Abstract

Just-In-Time (JIT) compilation is an acceleration technique that has played a central role in order-of-magnitude performance improvements of Java applications that run on high-end desktop computers and server workstations. In their pursuit of more speed, modern Java implementations have steadily grown in complexity, and as a result, resource consumption has escalated. The imbalance of this tradeoff must be addressed in order to bring the concept of JIT compilation to resource-constrained devices in a cost-effective manner. This text addresses this situation. A framework for JIT compilation that specifically caters to the resource constraints of new generations of small devices is presented. Preliminary results obtained from a prototype implementation show average speedup of 5.5 over a conventional interpreter-based Java implementation, with only a 15% increase in the static memory footprint and an adjustable, highly predictable dynamic footprint.

Acknowledgements

The author wishes to thank his advisor, Jon Olav Hauglid, for showing faith and keen interest in the task at hand, and for providing valuable feedback on this text.

Contents

Chapter 1

Introduction

The Java platform has shown a lot of promise in supporting the development and deployment of next-generation wireless devices and embedded systems. The adoption of Java reflects the computer industry's general move towards standards-based, reusable middleware, fueled by the desire to reduce costs and time-to-market. However, the high-level, platform-independent nature of Java entails runtime overhead not found in previous generations of development languages, and this overhead can be significant on platforms where memory and processing power is limited compared to the high-end desktop and server market. High performance at low footprint is crucial if dynamic languages like Java are to succeed on a diversity of platforms. Addressing this concern, this text presents a lightweight performance-enhancing mechanism suitable for Java runtime environments that are faced with significant memory limitations.

1.1 Background and Motivation

Whereas the 1990s belonged to the desktop computer, the new decade sees the exponential growth of a different class of computer. Wireless, mobile devices, such as cell phones and PDAs, have quickly become an integral part of everyday life throughout the world. Furthermore, in addition to the devices we see and normally think of as computers, there are now equally small and even smaller computers embedded in just about every electronic appliance in our homes, work offices – and, perhaps within another decade, in our bodies [NW01]. This situation – computers being anywhere and everywhere – is labeled as ubiquitous [TBS99, Cora], or pervasive [NW01], computing.

Despite their small physical size and cheap price, these devices are increasingly expected to deliver functionality and performance comparable to desktop computers. This is a natural evolution, as the user base becomes comfortable with the new technology. However, just as Moore's Law is not a law of nature, but rather a result of the relentless, collective efforts of the semiconductor industry, it is not a given that the implementers of small-scale devices will be able to continue to meet ever more sophisticated customer demands at an affordable price. Additionally, in a seemingly absurd twist, even as project complexity increases the time-to-market is forced to new lows, in order for companies to stay afloat in a highly competitive market.

In addition to more complex functionality becoming commonplace in this new wave of devices, they are also expected to be more *dynamic* than previous generations. Rather than hardwired circuitry that can only perform a single, narrow task, these devices implement a computation model very similar to a desktop computer: A general-purpose processor, memory, and a combination of general- and special-purpose software that runs on top, backed up by an operating system and additional domain-specific hardware. By accessing a network, the functionality of the device can be upgraded or customized as appropriate, in a seamless and transparent way. This allows a device to adapt and grow over time. Traditional implementation paradigms have proved incapable of supporting the development and deployment of such systems. Thus, the industry has looked towards new solutions.

Java is an attractive alternative for a number of reasons. Java is platform-independent, meaning that implementation of common behaviour can be moved to and shared by many similar devices, with practically all devices sharing a common core; this in turn provides for a high level of *interoperability* among devices. Java, due to its high-level, object-oriented nature, allows developers to abstract away from the details of hardware and low-level driver software, thus facilitating design reuse and the ability for applications to remain unaffected by the fluctuations in device and network technology. Java is *secure*, avoiding by design many of the security pitfalls inherent in previous generations of development platforms. Due to its success on the desktop computer, Java already enjoys a large base of developers whose competence can be leveraged. Furthermore, concerted standardization efforts are continually making the Java platform more relevant and easily accessible to a diversity of manufacturers and developers.

In order to achieve platform independence, the Java platform depends on an abstraction layer that masks the details of the underlying platform-specific software and hardware. The implementation of such a layer has an inherent price in terms of space and time overhead. Thus, a considerable challenge is associated with the adoption of Java on platforms where resources are scarce even *before* Java is factored into the equation. Due to the relentless miniaturization of many classes of device, manufacturers cannot simply sit on the fence and wait for Moore's law to kick in, either. First of all, price is a prime consideration, and it takes some time for memory prices to decline to the point where mass production of a small device with large memory becomes affordable; by that time, someone else might have already captured a large market share. Second, instead of getting more memory, heavily miniaturized devices may depend on Moore's law only to enable further shrinking in physical size, leaving memory capabilities relatively constant over a long time. Third, Moore's Law does not apply to battery life [Mic05].

Java implementations targeting high-end desktop and server systems have focused intently on speed, which has come at a high cost in terms of added complexity and the memory consumption that comes with it; this development may be viewed as a natural consequence of more and more resources becoming available to these classes of computer. However, the new generations of wireless and mobile devices are characterized first and foremost by small physical size, modest computing power and limited power supply; essentially, resources are bounded, not unbounded. Existing Java technology cannot simply be moved to these new platforms, because the resources needed to support such complex technology are not available. Implementing high-performance Java while catering to the premise of resource constraints requires a rethinking of the concepts that underlie performance enhancement, and an attention to memory concerns from the very beginning. The *tradeoff* between footprint and performance must be attacked in a way that avoids the escalation of system complexity in favour of speed.

1.2 Purpose

This work addresses the need for high performance in Java environments running on systems with so little memory that state-of-the-art Java implementation technology targeting desktop and server systems is not feasible to implement or port in a straightforward fashion. The purpose of the proposed system is to significantly accelerate the performance of a Java runtime environment targeting platforms with limited memory, *without* blowing the already highly stressed memory budget. Rather, the memory overhead due to the acceleration technique is intended to be low and predictable, so that the resulting system may be used on platforms with limitations similar to those of the original virtual machine. Accelerating the performance of the Java runtime environment yields faster response times for existing applications, which enhances the user experience on interactive platforms, such as mobile phones, and may even be critical in real-time embedded systems, such as hospital equipment. Furthermore, acceleration allows new applications to contain more sophisticated and performance-demanding features than what was previously possible.

1.3 Approach

The most effective way of improving the performance of Java implementations is to address the mechanism for execution of the Java instruction set (bytecode). The proposed system intends to enhance performance by introducing a well-known technique, Just-In-Time (JIT) compilation, into the execution engine of an interpreter-driven virtual machine. To reduce implementation efforts, existing Java virtual machine technology is leveraged. In recent years, several open-source projects enabling further research into virtual machine technology have surfaced, including efforts from major forces in the industry like IBM [IBM], Intel [Corb] and Sun. In this project, Sun's K Virtual Machine (KVM) [Mic00] forms the backbone of the system. KVM has successfully been used in several related projects, both academic [Sha02, ea05] and commercial [Cora]. KVM is the reference implementation for Sun's Connected Limited Device Configuration (CLDC) [Mica] platform, intended for use in resource-constrained environments; the "K" in KVM refers to machines that have hundreds of kilobytes of RAM, rather than tens or hundreds of megabytes. KVM offers all the basic services that are required of a virtual machine, and is thus a complete infrastructure upon which extensions, such as the one described in this text, may be built. Furthermore, since KVM already addresses memory concerns, it provides a set of conventions and policies for memory management that serve as the basis of further low-overhead implementation.

Following implementation, the potential of the acceleration technique will be evaluated by running a set of Java applications with the new technique enabled. The results will be compared against the results achieved by running the applications on the original KVM, and the improvements (or lack thereof) will be quantified and used to decide on future directions.

1.4 Scope

The work concerns the design, implementation and empirical assessment of a low memory footprint acceleration technique for Java bytecode execution. The solution includes the integration of the acceleration technique with an existing virtual machine. Memory behaviour and power utilization, while certainly of high interest, are outside the scope; low footprint and high performance are the main concern. Real-time considerations are not explicitly addressed.

1.5 Outline

The rest of this text is structured as follows. Chapter 2 gives an introduction to resource-constrained systems, and the motivations for bringing Java to such platforms. Subsequently, the focus turns to the challenges of implementing Java with low memory footprint while maintaining sufficient performance. Chapter 3 presents a lightweight architecture that addresses the performance of an existing virtual machine intended to run in little memory. Chapter 4 describes the implementation of a prototype that adheres to the architectural specification. Chapter 5 presents experimental results that have been obtained from the prototype. Finally, chapter 6 summarizes the important findings and suggests future work.

