1 + 1$, if $r_1 = r_2$, where r_1 and r_2 are the minimum number of registers required by each subexpression respectively. For the x86 architecture, we have implemented this technique using a *written registers set* for each expression instead of using r. This was done because of the x86 instruction set operand restrictions in which registers are not homogeneous (as past mentioned for the idivl instruction). Fortunately, the implementation of written registers sets could be done efficiently using 32-bit integers; and this was possible exactly because the x86 architecture register set is small.

```
private void eax()
<int cost, int reg, int kill> [@@.cost < cost]
  ...
  | IR.IDIV(eax,r32) [@3.reg != as.EDX && (@3.kill & @2.reg) == 0]
  { @@.cost = cost(6,3)+@2.cost+@3.cost;
    @@.reg = as.EAX;
    @@.kill = as.EAX|as.EDX|@2.kill|@3.kill; }
  = { @2();
      @3();
      as.cltd();
      as.idivl(@3.reg); }
  | IR.IDIV(eax,r32) [@3.reg != as.EDX && (@2.kill & @3.reg) == 0]
  { @@.cost = cost(6,3)+@2.cost+@3.cost;
    @@.reg = as.EAX;
    @@.kill = as.EAX|as.EDX|@2.kill|@3.kill; }
  = { @3();
      @2();
      as.cltd();
      as.idivl(@3.reg); }
  ...
  ;
```

Figure 7.2: Sample tree pattern rules extracted from the x86 specification.

Let's look a little closer at this grammar specification excerpt in Figure 7.2. Rules have the non-terminal *eax* on their left hand size because the result of the idivl instruction is always kept in %eax. The non-terminal *eax* synthetizes three attributes: a mixed time/space cost, the register that stores the result of the expression, and the set of registers *killed* during the computation of this expression. The *cost* attribute is used to choose the best tree matching. The tree pattern rules are similar, changing only the subexpressions scheduling. The tree patterns are simple, they match an idiv opcode requiring that the dividend must be %eax (*eax* non-terminal) and the divisor any general purpose register (*r32* non-terminal). It is important that the divisor must not be register %edx because the dividend will be sign-extended to the 64-bit register pair %eax:%edx (it can be seen that this is captured semantically rather than syntatically). Also, each match must only occur if the subexpression that is scheduled last does not overwrite the register storing the result of the subexpression scheduled first. The

attributes synthetization for both rules is very simple: the cost is the sum of the costs of each subexpression plus a 6 cycles and 3 bytes of the `cltd`/`idivl` instructions; the result register is `%eax`; the written registers set is the union of the written registers set of each subexpressions, including registers `%eax` and `%edx` for the current expression. The code generation for these tree pattern rules is also simple, first the code for each subexpression is generated according to the expected scheduling (methods `@2( )` and `@3( )`), then the instructions related to the division are generated.

The following assembly code was generated by the x86 back-end for the expression `idiv(iuse(%0),iadd(iuse(%5),iuse(%0)))`.

```
        movl    [%ebp+8],%eax    ;left subexpression, write %eax

        movl    [%ebp-8],%ebx    ;right subexpression, write %ebx
        addl    [%ebp+8],%ebx    ;must not kill %eax

        cltd                     ;division, write %eax,%edx
        idivl   %ebx             ;result kept in %eax
```

The rest of the x86 back-end grammar specification is vast and repetitive. Besides, the ideas used to implement each rule are exactly the same as described in the example above.

7.2   Cooperative Runtime Support

This section describes extra information provided by the x86 back-end as a requirement of the language runtime in order to carry out some of its tasks, namely: live references identification in the stack frame, stack traces printing, control transference to appropriate exception handler, and identification of references hard-coded in method text objects.

7.2.1   Live Frame References and Stack Tracing Tables

At certain times, the runtime has to inspect the thread call stack in order to gather information about it. This is only possible when the executing method does a *runtime callback*, then the runtime is able to look at the underlying stack. Also, information about each method in the stack frame must be made available by the code generator to be used by the runtime during callbacks. The runtime inspects the stack for two reasons: to discover the set of live root references in the thread stack, and to print a stack trace.

We have classified some IR opcodes as *inspection-point*, they mark points in the IR program where the stack frame may be inspected. The code generator identifies inspection-point opcodes and provides comprehensive information about the IR program during their execution. There are two types of inspection-point IR opcodes:

Call Opcodes Opcodes that cause another Java method invocation. Provides information for all frames in the call stack but the topmost. Namely: `call` and `callx`.

Runtime Callback Opcodes Opcodes that cause a runtime callback. Provides information for the topmost frame. Namely: `init`, `initx`, `lock`, `lockx`, `ncall`, `ncallx`, `newarray`, `newarrayx`, `newinstance`, `newinstancex`.

The information required by the runtime is generated using a return address indexed table. After each call instruction that is generated to implement an inspection-point IR opcode, the code generator provides a return address label. The return address indexed table is built associating an entry to each return address label, and its pointer is made available via the method text header. When necessary, the runtime steps through the stack collecting the desired information by looking for the return address of each call in the indexed table associated to each stack frame (see Chapter 8 for object headers/heap structures layout, and details about stack traversal). Each indexed table entry is a pair of pointers to two other tables: the *live frame references* table, and the *stack tracing* table. The table is sorted by return address to speedup the search. This can be seen in the following code sample.

```
    ...
    pushl  %eax              ;pass parameter
    call   _init_            ;initialize class, runtime callback
IPOINT_0:
    addl   $4,%esp           ;C protocol, caller pop parameters
    ...
    pushl  %eax
    call   _newinstance_     ;instantiate class, runtime callback
IPOINT_1:
    addl   $4,%esp           ;C protocol, caller pop parameters
    movl   %eax,12(%ebp)     ;save return value
    ...
    pushl  -20(%ebp)         ;pass parameter
    call   *CALLEE_CLASS+96  ;call method through table
IPOINT_2:
    movl   %eax,-16(%ebp)    ;save return value
    ...

INSPECTOR:
    .long  IPOINT_0          ;the class initialization info
    .long  TRACE_0           ;stack tracing table
    .long  0                 ;no live references

    .long  IPOINT_1          ;the class instantiation info
    .long  TRACE_0           ;stack tracing table, same so is shared
    .long  0                 ;no live references

    .long  IPOINT_2          ;the static method table call info
    .long  TRACE_1           ;stack tracing table
    .long  LIVES_0           ;live frame references table

    ...
```

The live frame references table is a zero-terminated array of signed 8-bit frame indices. It is used by the runtime to determine the set of live root references for the current stack frame.

It is built using data flow information gathered by liveness analysis during local variable binding (see Section 7.1.2). The set of indices that make up a live frame references table is the set of frame indices assigned to the IR registers live at the point immediately following the inspection-point IR opcode associated to current return address. The encoding of the 8-bit frame indices addresses words instead of bytes, also the return address and text reference indices are skipped to expand the indexing capacity (see Figure 7.1 for stack frame layout). Therefore, an 8-bit index with value 4 addresses the reference at `(4+1)*4(%ebp)`; an 8-bit index with value -2 addresses the reference at `(-2-1)*4(%ebp)`. The encoding of the live frame references table using signed 8-bit indices may sound very limited but it is not. It provides support for identifying up to 128 parameters and 255 local variables (including parameters storage being reused by local variables). This is rather enough though java methods may have up to 255 parameters and, in some pathological cases, as many simultaneous live references in a single inspection-point as required to exceed the supported limit. The encoding, however, has sufficed our first implementation needs, and can be easily reviewed. The following code sample is the live frame references table for the method translation shown previously.

```
      ...
   LIVES_0:
     .byte  -4              ;reference at -20(%ebp) is live
     .byte  2               ;reference at 12(%ebp) is live
     .byte  3               ;reference at 16(%ebp) is live
     .byte  0               ;marks the end of table
     ...
```

The stack tracing table is an array of three field records. Each record contains information about a source code point that must appear in the stack trace for the current return address. The first field of the record is a reference to the class that declares the method that must appear in the stack trace. The second field is the index of that method as it appears in the declaring class file. The third field is the source code line number that must appear in the stack trace. If the line number is not available the second field must have flag value 65535 and the third index must be the index of the method (methods are indexed from 0 to 65534 while line numbers, when available, are indexed from 0 to 65535, so this special encoding was chosen). Each stack tracing table end is marked with a null reference. The following code sample shows the stack tracing tables for the method translation shown above (note that the stack trace information for the first two inspection points was the same, so a single stack tracing table is shared by both).

```
      ...
   TRACE_0:
     .long  THIS_CLASS      ;class reference
     .short 18              ;method index
     .short 77              ;line number
     .long  0

   TRACE_1:
```

```
        .long   THIS_CLASS      ;same class reference
        .short 18               ;same method index
        .short 79               ;two lines below
        .long  0
        ...
```

For IR programs translated right after the conversion from bytecode, each stack tracing table has at most one three field record. However, multiple records are supported in order to correctly implement stack traces when *inlining optimizations* occur before the translation by the back-end takes place. Inlining optimizations replace *in loco* method calls by the contents of the callee methods. Then method calls are eliminated and related stack trace information would be lost. In order to keep stack trace integrity, the stack trace information of the eliminated method call is added to the stack trace information of each inspection point in the code being inlined. The following code sample shows the stack tracing table resulted from the inlining of a method call.

```
        ...
    TRACE_1:
        .long   CALLEE_CLASS    ;class that owns the callee method
        .short 34               ;callee method index
        .short 203              ;line number

        .long   THIS_CLASS      ;previous stack trace information
        .short 18               ;that would have been lost after applying
        .short 79               ;the inlining optimization

        .long  0
        ...
```

As in the case of the live frame reference table encoding, the stack tracing table encoding was chosen for simplicity. Better alternatives are certainly available.

7.2.2   Exception Catching Routine

IR programs that have exception catching entry points (`acatch` opcode) are translated to method texts that must provide an *exception catching routine*. An exception catching routine is a small code segment that, based on the return address of a call, catches and delegates an exception to its appropriate handler. The exception catching routine of a caller method is used by the runtime to search for a handler whenever an exception is thrown through the frame of the callee method.

When a method cannot handle an exception, it rethrows the exception which may be caught by any caller method in the call chain. This rethrown action is done by calling a runtime routine called `_athrow_`. The `_athrow_` routine receives as parameter the exception instance being thrown. Then the `_athrow_` routine steps through the stack looking for a method who catches the exception (see Section 8.3.2 for details about stack traversal). For

each frame visited, it uses the *text reference* to reach the associated method text instance and check if it implements an exception catching routine (a possibly null pointer field in the method text header). If the method text does not implement an exception catching routine then the step through the stack continues. Otherwise, the exception catching routine is invoked. The parameters to the exception catching routine are: the exception being thrown, the return address for the method associated to the current stack frame, and the current stack frame base pointer. A sketch code for the ⌟athrow⌟ routine is shown above.

```
⌟athrow⌟:
  popl   %ecx            ;discard return address,
                         ;⌟athrow⌟ never returns
  popl   %eax            ;save exception instance in %eax

NEXT:
  movl   4(%ebp),%edx    ;save return address in %edx
  movl   (%ebp),%ebp     ;goto caller frame
  movl   -4(%ebp),%ebx   ;save text reference in %ebx

  testl  %ebx,%ebx       ;if text reference is null
  je     C_FRAME         ;its not a java frame

  movl   -8(%ebx),%ecx   ;save method text header pointer in %ecx
  movl   -12(%ecx),%ebx  ;save exception catching routine in %ebx

  testl  %ebx,%ebx       ;if no exception catching available
  je     NEXT            ;proceed to caller frame

  pushl  %ebp            ;pass frame pointer as parameter
  pushl  %edx            ;pass return address as parameter
  pushl  %eax            ;pass exception instance as parameter

  call   *%ebx           ;jump to exception catching routine
                         ;it never returns...

C_FRAME:
  ...                    ;C frame, return to JNI
```

In order to implement the exception catching routine, the x86 back-end must keep track of all possible return addresses of *exception-prone* opcodes (e.g. `callx`, `newinstancex`, etc) for a particular IR program. During code generation an extra sequential label is placed after each call instruction associated to an exception-prone opcode. That can be seen in the code excerpt above for a direct method call.

```
  ...
  pushl  -12(%ebp)       ;push second parameter
  pushl  -8(%ebp)        ;push first parameter
  call   CALLEE_ENTRY    ;direct method call
XRETADDR_2:              ;return address label
  movl   %eax,8(%ebp)    ;save return value in local
```

```
      ...
   HANDLER_2:                    ;handler enclosing the call
     movl   %eax,-8(%ebp)       ;store exception instance in local
      ...                        ;handle exception
```

The code for the exception catching routine is generated by the back-end just after the actual method translation. The exception catching routine restores the current method stack frame and searches the exception-prone return labels using the return address. If any label matches the return address, then control is transfered to its handler entry point label. Otherwise, the exception was thrown in a point not enclosed by a exception handler, and is not handled by current method; it is then rethrown in caller frame by calling the _athrow_ routine. A sample exception catching routine is shown above, note that the return method search is done sequentially for the sake of clarity, actual implementation uses a lookup table method.

```
   CATCHER:
     popl   %ecx               ;discard return address, never returns
     popl   %eax               ;save exception instance in %eax
     popl   %edx               ;save return address in %edx
     popl   %ebp               ;restore current stack frame

     leal   -12(%ebp),%esp     ;restore top of the stack

     cmpl   $XRETADDR_0,%edx   ;search for return address
     je     HANDLER_0          ;and transfer control to handler
     cmpl   $XRETADDR_1,%edx   ;exception instance is kept in %eax
     je     HANDLER_1
     cmpl   $XRETADDR_2,%edx
     je     HANDLER_2
     ...
     pushl  %eax               ;exception not catched
     call   _athrow_           ;rethrow in caller frame
```

7.2.3 Method Text Reference Table

Each heap allocated class instance must provide information about the layout of references in its field area. The instance class is usually the common placeholder for that kind of information. During garbage collection, each class instance is visited, and using its class reference[2], it is possible to locate the references inside the field area, which are scheduled to be visited in the future according to the garbage collection scheme.

Since in our implementation method texts are first-class objects (instances of final class *MethodText*), there must be a way to locate references directly referenced in its body. The types of references directly referenced by a method text are:

- Method texts, used in direct calls.

---

[2]Some implementations do not store the class reference in each instance, but a class record pointer.

- String literals, *internalized* instances of class `String`.

- Meta class objects, instances of class `Class`.

In order to obtain that information, each method text instance has in its header a pointer to a table of references, the *method text reference table*. That table is simply a immutable null-terminated array of references. Some other possibilities for encoding such table are possible but we choose this encoding for simplicity rather than storage efficiency. For instance, storing a 16-bit offset in the method text, indicating a reference encoded as instruction immediate data, would save half the memory; however it would limit method text length to be near 64K, and method text reference offsets should be flagged since, as immediate data, they are stored as PC relative addresses (even after we have obtained the absolute address from the relative address, this absolute address still is not the method text reference (`SELF`), but a pointer to its entry point `ENTRY`; a constant value must be subtracted from it). Simplicity was then an adequate choice specially because, in practice, this table is rather small and duplicate entries can be easily eliminated.

In addition, each method text instance has in its header a reference to its declaring class. This reference is required to prevent the class from being garbage collected while the method text is still executing (reachable by a text reference in any stack frame). Object headers and heap structures layout are described in Section 8.1.

7.3 Relocation and Patch Tables

This section provides details about relocation and patches that must be done by the runtime whenever a new method text is instantiated. Relocation consists of updating memory locations inside the method text by adding its base offset to them. Patches are updates to memory locations inside the method text in order to reflect the actual reference of another object. The relocation and patch tables are part of each method text information sent to the client-side during the TRANSLATE phase using the x86 back-end.

7.3.1 Relocation Table

The *relocation table* specifies method text offsets that contain absolute addresses to memory locations relative to its base address. The table is required because the run-time address of a method text is not known prior to its instantiation, and relative addressing is not available. Usually, relative addressing is not available when a label has to be placed in a table or used as immediate data of a non control transfer instruction. For instance, all the contents of the return address indexing table described in Section 7.2.1 need to be relocated; the address of each word must be included in the relocation table.

Prior to relocation, relocatable addresses contents are the zero-based offset of each label inside the method text. Relocating an address means adding the base address of the method text to this offset, resulting in its absolute address.

Figure 7.3 shows how relocation is implemented. During code generation, each absolute address to a label inside the method text is initialized with its offset from the base offset
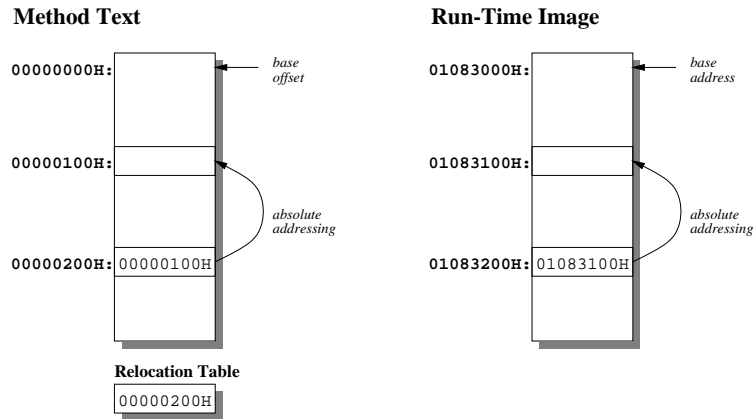
Method Text   **Run-Time Image**

Figure 7.3: Relocation of absolute addresses.

(`00000100H`); also the offset of the absolute address is recorded for relocation (`00000200H`). At run-time, each entry in the relocation table is visited and the method text base address is added to the absolute address at the associated offset.

$$01083000H + 00000100H = 01083100H$$

7.3.2   Runtime Callback Patch Table

The *runtime callback patch table* specifies method text offsets that contains PC relative addresses to runtime callback routines. The table is required because the address distance between the method text object and the callback entry point is not known before run-time. This address difference is exactly the value used as immediate data to relative calls. Relative addressing needs patching when target addresses are located outside the same block of code, which is the case.

Prior to patching, PC relative addresses contents are the difference between the method text base offset and the offset following the immediate data. Patching means adding the address difference between the callback entry point and the method text to that value.

Figure 7.4 shows how runtime callback patching is implemented. During code generation, each PC relative address to runtime callback is initialized with the difference between the base offset and the offset following the immediate data.

$$00000000H - 00000104H = FFFFFEFCH$$

$$00000000H - 00000204H = FFFFFDFCH$$

Also the offset of the immediate data is recorded for patching according to the associated runtime callback (`_athrow_`). At run-time, each entry in the runtime callback patch table is visited and the address distance between the runtime callback entry point and the method text base address is added to the immediate data at the associated offset.

Method Text   **Run-Time Image**
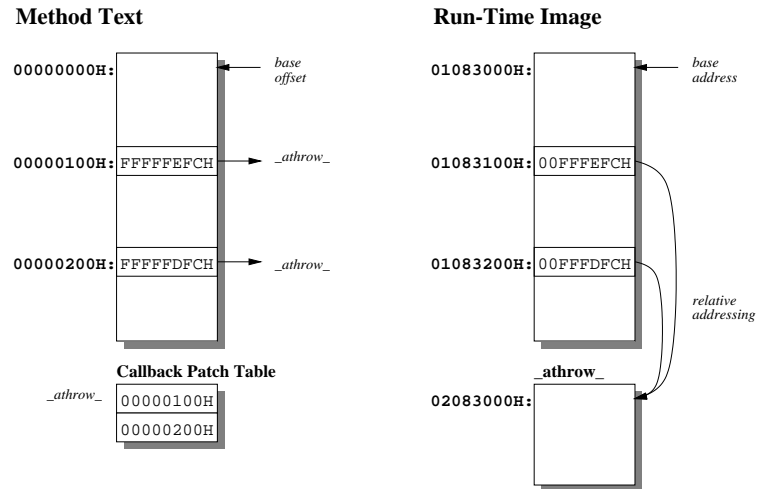
Figure 7.4: Patching of runtime callbacks.

$$02083000H - 01083000H = 01000000H$$

$$FFFFFEFCH + 01000000H = 00FFFEFCH$$

$$FFFFFDFCH + 01000000H = 00FFFDFCH$$

7.3.3   Method Text Patch Table

The *method text patch table* is similar to the runtime callback patch table but instead of specifying runtime calls, it specifies other directly called method texts. This table is also required because the address distance between the method text objects is not known before run-time. Method text references are symbolically identified as an entry in a class dispatch table (e.g. `java/lang/System`[3]).

As occurred in the runtime callback patch table, prior to patching, PC relative address contents are the difference between the method text base offset and the offset following the immediate data. Patching means adding the address difference between method texts to that value.

Figure 7.5 shows how method text patching is implemented. During code generation, each PC relative address to runtime callback is initialized with the difference between the base offset and the offset following the immediate data.

$$00000000H - 00000104H = FFFFFEFCH$$

$$00000000H - 00000204H = FFFFFDFCH$$
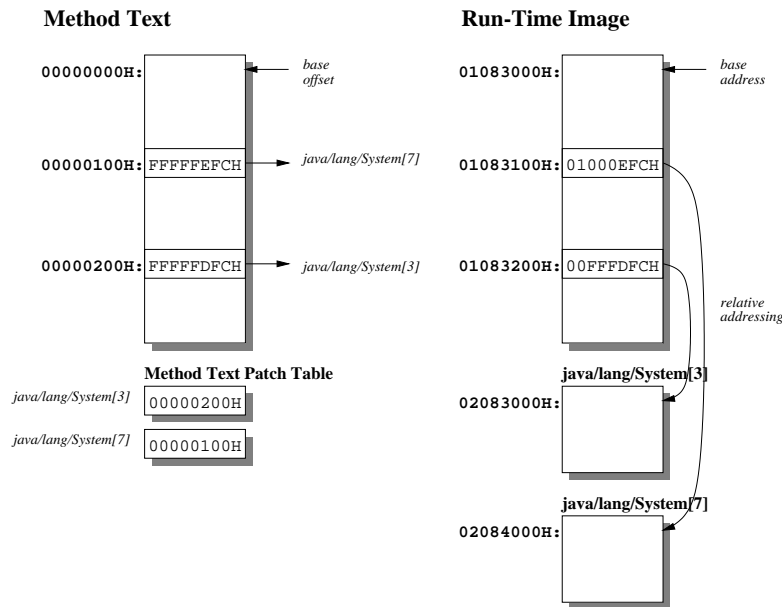
Figure 7.5: Patching of method text calls.