Chapter 2

Background

This chapter gives an overview of the central issues concerning the adoption of Java in resource-constrained environments, and the potential solutions that exist. The chapter is laid out as follows. Section 2.1 gives an introduction to systems where one or more of the resources memory, processing power and energy supply are limited, and offers a discussion of the principal differences between these classes of systems and the desktop computer. The financial and technical challenges faced by manufacturers and developers of ever more sophisticated generations of resource-constrained systems are highlighted, as are the primary paths that have been taken in recent years in order to meet these challenges. This leads into the motivation for using Java. Section 2.2 gives an overview of the Java platform, devoting particular attention to the branch of Java that caters to resource-constrained environments. The principal components of the Java runtime environment (JRE), the infrastructure that supports the execution of Java applications, are presented. The challenges that concern efficient implementation of the JRE and its core component, the Java virtual machine (JVM), are discussed. Section 2.3 discusses how a crucial component of the JVM, bytecode execution, can be accelerated. Particular attention is given to Just-In-Time (JIT) compilation, a highly effective, but proportionally complex, acceleration technique. The difficulties of adopting JIT compilation techniques in a low-memory environment are discussed. Section 2.4 contains a thorough discussion of levels of granularity, a very important matter in JIT compilation, and one that will be revisited often in subsequent chapters. Finally, section 2.5 concludes the chapter with a discussion of related work.

2.1 Resource-Constrained Devices

It seems that wireless, mobile computing is becoming more pervasive by the minute – one can almost smell it. After all, it does not seem all that long ago that a cellular phone was something privileged, rich businessmen got to carry around in a heavy backpack. Today, a similar apparatus is so small that the only way of operating it might very well be by pinning a particular pixel on a tiny display. This *relentless miniaturization* [NW01] is not likely to come to a halt anytime soon.

2.1.1 Smaller, Cheaper, More Advanced

Not only have devices shrunk in physical size and become affordable to laymen, however. The level of *sophistication* of even the tiniest computing device – and, along with it, the sophistication of end users – has grown at an unprecedented rate. Increasingly, functionality that we have become used to finding on desktop computers is making its way to smaller devices [D.M98] that we did not associate with online connectivity only few years ago; users now expect to be able to access multimedia and read email on their phone. This also illustrates the trend that functionality which was previously associated with separate devices are now being combined in the same device [Cora].

On the technical level, this revolution can be attributed to the *convergence* of the computer, telecommunication and associated industries [Mic03, DSCS03]. It is this convergence that enables the smooth interaction between so many conceptually diverse applications; today, or at least in the near future, "everything is connected to the Internet" [Mic00]. A major result of this convergence is that small devices are not only becoming similar to desktop computers in terms of functionality, but also in terms of the underlying architecture and hardware. Whereas these devices previously implemented much of their functionality directly in hardware, there has been a shift towards the use of more generic hardware with (several) layers of software on top [SBCK03]. This has opened up for the same level of inherent *dynamic* capabilities – soft upgrades, customizations and extensions – as the desktop computer has already enjoyed for quite some time. Sun [Mic03] formulates the difference as follows:

> The line between [more general-purpose computers and specialized computers] is defined more by the total memory budget and the physical screen size of the device, rather than by specific functionality or by a certain type of connectivity.

Thus, *resource constraints*, usually pertaining to one or more of memory, processing power and energy consumption, is a defining difference. Implementing the same functionality found on a desktop computer on a device facing these constraints presents major challenges; yet, increasingly, this is what users expect.

2.1.2 Challenges of a New Era

Less available memory ultimately means that less functionality can be included. Thus, applications must be very selective about what is actually included; configurability and customizability are two important keywords in this regard [Mic03].

However, there are other challenges besides available resources. A specialized computer must be *robust* [NW01]; it cannot afford to crash, especially if it is mission-critical. Due to its networkability, a *secure* environment is necessary so that malicious code or data cannot make its way into the system.

Collectively, the demands of these devices stipulate that memory limitations, networking, security and any other fundamental limitation or requirement must all be factored into the design from its conception and throughout its lifetime.

2.1.3 Consequences, Solutions and More Consequences

Increased system complexity has lead to larger production efforts, yet development cycles have decreased; it is a breakneck race to ship new products before the competitor does, at a lower price. In order to cope with these two factors, increased complexity and faster time-to-market, *object-oriented* development methodologies have emerged as a natural solution [SBCK03]. Object orientation inherently affords design reuse, and standardization based on interfaces. Interface-based software built from standardized components facilitates portability, not to mention that it eases the overall management of systems of high complexity.

However, object-oriented implementations bring with them the problem of overly general code, which may be viewed as a consequence of design reuse. There is a fundamental tradeoff in this regard, as pointed out by Schultz et al [SBCK03]: General code is big, slow and reuseable, while specialized code is small, fast and less reuseable.

The solution to this has been to design *layered* middleware architectures with *small interfaces* [NW01]. This enables developers to configure their particular system to a high degree. However, it is difficult for a standards committee to decide how much functionality to include at each layer, and whether or not to allow optional features; a consequence of this has been fragmentation [CD03], due to some standards' inability to be adopted to a highly specialized purpose in the real world. As stated by Corsaro et al [CSKO02], "[...] there is a growing mismatch between what is provided by the middleware and what is needed by any particular application."

2.1.4 The Status of Java

When Java first appeared in 1995, it did in fact target resource-constrained devices [TBS99]. However, it was not embraced by the community due to being too large, slow and unwieldy [CD03]. Instead, Java found its niche on more powerful desktop computers in capacity of delivering dynamic content to Web browsers. Next, when Java performance on the desktop was improved by orders of magnitude largely due to JIT compilation, Java became a platform for fullfledged desktop and server applications. In the meantime, the original target group of resource-constrained devices were starting to become more powerful as well. So, around 1998, the process of scaling the Java platform down to a low footprint began. In 2000, industry-concerted standards were published, and this time around reactions were a lot more positive, although the concerns related to speed, footprint and fragmentation remained. Nevertheless, since then, the platform has evolved and gained more popularity. Thus, Java may therefore be said to have "come full circle".

The popularity of Java can largely be attributed to its attention to security and networkability; its platform independence; its ease of use; its well-designed class libraries; and its compliance with, and contributions to, industry standards.

Comp and Dobbing [CD03] offer some predictions for Java in the wireless market:

> Java is predicted to be the dominant technology for wireless client devices through to 2007. 50% or more of the applications running on wireless client devices will be Java applications. In terms of numbers, it is predicted that there will be 691.6 million Java-enabled phones in the marketplace by 2007 out of a total 727.3 million handsets (95% of the market).

Java, like other standards-based middleware, has a price in terms of space and time overhead. Coming up with new techniques to mitigate this overhead will be key to Java's continued success.

2.1.5 Likely Future Directions

As the prices of memory drop over time, cellular phones and other devices of similar physical size can afford to have more memory; however, they are still far behind the desktop computer. Furthermore, with new high-speed networks such 2.5G and 3G [DSCS03] becoming available, they will need all the memory and processing power they can get in order to support new functionality; it is a well-established fact that the problem size tends to increase at least as fast as available resources do. Thus, improving performance at low cost will be no less important in the foreseeable future than it is today.

In the future, as computing becomes even more ubiquitous and miniaturization continues unabatedly, it is quite possible that new classes of miniscule devices will emerge that face the same resource constraints as the so-called constrained devices of today. Thus, many devices are likely to still be working in kilobytes of RAM rather than megabytes for quite some time. For instance, Sun's JavaCard [Micb] technology has long relied on 8- and 16-bit processors, only a few kilobytes of RAM and a tiny subset of standard Java functionality; however, the next generation of cards will include 32-bit processors and more memory (hundreds of kilobytes as opposed to tens), making them more comparable to the Java implementations currently found in cellular phones, and requiring new, efficient Java implementations that put the card's resources to optimal use. The JavaCard technology has already enjoyed large-scale commercial deployment in SIM cards, credit cards and ID cards, to name a few uses.

2.2 The Java Platform

Java is not just a programming language, but an infrastructure for development and deployment of applications. This section gives an overview of the Java platform. First, the concepts that lie behind Java's ability to be adopted across a wide spectrum of related and unrelated computers are presented. Next, a Java configuration that addresses the needs of resource-constrained systems, the Connected, Limited Device Configuration (CLDC), is introduced, followed by an overview of Sun's CLDC reference implementation, which is based on the K Virtual Machine (KVM). Lastly, an overview of the Java Runtime Environment (JRE) is given, and the difficulties of providing a high-performance JRE are highlighted.

2.2.1 Java Editions and Configurability

In order to extend the scope of Java to cover a wide spectrum of computing devices, the Java platform had to somehow be tailored to the resources and needs of different classes of computer. The solution has been to split the Java platform into *editions*. There are currently three major editions [Mic00]: J2EE (Enterprise Edition), targeting workstations and application servers, J2SE (Standard Edition), targeting desktop computers and laptops, and finally J2ME (Micro Edition), targeting resource-constrained systems. Figure 2.1 on the following page shows the major editions of the Java platform[1].

Sun [Mic00] describes the concept of an edition as follows:

> Each Java edition defines a set of technology and tools that can be used with a particular product: Java virtual machines that fit inside a wide range of computing devices; libraries and APIs specialized for each kind of computing device; and tools for deployment and device configuration.

One of the intents of the major editions is that they should be rooted in a common platform core, to ensure a high level of portability and interoperability; beyond the core, *extensions* are provided for each edition, so that developers can configure their platform according to their particular needs. The platform incarnations and their extensions are standardized by the Java Community Process (JCP).

On the J2ME platform, configurability becomes even more important than on desktop and server systems, because we simply cannot afford to include excess functionality. At the same time, functionality cannot be removed in an unscrupulous fashion that severely limits the applicability of the platform or precludes interoperability. One major concern that must be addressed is that the libraries defined for J2EE and J2SE require several megabytes of memory, and thus become unsuitable when memory is limited [Mic03]. In order to provide a high level of flexibility, J2ME defines a platform and its extensions along two axes: Configurations and profiles. Sun [Mic00] offers the following description of configurations, profiles and their relation:

[1]JavaCard [Micb], while based on Java technology, is still a bit different from the major editions in that it does not implement the Java specifications [LY99, GJSB00] to the extent that the major editions do, due to particularly severe resource constraints.

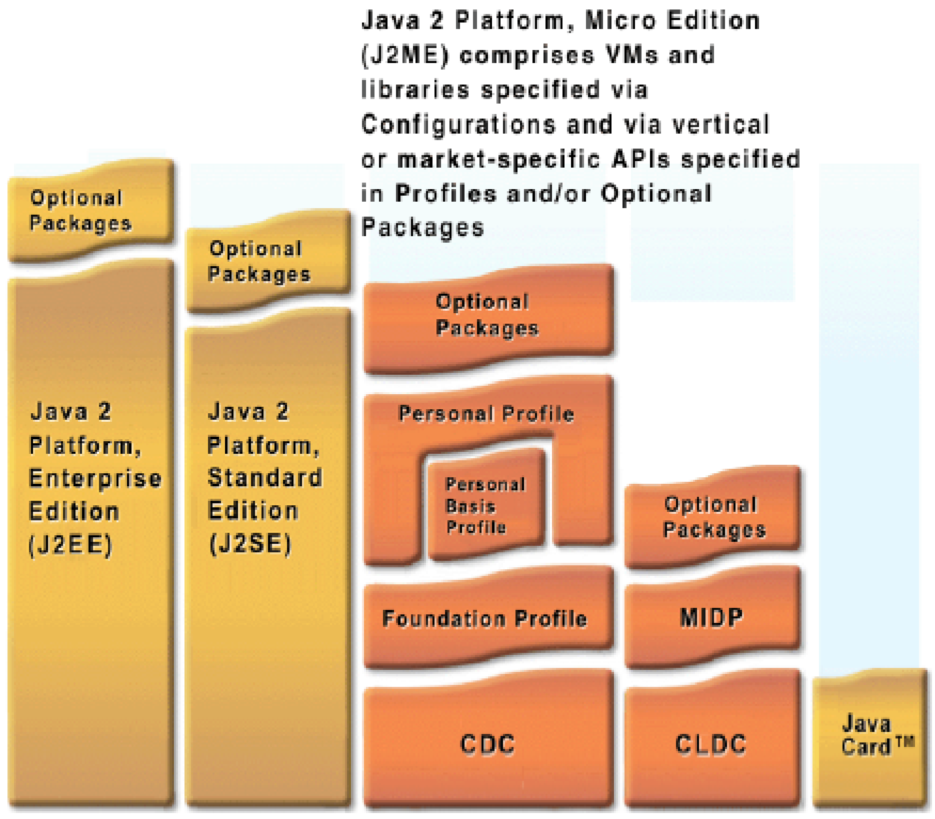

Java 2 Platform, Micro Edition (J2ME) comprises VMs and libraries specified via Configurations and via vertical or market-specific APIs specified in Profiles and/or Optional Packages
Optional Packages
Java 2 Platform, Enterprise Edition (J2EE)
Optional Packages
Java 2 Platform, Standard Edition (J2SE)
Optional Packages
Personal Profile
Personal Basis Profile
Foundation Profile
CDC
Optional Packages
MIDP
CLDC
Java Card™

Figure 2.1: The Java platform is composed of upwards compatible editions, each addressing the needs of a particular market, or class of computers. Each edition consists of standardized layers that enable flexible deployment. (From [Mic05])

> A J2ME configuration defines a minimum platform for a *horizontal* category or grouping of devices, each with similar requirements on total memory budget and processing power. [...] A J2ME profile is layered on top of (and thus extends) a configuration. A profile addresses the specific demands of a certain "vertical" market segment or device family.

Figure 2.2 on the next page illustrates the J2ME software layer stack. As indicated by the red bars, there is a tight connection between a configuration and the Java virtual machine (JVM); in particular, the JVM *implements* the behaviour that the configuration *specifies*, regarding Java programming language [GJSB00] and Java virtual machine [LY99] features supported, networking and security, and every other aspect covered by the configuration.

2.2.2 Connected, Limited Device Configuration

The Connected, Limited Device Configuration (CLDC) [Mic03] is a J2ME configuration that targets small connected devices, such as cellular phones and personal digital assistants. It is

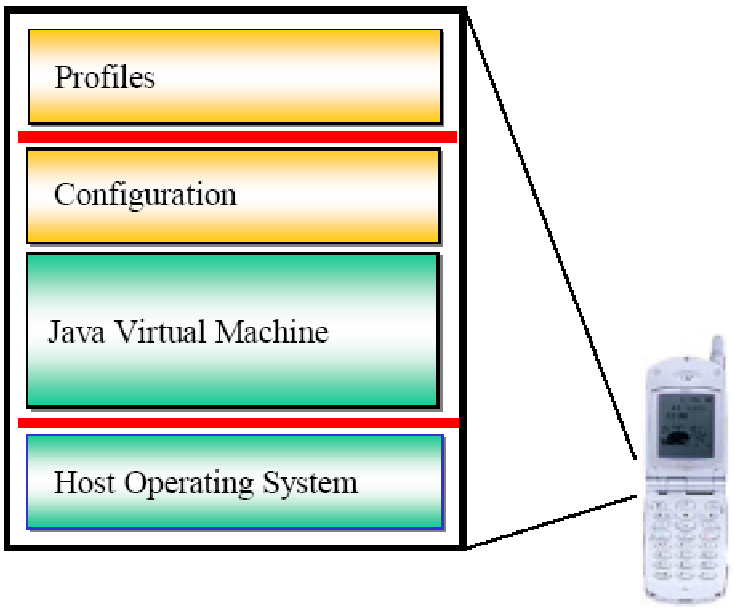


Figure 2.2: The J2ME software stack. (From [Mic00])

intended for devices that have only 192-512 KB of working memory (RAM), and a 16- or 32-bit processor. CLDC's main goal is to minimize the footprint due to the virtual machine and class libraries, while still maintaining a high level of portability. The scope of CLDC is restricted to Java language and virtual machine features, core Java libraries, input/output, networking, security and internationalization; graphical user interfaces, for example, are not covered. Since the scope of CLDC is limited, it is generally complemented by profiles; one such profile is the Mobile Information Device Profile (MIDP). MIDP addresses the limited screen size and battery power of mobile devices. An example of what a typical layered CLDC/MIDP architecture looks like is shown in figure 2.3 on the following page.

The heart of CLDC is the K Virtual Machine (KVM). KVM [Sun03] makes sure that the CLDC specification is adhered to. Sun offers a public distribution of the CLDC reference implementation, which contains KVM (with full source code), class libraries and a set of tools for platform development and customization. The footprint of KVM is made small by using more specialized security models than that found on J2EE and J2SE; additionally, a highly conventional bytecode interpreter is used. Due to the simplicity of the KVM interpreter, it does not give adequate performance for games and other highly interactive applicatons [Sha02]. However, the CLDC specification explicitly states that the addition of acceleration techniques, such as JIT compilation, are not precluded. The difficulty lies in implementing such techniques without exceeding the CLDC memory budget. In order to get a better understanding of this issue, a look at how Java applications are executed is necessary.

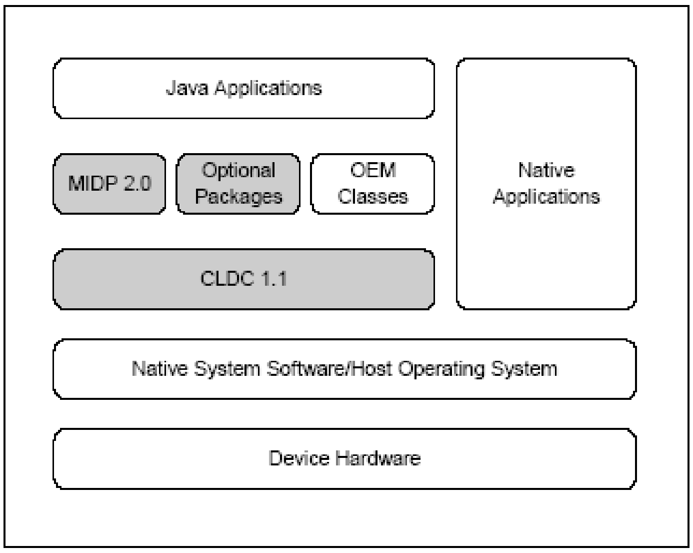


Figure 2.3: CLDC/MIDP architecture. (From [CD03])

2.2.3 The Java Runtime Environment

The purpose of the Java Runtime Environment (JRE) is to insulate code from differences of processor and operating system [Cora]. The cornerstone of the JRE is the Java Virtual Machine (JVM). Any platform that has a JVM implementation can run Java applications; it is the JVM that is the determining factor in Java's ability to "run anywhere, any time, on any device" [Mic03]. Figure 2.4 on the next page shows a high-level view of the JRE.