Also the offset of the immediate data is recorded for patching according to the associated method text (`java/lang/System`[3], `java/lang/System`[7]). At run-time, each entry in the method text patch table is visited and the address distance between the method texts addresses is added to the immediate data at the associated offset.

$$02083000H - 01083000H = 01000000H$$

$$02084000H - 01083000H = 01001000H$$

$$FFFFFEFCH + 01001000H = 01000EFCH$$

$$FFFFFDFCH + 01000000H = 00FFFDFCH$$

7.3.4 String Literal Patch Table

The *string literal patch table* specifies method text offsets that contains direct references to string literals. The table is required because the run-time address of a string literal (instance of *String*) is not known during code generation. Direct references to string literals are obtained from the translation of the `astring` IR opcode.

Prior to patching, string references are initialized with *null* (00000000H). Patching means overwriting that value with actual string reference. The actual string reference is obtained from the *internalized string table* (see *String.intern( )* API call).

Figure 7.6 shows how string literal patching is implemented.

Method Text   **Run-Time Image**

```
00000000H:        ┌──────┐← base           01083000H:        ┌──────┐← base
                  │      │   offset                          │      │   address
                  │      │                                   │      │
00000100H: 00000000H ─────→ "Hello, world!"  01083100H: 02083000H ┐
                  │      │                                   │      │ │
                  │      │                                   │      │ │ absolute
                  │      │                                   │      │ │ addressing
         String Literal Patch Table                "Hello, World!"  │
"Hello, world!"  00000100H                02083000H:        ┌──────┐←┘
                                                            │      │
```
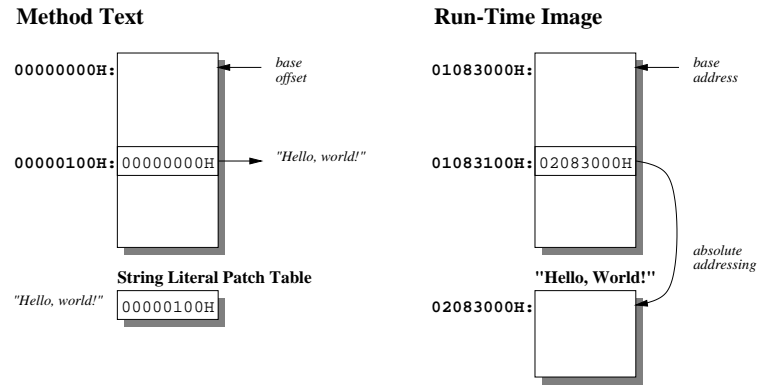
Figure 7.6: Patching of string literal references.

7.3.5   Meta Class Patch Table

The *meta class patch table* specifies method text offsets that contains direct references to meta class objects. The table is required because the run-time address of a meta class object (instance of `Class`) is not known during code generation. Direct references to meta class objects are obtained from the translation of the `aclass` IR opcode.

Prior to patching, meta class references are initialized with *null* (00000000H). Patching means overwriting that value with actual meta class reference.

Method Text   **Run-Time Image**

```
00000000H:        ┌──────┐← base           01083000H:        ┌──────┐← base
                  │      │   offset                          │      │   address
                  │      │                                   │      │
00000100H: 00000000H ─────→ java/lang/System 01083100H: 02083000H ┐
                  │      │                                   │      │ │
                  │      │                                   │      │ │ absolute
                  │      │                                   │      │ │ addressing
         Meta Class Patch Table                    java/lang/System │
java/lang/System  00000100H               02083000H:        ┌──────┐←┘
                                                            │      │
```
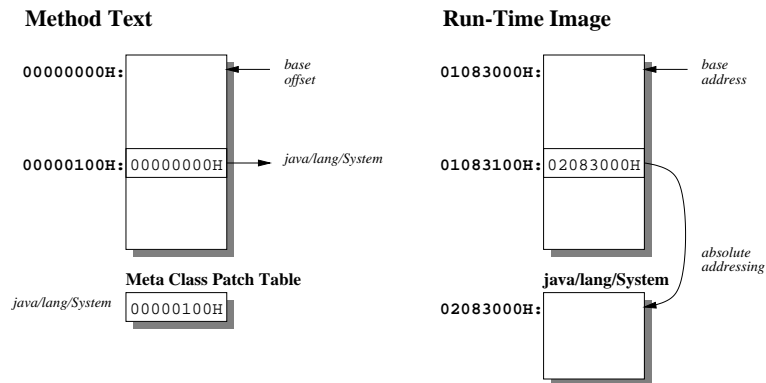
Figure 7.7: Patching of meta class references.

Figure 7.7 shows how meta class patching is implemented.

7.4   Back-End Improvements

The current implementation of the x86 back-end still lacks a lot of improvements. The most important improvements deal to:

Processor Specialization The back-end should be split in multiple back-ends sharing a common framework — one for each base processor of the Intel family. This would give us the opportunity to generate better code for machines that have processors with better hardware resources.

Instruction Selection The instruction selection must be simplified not to address expression tree register allocation. We noticed that trying to allocate registers to expression trees for a CISC machine results in large and heavy matchers. Removing this task from instruction selection is a better choice. The new implementation would be faster and consume less memory.

Global Register Allocation Since register allocation is not performed during instruction selection, a register allocator should be provided. Global register allocation does a map from virtual registers to machine registers trying to minimize memory accesses. It may be extended to allocate machine registers also to local variables that would be removed from the stack frame.

Peephole Optimizations Some peephole optimizations should be incorporated into the x86 back-end. Peephole optimization is a gain-proven cost-effective well-known technique used to implement simple optimizations based on a small window of code. Peephole optimization could be used in the x86 back-end to find and replace segments of code that can be rewritten as machine idioms (e.g. hardware loops and SIMD MMX instructions).

Instruction Scheduling For superscalars processors, instruction scheduling is an important optimization. It is basically the reordering of instructions in order to optimize the processor pipeline instruction flow.

Chapter 8

Runtime Environment

This chapter describes the Client JVM runtime implementation. The runtime is composed of a garbage-collected heap, multiple thread stacks, a monitor allocation table and the JNI implementation. Garbage collection, is the most complex runtime component, and thus deserves a separate chapter (Chapter 9).

8.1   Heap Structures

The garbage-collected heap is a linked-list of memory blocks. The allocation of memory blocks is done in page units, using the underlying operating system memory allocation interface. Each memory block, contains a sequence of word-aligned heap objects placed contiguously inside the block. There are six types of heap objects, namely:

Ordinary Objects Instances of ordinary classes (e.g. `String`, `Thread`, etc).

Array Objects Instances of array classes.

Method Text Objects Instances of class `MethodText`, each one representing a Java method binary translation.

Meta Class Objects Instances of class `Class`, each representing a loaded class.

Free Cells Memory blocks not currently associated to the storage of a Java object.

Block Records Information about the current heap memory block in the heap linked-list.

The first memory block in the heap linked list is the only block allocated ahead of time. It is allocated in the data segment and contains objects resulting from the core libraries embedding (see Chapter 10).

In addition to the memory block linked-list, the heap implementation provides a *free cell cache*, as being the usual implementation of the allocator. It is a table used to speed up the search for small free cells during allocation. Each cache entry points to the first element of a linked-list of fixed-size free cells. The allocator implementation is described in details in Section 8.2.

Each heap object has a two-word internal header which contains information associated to the heap implementation. The internal header has a negative offset and its contents varies according to the object type.

8.1.1 Ordinary Objects

Ordinary objects are instances of ordinary classes. The heap layout of ordinary objects, depicted in Figure 8.1, is logically divided in two areas: the inherited fields area and the new fields area.
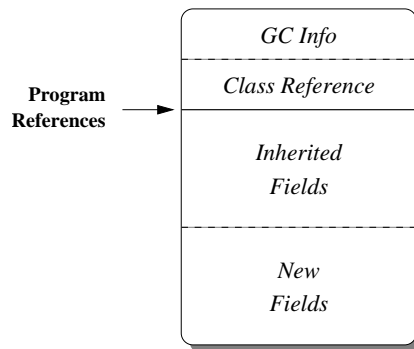

Figure 8.1: Ordinary objects layout.

The inherited field area is used to store instance fields declared in superclasses. Its layout must match the layout of both inherited and new field areas of the superclass. The new field area is used to store instance fields declared in the current class. However, sometimes new fields are placed in the inherited fields area to fill gaps left by word alignment. The placement of instance fields in the field areas is done by the Server JVM as described in Section 4.5.2.

For ordinary objects, the internal header is composed of its class reference, used to determine its type, its size and implement virtual calls, and the *GC Info* word, which is a bit field that contains information regarding monitor, garbage collection and heap implementations.

8.1.2 Array Objects

Array objects are instances of primitive and reference arrays. The heap layout for array objects is depicted in Figure 8.2. As in the case of ordinary objects, array objects have inherited and new field areas. In addition, it have a variable length area reserved for the array elements. The inherited and new fields areas are usually empty for array objects; therefore the access to the elements can be done directly using their reference as the base offset. Array classes are final, so its variable length layout does not need to be matched by subclasses.

The internal header of array objects is composed of its class reference and its length. The actual size of an array object is the sum of its instance size, retrieved from class, with its length scaled by its element width. Since there is no room in the internal header for the GC Info, it is placed after the array elements area.
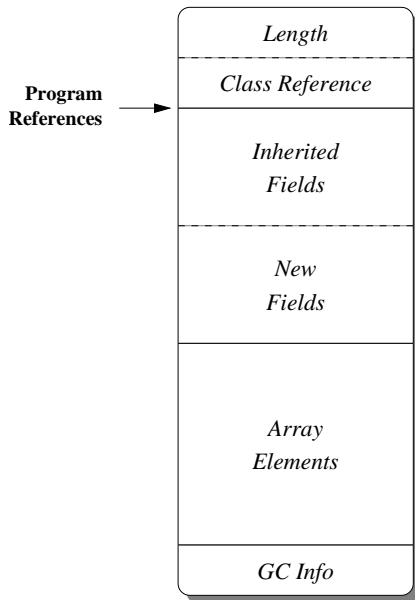
Figure 8.2: Arrays layout.

8.1.3 Method Text Objects

A method text object is a first-class object that represents the binary translation of a Java method. It has a special semantics, and its layout is depicted in Figure 8.3. Like arrays, the `MethodText` class is final, and thus no subclasses will have to match its layout. Also, the inherited and new fields areas are usually empty (it declares no fields and extends `Object`, which also declares no fields in its standard implementation).

The binary code is placed right after the new fields area. It contains not only the method translation, but also the exception catching routine and the live frame references, stack tracing, and method text reference tables, that were described in Section 7.2. Following the binary code area comes a four entry table that identifies these entities inside the method text. The reference to the class that declares the associated method is provided as well.

The internal header of method text objects is composed of its class reference (class `MethodText`) and a tail pointer. The tail pointer is required to determine the method text size since the binary area has a variable length. Also, it identifies the bottom of the four entry table described above used by the runtime to access the method text internals. That table has a fixed-size and could have been placed after the new fields area. However, once both inherited and new fields areas are usually empty, moving that table to the bottom makes the binary code entry point equals to the method text reference, simplifying method calls. At last, since no room is left for the GC Info in the method text internal header, it is placed after the four entry table.
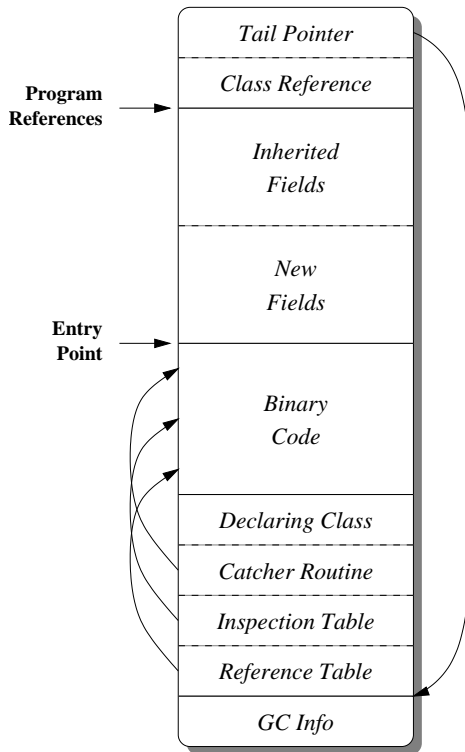
Figure 8.3: Method texts layout.

8.1.4   Meta Class Objects

Meta class objects are instances of final class `Class`. Apart from holding its instance fields, each meta class object is also the common placeholder for the dispatch table and static fields area of the Java class they represent. Also they provide room to store extra information required by the runtime to implement some of its operations (e.g. runtime type compatibility checks, linkage state, etc). Figure Figure 8.4 depicts the layout for meta class objects.

Following the inherited and new field areas comes the dispatch table which is composed of three parts. The first part is a single entry to the class initializer (`<clinit>`); this entry is null if the class does not provide a static initializer. The second part is the subset of dispatch entries associated to methods that may be overriden by subclasses. The layout of subclasses must match the layout of the superclass for these entries. The third part is the subset of method entries that will never be overriden by subclasses (i.e. static or final methods and constructors).

The next area in the meta class object layout is reserved for static fields, those fields declared static in the class represented by the current meta class object. On the sequence, comes the native pointer table, one for each native method declared in the associated class. Native methods are resolved and bound by the runtime during their first use; after resolution, the entry point address of the actual JNI native method implementation is stored into the native pointer table.

The internal fields area contains information used by the runtime to access and operate

Tail Pointer
Class Reference
Inherited Fields
New Fields
Class Initializer
Inherited Entries
Overridable New Entries
Non-Overridable New Entries
Static Fields
Native Pointers
Internal Fields
Meta Info Pointer
GC Info

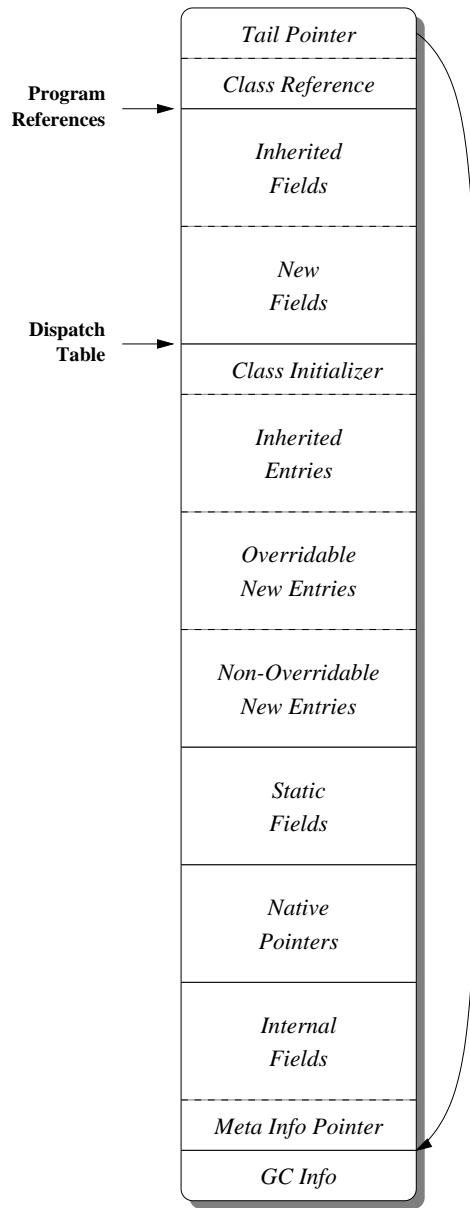**Program References**

Dispatch Table

Figure 8.4: Meta classes layout.

over the class object and its instances. This information comprises:

- Name and version number.

- Superclass reference.

- Interfaces references and associated dispatch table base offsets.

- Defining class loader reference.

- Element class reference, dimensions, and element width (for array classes).

- The linkage state, any of: loaded, linked, initialized or erroneus.

- The initialization thread id (refer to the class initialization procedure [LY99, §2.17.5]).

- A flag indicating if the instances need to be finalized, as part of garbage collection efficiency [LY99, 2.17.7].

- The offset and type of the *weak reference* (if the case).

- Static and instance sizes.

- Static and instance reference table offsets and sizes.

- Overridable dispatch table entry count.

- Non-overridable dispatch table entry count.

The meta info pointer points to a structure that contains meta information about the associated class. This meta class information is actually the information provided by the META phase, as described in Section 4.5.3, required to print stack traces and by the *Reflection API*.

The internal header of meta class objects is composed of its class reference (class `Class`) and a tail pointer. The tail pointer is required to determine the meta class size. Since no room is left for the GC Info in the method text internal header, it is placed just after all areas.

8.1.5   Free Cells

Free cells objects are heap objects whose storage was reclaimed by garbage collection and still was not assigned to an actual Java object. The layout of free cells is simple, as depicted in Figure 8.5. It is basically a size word followed by some uninitialized space followed by a trailing mirror size word.

For three-word free cells (including the size of internal header) the size words overlap. Two-word free cells does not provide a size word but a bit set in its GC Info word (see section Section 8.2.1). One-word free cells are not allowed since they cannot be represented as a heap object (every heap object must have a two-word internal header), this is done by preventing the allocator from fragmenting free cells that would leave just one-word remaining.

Non-Java heap objects, i.e. free cells and block records, are identified by having a null reference in their internal header, instead of a class reference as in the case of Java objects.
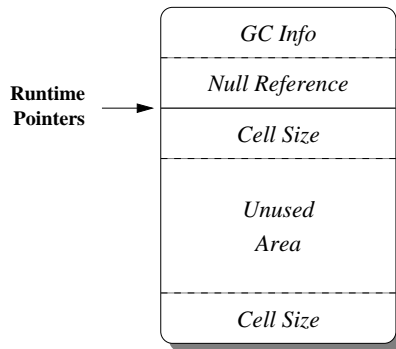
Figure 8.5: Free cells layout.

8.1.6 Block Records

Block records store information about a particular memory block of the heap linked list. It is placed at the bottom of the memory block and contains basically two information: the memory block size (including itself), and a pointer to the block record of the next memory block in the heap linked list. The layout of a block record can be seen in Figure 8.6. The pointer to the next block record is not shown because it is encoded as part of the GC Info word, as described further in Section 8.2.1.
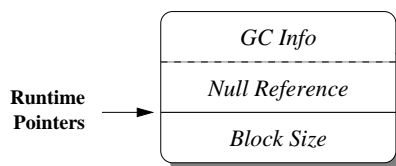
Figure 8.6: Block Records layout.

8.2 Allocator Implementation

This section describes the heap memory allocator implementation. It does not cover all the details behind the implementation, but gives the reader a broad idea about how it works. The allocator has no great innovations if compared to similar implementations for other language runtime implementations.

8.2.1 GC Info Word

Every heap object has a two-word internal header. One of the words holds a reference to the class of a Java object or a null reference for non-Java objects. The other word is the *GC Info* word, or an object specific value that enables the localization of the GC Info word inside it.

As the name implies, the GC Info word holds garbage collection information associated to each particular object. But it also holds information about the object monitor and heap flags as well[1]. Figure 8.7 shows the contents of the GC Info word for each object type.
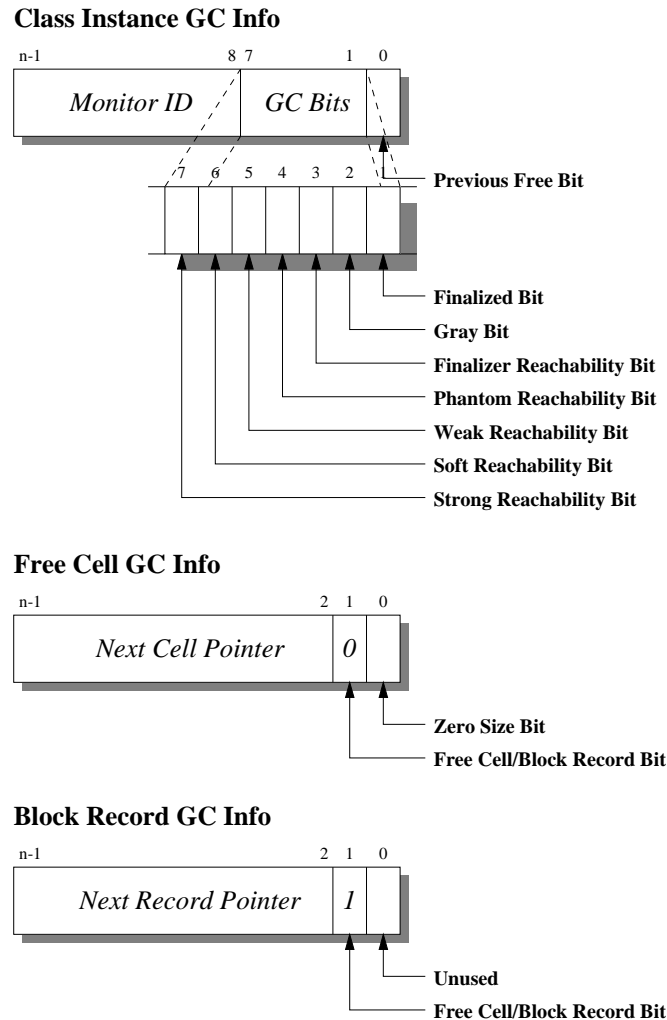
Figure 8.7: GC Info bits for each heap object.

The GC Info for Java class instances can be divided in three parts. The first part is composed of bit 0 which is set by the heap allocator to indicate that the area immediately preceding the object is a free cell. That information is used to merge two consecutive free blocks when an object is garbage collected. The second part goes from bit 1 to bit 7 and is information used by the garbage collection algorithm to store the state of the object. Bit 1 is used to mark objects who have been finalized. Bit 2 is a *gray* bit required by incremental garbage collection. Bits 3 to 7 makes up the the reachability for the associated object. Bit 3 is set whenever the object can become live through the execution of another object finalizer method. Bits 4 to 7

---

[1]The name *GC Info* may not be so appropriate. It was kept in this documentation because, since early implementation, it was used repeatedly in the source code.

are set according to the reachability type for the object, if none of these bits is set then the object is unreachable. Garbage collection is treated on Chapter 9.