Applications and libraries are stored in platform-independent class files. A class file is a binary representation of a Java class, and contains symbolic descriptions of the class's own methods and its dependencies on other classes, as well as the bytecode (behavioural specification) of the methods that the class implements. Each class file is a self-contained entity that can be moved to any computing device. The JVM is responsible for verifying the integrity of the class files as they are loaded, ensuring secure execution. Interaction with the underlying operating system is done exclusively through abstract, native interfaces; in this way, platform-specific details are hidden from the application.

Two principal ways of executing bytecode exist. One option is to implement the bytecode instruction set directly in silicon; examples of this approach are Sun's PicoJava [OT97, MO98] processors. The other approach is to, at some time, translate bytecode into an equivalent stream of instructions that an existing (e.g. RISC or CISC) processor understands. The advantage of executing bytecode in silicon is that there is minimal overhead associated with execution; no software-aided translation is necessary. Furthermore, a Java processor can provide direct hardware support for other Java mechanisms, such as garbage collection and synchronization [RVJ$^+$01]. However, the computer industry has already invested decades of time and unfathomable amounts of money in building hardware, operating systems and other systems software

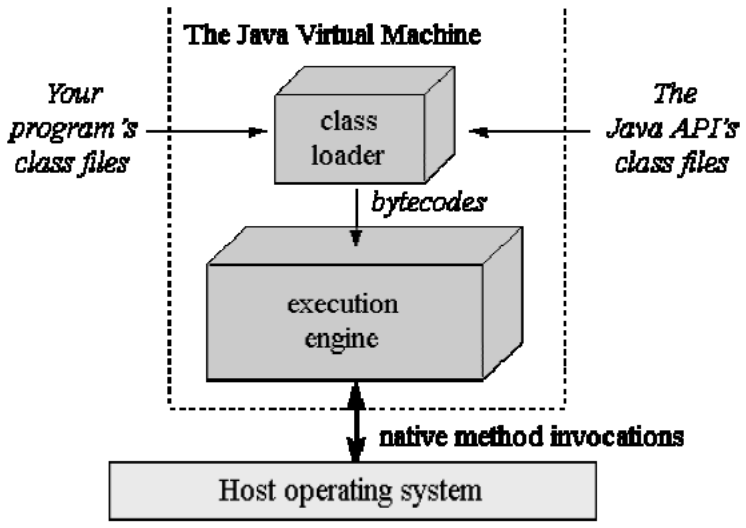


Figure 2.4: The Java virtual machine is effectively an abstraction layer that enables platform-independent bytecode to be executed on any hardware. (From [Raj02])

that cannot simply be moved to a purely Java-based platform. Thus, due to the cost efficiency of leveraging existing, mature technology, implementing the JVM as an abstraction layer above existing hardware is today the most popular alternative [Cora].

As a result of this choice, considerable research has been invested in developing technology that can provide efficient Java execution without requiring specialized hardware support. The semantics of the JVM are dictated by a detailed specification [LY99]. However, no particular implementation is mandated by this specification. While the platform-independent specification makes it challenging to offer high-performance implementations, it is also key to improving performance over time without changing the architecture that applications have come to rely on. The fact that the JVM specification has remained largely unchanged since the first edition in 1995 (the second, and latest, version was published in 1999, and was of the incremental, evolutionary kind) is a testament not only to its rigid design, but to the efforts of researchers that have devised new implementation techniques that, to the world outside the JVM, still result in behaviour that is consistent with the standard. Using the terminology of Valiant [Val90], the JVM acts as a *bridge* between the platform-independent and platform-dependent parts of the system. Valiant uses the von Neumann model of sequential computation as a prime example of a successful bridging model of computation, because “high-level languages can be efficiently compiled on to this model; yet it can be efficiently implemented in hardware” [Val90]. Von Neumann is certainly a tough act to follow, but Java can’t be accused of not trying.

In Java, many features which in older languages were the responsibilities of the programmer have become language primitives. This includes memory management (including automatic garbage collection), synchronization and exception detection. While this makes Java a safer language that eases the burden of developers, these primitives entail overhead that can affect the performance of the JVM. Figure 2.1 on the following page shows the average distribution

Application function	Execution time
Allocation and garbage collection	20%
Thread synchronization	19%
Running native methods	1%
Bytecode interpretation	60%

Table 2.1: Average division of labour in a virtual machine. (From [D.M98])

of execution time, as reported in [D.M98].

As can be seen, a significant portion of execution time typically goes into bytecode execution when bytecode is *interpreted*, as is the case in conventional virtual machines. This makes bytecode execution the area where efforts are most wisely invested in the pursuit of higher performance. The next section considers the principal techniques that aim to improve the performance of bytecode execution.

2.3 Accelerating Bytecode Execution

This section gives an introduction to techniques for accelerating the execution of bytecode, focusing in particular on Just-In-Time (JIT) compilation, the acceleration technique that our system will be based on. First, a motivation for accelerating the performance of bytecode execution is given, and the main approaches are discussed. This is followed by a more detailed presentation of the concepts and mechanisms that underlie JIT compilation in particular. Next, the evolution of JIT-related techniques is briefly recounted. Finally, the difficulties of adopting JIT compilation in resource-constrained environments are discussed.

2.3.1 Limitations of Interpretation

In the early days of Java, Java implementations earned a reputation of being slow [Cora]. This can largely be attributed to the fact that first-generation virtual machines achieved bytecode execution exclusively through interpretation. Bytecode interpretation is a software emulation of the virtual machine. The interpreter is based on a loop that fetches, decodes and executes bytecode instructions one at a time. Interpretation is simple to implement and has very low memory footprint; furthermore, the interpreter itself can be implemented in a portable language like C, making it possible to transfer the virtual machine to new platforms with minimal effort. However, due to its inherent execution overhead, interpretation is at least an order of magnitude slower than direct execution [Mic05, CO02]. If Java technology is to succeed on a diversity of platforms, it is important that the JVM provides efficient execution [RVJ+01]. There are techniques that can speed up interpretation, such as in-line threading [GH03], but in order to make a major leap in performance, the fundamental execution mechanism must be replaced.

Thus, in response to this situation, two major techniques emerged: Ahead-Of-Time (AOT) compilation and JIT compilation. Before we discuss JIT compilation, it is instructive to first consider the advantages and disadvantages of AOT compilation, a technique that lies at the opposite end of the spectrum relative to interpretation.

2.3.2 Ahead-Of-Time Compilation

In AOT (also called *static* or off-line) compilation, a Java application's class files are compiled to a platform-dependent binary executable *prior* to execution (hence the name). The major advantage of this approach is that the inherent overhead of interpretation is fully eliminated, allowing Java applications to approach speed comparable to that of applications written in traditional languages like C and C++ without much runtime overhead (of course, virtual machine services like garbage collection must still be included). Also, as the application's class files are combined into a single compact, chiefly pre-linked image, a lot of the inherent redundancy due to the self-contained nature of class files is eliminated [D.M98]. Furthermore, startup times are significantly reduced [Mic05], due to the elimination of class loading and the fact that machine code is readily available for execution. Many systems are faced with real-time requirements, in which fast response times is a must; AOT compilation can be used to address such concerns [SBCK03]. Lastly, static compilation techniques are highly mature, allowing AOT compilers to leverage vast amounts of existing compiler technology.

The primary disadvantage of AOT is that it goes against Java's mantra of platform independence. Unlike bytecode, the compiled application can no longer be transported seamlessly to other platforms over a network, neither in full nor in part. Furthermore, AOT increases the footprint of the application due to the compilation of code that might never actually be executed [PC97], and because even the most compact machine code is typically two or three times larger than the equivalent bytecode [RVJ+01]. This issue is of particular concern when memory is limited. AOT compilation does not facilitate Java's late binding model as readily as interpretation does, which can limit the system's dynamic class loading abilities [SBCK03]. Finally, the dynamic nature of Java inhibits an AOT compiler from applying aggressive optimizations that are effective when compiling older, more static languages [RVJ+01].

2.3.3 The Concept of Just-In-Time

JIT compilation can be viewed as a compromise between interpretation and AOT compilation. The goal of JIT compilation is to maintain the major advantages of interpretation, namely low footprint, platform independence of Java applications and full support of dynamic language features, but at the speed of AOT-compiled applications. The fundamental principle that underlies modern JIT compilation techniques is locality of reference [D.M98], which is based on the observation that an application typically spends most of its time in a small fraction of the code [AFG+04]. A JIT-enabled JVM aims to dynamically locate, compile and execute natively only the most frequently executed parts of the application – the so-called "hot" regions [SYK+01]

– while relying on the interpreter to still handle the less frequently executed parts. Ideally, this happens in a completely user-transparent way [Sch03]; bytecode is still the only input to the JVM, which takes care of encapsulating the JIT compilation, making sure that the user-visible state cannot be distinguished from that achieved through pure interpretation. A leading-edge JIT-enabled JVM is analogous to a modern microprocessor, such as the Intel Pentium 4; internally, the microprocessor performs register renaming, branch prediction, speculative execution and other optimizations, while externally, it presents a state that is consistent with a sequential execution model. Like a complex dynamic microprocessor, a JIT-enabled JVM must make good decisions in a miniscule fraction of the time that is available to AOT techniques, in order to justify the additional implementation complexity and actually achieve performance gains over simpler implementations.

Figure 2.5 gives a short summary of the differences between conventional (pure) interpretation, AOT compilation and JIT compilation, showing how JIT compilation strikes a middle ground.

	Pure Interpretation	Just-In-Time Compilation	Ahead-Of-Time Compilation
On-line	Interpretation	Interpretation Compilation Native Execution	Native Execution
Off-line			Compilation

Figure 2.5: A comparison of three bytecode execution techniques.

2.3.4 Approaches to Just-In-Time

Early JIT implementations did not contain a bytecode interpreter [AFG$^+$04]. Instead of relying on locality of reference, bytecode was compiled unconditionally the first time it was encountered, usually on a per-method basis. The advantage of this scheme over AOT is that only the parts of an application that are actually executed are compiled [PC97]. However, compilation is no longer an off-line process – instead it is moved to the critical path of execution. As a result, during application startup or phase shifts the latencies under this scheme can be significant. Execution of the Java application is effectively halted during compilation. During startup there may be large parts of the application that need to be compiled, thus causing the combined overhead due to compilation to become significant; one study referred to by Schilling [Sch03] found that 77% of compilation overhead occurs in the initial 10% of program execution. While startup latencies may be tolerable for long-running server applications, this is not the case for short-running and highly interactive applications, where user-perceivable pauses can detract from the experience.

A related problem when compiling every bytecode when it is first encountered is to decide on the level of compiler optimization to apply. Spending inordinate amounts of time analyzing and optimizing a method that will only be executed once and whose contribution to total execution time is potentially diminishing, is probably not the best choice, and only adds to the problem of startup latencies. Moreover, this illustrates a fundamental tradeoff in dynamic compilation: In general, greater code quality comes at the expense of longer compilation times [RVJ$^+$01, CO02]. The key to the success of JIT compilation is to amortize the cost of compilation over many subsequent executions of the compiled code.

Plezbert and Cytron [PC97] presented the *continuous compiler*, a scheme in which compilation is overlapped with interpretation and native execution in order to mitigate compilation latencies. The goal of the continuous compiler is to have the native machine code representation available immediately when it is needed. To achieve this, the system has two threads of control: One thread executing bytecode (either through interpretation or native execution), and the other thread compiling bytecode. If a method is not available in its native form when the bytecode execution thread reaches it, the compiler thread compiles the method while the bytecode execution thread interprets it. In this way, the latency due to compilation is tolerated because compilation is overlapped with other work, analogous to how a microprocessor attempts to tolerate memory access stalls by doing something else during the wait. The main drawback of the continuous compiler is that two physical processor cores are needed if the system is to operate at full potential speed. Once all bytecode has been compiled, the compiler thread will sit idle.

Plezbert and Cytron move on to consider the *smart JIT*, more generally labeled as selective, dynamic compilation [ea05], a solution that is a compromise between unconditional compilation and the continuous compiler. The goal is to reduce compilation latencies associated with unconditional compilation, but without requiring two dedicated threads of control. In this system, interpretation, compilation and native execution are intertwined in a single thread of control. Initially, the application is interpreted. Execution is monitored to dynamically determine which parts are likely to benefit most from compilation. Essentially, the execution profile of the past is combined with heuristics in order to estimate the application's likely behaviour in the future; based on these estimates, a well-informed decision can be made on which parts to compile.

Figure 2.6 on the following page illustrates the principal differences between the three techniques outlined above.

In recent years, even more sophisticated JIT techniques have emerged; Arnold et al [AFG$^+$04] provide an in-depth survey of these advancements. Newer techniques are not concerned with "just" finding the hot regions and compiling them; they aim to find such regions in a reliable way while incurring minimum overhead, and, furthermore, they focus intently on finding the right set of optimizations that should be applied during compilation of a particular application region.

Selective optimization [AFG$^+$04] is a technique that uses multiple levels of on-line application analysis and relies on several compilers, each with different tradeoffs regarding compilation cost and code quality. The basic idea is to first compile a hot method quickly, with only fair code quality. The method then continues to be monitored, even when executed natively. If further

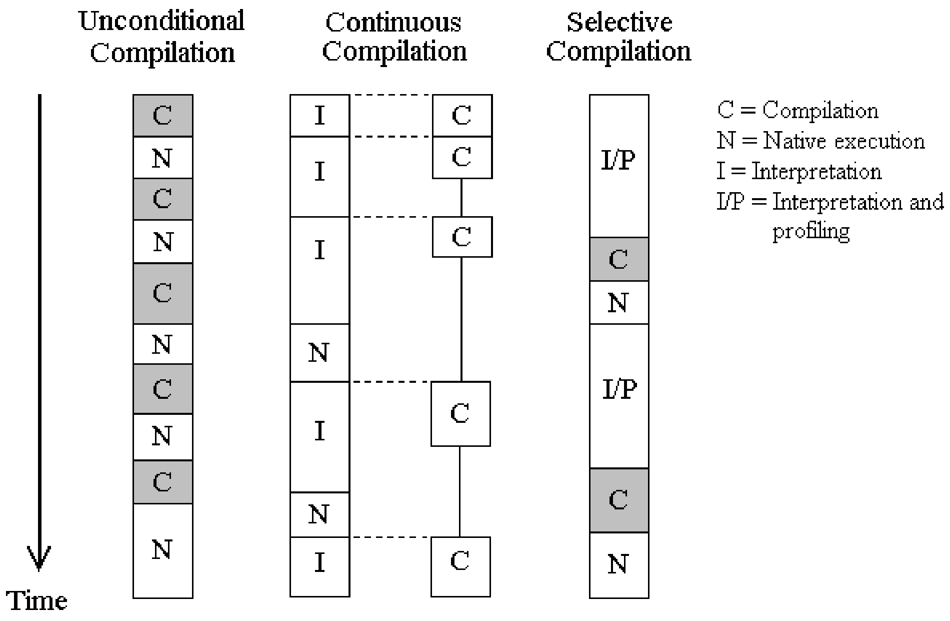


Figure 2.6: A comparison of three JIT compilation techniques. Compilation is shown in grey color whenever it leads to stalling of bytecode execution. During application startup, unconditional compilation will cause many execution pauses due to the need to always compile bytecode before it can be executed. Continuous compilation avoids the stalls completely by using two threads of control (each running on a dedicated processor), while selective compilation is a compromise that only compiles frequently executed code and interprets the rest.

monitoring reveals that the method can benefit from specific optimizations or higher code quality in general, the method is recompiled with a higher level of optimization. This process, called feedback-directed optimization, can continue in an incremental fashion. Such a scheme effectively brings together the advantages of cheap and expensive compilation: Any application can achieve fast startup times because interpretation coupled with fast code generation is employed early on, while at the same time long-running applications will benefit from the successive re-optimization of hot regions. Suganuma et al [SYK$^+$01] describe such a dynamic optimization framework where three optimization levels are used. Both this system and the system described by Arnold et al [AFG$^+$00] use several threads of control to profile and compile methods in an asynchronous fashion, as a way of minimizing latencies. In order to better appreciate the level of sophistication of such a system, the framework due to Suganuma et al is shown in figure 2.7 on the next page. However, we will not go into further detail on this framework since it is not intended for resource-constrained platforms.

Another noteworthy development is the merging of AOT and JIT compilation techniques [Mic05]. In such a scheme, system classes, which remain constant for a particular Java release, are typically compiled ahead-of-time, while the user application classes are loaded dynamically [Mic03]; this can significantly reduce startup latencies.

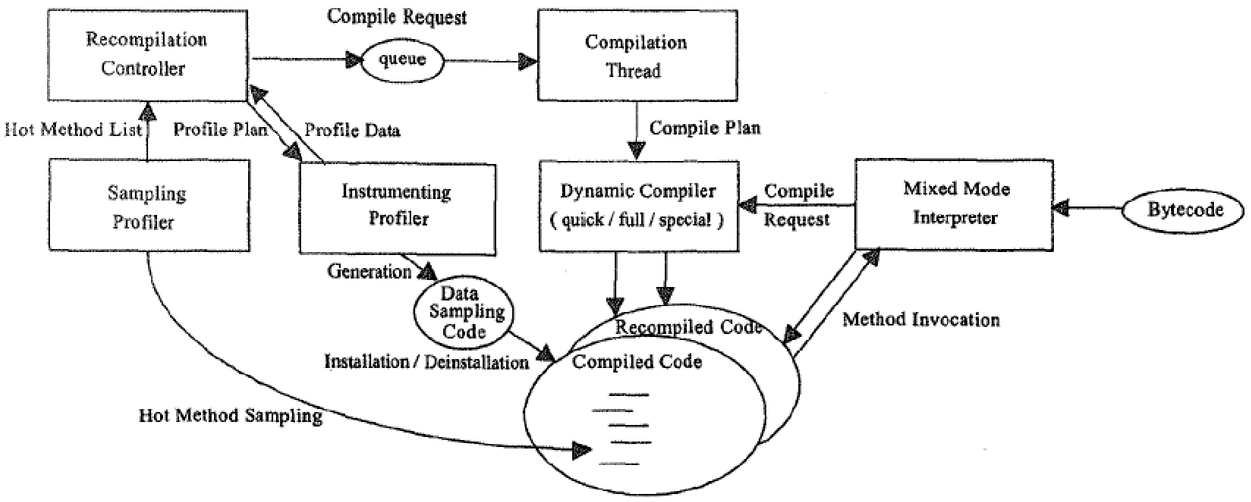


Figure 2.7: Architecture of an advanced dynamic optimization framework. (From [SYK+01])