The GC Info for free cells contains two flag bits and a pointer. The pointer points to the next free cell in the linked list of fixed size free cells. That linked list is reachable by the free cell cache table. Since all free cells are word aligned the least two bits (three bits for 64-bit systems) are always zero and their storage can be reused. One of these bits (bit 1) is used to distinguish free cells from block records. The other bit (bit 0) is set to identify empty free cells.

The GC Info for block records is similar to the GC Info for free cells except that bit 0 is unused and the pointer points to the block record of the next heap memory block in the heap linked list.

8.2.2   Allocation Procedure

This section describes the heap allocation procedure. Whenever a Java class is instantiated storage for it must be obtained from the garbage-collected heap. As seen before, the size of the storage area depends of the class being instantiated. The procedure herein described allocates an area given that its size is already provided.

First we describe the global variables required by the heap implementation. These variables comprise a pointer to the block record associated to the head memory block of the heap linked list; a free cell cache indexed by size for small objects; and a free cell cache for big objects (usually big objects are considered to have size greater than 1024 words).

```
blockrecord heaptop
freecell smallcache[SMALL_SIZE]
freecell bigcache
```

Next we provide a helper function that, given a free cell already removed from the free cell cache, adjusts the area size to the required size. This occurs because usually the search for an small area may demand fragmenting a larger one. The remains of the area are inserted back into the free cell cache. If the area has the expected size then the object placed right after the free area is modified to reflect the fact that its preceding area is not free anymore.

```
void ReclaimTail(freecell f, integer size)
    integer freesize ← f.size
    if freesize = size
        (f+size).previous_free ← false
    else
        f.size ← size
        f ← (f+size)
        size ← freesize-size
        f.size ← size
        if size < SMALL_SIZE
            f.next ← smallcache[size]
            smallcache[size] ← f
        else
```

```
f.next ← bigcache
bigcache ← f
```

The following two procedures are used to search for a free area in both small and big free cells caches. Attention should be given to the fact that free cells greater than the required size in one word cannot be used to allocate the associated object. This is denied because that would leave a one word free area which cannot be used to represent a heap object. If such restriction is obeyed and an object is found, it is removed from the cache, its size is adjusted and it is returned to the caller routine. If no appropriate free cell can be found in the caches, a null pointer is returned.

```
freecell SearchSmall(integer size)
   if size < SMALL_SIZE
      integer i ← size
      if smallcache[i] ≠ null
         freecell f ← smallcache[i]
         smallcache[i] ← f.next
         ReclaimTail(f, size)
         return f
      for i in size+2 to SMALL_SIZE-1 do
         if smallcache[i] ≠ null
            freecell f ← smallcache[i]
            smallcache[i] ← f
            ReclaimTail(f, size)
            return f
   return null


freecell SearchBig(integer size)
   if bigcache ≠ null
      if bigcache.size = size or bigcache.size ≥ size+2
         freecell f ← bigcache
         bigcache ← f.next
         ReclaimTail(f, size)
         return f
      else
         freecell prev ← bigcache
         freecell f ← prev.next
         while f ≠ null
            if f.size = size or f.size ≥ size+2
               prev.next ← f.next
               ReclaimTail(f, size)
               return f
            prev ← f
            f ← prev.next
   return null
```

The main allocation procedure is shown below. First it tries to search for a free cell in both small and big caches. If a free cell is not available then is requires the underlying operating

system to allocate fresh memory. The size of the memory area to be allocated is the smallest multiple of the memory page size that is big enough to hold the free cell (size), its header (2 words), a block record (3 words), and still does not leave a one word space left blank (2 words). The new memory block is inserted into the heap linked list and extra space is inserted into the free cell caches.

```
freecell Allocate(integer size)
    freecell f ← SearchSmall(size)
  if f = null
     f ← SearchBig(size)
    if f = null
       integer memsize ← OSpagealign(2+size+2+3)
       freecell f ← OScommit(memsize)
       if f = null
          return null

       blockrecord b ← (f+memsize-1)
       b.size ← memsize
       b.next ← heaptop
       heaptop ← b

       f ← (f+2)
       f.size ← memsize-2-3
       f.next ← null
       ReclaimTail(f, size)
  return f
```

8.2.3   Deallocation Procedure

This section details the heap deallocation procedure, it does the reverse of the allocation procedure. It reclaims used area, merging adjacent free cells, and inserts the new area in the free cell cache.

   The following routine searches and removes a particular free cell from the free cell caches. Free cell caches are implemented as simple linked lists.

```
boolean RemoveFree(freecell f)
    integer size ← f.size
    if size < SMALL_SIZE
       if smallcache[size] ≠ null
          if f = smallcache[size]
             smallcache[size] ← f.next
             return true
          else
             freecell prev ← smallcache[size]
             freecell p ← prev.next
             while p ≠ null
                if p = f
                   prev.next ← p.next
                   return true
```

```
                  prev ← p
                  p ← prev.next
            else
               if bigcache ≠ null
                  if f = bigcache
                     bigcache ← f.next
                     return true
                  else
                     prev ← bigcache
                     p ← prev.next
                     while p ≠ null
                        if p = f
                           prev.next ← p.next
                           return true
                        prev ← p
                        p ← prev.next
         return false
```

The deallocation procedure is shown below. First it checks if the area immediately preceding the area being deallocated is free, it does that in order to maximize free cells and thus avoid fragmentation. If the preceding area is free, it is removed from the free cell cache and merged with the object being reclaimed. The same thing occurs for the area immediately following the object; if it is free, it is removed from the free cell cache and merged. At last if inserts the resulting free cell into the caches, and updates the GC Info of the following object to record that the preceding area is a free cell.

```
void Deallocate(object o)
   integer size ← o.size

   if o.prev_free
      freecell f ← o
      integer prevsize ← (f-3).size
      f ← f-(2+prevsize)
      o ← f
      size ← size+2+prevsize
      RemoveFree(f)

   freecell f ← o+size+2
   if f.class = null
      if f.is_free
         size ← size+2+f.size
         RemoveFree(f)

   f ← o
   f.size ← size
   if size < SMALL_SIZE
      f.next ← smallcache[size]
      smallcache[size] ← f
   else
      f.next ← bigcache
```

```
        bigcache ← f

    o ← f+size+2;
    o.prev_free ← true
```

8.2.4   Heap Traversal Procedure

This section presents the procedure for walking through the garbage-collected heap. This procedure is required by the garbage collector to visit all objects, usually called when it is operating in incremental mode or collecting the garbage.

This first routine returns the reference to the first object of the heap. From the heap linked list head block record is starts scanning current block skipping free cells until it encounters a Java object or the block record. In the last case it continues the scan in the next memory block.

```
object HeapStart()
    blockrecord b ← heaptop
    object o ← (b+1-b.size+2)
    while o.class = null
      freecell f ← o
      if f.is_free
        o ← (f+f.size+2)
      else
        blockrecord b ← f.next
        if b = null
          return null
        o ← (b+1-b.size+2)
    return o
```

The following procedure is used to proceed to the next heap object given the current one. What it does is jumping the area of the current object and searching for the next object in memory blocks skipping free cells, as described above.

```
object HeapNext(object o)
    object class ← o.class
    if class = Class or class = MethodText
      o ← (class.tail_pointer+3)
    else
      integer size ← class.instance_size
      if class.dimensions ≠ 0
        size ← size+o.length*class.array_width+1;
      o ← (o+size+2)
    while o.class = null
      freecell f ← o
      if f.is_free
        o ← (f+f.size+2)
      else
        blockrecord b ← f.next
        if b = null
```

```
            return null
        o ← (b+1-b.size+2)
    return o
```

8.3   Thread Stacks

At run-time, each started Java thread has its own *thread stack* allocated in the virtual machine address space using underlying platform primitives. In our implementation, each Java thread is implemented as a native thread for simplicity. At first, we decided not to implement *green-threads* nor any other resource-aware alternate strategy, though, in the future, they may be incorporated with reasonable ease.

Different from other Java runtime implementations, our implementation stores Java stack frames in the native thread stack interleaved by native frames. The detection of stack overflow is not done explicitly on each method call, but captured in a native stack overflow signal handler, or whatever similar mechanism available in the underlying platform. Even though detecting stack overflow using this approach eliminates method calls overhead, it revealed limitations of our exception catching scheme when dealing with its assynchronous nature.

8.3.1   Stack Organization

The stack organization for our interleaved stack implementation is exemplified in Figure 8.8. Java stack frames are composed of five elements: the parameters, the return address, the previous frame pointer, a reference to the associated method text and the local variables. During native calls, the topmost Java frame pointer is stored in a *thread local storage* in order to be retrieved back when Java code starts executing again.

When the native implementation decides to do a Java method callback, it must create a pseudo frame in the stack used to separate the native and Java areas in the stack. The pseudo frame, similar to a Java frame, has parameters provided from JNI calls, a return address, and a frame pointer backup. In addition, it has a null reference, instead of a method text reference, used to identify the pseudo frame; a previous Java frame pointer read from the thread local storage during the callback; and a area reserved for saving registers assumed not to be written by callees using the standard native protocol.

8.3.2   Stack Traversal Procedure

In this section we provide the procedure for traversing a thread stack. The thread stack traversal is required when printing stack traces, collecting garbage collection root references, and discovering caller classes. This last operation is required by the Java security model.

The stack traversal procedure is fairly simple, and is below:

```
boolean TraverseStack(address top_frame, boolean process(address, address))
    address frame ← top_frame
    while frame ≠ null
```

Local Variables
Method Text
Previous Frame
Return Address
Parameters
Callee-Save Registers
Prev Java Frame
Null Reference
Previous Frame
Return Address
Parameters
Local Variables
Method Text
Previous Frame
Return Address
Parameters
Local Variables
Method Text
Previous Frame
Return Address
Parameters
Callee-Save Registers
Null Pointer
Null Reference
Previous Frame
Return Address
Parameters

Topmost Java Frame

Callback Pseudo Frame

Native Stack Organization

Native Method Java Frame

Ordinary Java Frame

Bottom Pseudo Frame

Native Stack Organization

stack growth direction
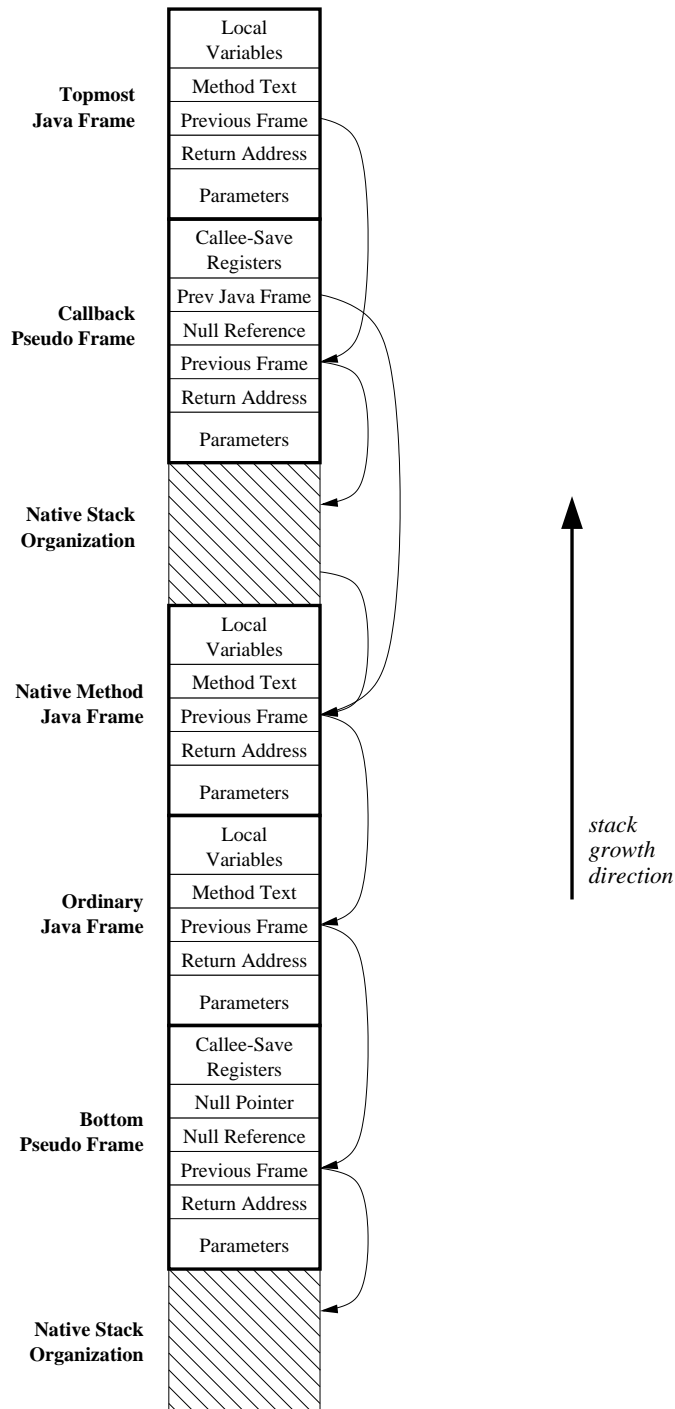
Figure 8.8: Stack organization.

```
            address ret_address ← frame.ret_address
            frame ← frame.previous
            object method_text ← frame.method_text
            if method_text = null
                frame ← frame.java_previous
            else
                boolean continue ← process(frame, ret_address)
                if not continue
                    return false
        return true
```

To traverse the thread stack we need two parameters, the topmost Java frame pointer, which can be read from the thread local storage, and an action function. Then we start visiting Java stack frames skipping pseudo frames until we find the bottom pseudo frame. For each Java frame the action function is called passing as parameter the frame pointer and the return address of its callee. The stack traversal continues while the action function returns false or the bottom pseudo frame is reached. It returns true if definately all Java frames have been processed.

8.3.3   Stack Overflow Detection

As said before, in our implementation the stack overflow detection is done by handling a native stack overflow signal, or a similar mechanism in the underlying platform. In the signal handler code, a new instance of class *StackOverflowError* must be created and thrown in the context of the currently executing method.

However, since native stack overflow may occur arbitrarily, it is possible that the *program counter* at that time assumes any value inside the address space of the method text. This turns out to be a problem because the exception catching mechanism we have implemented (described in Section 6.3) requires that exceptions occur at some prescribed points. Therefore it may not be possible to determine the exception catching entry point to transfer control.

The solution we provided to this problem is awkward. Whenever we cannot determine the exception catching entry point for a given program counter value, in the context of the topmost java frame, we throw the exception in the caller frame instead. Throwing the exception in the caller frame can always be done because every call site either has an associated exception catching entry point or does not catch exceptions at all.

The adoption of this scheme dictates that our implementation may sometimes ignore *StackOverflowError* exception windows. It was our belief that rarely a program catches stack overflow exceptions. However this was a fallacy. Although programs that catch stack overflow exception a rare, many programs catch all types of exception usually to free resources (*try*/*finally* construct). Ignoring these handlers may lead to unpredictable program behavior (e.g. object locks being hold when the stack overflow occurs in a synchronized method).

A better approach is to avoid the generation of code that may overflow the stack in the program points for which the exception catching entry point cannot be determined. We intend to use this approach in the future.

8.4   Monitor Implementation

In our implementation, each class instance has in its GC Info word 24 bits (56 bits for 64-bit machines) that are reserved to implement its associated monitor. These 24-bits are used to index a monitor table where the object monitor lies. Since most objects do not use their monitor, storing an index in its header — rather than the whole monitor — saves storage space. However, a monitor must be allocated to the object whenever it is first used, and that requires precise synchronization. Monitors are released, becoming available to others, when the object is garbage collected.

```
union monitor
    OSmutex mutex
    integer next

integer top_id ← 1
integer recycled ← 0
monitor monitors[MAX_MONITORS]
```

The monitor table is a table of platform dependent monitors. The monitor at index 0 is a special monitor used to synchronize during monitor allocation. The management of free monitors is done using a free monitor linked list. Since the monitor table is statically allocated in the data segment of the virtual machine, we always try to reuse free monitors at the bottom of the table prior to allocating a monitor on the top of the table. This is done in order to avoid the commitment, i.e. physical allocation, of the associated memory pages by the underlying operating system.

The procedure for entering an object monitor — including its compulsory allocation — is shown below:

```
boolean MonitorEnter(object o)
    integer id ← o.monitor_id
    OSlock(monitors[id].mutex)
    if id = 0
        id ← o.monitor_id
        if id = 0
            if recycled ≠ 0
                id ← recycled
                recycled ← monitor[recycled].next
            else
                if top_id = 0
                    OSunlock(monitors[0].mutex)
                    return false
                id ← top_id
                top_id ← (top_id+1) mod MAX_MONITORS
            o.monitor_id ← id
            OSinit(monitors[id].mutex)
        OSunlock(monitors[0].mutex)
        OSlock(monitors[id].mutex)
    return true
```

At first, it locks the monitor using the object monitor id. If no monitor has been allocated to the object it will lock monitor at entry 0. After locking, it tests the monitor id before locking. If it was non-zero then the object has a monitor allocated already. Otherwise, it reads the monitor id again to check if the monitor has been allocated concurrently while it was blocked by the lock operation. If this is the case, it releases monitor at index 0 and locks monitor at the specified index. If no monitor has been associated to the object yet, then a new monitor is allocated, initialized, the monitor at index 0 is released, and the new monitor is locked.

The procedure for exiting an object monitor is trivial, simply unlocks the associated monitor.

```
void MonitorExit(object o)
   integer id ← o.monitor_id
   OSunlock(monitors[id].mutex)
```

When an object is garbage collected, the monitor that has been associated to it, if any, is recycled. This is done synchronizedly using the monitor at index 0.

```
void MonitorRecycle(object o)
   integer id ← o.monitor_id
   if id ≠ 0
      OSlock(monitors[0].mutex)
      monitors[id].next ← recycled
      recycled ← id
      OSunlock(monitors[0].mutex)
      o.monitor_id ← 0
```

These routines where crafted to implement monitors efficiently while still preserving the correct behavior for all concurrency possibilities.

8.5   JNI Implementation

Our runtime implementation has full support for the *Java Native Interface* (JNI). JNI is used to instantiate the virtual machine from native programs as well as implementing native methods. We highlight the points we judge to be important:

No Reference Handles In our implementation, JNI object wrapper types are direct references to their target objects. It is a common implementation to use one level of indirection when implementing the wrapper types, so that the native program hold handles to the objects. These handles are indices to a table that holds the reference to the object. Representing wrapper types as direct references speeds up the implementation of most operations though it makes difficult the implementation of JNI weak references. JNI weak references are references automatically created by the virtual machine when an object becomes weakly reachable. In the handle based implementation this means writing a null reference in the associated table entry. In our implementation however, it

is not possible to clear each reference because they became available to the native program which may have copied it to other locations. Luckly, the only JNI call that requires checking if a weak reference has been cleared is `IsSameObject`. During this call, if one of the parameters is null and the other is not, we search all internal reference tables to check if the non-null reference is not strong and also not present in the weak reference table. In this case, the routine will be slower than usual. The policy we recommend and follow is to minimize the use and the functionality of the native methods. Native methods where not meant to improve performance, but to implement semantics not available in the bytecode.

String Manipulation in Java All string manipulation calls of the JNI are done by invoking the appropriate Java implemented methods of the class `String`. Including the implementation of the intern set of strings (see `String.intern()`).

Local References Allocation/Deletion is Fast Using Stack Protocol Local references are references made available to the native program during the liveness of a native method execution. They need to be recorded as live to implement proper garbage collection. In our implementation they are recorded in an array of references. New references are inserted at the end of the array. The deletion of local references requires scanning backwards the whole array and moving the reference at the top of the stack to the freed entry. Therefore deleting references in the reverse allocation order is the better choice since they will be found right in the first entry.

Arguments Array is Inefficient Invoking Java methods passing arguments through an array rather then the stack is inefficient. These calls are mapped to their equivalent stack based version requiring parsing method descriptors to find out which array elements uses one or two words.

8.6   JVMDI and JVMPI Support

In our design, we have focused on a high-performance end-used JVM, therefore we provide no implementation for the standard *Java Virtual Machine Debugging Interface* (JVMDI) and *Java Virtual Machine Profiling Interface* (JVMPI). Neither the support nor the impact of the implementation of these interfaces were part of our requirements when we set up our goals. Currently, the lack of knowledge about these interfaces requirements prevents us from measuring the difficulty of incorporating them in our implementation.

Chapter 9

The Garbage Collector

This chapter describes the *garbage collector* (GC) implemented as part of our JVM runtime. At first, we discuss the desired features of our garbage collection scheme, as they were defined during its design. In the sequence, we identify the GC runtime requirements, and describe our implementation. At last, we focus on the improvements to the current scheme.

9.1   Desired Features

The definition of the behavior and desired features of the garbage collector has a great impact over the runtime design (and the opposite is also true). For instance, the implementation of an accurate garbage collector is only possible if the available runtime provides support for discovering references in stack frames and objects. On the other hand, if the available GC does not handle concurrency, the runtime must provide proper synchronization, which may degrade performance. Therefore, the best strategy is design the GC having the runtime in mind, and vice-versa.

The features we have defined for our garbage collection scheme are listed below. All of them where conceived based on a GC/runtime co-design. Some of them are not yet available in our preliminary implementation.

Accurate The set of memory locations that contains object references can be obtained accurately. No pointer aliases will prevent unreachable objects from being collected, thus eliminating GC memory leaks.

Cooperation Without Run-Time Penalty Mutators, i.e. user threads, cooperate with the GC without run-time penalty (e.g. no reference counting).

Pinned Objects Objects cannot be moved to alternate memory locations. They must have the same address and use the same storage during all their lifetime. This denies the use of a copying collection scheme.

Incremental The garbage collection may be suspended and resumed, interleaved with mutator execution. The GC executes as a separate thread or co-routine.

Concurrent Mutators may execute during garbage collection.

Generational Objects may be divided in generations. Each generation deserves more or less
       GC attention based on the probability of its survival to yet another collection.

9.2   Runtime Requirements

In order to perform the GC task, the runtime must provide support to, or interact with, the
garbage collector. The runtime is required to cooperate with the GC exactly in two tasks:
to discover references inside objects and stack frames; to access each thread stack synchro-
nizedly.

    In order to implement an accurate garbage collector, the runtime generates extra GC infor-
mation embedded in meta class and method text objects. With that information the garbage
collector can identify roots and traverse the whole object graph. We have already described
how that information is computed and represented in Chapter 7 and Chapter 8. In Section 9.3,
we show how that information can be used to implement the garbage collection.

    At certain prescribed times, each thread must provide safe access to the contents of its
stack. Also, the GC must be able, at that time, to suspend the thread if desired. In our imple-
mentation, this runtime requirement is implemented non-preemptively; i.e., prior to proceed,
the GC must wait for each thread until it does a runtime callback. This occurs because we
renounce to use thread execution contexts in order to simplify and increase the runtime porta-
bility. However, the arguments we present to explain why this scheme is effective are similar
to the arguments we have presented for assynchronous exceptions in Section 6.6.

9.3   Implementation Details

This section provides details about our current garbage collection implementation. The current
implementation is simple and straightforward. It works in a *Mark-and-Sweep*[JL96, §2.2]
fashion, and is non-incremental nor concurrent. Before garbage collection, all threads are
suspended to avoid problems when collecting references in their stacks and when accessing
the heap.

    The *mark* phase is done in a mixed-mode between iterative and depth-first recursive. An
iterative mark phase works by doing multiple passes on the heap, marking grayed objects
and coloring their children, until all reachable objects are marked. A depth-first recursive
mark phase visits each root object and does a depth-first traversal until the complete reachable
object subgraph is marked. We adopt a mixed-mode mark phase, in which it iterates doing
a bounded depth-first traversal, because the convergence for the iterative method is too slow,
and the stack size for a pure recursive depth-first method can be prohibitive. As dictated
by the Java language, marking an object is not simply stating that it is reachable, one must
provide complete reachability information including its strength (strongly, softly, weakly, and
phantom) and its type (direct or finalizer)[GJS96, §12.6].

The *sweep* phase is simply a traversal of the heap, reclaiming the storage from the unreachable objects. For the Java garbage collector, this also means scheduling unreachable unfinalized objects to finalization, and clearing weak references.

The following routine marks an object based on a given reachability. It returns true if the object reachability has changed. Children are marked if the maximum depth was not yet reached.

```
boolean Mark(object o, set reachability, integer depth)
   if o ≠ null
      reachability ← reachability ∪ o.reachability
      if reachability ≠ o.reachability
         o.reachability ← reachability
         o.gray ← true
         if depth = MAX_DEPTH
            return true
         else
            return MarkChildren(o, depth)
   return false
```

The next routine marks the children of a gray object. Based on the object class (and its superclasses), it is possible to determine which memory locations inside it are references; they are marked with the same reachability as the current object. Array, meta class and method text objects are treated specially since they provide extra references. At last, we mark the target reference of each soft or weak wrapper object using its corresponding reachability.

```
boolean MarkChildren(object o, integer depth)
   boolean changed ← false
   if o.gray
      o.gray ← false
      set reachability ← o.reachability

      changed ←ᵒʳ Mark(o.class, reachability, depth+1)

      object class ← o.class
      while clazz ≠ null
         object[] refs ← o@class.instance_refs_offset
         for integer i in 0 to class.instance_refs_count-1
            changed ←ᵒʳ Mark(refs[i], reachability, depth+1)
         class ← class.superclass

      class ← o.class

      if class = "Class"
         object[] refs ← o@class.dispatch_table_offset
         for integer i in 0 to class.dispatch_table_entries-1
            changed ←ᵒʳ Mark(refs[i], reachability, depth+1)
         refs ← o@class.static_refs_offset
         for integer i in 0 to class.static_refs_count-1
            changed ←ᵒʳ Mark(refs[i], reachability, depth+1)
         changed ←ᵒʳ Mark(o.loader, reachability, depth+1)
```

```
            changed ←→ᵒʳ Mark(o.elementClass, reachability, depth+1)
            changed ←→ᵒʳ Mark(o.superClass, reachability, depth+1)
            for integer i in 0 to o.interfaces_count-1
               changed ←→ᵒʳ Mark(o.interfaces[i], reachability, depth+1)

         if class = "MethodText"
            changed ←→ᵒʳ Mark(o.declaring_class, reachability, depth+1)
            for integer i in 0 to o.references_count-1
               changed ←→ᵒʳ Mark(o.references[i], reachability, depth+1)

         if o.class.dimensions > 0
            if o.class.name[1] = 'L' or o.class.name[1] = '['
               object[] refs ← o@class.instance_size
               for integer i in 0 to o.length-1
                  changed ←→ᵒʳ Mark(refs[i], reachability, depth+1)

         if o.class.is_soft
            changed ←→ᵒʳ Mark(o.weak, { SOFTLY } ∪ (reachability ∩ { FINALIZER }), depth+1)
         if o.class.is_weak
            changed ←→ᵒʳ Mark(o.weak, { WEAKLY } ∪ (reachability ∩ { FINALIZER }), depth+1)
      return changed
```

The following routine marks all references live in the stack of a particular thread. It traverses the stack using return addresses to determine the live reference variables at the time each call was performed. Also the method text associated to each stack frame is marked.

```
      boolean MarkStack(address frame)
         boolean changed ← false

         boolean MarkFrame(address frame, address ret_address)
            changed ←→ᵒʳ Mark(frame.method_text, { STRONGLY }, 0)
            byte[] lives ← frame.method_text.lives[ret_address]
            integer i ← 0
            while lives[i] ≠ 0
               changed ←→ᵒʳ Mark(frame[lives[i]], { STRONGLY }, 0)
               i ←→⁺ 1
            return true

         TraverseStack(frame, MarkFrame)
         return changed
```

The main garbage collection routine is presented next. It may be divided in three phases: initialization, mark and sweep.

The initialization phase consists of traversing the entire heap reseting the reachability and clearing the gray bit for all objects.

```
      void GC(javavm jvm)

         /* Reset objects */
```

```
object o ← HeapStart()
while o ≠ null
   o.reachability ← ∅
   o.gray ← false
   o ← HeapNext(o)
```

The mark phase is subdivided in three parts: marking roots, marking finalizer reachable, and iterating. The marking roots part marks all direct references from the JVM, and its threads, to the heap. The marking finalizer reachable part marks all objects not yet finalized as finalizer reachable (by themselves). Finally, the iterating part marks, using the according reachability, all remaining objects in the object graph.

```
boolean changed ← false

/* Mark roots */
changed ←or Mark(jvm.system_thread_group, { STRONGLY }, 0)
for integer i in 0 to jvm.globals_count-1 do
   changed ←or Mark(jvm.globals[i], { STRONGLY }, 0)
for integer i in 0 to jvm.weaks_count-1 do
   changed ←or Mark(jvm.weaks[i], { WEAKLY }, 0)
jnienv env ← jvm.envs
while env ≠ null
   changed ←or Mark(env.thrown, { STRONGLY }, 0)
   changed ←or Mark(env.thread, { STRONGLY }, 0)
   changed ←or MarkStack(env.top_javaframe)
   jniframe frame ← env.top_jniframe
   while frame ≠ null
      for integer i in 0 to frame.entry_count-1 do
         changed ←or Mark(frame.entries[i], { STRONGLY }, 0)
      frame ← frame.previous
   env ← env.next

/* Mark finalizer reachable */
o ← HeapStart()
while o ≠ null
   if not o.finalized
      changed ←or Mark(o, { FINALIZER }, 0)
   o ← HeapNext(o)

/* Iterate marking all heap */
while changed
   changed ← false
   o ← HeapStart()
   while o ≠ null
      changed ←or MarkChildren(o, 0)
      o ← HeapNext(o)
```

The sweep phase is also subdivided into three parts: reclaiming unreachable objects, queueing unfinalized objects for finalization, and clearing weak references. The reclaiming

part traverses the heap looking for unreachable objects (reachability set is empty) and reclaim their storage and monitor. Unreachable objects are known to be already finalized because, if not, their reachability set would not be empty. The queueing unfinalized objects part traverses the heap looking for finalizer reachable objects whose strength is at most phantom. Those objects are marked as finalized and enqueue for finalization (they then become strongly reachable again). Finalization occurs when the method `Runtime.runFinalization()` is invoked by the user, or by the runtime under low memory conditions. At last, weak reference wrappers whose target reference has a reachability weaker than required are cleared and enqueued (see `ReferenceQueue` class in the standard API).

```
/* Reclaim unreachable */
o ← HeapStart()
while o ≠ null
  object moribund ← o
  o ← HeapNext(o)
  if moribund.reachability = ∅
    RecycleLock(moribund)
    Deallocate(moribund)


/* Enqueue unfinalized */
o ← HeapStart()
while o ≠ null
  set reachability ← o.reachability ∩ { STRONGLY, SOFTLY, WEAKLY, PHANTOM }
  if reachability = ∅ or reachability = { PHANTOM }
    if not o.finalized
      EnqueueForFinalization(o)
      o.finalized ← true
  o ← HeapNext(o)


/* Clear weak references */
for integer i in 0 to jvm.weaks_count-1 do
  object weak ← jvm.weaks[i]
  if weak ≠ null
    if weak.reachability ∩ { STRONGLY, SOFTLY } = ∅
      jvm.weaks[i] ← null
o ← HeapStart()
while o ≠ null
  if o.class.is_soft
    if o.weak ≠ null
      if o.weak.reachability ∩ { STRONGLY } = ∅
        o.weak ← null
        EnqueueReferenceObject(o)
  if o.class.is_weak
    if o.weak ≠ null
      if o.weak.reachability ∩ { STRONGLY, SOFTLY } = ∅
        o.weak ← null
        EnqueueReferenceObject(o)
  o ← HeapNext(o)


/* End of GC */
```

9.4   Future Improvements

The most important improvement our current garbage collection asks for is the incorporation of a generational strategy. Although generational collection wastes more memory, it reduces significantly the pause time of each collection. This has been proven to work on other Java runtime implementations[Sun98, FKR$^+$99].

An alternate possibility, is implementing a concurrent collector that runs in a separate thread. It is a good scheme specially if the JVM is targeted to a multiprocessor system. However, care must be taken to implement it correctly, without increasing execution contention.

Chapter 10

Automatic Machine Generation

This chapter covers automatic machine generation. Automatic machine generation consists of ahead-of-time linkage (including JIT compilation) of core libraries, which are embedded in the runtime. The machine generator simulates a JVM heap as it loads and links the classes specified in a configuration file. When all activities finish, an assembly file reflecting the heap image placement, according to the specified target architecture, is output.

There are basically two reasons that motivate us to implement the automatic machine generator. First, as seen in Chapter 8, many runtime tasks are implemented in Java, and some of them are required to be promptly available upon machine startup. Second, the off-line embedding of core libraries speeds up the machine bootstrap that occurs every time it is started up.

Off-line embedding core libraries is a technique that succeeds based on the premiss that core classes are not supposed to be replaced by users, nor will need to change before the next JVM release.

10.1   Static Heap Image

The *static heap image* is a heap image reflecting some ahead-of-time link-time activities symbolically performed and output by the machine generator. As expected, the static heap layout must conform with the memory heap layout described in Chapter 8.

The objects that compose a static heap image are:

- Meta class objects, instances of `Class`, representing the classes embedded.

- Method text objects, instances of `MethodText`, representing methods declared in those classes.

- String objects, instances of `String`, implementing string literals directly referenced from method texts.

- Array of char objects, instances of `char[]`, used to store the contents of string objects.

Two details must be highlighted about the static heap image. First, method text objects are translated using the appropriate back-end for the underlying architecture being targeted by the machine generator. Second, strings and their associated array of chars must be placed contiguously, so that the runtime can identify that association during the heap initialization procedure (see Section 10.4).

The static heap image must provide imported and exported symbols information in order to be linked with the C runtime by the platform linker. It exports a single symbol, `_heapstart_`, that is used by the runtime to locate the static heap in its address space. The symbols imported by the static heap image are exactly the labels for runtime callback entry points, namely:

- Exception throwing: `_athrow_`.

- Long integer division and remainder: `_ldiv_` and `_lrem_`.

- Class initialization: `_init_`.

- Instance and array instantiation: `_newinstance_` and `_newarray_`.

- Synchronization primitives: `_lock_`, `_unlock_` and `_islocked_`.

- Type testing: `_subtypeof_` and `_comptypeof_`.

- Interface method lookup: `_imlookup_`.

- Native method call: `_ncalll_`, `_ncallz_`, `_ncallb_`, `_ncallc_`, `_ncalls_`, `_ncalli_`, `_ncallj_`, `_ncallf_`, `_ncalld_` and `_ncallv_`.

In our current implementation, the static heap image is generated in a data segment and, at run-time, its contents are modified as the machine executes. This means that only a single JVM instance can be created in the context of a process, providing a limited implementation of the `JNI_CreateJavaVM` JNI call[1]. A workaround to this limitation can be done using dynamic mapping of process segments by copying the original heap layout from the executable binary to a different memory location every time a JVM is created. For platforms that do not support dynamic segment mapping, this could be done explicitly by the application. In both cases, some patching is required to update the runtime callback addresses.

10.2   Machine Generation Configurations

The automatic machine generation is a configuration driven process. The machine generator executes and produces a static heap image based on a input configuration file. The configuration file defines the behavior of the machine to be generated by providing information about the classes to be off-line embedded in it.

A machine generation configuration comprises the following information:

---

[1]This is a common limitation in most JVM implementations including Sun's reference implementation.

- The path list from where class files will be read during the generation process (usually referenced to as *CLASSPATH*).

- The name of the classes to be embedded into the static heap image. In addition each class has two attributes:

    1. An attribute indicating if the class must be linked off-line.
    2. An attribute indicating if meta class information must be included beforehand.

In a configuration file, the list of classes can partitioned into two sets: *behavior defining classes* and *dependent classes*. Behavior defining classes are classes that implement a particular JVM operation following some desired strategy or algorithm. Dependent classes are more generic classes used by behavior defining classes to complete or help in their implementation. Since some Java implemented extensions of the runtime must be promptly available upon machine startup, some dependent classes — in conjunction with behavior defining classes — need to be embedded avoiding chicken-egg problems (e.g. linking a class that is part of class linkage implementation).

However, the detection of the minimum set of dependent classes required to bootstrap the machine is a difficult task. It is difficult because we have to find the closure set of the methods reachable from all methods directly called by the runtime during bootstrap. We do that in order to ensure that all execution paths required to bootstrap the machine are already present in the binary executable before the bootstrap (otherwise the machine shuts down unpredictably). Actually, the difficulty is not in finding the whole closure set, which can be done conservatively, but in finding a subset of the closure set that is an approximate superset of the minimum set of methods possibly executed during bootstrap. Automatic conservative detection of bootstrap dependencies generates a huge static heap image. This has impact in the amount of memory consumed by the JVM[2], sometimes wasted by the storage of classes linked beforehand and never used.

In our configurations, we have decided to detect bootstrap dependencies by hand. This task is repetitive and time-consuming but, once the core libraries basic structure is kept, it will need few revisions. As foreseen in our design (see Chapter 3), we have written two machine generation configurations: the Thin-Client Client JVM configuration and the Standalone Client JVM configuration.

10.3   Machine Generator Functionality

The machine generator is a tool that uses a configuration to generate a static heap image to be incorporated by the runtime. The tool is very simple, it simulates the bootstrap loading and linking activities of the JVM using a simulated heap; when all classes in the configuration are processed, it uses the target back-end to produce an assembly output.

The simulation of the JVM bootstrap is very simple. Reading classes from the CLASS-PATH provided in the configuration file, it allocates storage in the simulated heap for each meta class being loaded, their method texts, string literals and associated arrays of chars.

---

[2]It also has a strong impact in the machine generation time. Luckly, machines are generated once.

Two tables play an important role in machine generation process: the *bootstrap class loader table* and the *string intern set table*. The former table records the classes already loaded during generation. The latter table is a map between string literals and simulated heap allocated string instances, avoiding the occurrence of duplicate string instances associated with the same string literal. Both tables are implemented internally by the generator — and not allocated in the simulated heap — even though they will be instantiated and initialized as soon as possible during the machine bootstrap. They cannot be allocated into the simulated heap because their initialization requires calling their methods and bytecode interpretation is not supported during machine generation (we foresee bytecode interpretation as part of the future improvements to the generator, most of its complexity is due to native methods existence).

The internal representation of the simulated heap is symbolic. Instead of actually reserving storage for class instances, we retain the size required for each object in the heap. Meta class and method text instances have extra internal fields that store information about their link state and binary translation, respectively. This extra information is exactly the extra information stored in the instance headers or extended bodies as described in Chapter 8.

All the generation process is done through a Server JVM used to register, load, link and translate class files. Linkage errors during generation make the generator abort with a detailed message.

10.4   Heap Initialization Procedure

The heap initialization procedure, that must be done by the runtime during machine creation, is fairly simple. The initialization consists of a heap traversal executing one of three actions:

Construct and Load Class Meta class instance constructor (see `Class()`) is executed and the class is recorded to be loaded by the bootstrap class loader. Usually the meta class constructor does nothing but return, if the case, may be omitted.

Construct Method Text Method text instance constructor (see `MethodText()`) is executed. Usually the method text constructor does nothing but return, if the case, may be omitted.

Construct and Internalize String String literal constructor is executed using the subsequent array of chars as parameter (see `String(char[])`), afterwards the string is internalized (see `String.intern()`).

Chapter 11

Conclusions

This document has described an alternate implementation of the Java Virtual Machine. The most important feature of that implementation is its ability to externally hoist and cache link-time activities (specially JIT compilation) on a network based computer. It improves the performance of JIT produced code, decreases runtime overheads, and makes better use of hardware resources. The explanation goes beyond the solutions proposed to implement this innovative approach. It comprises:

- Techniques for detecting and caching repetitive link-time contexts.

- An alternate, off-line, bytecode verification procedure.

- The design and implementation of a Java specific intermediate representation. Its conversion from Java bytecodes, the manipulation tool, some analyses and transformation algorithms.

- A simple unified back-end for the Intel family of 32-bit processors.

- Runtime data organization on heap and thread stack. Including support for full-fledged stack traces on optimized compiled code.

- Accurate garbage collection requirements and implementation issues.

- Off-line embedding of core libraries in the virtual machine runtime.

There were two important simplification problems in our work. However, we noticed that they can be solved without significant impact on the overall system design. The alternatives, in both cases, speed up execution and save memory.

The first problem regards *lazy resolution*. In an early specification, we decided to adopt *eager resolution* in our preliminary implementation, as allowed by the JVM specification. Eager resolution is the premature resolution, at link-time, of classes, fields, and methods symbolically referenced by a class. On the other hand, lazy resolution states that those symbolic entities need only to be resolved on their first use during execution. We indeed verified that, using eager resolution, a large amount of link-time activities tends to concentrate on applications startup. Eager resolution generates a startup delay that should be avoided on short-lived

applications. The solution to this problem can be achieved by extending the IR to support lazy resolution operations. By adopting lazy resolution, context classes need not to be loaded on the LINK phase, saving memory.

The second problem regards *method compilation*. According to what has been described, all class methods are converted to IR during the LINK phase. In a similar manner, all methods are translated to machine code during the TRANSLATE phase. That means client JVMs have to wait for the compilation of all methods even if they need to execute just some of them. This batch compilation scheme introduces delay on link-time operations. The solution to this problem is tricky. Instead of using the compiled version of methods to initialize dispatch tables, the runtime uses synthetic versions. Each synthetic version is responsible for, synchronizedly, replacing itself by the actual version of the method. The TRANSLATE phase must then be modified to handle requests on a method granularity. Synthetic methods are usually smaller and save memory.

Due to time constraints, we have not implemented a mid-level optimizer. It is a key component of the system and is part of the future work.

Despite these problems, we have successfully met our goals. The prototype has been used to run real-world applications and benchmarks. It still lacks profiling and tuning but, as a first implementation, it exhibited surprisingly good performance. At least, it outstripped the interpreted implementation performance for many standard vendors.

Appendix A

Intermediate Representation Specification

The intermediate representation, for short *IR*, consists of a set of *IR opcodes*. An IR opcode defines a simple but semantically clear operation. Each IR opcode may have arguments, attributes and optionally provide a result. The arguments of an IR opcode are actually the result of other IR opcodes coupled to it. The attributes of an IR opcode extend its semantics. IR opcodes that provide a result are used as arguments by others. IR opcodes that do not provide a result define *IR statements*. An *IR program* is the sequence of IR statements that implements a particular Java method. This appendix gives details about the syntax of each IR opcode and the semantics of the operation it performs.

A.1   Grammar

The following *IR Grammar* defines syntatically how IR opcodes can be coupled to form an IR statement. An IR program is a sequence of IR statements and a virtually infinite set of registers. The IR is typed, which means that IR opcodes can only couple to and work with entities of the expected type.

The IR supports five types: signed 32-bit integers, signed 64-bit long integers, floats, doubles and object references. Floats and doubles are encoded and handled as the 32-bit and 64-bit IEEE standards. Whence on registers, those types may have an extended exponent[LY99, §3.3.2] which is a choice of interpretation. However, at some points, those extended values may need to be mapped to non-extended values (`fstrict`, `dstrict`). The reference type does not demand an encoding and is left unspecified.

The IR opcodes are partitioned into six sets, one for each type described above and one associated with IR statements. Each opcode may provide a result, and thus it may be classified according to the type of the result. If the opcode does not provide a result then it defines an IR statement.

Each IR statement is a tree. The root of the tree must be an opcode that does not provide a result and is represented in the IR Grammar by the S non-terminal. The rest of the tree is obtained by applying the rules present in the IR Grammar to the arguments of the root opcode. This happens until no more arguments are left for expansion.

The IR Grammar defines only part of the constraints which the IR is required to obey,

precisely those constraints that could be captured by syntax. Constraints regarding the IR semantics are exposed on next section when each opcode is revisited.