2.3.5 Just-In-Time and Resource Constraints

Modern JIT compilation introduces substantial complexity into a virtual machine. The advancements of new, more sophisticated dynamic compilation techniques that throw additional memory onto the fire do not improve this situation. The added complexity brings with it a memory footprint that can be significant when memory is limited. In addition to the extra static memory needed to hold the implementation, the JIT compilation system requires dynamic memory for holding profiling data and bytecode translations. As mentioned earlier, the native representation of bytecode can be several times that of the original bytecode. Many effective optimizations, such as code specialization [SYK+01] and loop unrolling, trade space for speed. Furthermore, there is the possibility of redundancy due to two forms of represention, bytecode and machine code, being maintained. In tandem with the evolution of JIT compilation techniques, the complexity of other important components of the JVM has increased as well; this becomes a necessary pursuit when bytecode execution becomes so fast that the bottlenecks in the system shift, e.g. towards garbage collection. The total increase in footprint may force an implementation to use a conventional interpreted JVM [RVJ+01].

In order to bring JIT technology to constrained environments, the tradeoff between speed and memory footprint must be addressed at an early stage; this leads us to the realization that different techniques to those used in larger systems are needed to make JIT compilation feasible [Sha02]. Sun [Mic05] note that simply porting their existing JIT compilation system, originally designed with the high-end client and server markets in mind, would result in a memory footprint far too large.

Additionally, there are other considerations besides speed and footprint. Systems with limited energy, e.g. those that are battery-driven, should use their power conservatively in order to ensure long operation. JIT compilation is known to exhibit poor cache behaviour compared to interpretation [Raj02, RVJ$^+$01], which directly affects dynamic energy consumption due to more frequent accesses to main memory [CKV$^+$02]. However, faster execution can bring energy consumption down, quite simply because the JIT system finishes the job in a shorter time than an interpreter is capable of [Mic05].

Several studies and real-world projects have shown that there are good opportunities for implementing JIT compilation without blowing the memory budget. Radhakrishnan et al [RVJ$^+$01] found that the performance improvement due to advanced logic for selection of the application regions to compile is 15% at most, which means that it is possible to do without some of this complexity. Similarly, Schilling [Sch03] found that, when the overhead is taken into account, simple heuristics based on execution frequency and method length proved to be more effective than more complicated heuristics. Chen and Olukotun [CO02] challenged the generally held notion that small, fast compilers cannot generate code of good quality. Their contributions and others will be further discussed in the section on related work.

The theory and practice of JIT compilation is not something that is unique to Java. In fact, the techniques used by Java implementations today are based on earlier techniques for the languages that inspired Java, such as the SELF language [AFG$^+$04]. Somewhat ironically, these techniques were developed at a time when memory was still a concern even on desktop systems; this has led to a rekindled interest in the "forgotten" memory-conservative JIT compilation techniques of yesteryear, and how they can be mapped onto Java at little extra cost.

2.4 Levels of Granularity

The issue of the *level of granularity* at which profiling, compilation and mixed-mode execution occurs in a JIT-enhanced virtual machine is a fundamental decision. Much of such a system – including the one described in this text – follows logically from, is constrained by, and can be explained in terms of, the levels of granularity; therefore, this topic deserves a thorough discussion. The author is not aware of any similar comprehensive, unified treatment of the subject, which he found a bit odd considering the wealth of interesting aspects and theories that emerge.

Informally stated, the level of granularity says something about the scope and detail of processing, and, usually, how often such processing occurs. A coarse granularity corresponds to a large scope. Some of the possible levels of granularity for each of the three aforementioned mechanisms – profiling, compilation and mixed-mode execution – are shown in figure 2.8 on the facing page.

Naturally, there are tradeoffs involved with each possible choice, and these concern both space and time requirements and implementation complexity of the system. In the sequel, the inherent granularity-related tradeoffs for each mechanism are highlighted.

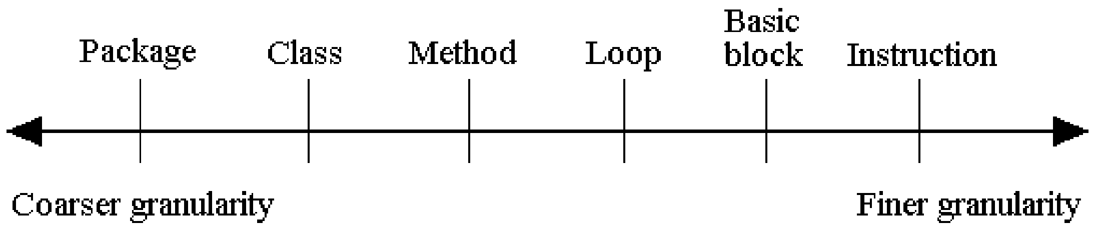


Figure 2.8: Some possible levels of granularity at which JIT-related mechanisms may operate.

2.4.1 Profiling

Recall that the objective of profiling is to discover the "hot" parts of an application, where compilation followed by anticipated subsequent (re)execution of the resulting native code is likely to yield the biggest performance gain over straightforward interpretation. Too coarse-grained profiling means there is an increased likelyhood of compiling program parts that aren't performance critical, since the decision-making logic for compilation has less precise information on which to base its decisions on. Too fine-grained profiling, on the other hand, may lead to excessive space and time overhead related to the collection, representation and processing of profiling data.

As an example for sampling-based profiling, a coarse-grained profiler might only record the *classes* participating in the execution at the time of sampling; a medium-grained profiler would concentrate on the *methods* participating; while a fine-grained profiler would consider local parts of a method, such as the *basic blocks*[2] that make up a loop.

2.4.2 Compilation

Too coarse-grained compilation means that a large amount of memory will be needed to hold the native code produced by the compiler, especially since bytecode-to-native code expansion is typically around 2-8 [RVJ$^+$01, ea05]. If the compiler generates intermediate representations prior to actual code generation, more temporary memory is needed for storing those; for instance, Chen and Olukotun [CO02] note that the bulk of dynamic memory used by their JIT compiler is for the intermediate representation. This increases the compiler's peak memory demands, which adversely affects the *predictability* of its memory usage. Finally, compiling large amounts of code "atomically" leads to longer compilation times, since the time spent in compilation is linear to the number of bytecodes compiled [PC97]. In an interactive environment, this may result in slow startup times and unpredictable, user-perceivable pauses [Sha02, Sch03].

Too fine-grained compilation may inhibit the possibility of many popular and effective optimizations, such as copy propagation, code motion and intra-method register allocation, since

[2]A *basic block* is a sequence of consecutive instructions "in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end" [ASU86].

only a small window of instructions are considered. This means that, while the resulting native code can be generated quickly and will be *relatively* small (since the original bytecode sequence is small), it might not be of very high quality. However, many optimizations that are standard in desktop- and server-based JIT compilers are arguably too expensive to be performed with limited memory and processing power at any rate [CO02, ea05]. The downside is that poor quality code contains redundancy and utilizes the underlying hardware resources poorly. Thus, the possible performance gains over normal interpretation might be severely limited. Furthermore, the redundancy makes the code bigger than necessary. This increases the likelyhood that the *set of translations* corresponding to the entire working set of bytecodes does not fit in software-managed translation caches nor processor caches. In addition to performance degradation, the consequent increase in memory traffic is particularly severe for a battery-driven device, since it drains its power.

2.4.3 Mixed-Mode Execution

Once a unit of bytecodes has been compiled, the resulting native code is ready to be run directly on the target processor. However, this typically necessitates a preceding *context switch* from the interpreter mode to native mode, in order to transfer the interpreter's state to the format and locations (registers and stack) expected by the native code [ea05, Sha02]. Similarly, following execution of the native code, the state of the interpreter must be transformed so the final result is identical to the state normal interpretation of the bytecodes would produce. The interleaving of interpretation and native execution is called *mixed-mode* execution.

Too fine-grained units of bytecode – that is, only very short sequences – means that the context switches back and forth between interpretation and native execution will be very frequent. This might compromise the potential speedup. Indeed, one of the reasons why native execution affords a performance increase over interpretation is that the combined translation of *several* successive bytecodes removes the inherent overheads of the fetch-decode-execute loop of the interpreter. Fine-grained mixed-mode execution demands that the context switches are fast and simple.

As an example of what an impact fine-grained mixed-mode execution can have on performance and, consequently, the system implementation, Shaylor [Sha02] had to expand his compiler to also handle some of the more complex opcodes (initially handled exclusively by the interpreter), in order to bring the number of context switches down to an acceptable level.