Syntax	*Page*
S: `ireceive(%i)`	171
S: `lreceive(%l)`	178
S: `freceive(%f)`	164
S: `dreceive(%d)`	159
S: `areceive(%a,#c)`	152
S: `ipass(`*I*`)`	171
S: `lpass(`*L*`)`	178
S: `fpass(`*F*`)`	164
S: `dpass(`*D*`)`	159
S: `apass(`*A*`)`	151
S: `call(`*A*`,#t)`	155
S: `callx(`*A*`,#t,@l)`	156
S: `ncall(#c,#s,#t)`	180
S: `ncallx(#c,#s,#t,@l)`	181
S: `iresult(%i)`	171
S: `lresult(%l)`	178
S: `fresult(%f)`	165
S: `dresult(%d)`	160
S: `aresult(%a,#c)`	152
S: `label(@l)`	174
S: `jump(@l)`	174
S: `ajump(#x,`*A*`,`*A*`,@l)`	151
S: `ijump(#x,`*I*`,`*I*`,@l)`	168
S: `iswitch(`*I*`,[$i,@l]...)`	173
S: `acatch(%a)`	150
S: `athrow(`*A*`)`	153
S: `ireturn(`*I*`)`	172
S: `lreturn(`*L*`)`	179
S: `freturn(`*F*`)`	165
S: `dreturn(`*D*`)`	160
S: `areturn(`*A*`)`	152
S: `vreturn()`	185
S: `idefine(%i,`*I*`)`	168
S: `ldefine(%l,`*L*`)`	176
S: `fdefine(%f,`*F*`)`	163
S: `ddefine(%d,`*D*`)`	158

S:	adefine**(**%a,*A***)**	150
S:	bstore**(***A*,#o,#v,*I***)**	155
S:	sstore**(***A*,#o,#v,*I***)**	185
S:	istore**(***A*,#o,#v,*I***)**	172
S:	lstore**(***A*,#o,#v,*L***)**	179
S:	fstore**(***A*,#o,#v,*F***)**	165
S:	dstore**(***A*,#o,#v,*D***)**	160
S:	astore**(***A*,#o,#v,*A***)**	153
S:	bastore**(***A*,*I*,*I***)**	154
S:	sastore**(***A*,*I*,*I***)**	184
S:	iastore**(***A*,*I*,*I***)**	168
S:	lastore**(***A*,*I*,*L***)**	175
S:	fastore**(***A*,*I*,*F***)**	162
S:	dastore**(***A*,*I*,*D***)**	157
S:	aastore**(***A*,*I*,*A***)**	150
S:	init**(***A*,#t**)**	170
S:	initx**(***A*,#t,@l**)**	170
S:	newinstance**(***A*,#t**)**	183
S:	newinstancex**(***A*,#t,@l**)**	183
S:	newarray**(***A*,*I*,#t**)**	182
S:	newarrayx**(***A*,*I*,#t,@l**)**	182
S:	lock**(***A*,#t**)**	177
S:	lockx**(***A*,#t,@l**)**	177
S:	unlock**(***A***)**	185
S:	readbarrier**()**	183
S:	writebarrier**()**	186
A:	getclass**(***A***)**	166
A:	aload**(***A*,#o,#v,#c**)**	151
A:	aaload**(***A*,*I***)**	149
A:	mlookup**(***A*,#i**)**	180
A:	imlookup**(***A*,*A*,#i**)**	169
A:	ause**(**%a**)**	154
A:	anull**()**	151
A:	aclass**(**$c**)**	150
A:	astring**(**$s**)**	153
I:	i2b**(***I***)**	166
I:	i2c**(***I***)**	166
I:	i2s**(***I***)**	167
I:	l2i**(***L***)**	174
I:	f2i**(***F***)**	161
I:	d2i**(***D***)**	156

```
I:  iadd(I,I)                                  167
I:  isub(I,I)                                  173
I:  imul(I,I)                                  169
I:  idiv(I,I)                                  168
I:  irem(I,I)                                  171
I:  ineg(I)                                    170
I:  ishl(I,I)                                  172
I:  ishr(I,I)                                  172
I:  iushr(I,I)                                 173
I:  iand(I,I)                                  168
I:  ior(I,I)                                   170
I:  ixor(I,I)                                  173
I:  lcmp(L,L)                                  175
I:  fcmpg(F,F)                                 162
I:  fcmpl(F,F)                                 163
I:  dcmpg(D,D)                                 158
I:  dcmpl(D,D)                                 158
I:  length(A)                                  176
I:  bload(A,#o,#v)                             154
I:  sload(A,#o,#v)                             184
I:  iload(A,#o,#v)                             169
I:  baload(A,I)                                154
I:  saload(A,I)                                184
I:  iaload(A,I)                                167
I:  islocked(A)                                172
I:  subtypeof(A,A)                             185
I:  comptypeof(A,A)                            156
I:  iuse(%i)                                   173
I:  iconst($i)                                 168

L:  i2l(I)                                     167
L:  f2l(F)                                     162
L:  d2l(D)                                     157
L:  ladd(L,L)                                  174
L:  lsub(L,L)                                  180
L:  lmul(L,L)                                  176
L:  ldiv(L,L)                                  176
L:  lrem(L,L)                                  178
L:  lneg(L)                                    177
L:  lshl(L,I)                                  179
L:  lshr(L,I)                                  179
L:  lushr(L,I)                                 180
L:  land(L,L)                                  175
L:  lor(L,L)                                   178
```

```
L:   lxor(L,L)                                    180
L:   lload(A,#o,#v)                               176
L:   laload(A,I)                                  175
L:   luse(%l)                                     180
L:   lconst($l)                                   175

F:   i2f(I)                                       167
F:   l2f(L)                                       174
F:   d2f(D)                                       157
F:   fstrict(F)                                   165
F:   fadd(F,F)                                    162
F:   fsub(F,F)                                    166
F:   fmul(F,F)                                    164
F:   fdiv(F,F)                                    163
F:   frem(F,F)                                    165
F:   fneg(F)                                      164
F:   fload(A,#o,#v)                               163
F:   faload(A,I)                                  162
F:   fuse(%f)                                     166
F:   fconst($f)                                   162

D:   i2d(I)                                       167
D:   l2d(L)                                       174
D:   f2d(F)                                       161
D:   dstrict(D)                                   161
D:   dadd(D,D)                                    157
D:   dsub(D,D)                                    161
D:   dmul(D,D)                                    159
D:   ddiv(D,D)                                    158
D:   drem(D,D)                                    160
D:   dneg(D)                                      159
D:   dload(A,#o,#v)                               159
D:   daload(A,I)                                  157
D:   duse(%d)                                     161
D:   dconst($d)                                   158
```

A.2   Opcodes

aaload

Operation   Load reference from array

Syntax
```
A:   aaload(A,I)
```

Description The `aaload` opcode reads a reference from array A at index I. A must be a non-null reference to a reference array instance. I must be non-negative and less than A length.

Class *memory-accessing*

aastore

Operation Store into reference array

Syntax $\boxed{S: \quad \texttt{aastore(}A_1\texttt{,}I\texttt{,}A_2\texttt{)}}$

Description The `aastore` opcode writes a reference A_2 into array A_1 at index I. A_1 must be a non-null reference to a reference array instance. I must be non-negative and less than A_1 length. A_2 must be a subtype of A_1 component type.

Class *memory-accessing*

acatch

Operation Catch exception

Syntax $\boxed{S: \quad \texttt{acatch(\%a)}}$

Description The `acatch` opcode defines an exception handler entry point in an IR program. It stores a non-null reference to the exception thrown in register `%a`.

Notes The `acatch` is only reachable by *exception-prone* opcodes: `callx`, `ncallx`, `initx`, `newinstancex`, `newarrayx` and `lockx`.

The `acatch` opcode simply defines a unified exception handler entry point for a set of *exception-prone* opcodes. The check for exception subtyping and delegation to appropriate handler is responsibility of the code following it.

aclass

Operation Provide class reference

Syntax $\boxed{A: \quad \texttt{aclass(\$c)}}$

Description The `aclass` opcode provides the reference for the `Class` instance associated to type `$c`.

The type `$c` is represented by the name of a class or interface in its *extended fully-qualified internal* form (Section 4.3.1).

adefine

Operation Write reference register

Syntax

```
S:  adefine(%a,A)
```

Description The adefine opcode stores the value of expression A in register %a.

ajump

Operation Transfer control if reference comparison succeeds

Syntax

```
S:  ajump(#x,A₁,A₂,@l)
```

Description The ajump opcode transfers execution to label @l if the comparison of A_1 and A_2 succeeds.

The attribute #x defines the behavior of the ajump opcode in the following way:

#x	TRANSFER CONTROL IF
EQ	$A_1 = A_2$
NE	$A_1 \neq A_2$

aload

Operation Load reference from field

Syntax

```
A:  aload(A,#o,#v,#c)
```

Description The aload opcode reads a reference field from A at offset #o. A must be a non-null reference.

The attribute #o specifies the reference field offset using the encoding described on Section 4.4. It also tells, using a boolean flag, which of the static or instance field tables must be used.

The attribute #v indicates if the read is *volatile*, i.e. cannot be cached.

The attribute #c is the type of the reference provided by aload. It is represented by the name of a class or interface in its *extended fully-qualified internal* form (Section 4.3.1).

Class *memory-accessing*

anull

Operation Provide null reference

Syntax | *A:* anull**()** |

Description The anull opcode provides a null object reference.

apass

Operation Pass reference as parameter to method call

Syntax | *S:* apass**(***A***)** |

Description The apass opcode passes a reference *A* as parameter to a subsequent method call.

Class *parameter-passing*

Notes The parameter passing protocol in an IR program is *right-to-left* rather than *left-to-right*, as adopted in Java bytecodes.

The number and types of *parameter-passing* opcodes must match the number and types of the method being called. They should appear in the expected order just before the associated *method-call* opcode.

areceive

Operation Write method parameter to reference register

Syntax | *S:* areceive**(%**a**,#**c**)** |

Description The areceive opcode stores the parameter received by a method, upon its call, in reference register %a.

The reference written to reference register %a points to an object of type #c or one of its subtypes. The type #c is represented by the name of a class or interface in its *extended fully-qualified internal* form (Section 4.3.1).

Class *parameter-receiving*

Notes *Parameter-receiving* opcodes must appear on top of IR programs, before all other opcodes. They must be placed according to the method signature in *left-to-right* order.

aresult

Operation   Write method result to reference register

Syntax $S:$ `aresult(%a,#c)`

Description   The `aresult` opcode stores the result of a method call in reference register `%a`. It must appear just after the method call opcode and be used only when calling reference methods.

The reference written to reference register `%a` points to an object of type `#c` or one of its subtypes. The type `#c` is represented by the name of a class or interface in its *extended fully-qualified internal* form (Section 4.3.1).

Class   *result-saving*

Notes   The `aresult` opcode is also used after *memory-allocation* opcodes to store the reference to the newly allocated object in a reference register.

areturn

Operation   Return from reference method

Syntax $S:$ `areturn(A)`

Description   The `areturn` opcode returns from the current executing method. The current executing method must be a reference method. The value of expression A is used as the return value.

Class   *method-returning*

Notes   *Method-returning* opcodes may appear anywhere in an IR program, not simply at the end as one would expect.

astore

Operation   Store into reference field

Syntax $S:$ `astore(A_1,#o,#v,A_2)`

Description   The `astore` opcode writes A_2 into a reference field of A_1 at offset `#o`. A_1 must be a non-null reference.

The attribute `#o` specifies the reference field offset using the encoding described on Section 4.4. It also tells, using a boolean flag, which of the static or instance field tables must be used.

The attribute `#v` indicates if the write is *volatile*, i.e. cannot be cached.

Class   *memory-accessing*

astring

Operation   Provide string reference

Syntax | *A:*   `astring(`$s`)` |

Description   The `astring` opcode provides the reference for the *String* instance result of $s internalization. $s is any Java string literal.

athrow

Operation   Throw exception

Syntax | *S:*   `athrow(`*A*`)` |

Description   The `athrow` opcode throws an exception in the frame of the caller method. The expression *A* must evaluate to a non-null reference.

Notes   The `athrow` opcode is used only when throwing or rethrowing exceptions outside the current frame. The `athrow` opcode is not used when throwing exceptions in code protected by the current frame. In this case, the exception handling is done explicitly by transfering the control to the appropriate handler.

ause

Operation   Read reference register

Syntax | *A:*   `ause(`%a`)` |

Description   The `ause` opcode loads the value of register %a.

baload

Operation   Load boolean or byte from array

Syntax | *I:*   `baload(`*A*`,`*I*`)` |

Description   The `baload` opcode reads the lower 8 bits of integer from array *A* at index *I*. *A* must be a non-null reference to a boolean or byte array instance. *I* must be non-negative and less than *A* length. The higher 24 bits of the integer provided by `baload` are left unspecified.

Class   *memory-accessing*

Notes   Booleans are coded as integers. Non-zero integers when treated as booleans have *true* semantics, otherwise they have *false* semantics.

bastore

Operation Store into boolean or byte array

Syntax $S:$ `bastore(`A`,`I_1`,`I_2`)`

Description The `bastore` opcode writes the lower 8 bits of integer I_2 into array A at index I_1. A must be a non-null reference to a boolean or byte array instance. I_1 must be non-negative and less than A length.

Class *memory-accessing*

Notes Booleans are coded as integers. Non-zero integers when treated as booleans have *true* semantics, otherwise they have *false* semantics.

bload

Operation Load boolean or byte from field

Syntax $I:$ `bload(`A`,#o,#v)`

Description The `bload` opcode reads the lower 8 bits of the integer from boolean or byte field of A at offset #o. A must be a non-null reference. The higher 24 bits of integer provided by `bload` are left unspecified.

The attribute #o specifies the boolean or byte field offset using the encoding described on Section 4.4. It also tells, using a boolean flag, which of the static or instance field tables must be used.

The attribute #v indicates if the read is *volatile*, i.e. cannot be cached.

Class *memory-accessing*

Notes Booleans are coded as integers. Non-zero integers when treated as booleans have *true* semantics, otherwise they have *false* semantics.

bstore

Operation Store into boolean or byte field

Syntax $S:$ `bstore(`A`,#o,#v,`I`)`

Description The `bstore` opcode writes the lower 8 bits of integer I into a boolean or byte field of A at offset #o. A must be a non-null reference.

The attribute #o specifies the boolean or byte field offset using the encoding described on Section 4.4. It also tells, using a boolean flag, which of the static or instance field tables must be used.

The attribute #v indicates if the write is *volatile*, i.e. cannot be cached.

Class   *memory-accessing*

Notes   Booleans are coded as integers. Non-zero integers when treated as booleans have *true* semantics, otherwise they have *false* semantics.

call

Operation Call method

Syntax
```
S:   call(A,#t)
```

Description The `call` opcode calls method `A`. `A` must evaluate to a non-null reference of class `MethodText`. The parameters to the call are passed before this opcode using *parameter-passing* opcodes. If the callee method is not void, a *result-saving* opcode may be used just after the `call` opcode to store the result in a register.

The attribute `#t` contains information regarding stack inspection and tracing.

Class   *method-call*, *inspection-point*

Notes   The parameter passing protocol in an IR program is *right-to-left* rather than *left-to-right*, as adopted in Java bytecodes.

The number and types of *parameter-passing* opcodes must match the number and types of the method being called. They should appear in the expected order just before the associated *method-call* opcode.

The type of the *result-saving* opcode must match the type of the method being called.

This opcode is not *exception-prone*, hence, upon failure an exception is thrown in the frame of the caller method.

callx

Operation Call method, handle failure

Syntax
```
S:   callx(A,#t,@l)
```

Description The `callx` opcode calls method `A`. `A` must evaluate to a non-null reference of class `MethodText`. The parameters to the call are passed before this opcode using *parameter-passing* opcodes. If the callee method is not void, a *result-saving* opcode may be used just after the `callx` opcode to store the result in a register.

The attribute `#t` contains information regarding stack inspection and tracing.

Upon failure, the control is transfered to label `@l`. An `acatch` opcode must be the first statement following the label `@l`.

Class *method-call*, *inspection-point*, *exception-prone*

Notes The parameter passing protocol in an IR program is *right-to-left* rather than *left-to-right*, as adopted in Java bytecodes.

The number and types of *parameter-passing* opcodes must match the number and types of the method being called. They should appear in the expected order just before the associated *method-call* opcode.

The type of the *result-saving* opcode must match the type of the method being called.

comptypeof

Operation Determine array component subtyping

Syntax $\boxed{I: \quad \texttt{comptypeof(}A_1\texttt{,}A_2\texttt{)}}$

Description The `comptypeof` opcode checks if class or interface A_1 is a subtype of the component type of array class A_2. Both expressions must evaluate to non-null references of class *Class*.

d2i

Operation Convert double to integer

Syntax $\boxed{I: \quad \texttt{d2i(}D\texttt{)}}$

Description The `d2i` opcode converts the double expression D to integer.

d2f

Operation Convert double to float

Syntax $\boxed{F: \quad \texttt{d2f(}D\texttt{)}}$

Description The `d2f` opcode converts the double expression D to float.

d2l

Operation Convert double to long integer

Syntax $\boxed{L: \quad \texttt{d2l(}D\texttt{)}}$

Description The `d2l` opcode converts the double expression D to long integer.

dadd

Operation   Add doubles

Syntax $\boxed{D:\quad \mathtt{dadd(}D_1\mathtt{,}D_2\mathtt{)}}$

Description   The $\mathtt{dadd}$ opcode provides the result of addition $D_1 + D_2$.

daload

Operation   Load double from array

Syntax $\boxed{D:\quad \mathtt{daload(}A\mathtt{,}I\mathtt{)}}$

Description   The $\mathtt{daload}$ opcode reads a double from array A at index I. A must be a non-null reference to a double array instance. I must be non-negative and less than A length.

Class   *memory-accessing*

dastore

Operation   Store into double array

Syntax $\boxed{S:\quad \mathtt{dastore(}A\mathtt{,}I\mathtt{,}D\mathtt{)}}$

Description   The $\mathtt{dastore}$ opcode writes a double D into array A at index I. A must be a non-null reference to a double array instance. I must be non-negative and less than A length.

Class   *memory-accessing*

dcmpg

Operation   Compare doubles

Syntax $\boxed{I:\quad \mathtt{dcmpg(}D_1\mathtt{,}D_2\mathtt{)}}$

Description   The $\mathtt{dcmpg}$ opcode compares doubles D_1 and D_2. If $D_1 < D_2$, a negative integer value is provided. If $D_1 = D_2$, the integer value 0 is provided. If $D_1 > D_2$, a positive integer value is provided. If at least one of D_1 or D_2 is *NaN*, a positive integer value is provided.

Notes   The $\mathtt{dcmpg}$ and $\mathtt{dcmpl}$ instructions differ only when treating *NaN*.

dcmpl

Operation   Compare doubles

Syntax $\boxed{I:\quad \text{dcmpl}(D_1,D_2)}$

Description   The dcmpl opcode compares doubles D_1 and D_2. If $D_1 < D_2$, a negative integer value is provided. If $D_1 = D_2$, the integer value 0 is provided. If $D_1 > D_2$, a positive integer value is provided. If at least one of D_1 or D_2 is *NaN*, a negative integer value is provided.

Notes   The dcmpg and dcmpl instructions differ only when treating *NaN*.

dconst

Operation   Provide double constant

Syntax $\boxed{D:\quad \text{dconst(\$d)}}$

Description   The dconst opcode provides the double constant $d.

ddefine

Operation   Write double register

Syntax $\boxed{S:\quad \text{ddefine}(\%d,D)}$

Description   The ddefine opcode stores the value of expression D in register %d.

ddiv

Operation   Divide doubles

Syntax $\boxed{D:\quad \text{ddiv}(D_1,D_2)}$

Description   The ddiv opcode provides the result of division D_1/D_2.

dload

Operation   Load double from field

Syntax $\boxed{D:\quad \text{dload}(A,\#o,\#v)}$

Description   The dload opcode reads a double field from A at offset #o. A must be a non-null reference.

The attribute #o specifies the double field offset using the encoding described on Section 4.4. It also tells, using a boolean flag, which of the static or instance field tables must be used.

The attribute #v indicates if the read is *volatile*, i.e. cannot be cached.

Class   *memory-accessing*

dmul

Operation   Multiply doubles

Syntax $\boxed{D: \quad \mathtt{dmul(}D_1\mathtt{,}D_2\mathtt{)}}$

Description The `dmul` opcode provides the result of multiplication $D_1 * D_2$.

dneg

Operation   Negate double

Syntax $\boxed{D: \quad \mathtt{dneg(}D\mathtt{)}}$

Description The `dneg` opcode provides the result of negation $-D$.

dpass

Operation   Pass double as parameter to method call

Syntax $\boxed{S: \quad \mathtt{dpass(}D\mathtt{)}}$

Description The `dpass` opcode passes a double D as parameter to a subsequent method call.

Class   *parameter-passing*

Notes   The parameter passing protocol in an IR program is *right-to-left* rather than *left-to-right*, as adopted in Java bytecodes.

                 The number and types of *parameter-passing* opcodes must match the number and types of the method being called. They should appear in the expected order just before the associated *method-call* opcode.

dreceive

Operation   Write method parameter to double register

Syntax $\boxed{S: \quad \mathtt{dreceive(\%d)}}$

Description The `dreceive` opcode stores the parameter received by a method, upon its call, in double register `%d`.

Class   *parameter-receiving*

Notes   *Parameter-receiving* opcodes must appear on top of IR programs, before all other opcodes. They must be placed according to the method signature in *left-to-right* order.

drem

Operation   Remainder doubles

Syntax $D\mathtt{:}$ `drem(`D_1`,`D_2`)`

Description The `drem` opcode provides the result of remainder $D_1 \% D_2$.

dresult

Operation   Write method result to double register

Syntax $S\mathtt{:}$ `dresult(%d)`

Description The `dresult` opcode stores the result of a method call in double register `%d`. It must appear just after the method call opcode and be used only when calling double methods.

Class   *result-saving*

dreturn

Operation   Return from double method

Syntax $S\mathtt{:}$ `dreturn(`D`)`

Description The `dreturn` opcode returns from the current executing method. The current executing method must be a double method. The value of expression D is used as the return value.

Class   *method-returning*

Notes   *Method-returning* opcodes may appear anywhere in an IR program, not simply at the end as one would expect.

dstore

Operation   Store into double field

Syntax $S\mathtt{:}$ `dstore(`A`,#o,#v,`D`)`

Description The dstore opcode writes D into a double field of A at offset #o. A must be a non-null reference.

The attribute #o specifies the double field offset using the encoding described on Section 4.4. It also tells, using a boolean flag, which of the static or instance field tables must be used.

The attribute #v indicates if the write is *volatile*, i.e. cannot be cached.

Class *memory-accessing*

dstrict

Operation Convert to double value set

Syntax $\boxed{D: \quad \texttt{dstrict(}D\texttt{)}}$

Description The dstrict opcode provides the nearest element of the *double value set* representing the *double-extended-exponent value set* element D. The *value set conversion* procedure is described on [LY99, §2.6.6].

dsub

Operation Subtract doubles

Syntax $\boxed{D: \quad \texttt{dsub(}D_1\texttt{,}D_2\texttt{)}}$

Description The dsub opcode provides the result of subtraction $D_1 - D_2$.

duse

Operation Read double register

Syntax $\boxed{D: \quad \texttt{duse(\%d)}}$

Description The duse opcode loads the value of register %d.

f2d

Operation Convert float to double

Syntax $\boxed{D: \quad \texttt{f2d(}F\texttt{)}}$

Description The f2d opcode converts the float expression F to double.

f2i

Operation   Convert float to integer

Syntax
```
I:   f2i(F)
```

Description The f2i opcode converts the float expression F to integer.

f2l

Operation   Convert float to long integer

Syntax
```
L:   f2l(F)
```

Description The f2l opcode converts the float expression F to long integer.

fadd

Operation   Add floats

Syntax
```
F:   fadd(F₁,F₂)
```

Description The fadd opcode provides the result of addition $F_1 + F_2$.

faload

Operation   Load float from array

Syntax
```
F:   faload(A,I)
```

Description The faload opcode reads a float from array A at index I. A must be a non-null reference to a float array instance. I must be non-negative and less than A length.

Class   *memory-accessing*

fastore

Operation   Store into float array

Syntax
```
S:   fastore(A,I,F)
```

Description The fastore opcode writes a float F into array A at index I. A must be a non-null reference to a float array instance. I must be non-negative and less than A length.

Class   *memory-accessing*

fconst

Operation   Provide float constant

Syntax $\boxed{F:\quad \texttt{fconst(\$f)}}$

Description The `fconst` opcode provides the float constant `$f`.

fcmpg

Operation   Compare floats

Syntax $\boxed{I:\quad \texttt{fcmpg(}F_1\texttt{,}F_2\texttt{)}}$

Description The `fcmpg` opcode compares floats F_1 and F_2. If $F_1 < F_2$, a negative integer value is provided. If $F_1 = F_2$, the integer value 0 is provided. If $F_1 > F_2$, a positive integer value is provided. If at least one of F_1 or F_2 is *NaN*, a positive integer value is provided.

Notes   The `fcmpg` and `fcmpl` instructions differ only when treating *NaN*.

fcmpl

Operation   Compare floats

Syntax $\boxed{I:\quad \texttt{fcmpl(}F_1\texttt{,}F_2\texttt{)}}$

Description The `fcmpl` opcode compares floats F_1 and F_2. If $F_1 < F_2$, a negative integer value is provided. If $F_1 = F_2$, the integer value 0 is provided. If $F_1 > F_2$, a positive integer value is provided. If at least one of F_1 or F_2 is *NaN*, a negative integer value is provided.

Notes   The `fcmpg` and `fcmpl` instructions differ only when treating *NaN*.

fdefine

Operation   Write float register

Syntax $\boxed{S:\quad \texttt{fdefine(\%f,}F\texttt{)}}$

Description The `fdefine` opcode stores the value of expression F in register `%f`.

fdiv

Operation   Divide floats

Syntax $\boxed{F:\quad \texttt{fdiv(}F_1\texttt{,}F_2\texttt{)}}$

Description The `fdiv` opcode provides the result of division F_1/F_2.

fload

Operation   Load float from field

Syntax   `F:   fload(A,#o,#v)`

Description   The `fload` opcode reads a float field from *A* at offset `#o`. *A* must be a non-null reference.

The attribute `#o` specifies the float field offset using the encoding described on Section 4.4. It also tells, using a boolean flag, which of the static or instance field tables must be used.

The attribute `#v` indicates if the read is *volatile*, i.e. cannot be cached.

Class   *memory-accessing*

fmul

Operation   Multiply floats

Syntax   `F:   fmul(`F_1`,`F_2`)`

Description   The `fmul` opcode provides the result of multiplication $F_1 * F_2$.

fneg

Operation   Negate float

Syntax   `F:   fneg(F)`

Description   The `fneg` opcode provides the result of negation $-F$.

fpass

Operation   Pass float as parameter to method call

Syntax   `S:   fpass(F)`

Description   The `fpass` opcode passes a float *F* as parameter to a subsequent method call.

Class   *parameter-passing*

Notes   The parameter passing protocol in an IR program is *right-to-left* rather than *left-to-right*, as adopted in Java bytecodes.

The number and types of *parameter-passing* opcodes must match the number and types of the method being called. They should appear in the expected order just before the associated *method-call* opcode.

freceive

Operation   Write method parameter to float register

Syntax S**:**   `freceive(%f)`

Description The `freceive` opcode stores the parameter received by a method, upon its call, in float register `%f`.

Class   *parameter-receiving*

Notes   *Parameter-receiving* opcodes must appear on top of IR programs, before all other opcodes. They must be placed according to the method signature in *left-to-right* order.

frem

Operation   Remainder floats

Syntax F**:**   `frem(`F_1**,**F_2`)`

Description The `frem` opcode provides the result of remainder $F_1 \% F_2$.

fresult

Operation   Write method result to float register

Syntax S**:**   `fresult(%f)`

Description The `fresult` opcode stores the result of a method call in float register `%f`. It must appear just after the method call opcode and be used only when calling float methods.

Class   *result-saving*

freturn

Operation   Return from float method

Syntax S**:**   `freturn(`F`)`

Description The `freturn` opcode returns from the current executing method. The current executing method must be a float method. The value of expression F is used as the return value.

Class   *method-returning*

Notes   *Method-returning* opcodes may appear anywhere in an IR program, not simply at the end as one would expect.

fstore

Operation   Store into float field

Syntax $\boxed{S: \quad \texttt{fstore(}A\texttt{,\#o,\#v,}F\texttt{)}}$

Description   The `fstore` opcode writes F into a float field of A at offset #o. A must be a non-null reference.

The attribute #o specifies the float field offset using the encoding described on Section 4.4. It also tells, using a boolean flag, which of the static or instance field tables must be used.

The attribute #v indicates if the write is *volatile*, i.e. cannot be cached.

Class   *memory-accessing*

fstrict

Operation   Convert to float value set

Syntax $\boxed{F: \quad \texttt{fstrict(}F\texttt{)}}$

Description   The `fstrict` opcode provides the nearest element of the *float value set* representing the *float-extended-exponent value set* element F. The *value set conversion* procedure is described on [LY99, §2.6.6].

fsub

Operation   Subtract floats

Syntax $\boxed{F: \quad \texttt{fsub(}F_1\texttt{,}F_2\texttt{)}}$

Description   The `fsub` opcode provides the result of subtraction $F_1 - F_2$.

fuse

Operation   Read float register

Syntax $\boxed{F: \quad \texttt{fuse(\%f)}}$

Description   The `fuse` opcode loads the value of register %f.

getclass

Operation   Get object class

Syntax | *A:* getclass*(A)* |

Description   The getclass opcode provides a reference to a *Class* instance representing the class of *A*. *A* expression must evaluate to a non-null reference.

Notes   The getclass opcode may be handled as if it were an arithmetic expression. It does not have side effects and will always provide the same value once its argument is fixed.

i2b

Operation   Convert integer to byte

Syntax | *I:* i2b*(I)* |

Description   The i2b converts the integer expression *I* to byte. The integer value provided by i2b is the result of sign-extending *I* lower 8 bits.

i2c

Operation   Convert integer to char

Syntax | *I:* i2c*(I)* |

Description   The i2c converts the integer expression *I* to char. The integer value provided by i2c is the result of zero-extending *I* lower 16 bits.

i2d

Operation   Convert integer to double

Syntax | *D:* i2d*(I)* |

Description   The i2d opcode converts the integer expression *I* to double.

i2f

Operation   Convert integer to float

Syntax | *F:* i2f*(I)* |

Description   The i2f opcode converts the integer expression *I* to float.

i2l

Operation Convert integer to long integer

Syntax `L:  i2l(I)`

Description The `i2l` opcode converts the integer expression `I` to long integer.

i2s

Operation Convert integer to short

Syntax `I:  i2s(I)`

Description The `i2s` converts the integer expression `I` to short. The integer value provided by `i2s` is the result of sign-extending `I` lower 16 bits.

iadd

Operation Add integers

Syntax `I:  iadd(I₁,I₂)`

Description The `iadd` opcode provides the result of addition $I_1 + I_2$.

iaload

Operation Load integer from array

Syntax `I:  iaload(A,I)`

Description The `iaload` opcode reads an integer from array `A` at index `I`. `A` must be a non-null reference to a integer array instance. `I` must be non-negative and less than `A` length.

Class *memory-accessing*

iand

Operation Bitwise and integers

Syntax `I:  iand(I₁,I₂)`

Description The `iand` opcode provides the result of operation $I_1 \& I_2$.

iastore

Operation   Store into integer array

Syntax $\boxed{S: \quad \texttt{iastore(}A\texttt{,}I_1\texttt{,}I_2\texttt{)}}$

Description The `iastore` opcode writes an integer I_2 into array A at index I_1. A must be a non-null reference to a integer array instance. I_1 must be non-negative and less than A length.

Class   *memory-accessing*

iconst

Operation   Provide integer constant

Syntax $\boxed{I: \quad \texttt{iconst(\$i)}}$

Description The `iconst` opcode provides the integer constant $i.

idefine

Operation   Write integer register

Syntax $\boxed{S: \quad \texttt{idefine(\%i,}I\texttt{)}}$

Description The `idefine` opcode stores the value of expression I in register `%i`.

idiv

Operation   Divide integers

Syntax $\boxed{I: \quad \texttt{idiv(}I_1\texttt{,}I_2\texttt{)}}$

Description The `idiv` opcode provides the result of division I_1/I_2. I_2 must be non-zero.

ijump

Operation   Transfer control if integer comparison succeeds

Syntax $\boxed{S: \quad \texttt{ijump(\#x,}I_1\texttt{,}I_2\texttt{,@l)}}$

Description The `ijump` opcode transfers execution to label `@l` if the comparison of I_1 and I_2 succeeds.

The attribute `#x` defines the behavior of the `ijump` opcode in the following way:

#x	TRANSFER CONTROL IF
EQ	$I_1 = I_2$
NE	$I_1 \neq I_2$
LT	$I_1 < I_2$
LE	$I_1 \leq I_2$
GE	$I_1 \geq I_2$
GT	$I_1 > I_2$
B	$I_1 <_{(unsigned)} I_2$
BE	$I_1 \leq_{(unsigned)} I_2$
AE	$I_1 \geq_{(unsigned)} I_2$
A	$I_1 >_{(unsigned)} I_2$

iload

Operation Load integer from field

Syntax $I:$ `iload(`A`,#o,#v)`

Description The `iload` opcode reads an integer field from A at offset #o. A must be a non-null reference.

The attribute #o specifies the integer field offset using the encoding described on Section 4.4. It also tells, using a boolean flag, which of the static or instance field tables must be used.

The attribute #v indicates if the read is *volatile*, i.e. cannot be cached.

Class *memory-accessing*

imlookup

Operation Search for interface method

Syntax $A:$ `imlookup(`A_1`,`A_2`,#i)`

Description The `imlookup` provides the method at dispatch table index #i, from base offset of interface A_2, of class A_1. Both the expressions A_1 and A_2 must evaluate to non-null references of class *Class*. The `imlookup` provides a non-null reference of class *MethodText* if class A_1 implements interface A_2. Otherwise, the *null* reference is provided.

Notes The `imlookup` opcode may be handled as if it were an arithmetic expression. It does not have side effects and will always provide the same value once its arguments and attribute are fixed.

imul

Operation Multiply integers

Syntax $I:$ `imul(`I_1`,`I_2`)`

Description The `imul` opcode provides the result of multiplication $I_1 * I_2$.

ineg

Operation Negate integer

Syntax $I:$ `ineg(`I`)`

Description The `ineg` opcode provides the result of negation $-I$.

init

Operation Initialize class

Syntax $S:$ `init(`A`,#t)`

Description The `init` opcode triggers the initialization procedure ([LY99, §2.17.5]) for class A. The expression A must evaluate to a non-null reference of an instance of class `Class`.

The attribute `#t` contains information regarding stack inspection and tracing.

Class *inspection-point*

Notes This opcode is not *exception-prone*, hence, upon failure an exception is thrown in the frame of the caller method.

initx

Operation Initialize class, handle failure

Syntax $S:$ `initx(`A`,#t,@l)`

Description The `initx` opcode triggers the initialization procedure ([LY99, §2.17.5]) for class A. The expression A must evaluate to a non-null reference of an instance of class `Class`.

The attribute `#t` contains information regarding stack inspection and tracing.

Upon failure, the control is transfered to label `@l`. An `acatch` opcode must be the first statement following the label `@l`.

Class *inspection-point*, *exception-prone*

ior

Operation   Bitwise or integers

Syntax   `I:   ior(`I_1`,`I_2`)`

Description The `ior` opcode provides the result of operation $I_1 | I_2$.

ipass

Operation   Pass integer as parameter to method call

Syntax   `S:   ipass(`I`)`

Description The `ipass` opcode passes an integer I as parameter to a subsequent method call.

Class   *parameter-passing*

Notes   The parameter passing protocol in an IR program is *right-to-left* rather than *left-to-right*, as adopted in Java bytecodes.

The number and types of *parameter-passing* opcodes must match the number and types of the method being called. They should appear in the expected order just before the associated *method-call* opcode.

ireceive

Operation   Write method parameter to integer register

Syntax   `S:   ireceive(%i)`

Description The `ireceive` opcode stores the parameter received by a method, upon its call, in integer register `%i`.

Class   *parameter-receiving*

Notes   *Parameter-receiving* opcodes must appear on top of IR programs, before all other opcodes. They must be placed according to the method signature in *left-to-right* order.

irem

Operation   Remainder integers

Syntax   `I:   irem(`I_1`,`I_2`)`

Description The `irem` opcode provides the result of remainder $I_1 \% I_2$. I_2 must be non-zero.

iresult

Operation   Write method result to integer register

Syntax | *S:*   iresult**(%i)** |

Description The iresult opcode stores the result of a method call in integer register %i. It must appear just after the method call opcode and be used only when calling integer methods.

Class   *result-saving*

ireturn

Operation   Return from integer method

Syntax | *S:*   ireturn**(***I***)** |

Description The ireturn opcode returns from the current executing method. The current executing method must be an integer method. The value of expression I is used as the return value.

Class   *method-returning*

Notes   *Method-returning* opcodes may appear anywhere in an IR program, not simply at the end as one would expect.

ishl

Operation   Shift left integer

Syntax | *I:*   ishl**(***I_1***,***I_2***)** |

Description The ishl opcode provides the result of operation $I_1 << I_2$.

ishr

Operation   Arithmetic shift right integer

Syntax | *I:*   ishr**(***I_1***,***I_2***)** |

Description The ishr opcode provides the result of operation $I_1 >> I_2$.

islocked

Operation   Determine if lock is acquired

Syntax   `I:  islocked(A)`

Description The `islocked` opcode determines if the lock for object `A` has been acquired by the *current thread*. The expression `A` must evaluate to a non-null reference. If the lock has been acquired the `islocked` opcode provides a non-zero integer value. Otherwise, it provides the integer `0` value.

istore

Operation   Store into integer field

Syntax   `S:  istore(A,#o,#v,I)`

Description The `istore` opcode writes `I` into an integer field of `A` at offset `#o`. `A` must be a non-null reference.

The attribute `#o` specifies the integer field offset using the encoding described on Section 4.4. It also tells, using a boolean flag, which of the static or instance field tables must be used.

The attribute `#v` indicates if the write is *volatile*, i.e. cannot be cached.

Class   *memory-accessing*

isub

Operation   Subtract integers

Syntax   `I:  isub(I₁,I₂)`

Description The `isub` opcode provides the result of subtraction $I_1 - I_2$.

iswitch

Operation   Transfer control based on integer jump table

Syntax   `S:  iswitch(I,[$i,@l]...)`

Description The `iswitch` opcode uses the result of expression `I` to transfer execution based on an integer jump table. Each entry `$i` defines a label `@l` that will be used to transfer execution if there is a match. If there is no match, execution is not transfered and continues sequentially. No entry `$i` may appear twice on the jump table.

iuse

Operation   Read integer register

Syntax $I:$ `iuse(%i)`

Description   The `iuse` opcode loads the value of register `%i`.

iushr

Operation   Logical shift right integer

Syntax $I:$ `iushr(`I_1`,`I_2`)`

Description   The `iushr` opcode provides the result of operation $I_1 >>> I_2$.

ixor

Operation   Bitwise xor integers

Syntax $I:$ `ixor(`I_1`,`I_2`)`

Description   The `ixor` opcode provides the result of operation $I_1\verb|^|I_2$.

jump

Operation   Transfer control unconditionally

Syntax $S:$ `jump(@l)`

Description   The `jump` opcode transfer execution to label `@l`, unconditionally.

l2d

Operation   Convert long integer to double

Syntax $D:$ `l2d(`L`)`

Description   The `l2d` opcode converts the long integer expression L to double.

l2f

Operation   Convert long integer to float

Syntax $F:$ `l2f(`L`)`

Description   The `l2f` opcode converts the long integer expression L to float.

l2i

Operation Convert long integer to integer

Syntax $\boxed{\texttt{I: l2i(}L\texttt{)}}$

Description The `l2i` opcode converts the long integer expression L to integer.

label

Operation Declare label

Syntax $\boxed{\texttt{S: label(@l)}}$

Description The `label` opcode associates the current point in the IR program to label `@l`.

ladd

Operation Add long integers

Syntax $\boxed{\texttt{L: ladd(}L_1\texttt{,}L_2\texttt{)}}$

Description The `ladd` opcode provides the result of addition $L_1 + L_2$.

laload

Operation Load long integer from array

Syntax $\boxed{\texttt{L: laload(}A\texttt{,}I\texttt{)}}$

Description The `laload` opcode reads a long integer from array A at index I. A must be a non-null reference to a long integer array instance. I must be non-negative and less than A length.

Class *memory-accessing*

land

Operation Bitwise and long integers

Syntax $\boxed{\texttt{L: land(}L_1\texttt{,}L_2\texttt{)}}$

Description The `land` opcode provides the result of operation $L_1 \& L_2$.

lastore

Operation   Store into long integer array

Syntax $S:$ `lastore(`A,I,L`)`

Description   The `lastore` opcode writes a long integer L into array A at index I. A must be a non-null reference to a long integer array instance. I must be non-negative and less than A length.

Class   *memory-accessing*

lcmp

Operation   Compare long integers

Syntax $I:$ `lcmp(`L_1,L_2`)`

Description   The `lcmp` opcode compares signed long integers L_1 and L_2. If $L_1 < L_2$, a negative integer value is provided. If $L_1 = L_2$, the integer value 0 is provided. If $L_1 > L_2$, a positive integer value is provided.

lconst

Operation   Provide long integer constant

Syntax $L:$ `lconst($l)`

Description   The `lconst` opcode provides the long integer constant `$l`.

ldefine

Operation   Write long integer register

Syntax $S:$ `ldefine(%l,`L`)`

Description   The `ldefine` opcode stores the value of expression L in register `%l`.

ldiv

Operation   Divide long integers

Syntax $L:$ `ldiv(`L_1,L_2`)`

Description   The `ldiv` opcode provides the result of division L_1/L_2. L_2 must be non-zero.

length

Operation   Get length of array

Syntax   `I:   length(A)`

Description The length opcode provides the number of elements of an array instance. The expression `A` must evaluate to a non-null array reference. It provides a non-negative integer.

Notes   The length opcode may be handled as if it were an arithmetic expression. It does not have side effects and will always provide the same value once its argument is fixed.

lload

Operation   Load long integer from field

Syntax   `L:   lload(A,#o,#v)`

Description The lload opcode reads a long integer field from `A` at offset `#o`. `A` must be a non-null reference.

The attribute `#o` specifies the long integer field offset using the encoding described on Section 4.4. It also tells, using a boolean flag, which of the static or instance field tables must be used.

The attribute `#v` indicates if the read is *volatile*, i.e. cannot be cached.

Class   *memory-accessing*

lmul

Operation   Multiply long integers

Syntax   `L:   lmul(L₁,L₂)`

Description The lmul opcode provides the result of multiplication $L_1 * L_2$.

lneg

Operation   Negate long integer

Syntax   `L:   lneg(L)`

Description The lneg opcode provides the result of negation $-L$.

lock

Operation Acquire lock for object

Syntax | S: lock(*A*,#t) |

Description The lock opcode acquires the lock for object *A*. The expression *A* must evaluate to a non-null reference. If the lock has been previously acquired by another *thread* it will block until the lock is released.

The attribute #t contains information regarding stack inspection and tracing.

Class *inspection-point*

Notes Locks are recursive.

The lock opcode has no semantics regarding the memory model. The invalidation of cached memory reads is done using the readbarrier opcode.

This opcode is not *exception-prone*, hence, upon failure an exception is thrown in the frame of the caller method.

lockx

Operation Acquire lock for object, handle failure

Syntax | S: lockx(*A*,#t,@l) |

Description The lockx opcode acquires the lock for object *A*. The expression *A* must evaluate to a non-null reference. If the lock has been previously acquired by another *thread* it will block until the lock is released.

The attribute #t contains information regarding stack inspection and tracing.

Upon failure, the control is transfered to label @l. An acatch opcode must be the first statement following the label @l.

Class *inspection-point*, *exception-prone*

Notes Locks are recursive.