2.4.4 Combining Levels of Granularity

While choosing the same level of granularity for all three mechanisms may seem the most intuitive approach, there are other reasonable alternatives. For instance, Plezbert and Cytron [PC97] describe a system where profiling is done at the method level, while compilation occurs at the class level, based on the sum of accumulated execution times of all methods in a class;

this allows for intra-method compiler optimizations. In Shaylor's [Sha02] system, compilation occurs at the method level, while mixed-mode execution is done at the basic block level.

Some more recent, sophisticated systems even employ multi-level profiling schemes; Arnold et al [AFG$^+$00] describe a profiling system where a coarse-grained profiling is used to determine candidates for further fine-grained analysis.

2.5 Related Work

In this section, some existing systems that target JIT compilation in little memory are considered. We will look especially at the footprint, the levels of granularity, profiling mechanisms, and compilation strategies employed by these systems, as well as the interaction between interpretation and native execution. The purpose of the discussion is to get an overview of techniques that may form a solution space when our own architecture is crafted.

First, the term "footprint" deserves a definition. There are two types of footprint that are important to memory-constrained systems: *Static* footprint and *dynamic* footprint. Corsaro et al [CSKO02] defines them as follows:

> Static footprint [...] is the amount of storage needed to hold an image of the given application. [It] is a time-invariant measure.
>
> Dynamic footprint [...] represents the amount of memory used by a running instance of the application. [It] is time-dependent.

Dynamic footprint is the sum of the static footprint and additional runtime data. As Corsaro et al point out, it is essential for memory-constrained systems to have a hard upper bound on dynamic footprint.

2.5.1 The Small Hybrid JIT

Manjunath and Krishnan [MK00] describe a small hybrid JIT for embedded systems that combines interpretation with compilation. Based on the rule of thumb that most programs spend 80% of their time in 20% of the code, the authors' hypothesis is that it is not necessary to compile entire methods to gain execution performance; that is, they bring the principle of locality of reference down from the application level to the individual method level. The authors argue that focus should be directed towards those parts of a method that are most frequently executed, such as loops. Thus, in this system, the basic block, rather than the method, is the unit of profiling and compilation. This is believed to reduce both the memory required to store native code and the time for compilation, since only a subset of the full method is considered. A prototype of the JIT showed performance improvements of 5-10% over regular interpretation. The static footprint due to the compiler is reported to be only 6.7 KB, while nothing is said about the dynamic footprint.

The hybrid JIT system starts by interpreting all methods. Once a method has been invoked a predefined number of times, a bytecode analyzer is run that identifies the basic blocks of the method and estimates which ones are good candidates for compilation, based on a heuristic involving size, opcodes used and usage count. The next time the basic block is encountered by the interpreter, it is compiled and henceforth executed natively.

While no direct reference is made to it, the approach to code generation is *selective inlining*, first introduced in [PR98]. Instead of compiling bytecodes in the traditional sense, the "compiler" simply concatenates the native code that the host compiler generates for each of the `case` statements in the interpreter's main `switch` statement. This eliminates the overhead of both the switch and the jump back to the top of the interpreter loop. While not implemented, the authors suggest that a peephole optimizer may be a simple and effective way of improving the quality of the resulting code, since the stack-based nature of JVM execution leads to many redundant loads and stores when bytecode is naïvely translated to a register-based form.

Overall, the hybrid JIT seems very simple; perhaps *too* simple. Performance improvement of 5-10% over regular interpretation is a far cry from the order of magnitude improvement we have come to expect from JIT compilers. Even though the results are very preliminary, the system does not appear to be very scalable; it is unclear how further *significant* performance improvements can be achieved. Since the native code is copied from the host compiler's implementation of the interpreter, rather than generated "manually", one is completely at the mercy of the host compiler as far as code quality goes. Furthermore, as the authors state, the code is not even guaranteed to be directly executable, probably due to the idiosynchrasies of the particular host compiler used and optimization levels specified. Even if peephole optimization is applied, the resulting code will still be stack-based, which is typically of much lower quality than what a code generator with a simple concurrent register allocation scheme, such as the one in [ASU86], is able to produce. While the small static footprint of the hybrid JIT is impressive, the apparent inability to refine and improve the system further limits its relevance. In short, the hybrid JIT actually takes the concept of low-footprint JIT compilation *too far*: The complexity is quite simply too low to enable significant performance improvement.

The reason why the hybrid JIT is still interesting is because it is the only system encountered that performs profiling, compilation and mixed-mode execution exclusively at the basic block level. The arguments for choosing such a path in a memory-constrained JVM are convincing, and deserving of further investigations. Thus, the premise is appealing, but the particular approach taken seems to be a limiting factor for success.

2.5.2 KJIT

Shaylor [Sha02], employed at Sun Microsystems and a contributor to recent versions of KVM, describes a JIT compiler called KJIT, designed with memory constrained low-power devices in mind. Its foundation is KVM. The compiler is purposely "incomplete"; that is, it can only compile a subset of the JVM instruction set. The full instruction set is supported by leveraging the functionality of the existing, complete KVM interpreter. This way, efforts could be directed towards compiling those JVM opcodes which are the most performance critical. As a result, the compiler is relatively small and simple, and its interaction with KVM as a whole is less complex than a complete JIT would be. Speedups in the range 5.7 to 10.7 were reported, with a static footprint of 64 KB and additional dynamic footprint of 128 KB. The target processor is ARM.

KJIT compiles bytecodes at the method granularity, but can be said to perform mixed-mode execution at the individual bytecode granularity. The fact that not all JVM opcodes are compiled implies that control must be able to pass between interpretation and native execution at any time, not just at method entry or exit. Since the switch between native mode and interpretation mode can occur quite frequently in this scheme, an efficient switch mechanism was reported to be crucial for achieving good speedup. To this end, KJIT preprocesses a method's bytecodes in a way that effectively removes the evaluation stack, ensuring that no operands are on the stack at the end of a basic block. This transformation makes the originally stack-based JVM state more readily transferable to the register-based state of the target processor and vice versa. However, the resulting bytecode is reported to be 30% bigger than the original [ea05].

No heuristics are used for deciding when to compile a method; methods are presumably compiled as they are first encountered. Debbabi et al [ea05] argue that this scheme seems very heavyweight and may only be suitable for desktop or server systems. The motivation behind KJIT's choice of no heuristics is the assumption that J2ME applications are very small, comprising only tens of classes, thus heuristics for detecting performance critical parts is presumed to be less important than in large, long-running server applications comprising hundreds of classes. Since KJIT actually achieves speedups up to 10.7 over interpretation, Shaylor's hypothesis may indeed hold water. By not relying on profiling to guide compilation, KJIT avoids the space and time overhead due to the collection and processing of profiling data; if it is the case that profiling is less important for small applications, removing the profiling actually improves performance. However, it is still a fact that compiling every method results in more native code being generated, increasing the amount of memory needed to store the code. A quantitative investigation of the effects of incorporating a selective compilation strategy would be an interesting future development.

The code management strategy employed by KJIT is to allocate a single static code buffer, and clear it once it becomes full. This simple scheme is justified by the claim that (re)compilation can be done quickly, should the method whose code has been discarded ever be executed again. It also makes it easy to discard aggressive optimizations; if an optimization needs to be invalidated at some point, the code buffer is simply cleared. This strategy seems like a good choice when low complexity is desired: Implement the simplest possible solution, and if it turns out to not have a negative impact on performance, pursuing more advanced solutions is not necessary.

Experimental results for KJIT are presented that show the speedups using code buffers that range in size from 128KB down to 8KB. The results indicate that the amount of performance degradation as buffer size is reduced is heavily dependent on the characteristics (such as working set size) of the individual Java application. This is a problem, as it yields unpredictable performance and may require per-application manual tuning in order to achieve good speedup.

Allocating a static buffer so large that it is assumed to be "big enough" for all current and future applications would be too wasteful in the general case. On the other hand, too conservative buffer allocation would lead to heavy thrashing, as the buffer is frequently and repeatedly cleared, filled and cleared again. This has a severe impact on the memory traffic of the system, which is something to be conscious of in a low-power environment.

2.5.3 E-Bunny

Debbabi et al [ea05] describe E-Bunny, a selective dynamic compiler that has been integrated with KVM. E-Bunny was developed by a group of people at the Concordia University in Canada. Its intended use is in embedded systems with memory limitations similar to those targeted by the original KVM. The extra footprint due to the E-Bunny integration does not exceed 150 KB, roughly evenly divided between static and dynamic memory needs. Experimental results have shown an overall speedup of 4 over KVM version 1.0.4.

The E-Bunny system implements a highly conventional mixed-mode execution. A method is fully interpreted until it has been invoked a predefined number of times, at which time the entire method is compiled to native code and installed in a translation cache. Upon subsequent invocations the method is executed natively rather than interpreted. Control flows between interpretation and native method execution as appropriate. If the translation cache becomes full, an LRU replacement policy is used to decide which method's native representation to discard.

Concerned with the overheads of sophisticated compilation techniques in an environment scarce on resources, E-Bunny opts for a simple, lightweight compilation strategy. The E-Bunny compiler itself is a single-pass compiler targeting the Intel IA-32 architecture. It generates stack-based native code that very much resembles the original bytecode sequence. This simple scheme precludes all but the simplest optimizations.