The lock opcode has no semantics regarding the memory model. The invalidation of cached memory reads is done using the readbarrier opcode.

lor

Operation Bitwise or long integers

Syntax | L: lor(L_1,L_2) |

Description The lor opcode provides the result of operation $L_1 | L_2$.

lpass

Operation Pass long integer as parameter to method call

Syntax `S:  lpass(L)`

Description The `lpass` opcode passes a long integer L as parameter to a subsequent method call.

Class *parameter-passing*

Notes The parameter passing protocol in an IR program is *right-to-left* rather than *left-to-right*, as adopted in Java bytecodes.

The number and types of *parameter-passing* opcodes must match the number and types of the method being called. They should appear in the expected order just before the associated *method-call* opcode.

lreceive

Operation Write method parameter to long integer register

Syntax `S:  lreceive(%l)`

Description The `lreceive` opcode stores the parameter received by a method, upon its call, in long integer register `%l`.

Class *parameter-receiving*

Notes *Parameter-receiving* opcodes must appear on top of IR programs, before all other opcodes. They must be placed according to the method signature in *left-to-right* order.

lrem

Operation Remainder long integers

Syntax `L:  lrem(L_1,L_2)`

Description The `lrem` opcode provides the result of remainder $L_1 \% L_2$. L_2 must be non-zero.

lresult

Operation Write method result to long integer register

Syntax `S:  lresult(%l)`

Description The `lresult` opcode stores the result of a method call in long integer register `%l`. It must appear just after the method call opcode and be used only when calling long integer methods.

Class *result-saving*

lreturn

Operation Return from long integer method

Syntax | *S:* `lreturn(`*L*`)` |

Description The `lreturn` opcode returns from the current executing method. The current executing method must be a long integer method. The value of expression *L* is used as the return value.

Class *method-returning*

Notes *Method-returning* opcodes may appear anywhere in an IR program, not simply at the end as one would expect.

lshl

Operation Shift left long integer

Syntax | *L:* `lshl(`*L*`,`*I*`)` |

Description The `lshl` opcode provides the result of operation $L << I$.

lshr

Operation Arithmetic shift right long integer

Syntax | *L:* `lshr(`*L*`,`*I*`)` |

Description The `lshr` opcode provides the result of operation $L >> I$.

lstore

Operation Store into long integer field

Syntax | *S:* `lstore(`*A*`,#o,#v,`*L*`)` |

Description The `lstore` opcode writes *L* into a long integer field of *A* at offset #o. *A* must be a non-null reference.

The attribute #o specifies the long integer field offset using the encoding described on Section 4.4. It also tells, using a boolean flag, which of the static or instance field tables must be used.

The attribute #v indicates if the write is *volatile*, i.e. cannot be cached.

Class   *memory-accessing*

lsub

Operation   Subtract long integers

Syntax
```
L:   lsub(L₁,L₂)
```

Description The lsub opcode provides the result of subtraction $L_1 - L_2$.

luse

Operation   Read long integer register

Syntax
```
L:   luse(%l)
```

Description The luse opcode loads the value of register %l.

lushr

Operation   Logical shift right long integer

Syntax
```
L:   lushr(L,I)
```

Description The lushr opcode provides the result of operation $L >>> I$.

lxor

Operation   Bitwise xor long integers

Syntax
```
L:   lxor(L₁,L₂)
```

Description The lxor opcode provides the result of operation $L_1\char`^L_2$.

mlookup

Operation   Search for method

Syntax
```
A:   mlookup(A,#i)
```

Description The `mlookup` provides the method at dispatch table index #i of class *A*. The expression *A* must evaluate to a non-null reference of class *Class*. The `mlookup` provides a non-null reference of class *MethodText*.

Notes The `mlookup` opcode may be handled as if it were an arithmetic expression. It does not have side effects and will always provide the same value once its argument and attribute are fixed.

ncall

Operation Call native method

Syntax | *S:*   `ncall(#c,#s,#t)` |

Description The `ncall` opcode calls the native implementation of native method identified by signature #s declared on class #c. The parameters to the call are passed before this opcode using *parameter-passing* opcodes. If the callee method is not void, a *result-saving* opcode may be used just after the `call` opcode to store the result in a register.

The attribute #c identifies the declaring class of the method to be called by `ncall`. It is represented by the name of a class or interface in its *extended fully-qualified internal* form (Section 4.3.1).

The attribute #s contains the name and descriptor of the callee method.

The attribute #t contains information regarding stack inspection and tracing.

Class *method-call*, *inspection-point*

Notes The parameter passing protocol in an IR program is *right-to-left* rather than *left-to-right*, as adopted in Java bytecodes.

The number and types of *parameter-passing* opcodes must match the number and types of the method being called. They should appear in the expected order just before the associated *method-call* opcode.

It the callee method is static, an extra reference to the class that declares it must be passed right before the `ncall` opcode.

The type of the *result-saving* opcode must match the type of the method being called.

This opcode is not *exception-prone*, hence, upon failure an exception is thrown in the frame of the caller method.

ncallx

Operation Call native method, handle failure

Syntax | *S:*   `ncallx(#c,#s,#t,@l)` |

Description The `ncallx` opcode calls the native implementation of native method identified by signature #s declared on class #c. The parameters to the call are passed before this opcode using *parameter-passing* opcodes. If the callee method is not void, a *result-saving* opcode may be used just after the `call` opcode to store the result in a register.

The attribute #c identifies the declaring class of the method to be called by `ncall`. It is represented by the name of a class or interface in its *extended fully-qualified internal* form (Section 4.3.1).

The attribute #s contains the name and descriptor of the callee method.

The attribute #t contains information regarding stack inspection and tracing.

Upon failure, the control is transfered to label @l. An `acatch` opcode must be the first statement following the label @l.

Class   *method-call*, *inspection-point*, *exception-prone*

Notes   The parameter passing protocol in an IR program is *right-to-left* rather than *left-to-right*, as adopted in Java bytecodes.

The number and types of *parameter-passing* opcodes must match the number and types of the method being called. They should appear in the expected order just before the associated *method-call* opcode.

It the callee method is static, an extra reference to the class that declares it must be passed right before the `ncallx` opcode.

The type of the *result-saving* opcode must match the type of the method being called.

newarray

Operation   Allocate new array

Syntax | *S:*   `newarray(A,I,#t)` |

Description The `newarray` opcode allocates space for a new array in the garbage collected heap. The expression *A* must evaluate to a non-null reference of an array type instance of class *Class*. The expression *I* must evaluate to a non-negative value. The new array will have room for *I* elements of the appropriate type, initialized with default values.

The attribute #t contains information regarding stack inspection and tracing.

Class   *memory-allocation*, *inspection-point*

Notes   This opcode is not *exception-prone*, hence, upon failure an exception is thrown in the frame of the caller method.

newarrayx

Operation   Allocate new array, handle failure

Syntax   `S:   newarrayx(A,I,#t,@l)`

Description   The `newarrayx` opcode allocates space for a new array in the garbage collected heap. The expression `A` must evaluate to a non-null reference of an array type instance of class `Class`. The expression `I` must evaluate to a non-negative value. The new array will have room for `I` elements of the appropriate type, initialized with default values.

The attribute `#t` contains information regarding stack inspection and tracing.

Upon failure, the control is transfered to label `@l`. An `acatch` opcode must be the first statement following the label `@l`.

Class   *memory-allocation*, *inspection-point*, *exception-prone*

newinstance

Operation   Allocate new object

Syntax   `S:   newinstance(A,#t)`

Description   The `newinstance` opcode allocates space for a new instance in the garbage collected heap. The expression `A` must evaluate to a non-null reference of a non-abstract instance of class `Class`. The new object have all its fields initialized with default values.

The attribute `#t` contains information regarding stack inspection and tracing.

Class   *memory-allocation*, *inspection-point*

Notes   This opcode is not *exception-prone*, hence, upon failure an exception is thrown in the frame of the caller method.

newinstancex

Operation   Allocate new object, handle failure

Syntax   `S:   newinstancex(A,#t,@l)`

Description   The `newinstancex` opcode allocates space for a new instance in the garbage collected heap. The expression `A` must evaluate to a non-null reference of a non-abstract instance of class `Class`. The new object have all its fields initialized with default values.

The attribute `#t` contains information regarding stack inspection and tracing.

Upon failure, the control is transfered to label @l. An `acatch` opcode must be the first statement following the label @l.

Class   *memory-allocation*, *inspection-point*, *exception-prone*

readbarrier

Operation Discards cached reads

Syntax $\boxed{S:\quad \texttt{readbarrier()}}$

Description The `readbarrier` opcode marks a point in the IR program where all cached memory reads must be discarded.

Notes   Once its single-threaded semantics does not change, the IR program may be transformed to anticipate memory reads, keeping values cached on registers. The `readbarrier` opcode prevents those values from staying cached.

saload

Operation Load char or short from array

Syntax $\boxed{I:\quad \texttt{saload(}A\texttt{,}I\texttt{)}}$

Description The `saload` opcode reads the lower 16 bits of the integer from array A at index I. A must be a non-null reference to a char or short array instance. I must be non-negative and less than A length. The higher 16 bits of the integer provided by `saload` are left unspecified.

Class   *memory-accessing*

sastore

Operation Store into char or short array

Syntax $\boxed{S:\quad \texttt{sastore(}A\texttt{,}I_1\texttt{,}I_2\texttt{)}}$

Description The `sastore` opcode writes the lower 16 bits of integer I_2 into array A at index I_1. A must be a non-null reference to a char or short array instance. I_1 must be non-negative and less than A length.

Class   *memory-accessing*

sload

Operation   Load char or short from field

Syntax $I:$ sload$(A,$#o$,$#v$)$

Description   The sload opcode reads the lower 16 bits of the integer from char or short field of A at offset #o. A must be a non-null reference. The higher 16 bits of the integer provided by sload are left unspecified.

The attribute #o specifies the char or short field offset using the encoding described on Section 4.4. It also tells, using a boolean flag, which of the static or instance field tables must be used.

The attribute #v indicates if the read is *volatile*, i.e. cannot be cached.

Class   *memory-accessing*

sstore

Operation   Store into char or short field

Syntax $S:$ sstore$(A,$#o$,$#v$,I)$

Description   The sstore opcode writes the lower 16 bits of integer I into a char or short field of A at offset #o. A must be a non-null reference.

The attribute #o specifies the char or short field offset using the encoding described on Section 4.4. It also tells, using a boolean flag, which of the static or instance field tables must be used.

The attribute #v indicates if the write is *volatile*, i.e. cannot be cached.

Class   *memory-accessing*

subtypeof

Operation   Determine subtyping

Syntax $I:$ subtypeof(A_1,A_2)

Description   The subtypeof opcode checks if class or interface A_1 is a subtype of class or interface A_2. Both expressions should evaluate to non-null references of class Class.

unlock

Operation Release lock for object

Syntax | `S:  unlock(A)` |

Description The `unlock` opcode releases lock for object `A`. The expression `A` must evaluate to a non-null reference. The lock must have been previously acquired before this opcode is reached.

Notes Locks are recursive.

The `unlock` opcode has no semantics regarding the memory model. The flush of cached memory writes is done using the `writebarrier` opcode.

vreturn

Operation Return from void method

Syntax | `S:  vreturn()` |

Description The `vreturn` opcode returns from the current executing method. The current executing method must be a void method.

Class *method-returning*

Notes *Method-returning* opcodes may appear anywhere in an IR program, not simply at the end as one would expect.

writebarrier

Operation Flush cached writes

Syntax | `S:  writebarrier()` |

Description The `writebarrier` opcode marks a point in the IR program where all cached memory writes must be flushed.

Notes Once its single-threaded semantics does not change, the IR program may be transformed to handle memory writes lazily, keeping values cached on registers. The `writebarrier` opcode prevents those values from staying cached.

Appendix B

Yet Another Tree Rewriting Tool

This chapter describes RING (Rewriting for INtermediate Grammar), a *tree rewriting tool* that uses *tree pattern matching*[HO82, Cha87, PL87] and *dynamic programming*. Its design is based on *iburg*[FHP92a, Fra89], although it has a different language and interface. It outputs a *hard-coded* matcher that does *dynamic programming* at compile time, similarly as *iburg*. The input specification is a proper superset of the *Java Programming Language*[AG00]. It supercedes *iburg* functionality since it supports *default rules* and *non-terminal templates*. Also, it provides a limited, but still useful, *non-terminal inlining* facility. Matchers are generated targeting the *Java Programming Language*.

This tree rewriting tool serves not only as a *code generator generator*[Fra77], but also as an intermediate representation manipulator. The tool is tailored to the intermediate representation (described in Appendix A), therefore it provides no syntax for declaring *grammar terminals*. For the same reason, it supports at most ternary arity tree patterns.

B.1   Specifications

The language for RING specification is an extension of the Java Programming Language. Figure B.1 shows the subset of the rules of a Java EBNF grammar[GJS96, §8] that defines the specification language extensions. To specify a tree matcher, the user must declare tree pattern rules in the class that she chooses to implement the matcher. Every tree pattern rule is associated to a non-terminal. Non-terminals are declared in the same way as fields and methods are declared in the body of the matcher class.

A non-terminal declaration consists of some access flags, a name, a signature, template parameters, non-terminal attributes, optimality expression, and rule declarations. The access flags define the visibility of the non-terminal. In the generated matcher, any non-terminal can be used as *start* symbol. Visibility flags may be used to hide non-terminals not meant to be start symbols. The *abstract* access flags may be used to mark the non-terminal as *inline* (see Section B.3.3). The name is used to identify the non-terminal. The non-terminal signature defines the signature of the *action function* that users call when the optimal match is found. Template parameters can be declared when creating template non-terminals (see Section B.3.2). A non-terminal may declare attributes which are synthetized during pattern

matching. The optimality expression is a boolean expression, based on synthetized attributes, used to test and replace matches for the non-terminal. The set of rule declarations defines the possible matches for that particular non-terminal.

ClassBodyDeclaration	→	*Initializer*
	|	*NestedClassDeclaration*
	|	*NestedInterfaceDeclaration*
	|	*ConstructorDeclaration*
	|	*MethodDeclaration*
	|	*FieldDeclaration*
	|	*NonTerminalDeclaration*
NonTerminalDeclaration	→	("**public**" | "**protected**" | "**private**" | "**abstract**")*
		ResultType Identifier [*TemplateId*]
		FormalParameters ("**[**" "**]**")* ["**throws**" *NameList*]
		[*NonTerminalVars* ["**[**" *Expression* "**]**"]]
		"**:**" *RuleDeclaration* ("**|**" *RuleDeclaration*)* "**;**"
NonTerminalVars	→	"<" *NonTerminalVar* ("**,**" *NonTerminalVar*)* ">"
NonTerminalVar	→	*Type VariableDeclaratorId*
RuleDeclaration	→	*TreePattern* ["**[**" *Expression* "**]**"] [*Block*] ["**=**" *Block*]
	|	"**default**" [*Block*] ["**=**" *Block*]
TreePattern	→	*Name*
		["**(**" *TreePattern* ("**,**" *TreePattern*)* "**)**"]
		[*TreePattern*]
	|	*Identifier* [*TemplateId*]
TemplateId	→	"<" *Identifier* ("**,**" *Identifier*)* ">"

Figure B.1: EBNF grammar excerpt for Java based matcher specifications.

Each rule declaration consists of a tree pattern and three optional functions: a predicate expression, a synthetize function and an action function. It is also possible to declare default rules (see Section B.3.1). The tree pattern is constructed using terminals as parent nodes and non-terminals as child nodes. Each terminal is associated to an intermediate representation opcode of fixed arity. Instead of a tree pattern, a rule may have a single non-terminal on its right hand side. Rules with a single non-terminal on its right hand side are known as *chain rules*. Tree patterns provide syntax based matching which suffices on many cases. Sometimes extra information must be considered when matching, and this can be done by writing a predicate expression. Predicate expressions are boolean expressions that may deny a match based on an undesired semantic property. The synthetize function is used to synthetize the non-terminal attributes, normally using attributes inherited from its children, once a match

takes place. The action function is the function called by the user when the optimal match is found. It is responsible for calling the action function of the tree pattern children.

```
public class Sample {

  public void stmt(PrintStream out)
  <int cost> [@@.cost < cost]
     :   IR.ISTORE(disp,reg)
    { @@.cost = 2+@2.cost+@3.cost; }
    = { @2(out);
        @3(out);
        out.print("st "+RA.name(@3.reg));
        out.println(",("+RA.name(@2.reg)+")"+@1.getOfs()); }
    | IR.IDEFINE(reg)
    { @@.cost = @2.cost; }
    = { @2(out); }
    ;

  private void reg(PrintStream out)
  <int cost, int reg> [@@.cost < cost]
     :   IR.IADD(reg,rc)
    { @@.cost = 1+@2.cost+@3.cost;
      @@.reg = @2.reg; }
    = { @2(out);
        @3(out);
        out.println("add "+@3.addr+","+RA.name(@2.reg)); }
    | IR.I2B(IR.ILOAD(disp))
    { @@.cost = 3+@3.cost;
      @@.reg = @3.reg; }
    = { @3(out);
        out.print("ld ("+RA.name(@3.reg)+")+"+@2.getOfs());
        out.println(","+RA.name(@3.reg));
        out.println("sx "+RA.name(@3.reg)); }
    | IR.ICONST [@1.getValue() == 0]
    { @@.cost = 0;
      @@.reg = RA.zero(); }
    = { }
    | disp
    { @@.cost = @1.cost;
      @@.reg = @1.reg; }
    = { @1(out); }
    ;
```

Figure B.2: Sample matcher specification.

Figures B.2 and B.3 show a sample matcher specification. It implements a toy back-end for a generic RISC machine based on a subset of the intermediate representation. Five non-terminals are defined: *stmt* to match statements, *reg* to match register expressions, *disp* to match reference expressions, *rc* to match register or constant expressions, and *con* to

match constant expressions. We assume the reader is familiar with tree rewriting systems, like [FHP92a, FHP92b, AGT89, Tji86, Tji93, ESL89, BDB90].

The `stmt` non-terminal is the only non-terminal defined as public, so it is the start symbol. It has a `cost` attribute that stores the number of cycles required to execute the whole statement. The optimality expression `@@.cost < cost` is used to choose the match with the smallest cost. The `@@.cost` identifies the cost of a recent match, `cost` is the best cost so far. The `stmt` non-terminal declares two tree pattern rules. The first rule matches integer field assignment (`istore`). For that rule, the cost of the match is computed by the sum of subexpressions `disp` and `reg` plus `2` cycles of memory access. In the action function, the code for each subexpression is generated first, then a *store* instruction is emitted on `out`. The second rule matches integer register definition (`idefine`). Since the rule already requires that the definition expression stays on a register, `reg`, the cost of the match is the cost inherited from `reg`. Similarly, the action function does not require an operation different from calling the `reg` action function.

The `reg` non-terminal matches expressions whose values are kept in registers. It has a `cost` attribute similar to the `stmt` non-terminal. In addition, it provides a `reg` attribute which stores the index of the register that holds the expression result value. The optimality expression defines the same behavior as in `stmt`, it chooses the match with the smallest cost. The `reg` non-terminal declares three tree pattern rules. The first matches an `iadd` opcode generating an `add reg,reg` or `add imm,reg` instruction, keeping the result in the register synthetized by `reg` with cost `1`. The second rule matches a zero valued expression. It uses a predicate expression to accept the match only if the `iconst` constant value is `0`. This rule has cost zero and no action, so the `reg` attribute is synthetized with the read-only zero valued register. The third rule is a chain rule, it means that matches for `disp` non-terminal will apply similarly for the `reg` non-terminal.

The `disp` non-terminal acts like the `reg` non-terminal, however, it deals with reference expressions. The first rule matches an expression that reads a reference field. The result value is kept in the register provided by a `reg` expression, the cost is increased by `2` and it outputs an `ld` instruction. The second rule matches a reference register use. It has no cost and an empty action function. The synthetized attribute `reg` is determined by the register allocator.

The `rc` non-terminal matches a constant or register expression. It computes the cost in the usual way, and has an assembly `String` attribute with the result operand. The first rule is a chain rule that matches constant expressions. It synthetizes the constant with an immediate syntax `$`. The second rule is also a chain rule that matches register expressions. The `addr` attribute is synthetized with the register name.

The `con` non-terminal matches integer constant expressions. There is no need for a cost attribute since there is no runtime cost for constant expressions. The only attribute synthetized by the `con` non-terminal is `value`, which holds the constant value of the expression. The first rule matches the `iconst` opcode which provides a constant value. The second rule performs *constant folding* for the `iadd` opcode, by computing the constant result of an expression at compile time.

The sample matcher, shown in Figures B.2 and B.3, provides a broader idea of basic matcher specification. The usage of the matcher is very simple: The user must instantiate a

```
  private void disp(PrintStream out)
  <int cost, int reg> [@@.cost < cost]
    :   IR.ALOAD(reg)
    { @@.cost = 2+@2.cost;
      @@.reg = @2.reg; }
    = { @2(out);
        out.print("ld ("+RA.name(@2.reg)+")+"+@1.getOfs());
        out.println(","+RA.name(@2.reg)); }
    | IR.AUSE
    { @@.cost = 0;
      @@.reg = RegAlloc.alloc(@1.getReg()); }
    = { }
    ;

  private void rc(PrintStream out)
  <int cost, String addr> [@@.cost < cost]
    :   con
    { @@.cost = 0;
      @@.addr = "$"+@1.value; }
    = { }
    | reg
    { @@.cost = @1.cost;
      @@.addr = RA.name(@1.reg); }
    = { @1(out); }
    ;

  private void con()
  <int value>
    :   IR.ICONST
    { @@.value = @1.getValue(); }
    | IR.IADD(con,con)
    { @@.value = @2.value+@3.value; }
    ;

}
```

Figure B.3: Sample matcher specification (continued).

```
public class Main {
  public static void main(String[] args) {
    IR.snode stmt = /* ...  snip ...  */;
    Sample matcher = new Sample(stmt);
    System.out.println("cost = "+matcher.stmt.cost);
    matcher.stmt(System.out);
  }
}
```

Figure B.4: Sample matcher usage.

matcher passing as argument the IR tree to be processed. After construction, the matcher will have performed tree pattern matching and dynamic programming on `stmt`. The attributes and action functions for each *accessible* non-terminal become available for usage. A sample usage can be seen on Figure B.4.

B.2   Implementation

The matcher generated from the `Sample` specification is implemented by the class `Sample`. An instance of class `Sample` is associated to each IR node in the IR tree, it stores attributes and best matches. For each non-terminal, two fields are added to the matcher class: the non-terminal rule index and a non-terminal attribute class reference. The non-terminal attribute class is an *inner class* that declares the attributes as fields and the optimality expression as a method. Figure B.5 shows the declaration of those fields and inner classes for non-terminals `stmt` and `reg`.

```
private byte stmt$id;
public stmt stmt;

public static final class stmt {

  public int cost;

  private boolean better$(final stmt $$) {
    return $$.cost < cost;
  }

}

private byte reg$id;
private reg reg;

private static final class reg {

  public int cost;
  public int reg;

  private boolean better$(final reg $$) {
    return $$.cost < cost;
  }

}
```

Figure B.5: Structures generated for the `reg` rule.

Additional information is generated on the matcher class. The dynamic programming algorithm is implemented on the constructor of the class. Extra instance fields are generated

in the class to implement a mirror of the IR tree. Figure B.6 shows the fields and constructors generated for class *Sample*. Field *node$* points to the IR opcode that the current node mirrors. Fields *left$*, *middle$* and *right$* are used to store the mirrors for the children of the IR opcode, according to its arity. Two constructors are generated, the public one is used by users to instantiate a matcher and the private one implements the *bottom-up* matching recursively.

```
private final TreeNode node$;
private Sample left$, middle$, right$;

public Sample(TreeNode node$) {
  this(null, node$);
}

private Sample(TreeNode root$, TreeNode node$) {
  this.node$ = node$;
  switch (node$.op()) {
  case IR.I2B: {
    final IR.i2b $1 = (IR.i2b)node$;
    left$ = new Sample(root$, $1.left());
    tree$4(root$, $1);
    break;
  }
  case IR.IADD: {
    final IR.iadd $1 = (IR.iadd)node$;
    left$ = new Sample(root$, $1.left());
    right$ = new Sample(root$, $1.right());
    tree$1(root$, $1);
    tree$2(root$, $1);
    break;
  }
  /* ..  snip ...*/
  default:
    if (node$.hasNext())
      root$ = node$;
    switch (node$.arity()) {
    case 0:  break;
    case 1:  left$ = new Sample(root$, node$.left()); break;
    case 2:  left$ = new Sample(root$, node$.left());
             right$ = new Sample(root$, node$.right()); break;
    case 3:  left$ = new Sample(root$, node$.left());
             middle$ = new Sample(root$, node$.middle());
             right$ = new Sample(root$, node$.right()); break;
    default:  throw new Error("Illegal arity");
    }
  }
}
```

Figure B.6: Matcher variables and constructors.

Bottom up matching is achieved by first *switching* on the opcodes that appear on the root of tree patterns. Once the opcode has been identified, a matcher node is created for each of its children by invoking the constructor recursively. After that, each tree pattern is tested and possibly replaced by calling `tree$` methods.

The action function is implemented by testing the rule index of a particular non-terminal for the current matcher node. If the rule index is `0` then there was no match and an `Error` is thrown. Otherwise the associated action code is executed, if provided by the user. Figure B.7 shows the action function implementation generated for the non-terminal `reg` of the `Sample` matcher.

```
private final void reg(PrintStream out) {
  final Sample $$ = this;
  switch (reg$id) {
  case 0:  throw new Error("No match");
  case 1:  {
    final IR.iadd $1 = (IR.iadd)$$.node$;
    final Sample $2 = $$.left$;
    final Sample $3 = $$.right$;
    $2.reg(out);
    $3.rc(out);
    out.println("add "+$3.rc.addr+","+RA.name($2.reg.reg));
    break;
  }
  case 2:  {
    final IR.i2b $1 = (IR.i2b)$$.node$;
    final IR.iload $2 = (IR.iload)$$.left$.node$;
    final Sample $3 = $$.left$.left$;
    $3.disp(out);
    out.print("ld ("+RA.name($3.disp.reg)+")+"+$2.getOfs());
    out.println(","+RA.name($3.disp.reg));
    out.println("sx "+RA.name($3.disp.reg));
    break;
  }
  case 3:  {
    final IR.iconst $1 = (IR.iconst)$$.node$;

    break;
  }
  case 4:  {
    final Sample $1 = $$;
    $1.disp(out);
    break;
  }
  default:  throw new Error("Unimplemented rule");
  }
}
```

Figure B.7: Action method generated for the `reg` rule.

Figure B.8 shows some of the `tree$` functions generated for the `Sample` matcher. The rule match is computed by first checking if there is a pattern match. Then, for each rule that declares that pattern as the right hand side, the predicate expression is checked for a semantic test. If the semantic test passes, the synthetized function is used to synthetize the attributes of the non-terminal. At last, the optimality function for that non-terminal is used to compare the new match with the best match so far. If no best match results is found, the new match is accepted automatically.

The function `tree$0` computes the match for pattern `IR.ICONST`. This pattern appears in a rule of non-terminal `con` and a rule of non-terminal `reg`. Since this pattern is composed of a single root terminal — and has no children — the syntatic match has already been completely computed. In a first moment, the rule of non-terminal `con` is handled. Its `value` attribute is computed in a temporary register and, since `con` does not define an optimality expression, the match is only registered if it is the first occurred (`con$id = 0`). Since `con` is the right hand side of chain rules, upon a match we must check the match on the letf hand side of each of those chain rules. This is done by calling method `con$closure`. Next, the same occurs when the rule of non-terminal `reg` is handled. The associated predicate expression is checked, and the match occurs only if the constant value is `0`. The attributes of `reg` are synthetized and the rule is accepted if no match has been accepted so far, or if it is better than the current best match. The chain rules that have `reg` as right hand side are checked by calling method `reg$closure`.

For the patterns `IR.IADD(reg,rc)` and `IR.I2B(IR.ILOAD(disp))`, similar code is generated to check matches on methods `tree$1` and `tree$4` respectively. However, since those patterns are a bit more than just childless terminals, the code is generated enclosed by a syntatic test expression. The syntatic test expression checks if the non-root terminals of the pattern occur in the IR tree and if there is a match in each IR subtree associated to every non-terminal.

For each non-terminal that appears in the right hand side of a chain rule, a *closure* method is generated. Figure B.9 shows the implementation of closure methods for non-terminals `rc`, `disp` and `reg`. Those methods are implemented just like tree matching methods. Chain rules may include cycles in the grammar, which are implemented by recursive calls of closure methods. To avoid infinite looping during tree matching, the cost (or whatever metric used to achieve optimality) must increase when applying a direct or indirect recursive chain rule.

B.3 RING Extensions

This section describes the extensions to the bare tree rewriting tool described above. These extensions were designed to reduce the developing time of large complex matchers.

B.3.1 Default Rules

Default rules are rules that match only if no other rule matches. Although they are not considered to be a match by the rules that use the associated non-terminal, default rules provide a mechanism to synthetize and write actions when no match takes place.

```
private final void tree$0(TreeNode root$, IR.iconst $1) {
  final con $$ = new con();
  $$.value = $1.getValue();
  if (con$id == 0) {
    con = $$; con$id = 1; con$closure(root$);
  }
  if ($1.getValue() == 0) {
    final reg $$ = new reg();
    $$.cost = 0;
    $$.reg = RA.zero();
    if (reg$id == 0 || reg.better$($$)) {
      reg = $$; reg$id = 3; reg$closure(root$);
    }
  }
}

private final void tree$1(TreeNode root$, IR.iadd $1) {
  if (left$.reg$id != 0 && right$.rc$id != 0) {
    final Sample $2 = left$;
    final Sample $3 = right$;
    final reg $$ = new reg();
    $$.cost = 1+$2.reg.cost+$3.rc.cost;
    $$.reg = $2.reg.reg;
    if (reg$id == 0 || reg.better$($$)) {
      reg = $$; reg$id = 1; reg$closure(root$);
    }
  }
}

private final void tree$4(TreeNode root$, IR.i2b $1) {
  if (left$.node$.op() == IR.ILOAD
      && left$.left$.disp$id != 0) {
    final IR.iload $2 = (IR.iload)left$.node$;
    final Sample $3 = left$.left$;
    final reg $$ = new reg();
    $$.cost = 3+$3.disp.cost;
    if (reg$id == 0 || reg.better$($$)) {
      reg = $$; reg$id = 2; reg$closure(root$);
    }
  }
}
```

Figure B.8: Tree matching methods.

```
private final void con$closure(final TreeNode root$) {
  final Sample $1 = this;
  final rc $$ = new rc();
  $$.cost = 0;
  $$.addr = "$"+$1.con.value;
  if (rc$id == 0 || rc.better$($$)) {
    rc = $$; rc$id = 1;
  }
}

private final void disp$closure(final TreeNode root$) {
  final Sample $1 = this;
  final reg $$ = new reg();
  $$.cost = $1.disp.cost;
  $$.reg = $1.disp.reg;
  if (reg$id == 0 || reg.better$($$)) {
    reg = $$; reg$id = 4; reg$closure(root$);
  }
}

private final void reg$closure(final TreeNode root$) {
  final Sample $1 = this;
  final rc $$ = new rc();
  $$.cost = $1.reg.cost;
  $$.addr = RA.name($1.reg.reg);
  if (rc$id == 0 || rc.better$($$)) {
    rc = $$; rc$id = 2;
  }
}
```

Figure B.9: Closure methods for chain rules.

```
public void stmt(PrintStream out) throws NoMatchException
<int cost> [@@.cost < cost]
  :  IR.ISTORE(disp,reg)
  /* ...  snip ...  */
  | default
  { @@.cost = Integer.MAX_VALUE; }
  = { throw new NoMatchException(@1.toString()); }
  ;

public class NoMatchException extends Exception {

  public NoMatchException() { }

  public NoMatchException(String message) {
    super(message);
  }

}
```

Figure B.10: Default rule syntax.

Figure B.10 shows a default rule added to non-terminal `stmt` of matcher `Sample`. This default rule was declared to throw a `NoMatchException` (instead of internal `Error`) when there is no match.

The implementation of default rules is very simple. In the matcher constructor, before returning, we check the rule indices for all non-terminals that declare a default rule. For those whose index is `0`, we apply the default rule. In the action function, the code declared for default rules is emitted for the c ase value `0`. That can be seen on Figure B.11.

B.3.2   Non-Terminal Templates

Non-terminal templates are useful to declare multiple similar non-terminals with the same rules. Instead of writing a declaration for each of many similar non-terminals, the user writes the template declaration and instantiates it by using it with a defined parameter.

Figure B.12 shows a template, `reg<RID>`, which is instantiated as `reg<GR0>` and `reg<GR1>` on `stmt` non-terminal. That way, instead of having a match for generic registers, appropriate for RISC machines, you can make the matching for each register separately, what saves time when describing CISC matchers.

The implementation of non-terminal templates is straightforward. It is done by substituting non-terminal templates by multiple specialized non-terminals in the grammar. For each different combination of template parameters in a non-terminal template, an associated non-terminal is declared to implement that template instance. In the declaration of this new non-terminal, its actual template parameters are replaced on subsequent template uses on the right hand side of its rules. This may produce new template instances which are processed the same way. Once all non-terminal template uses are replaced by new non-terminals, they can be

```
private Sample(TreeNode root$, TreeNode node$) {
  this.node$ = node$;
  /* ...  snip ...  */
  if (stmt$id == 0) {
    final stmt $$ = new stmt();
    final TreeNode $1 = node$;
    $$.cost = Integer.MAX_VALUE;
    stmt = $$;
  }
}

public final void stmt(PrintStream out) throws NoMatchException {
  final Sample $$ = this;
  switch (stmt$id) {
  case 0:  {
    final TreeNode $1 = $$.node$;
    if (true) {
      throw new NoMatchException($1.toString());
    }
    break;
  }
  /* ...  snip ...  */
}
```

Figure B.11: Default rule implementation.

```
public static final int ridGR0 = 16, ridGR1 = 17;

public void stmt(PrintStream out)
<int cost> [@@.cost < cost]
  /* ...  snip ...  */
  | IR.IDEFINE(reg<GR0>) { @@.cost = @2.cost; }
  = { @2(out, ridGR0); }
  | IR.IDEFINE(reg<GR1>) { @@.cost = @2.cost; }
  = { @2(out, ridGR1); }
  ;

private void reg<RID>(PrintStream out, int rid)
<int cost> [@@.cost < cost]
  :  IR.IADD(reg<RID>,rc) { @@.cost = 1+@2.cost+@3.cost; }
  = { @2(out, reg); @3(out);
      out.println("add "+@3.addr+","+RA.name(rid)); }
  /* ...  snip ...  */
  ;
```

Figure B.12: Rule template syntax.

```
public void stmt(PrintStream out)
<int cost> [@@.cost < cost]
  /* ...  snip ...  */
  | IR.IDEFINE(reg_GR0_) { @@.cost = @2.cost; }
  = { @2(out, ridGR0); }
  | IR.IDEFINE(reg_GR1_) { @@.cost = @2.cost; }
  = { @2(out, ridGR1); }
  ;

private void reg_GR0_(PrintStream out, int rid)
<int cost> [@@.cost < cost]
  :  IR.IADD(reg_GR0_,rc) { @@.cost = 1+@2.cost+@3.cost; }
  = { @2(out, reg); @3(out);
      out.println("add "+@3.addr+","+RA.name(rid)); }
  /* ...  snip ...  */
  ;

private void reg_GR1_(PrintStream out, int rid)
<int cost> [@@.cost < cost]
  :  IR.IADD(reg_GR1_,rc) { @@.cost = 1+@2.cost+@3.cost; }
  = { @2(out, reg); @3(out);
      out.println("add "+@3.addr+","+RA.name(rid)); }
  /* ...  snip ...  */
  ;
```

Figure B.13: Rule template implementation.

removed from the specification. The result grammar is show in Figure B.13.

B.3.3   Non-Terminal Inlining

Non-terminal inlining allows the user to declare a non-terminal that comprises a set of sub-patterns to be used by other non-terminals. This saves times when writing patterns having subpatterns. The special non-terminal, marked with an "abstract" modifier, will not be considered by the matcher as a point of best match choice. For this reason, "abstract" non-terminals cannot declare optimality expressions. This "abstract" non-terminal feature provides the same behavior as if the right hand side of the rules were "inlined" in the patterns that use the "abstract" non-terminal. That is why it is called non-terminal inlining.

```
public void stmt(PrintStream out)
<int cost> [@@.cost < cost]
  /* ...  snip ...  */
  | IR.IDEFINE(greg) { @@.cost = @2.cost; } = { @2(out); }
  ;


private abstract void greg(PrintStream out)
<int cost, int reg>
  :  reg<GR0>
  { @@.cost = @1.cost; @@.reg = ridGR0; } = { @1(out, ridGR0); }
  | reg<GR1>
  { @@.cost = @1.cost; @@.reg = ridGR1; } = { @1(out, ridGR1); }
  ;
```

Figure B.14: Non-terminal inlining syntax.

Figure B.14 shows the use of non-terminal inlining to create a unified generic register non-terminal, *greg*, while keeping independent matching for each one of them. This way, instead of declaring a different pattern for each generic register template instance, the user writes one single pattern that captures all generic registers.

Similarly to non-terminal templates, non-terminal inlining implementation is straightforward and can be achieved by rewriting the matcher specification. For each "abstract" non-terminal rule, a new non-terminal is declared to match only the associated pattern. This prevents the matcher from choosing patterns at that point, since at most one match will occur. Then, each rule that uses an "abstract" non-terminal is replaced by many similar rules, one for each non-terminal recently associated to each right hand side. This can be seen on Figure B.15.

Non-terminal inlining is a powerful mechanism to express many complex patterns without having to write all the combinations. "Abstract" non-terminals reachable by themselves are not allowed. This would require the generation of an infinite pattern matcher which is not currently supported by RING.

```
public void stmt(PrintStream out)
<int cost> [@@.cost < cost]
  /* ...  snip ...  */
  | IR.IDEFINE(greg0) { @@.cost = @2.cost; }
  = { @2(out); }
  | IR.IDEFINE(greg1) { @@.cost = @2.cost; }
  = { @2(out); }
  ;

private void greg0(PrintStream out)
<int cost, int reg>
  :  reg<GR0>
  { @@.cost = @1.cost; @@.reg = ridGR0; } = { @1(out, ridGR0); }
  ;

private void greg1(PrintStream out)
<int cost, int reg>
  :  reg<GR1>
  { @@.cost = @1.cost; @@.reg = ridGR1; } = { @1(out, ridGR1); }
  ;
```

Figure B.15: Non-terminal inlining implementation.

List of Figures

List of Tables

Bibliography

[ADM98]   Ole Agesen, David Detlefs, and J. Elitot B. Moss. Garbage collection and local variable type-precision and liveness in Java Virtual Machines. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998.

[AG00]   Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, third edition, June 2000.

[AJ76]   A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[ASU86]   Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Aze99]   Ana Azevedo. Java anotation — aware Just-In-Time (AJIT) compilation system. In *ACM Java Grande Conference*, June 1999.

[Bot97]   Per Bothner. *A GCC-based Java Implementation*. Cygnus Solutions, February 1997.

[BW88]   Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software — Practice & Experience*, 18(9):807–820, September 1988.

[CAC+81]   Gregory Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[CFR+91]   Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[Cha87]   David R. Chase. An improvement to bottom up tree pattern matching. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 168–177, January 1987.

[CL97]   Patrick Chan and Rosanna Lee. *The Java Class Libraries, Volume 2: java.applet, java.awt, java.beans*. The Java Series. Addison-Wesley, second edition, October 1997.

[CLK98]   Patrick Chan, Rosanna Lee, and Doug Kramer. *The Java Class Libraries, Volume 1: java.io, java.lang, java.math, java.net, java.text, java.util*. The Java Series. Addison-Wesley, second edition, March 1998.

[CLK99]   Patrick Chan, Rosanna Lee, and Doug Kramer. *The Java Class Libraries, Volume 1: 1.2 Supplement*. The Java Series. Addison-Wesley, second edition, May 1999.

[Coc70]   J. Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20–25, July 1970.

[CU90]   Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the ACM SIGPLAN'90 Symposium on Compiler Construction*, 1990.

[CUL91]   C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF, a dynamic-typed object-oriented language based on prototypes. *Lisp and Symbolic Computation*, 4(3):243–281, 1991.

[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(3):213–226, September 1992.

[FKR+99]   Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for java. Technical Report MSR-TR-99-33, Microsoft Research, June 1999.

[Fra77]   Christopher W. Fraser. *Automatic Generation of Code Generators*. PhD thesis, Yale University, 1977.

[Fra89]   Christopher W. Fraser. A language for writing code generators. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 238–245, July 1989.

[GJS96]   James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, June 1996.

[GRI83]   A. Goldberg, D. Robson, and D. H. H. Ingalls. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[GS99]   David Gay and Bjarne Steensgaard. Stack allocating objects in java. Technical report, Microsoft Research, 1999.

[GYT96a]   James Gosling, Frank Yellin, and The Java Team. *The Java Application Programming Interface, Volume 1: Core Packages*. The Java Series. Addison-Wesley, May 1996.

[GYT96b]   James Gosling, Frank Yellin, and The Java Team. *The Java Application Programming Interface, Volume 2: Window Toolkit and Applets*. The Java Series. Addison-Wesley, June 1996.

[HO82]   Christoph Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.

[Hop71]   John E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971.

[IBM98]   IBM Corporation. *IBM High Performance Compiler for Java: An Optimizing Native Code Compiler for Java Applications*, July 1998.

[Int97a]   Intel Corporation. *Intel Architecture Software Developer's Manual: Volume 1: Basic Architecture*, 1997. Order Number 243190.

[Int97b]   Intel Corporation. *Intel Architecture Software Developer's Manual: Volume 2: Instruction Set Reference*, 1997. Order Number 243191.

[JL96]   Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[KG97]   A. Krall and R. Grafi. CACAO a 64-bit JavaVM just in time compiler. *Java for Computational Science and Engineering — Simulation and Modeling II*, 9(11):1017–1030, November 1997.

[KR88]   Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[LB98]   Sheng Liang and Gilad Bracha. Dynamic class loading in the Java Virtual Machine. In *Proceedings of the 13th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98)*, Vancouver, BC, Canada, October 1998.

[Les99]   Dmitry Leskov. *JET, Deployment Environment that Boosts Performance and Saves Resources*. Excelsior, December 1999. Whitepaper.

[LY96]   Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, June 1996.

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

[Mey91]   Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, 1991.

[MMBC97] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: a flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the Third Conference on Object-Oriented Technologies and Systems (COOTS'97)*, 1997.

[MMS98] Samuel P. Midkiff, José E. Moreira, and Marc Snir. Optimizing array reference checking in java programs. Research Report RC 21184(94652), IBM Research Division, May 1998.

[Mor98] Robert Morgan. *Building an Optimizing Compiler*. Butterworth-Heinemann, 1998.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[Nat95] National Institute of Standards and Technology, U.S. Department of Commerce. *Secure Hash Standard*, April 1995. Federal Information Processing Standards Publication (FIPS PUB) 180-1.

[Ner58] Anil Nerode. Linear automaton transformations. In *Proceedings of the American Mathematical Society*, pages 541–544, 1958.

[OSM⁺00] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT: An open-ended, reflective JIT compiler framework for Java. In *ECOOP 2000*, pages 362–387, 2000.

[PL87] Eduardo Pelegrí-Llopart. *Tree Transformations in Computer Systems*. PhD thesis, University of California, December 1987.

[PTB⁺97] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications: A Way Ahead of Time (WAT) compiler. In *Proceedings of the Third Conference on Object-Oriented Technologies and Systems (COOTS'97)*, 1997.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[Sun97] Sun Microsystems. *Java Native Interface Specification*, May 1997.

[Sun98] Sun Microsystems. *The Java HotSpot Virtual Machine Architecture, A White Paper About Sun's Second Generation Java Virtual Machine*, March 1998. Whitepaper.

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, 1984.

[Yel96]   Frank Yellin. *The JIT Compiler API*. Sun Microsystems, October 1996.

[YMP$^+$99] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik Altman. LaTTe: A Java VM Just-In-Time compiler with fast and efficient register allocation. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, New Port Beach, October 1999.

Index