E-Bunny distinguishes between simple JVM opcodes – those which can directly and compactly be translated to equivalent native code – and complex opcodes, which have a higher level of abstraction and whose execution is therefore more involved. Complex opcodes are handled by generating native code that calls a distinguished helper function (one for each opcode) written in C. In this way, the entire JVM instruction set is supported without involving the interpreter. Actually, in E-Bunny this is absolutely necessary, since both compilation and native execution occurs exclusively at the method level of granularity.

E-Bunny extends the KVM thread context to include a native stack, on which translated methods store their activation records, in a fashion similar to how the interpreter manipulates its stack. During the switch between interpretation and native execution, value transfer from one stack to the other is necessary. Also, since garbage collection can happen during native execution, the KVM garbage collector was extended to scan the native stack for object references as well.

The current version of E-Bunny compiles every method that is invoked by the top-level ("hot") method. This choice is attributed to the complexity of implementing the switch from native execution back to interpretation. As stated by the authors, this can lead to translation of methods which are not performance critical. What is not pointed out, however, is that Java libraries tend to have strong internal dependencies [Mic03, TBS99]. While an effort has been made to reduce dependencies in the J2ME core libraries, some level of dependency will still result from the criteria of maintaining upward compatibility with J2SE and J2EE. Also, Java's object-oriented nature dictates that writing small, high-level methods that invoke lower-level methods to do the real work is good practice. In the context of E-Bunny's compilation scheme, this means that immediately compiling, recursively, every method that one method invokes can lead to a potentially huge number of methods being compiled, which in turn can lead to long, unpredictable compilation times. Furthermore, increased compilation leads to increased memory traffic (to install the generated native code), which can have a severe impact on cache utilization [Raj02, RVJ+01] and energy consumption [VKK+01]. Lastly, Schilling's measurements [Sch03] indicate that no "locality of reference" exists between a method and the methods it invokes; no tests ran faster using this heuristic. Thus, in the worst case, this scheme can defeat the original intents of the E-Bunny system.

Since E-Bunny compiles entire methods, it will also compile any exception handling code that is part of the method. If, as in [LYK+00], we assume that exception handlers are rarely executed compared to code on the normal flow path, translation of exception handlers can be wasteful. Not only does it add to compilation time, it also increases the size of the native code and hurts code locality.

No details are given regarding the choice of method counter treshold that is used to determine when a method should be compiled. The process of determining such a treshold is labeled in [AFG+04] as being ad hoc and demanding extensive tuning. We may only assume that the treshold has been selected so as to give the best results for the particular benchmark suite (CaffeineMark) used to measure speedups due to E-Bunny.

Similarly, a justification for choosing a translation cache size of exactly 64KB, no more and no less, would also have been of interest. Nothing is said about the size of native code compared to the original bytecodes.

2.5.4 microJIT

Chen and Olukotun [CO02] describe microJIT, a fast, portable and small JIT compiler that targets embedded environments. microJIT challenges the notion that a fast compiler will produce code of inferior quality compared to the code produced by a slow, more complex compiler. microJIT achieves fast compilation times by reducing the number of major compilation passes; a standard set of optimizations, both platform-independent and target processor-specific, are applied as the intermediate representation is generated.

microJIT operates at the method level of granularity. The authors recognize that the intermediate representations generated during compilation accounts for the bulk of dynamic memory needs. To reduce the footprint, the authors suggest a partial compilation scheme, where only the intermediate representation for one subsection of the method is kept in memory at a time. The tradeoff is that this scheme precludes some global optimizations.

microJIT is built on KaffeVM. The static footprint of the compiler is 200 KB, and the dynamic memory recommended for storing native code is 250 KB. Initially targeting the MIPS IV ISA, the compiler has later been ported to SPARC and StrongARM. RISC architectures have been favored over CISC, such as Intel IA-32, so as not to eliminate most possibilities for register allocation. Compared to the original bytecodes, the size of code generated by microJIT is bigger by a factor of 3-4 for the benchmarks considered. microJIT's fast compilation times resulted in short running applications starting up quickly, and the performance also proved comparable to that of more heavyweight compilers for many longer running applications.

Selective compilation and the effects of a limited code buffer were outside the scope of the microJIT study.

While microJIT is lightweight compared to most desktop and server solutions, it seems too large to fit into the tight memory budget of a CLDC-compliant device. As the authors point out, CDC is the primary intended platform, which has more memory than CLDC by an order of magnitude.

2.5.5 Summary

The systems that have been studied have focused on keeping the footprint low and compilation times fast by implementing simple, counter-based profiling schemes – if any profiling at all – and lightweight compilation strategies that do not include very sophisticated optimizations. The dominant level of granularity is the method, although KJIT [Sha02] implemented mixed-mode execution at a lower level of granularity that allowed a much simpler interaction with the interpreter. In comparison, the cost in complexity due to E-Bunny's [ea05] decision to perform mixed-mode execution at the method level seemed to be unwarranted. The same can be said regarding KJIT's choice of a static code buffer that is cleared when full versus E-Bunny's LRU scheme. The major indication might be the following: When memory is scarce, don't implement a sophisticated solution until a simpler one has first been attempted – it might turn out to be perfectly adequate. This way, the extra kilobytes saved can go into addressing the *real* bottlenecks, which undoubtedly will emerge once empirical data is available.

Chapter 3

Architecture

This chapter gives a high-level presentation of an architecture that aims to accelerate Java bytecode execution. The architecture encompasses a selective, on-line compiler and its integration and interaction with an existing virtual machine, namely Sun's K Virtual Machine (KVM). KVM is able to provide a complete Java runtime environment within a total minimum memory budget of 200 KB of RAM. The goal of this project is to create a lightweight extension to KVM that accelerates the virtual machine's performance significantly, without introducing excessive memory overheads. Specifically, the challenge faced by the architecture is to facilitate implementations of the acceleration mechanism that does not invalidate the virtual machine's applicability in systems facing constraints similar to those addressed by the original KVM. The acceleration will allow those systems to improve their response times for existing applications, as well as support new applications with higher performance demands.

There are several ways to improve the performance of a virtual machine, even when memory is limited. One way is to improve the performance of individual components, such as the garbage collector [SMB04], or the implementation of dynamic Java features such as virtual method invocation. Another approach is to improve the performance of the bytecode interpreter, by introducing techniques such as inline threading [GH03], or by implementing the interpreter in assembly language. A third approach, which is the one taken here, is to speed up basic bytecode execution by introducing execution techniques that augment interpretation. Since interpretation can be an order of magnitude slower than native execution, addressing KVM's conventional bytecode interpreter arguably has the greatest potential for achieving speedup, because executing bytecode is what an interpreter-based virtual machine does 60-80% of the time, on average [D.M98, VMK02].

The acceleration of KVM is to be achieved by augmenting KVM's bytecode execution engine with JIT compilation. Specifically, KVM's capabilities are extended from pure interpretation to a scheme where interpretation, selective compilation and native execution are interleaved in time. The resulting architecture is completely independent of target processor; thus, processor-specific issues, such as the exact details of context switching and compilation, are not covered by the architecture. These issues are instead addressed in the next chapter.

The rest of this chapter is laid out at follows. Section 3.1 introduces the architecture's top-level components and interactions. Section 3.2 decides on the fundamental issue of the levels of granularity at which on-line profiling, compilation and mixed-execution should occur. Section 3.3 introduces a new bytecode execution engine that facilitates mixed-mode execution. Following a high-level presentation, the on-line profiling mechanism and selective compilation logic that the engine relies on are decomposed. Section 3.4 discusses the general approach to bytecode compilation, and how to manage the code that the compiler produces, given that little dynamic memory is available for holding the code. Section 3.5 concludes the chapter with a discussion of the importance of introspective profiling capabilities, a mechanism that allows a researcher or developer to inspect the behaviour of the system and use the information to assess the effectiveness of an implementation and the architecture it is based on.

3.1 Top-Level Components and Interactions

Figure 3.1 on the next page shows the architecture's top-level components and interactions. The components are:

Instrumented Interpreter An extended version of the KVM interpreter that facilitates mixed-mode execution based on on-line profiling and selective compilation.

Target Processor The processor that directly executes the machine code corresponding to compiled bytecode[1].

Compiler Compiles bytecode to machine code upon the interpreter's request.

Translation Cache Manages the (limited) memory used for holding compiled code. When this memory becomes full, the translation cache must make room for a new translation by discarding one or more existing translations.

The system runs in a single thread of control, as in Plezbert and Cytron's Smart JIT [PC97] discussed in section 2.3; using several threads, as in more recent systems, would go against KVM's green threads implementation, introduce nondeterminism and complicate implementations derived from the architecture significantly. Thus, on-line profiling of the application being executed is integrated into the interpreter, guiding the selection of application parts to be compiled. The compiler translates the selected bytecode to target processor-specific machine code, which is installed in the software-managed translation cache for subsequent retrieval and native execution. The responsibility of bytecode execution now becomes divided between the interpreter and machine code, with the interpreter continuing to interpret the rarely executed bytecode while the frequently executed bytecode is executed directly in a native form, eliminating the overhead of interpretation.

[1]The interpreter also runs on the target processor, but this is a detail that is not important in a conceptual sense.

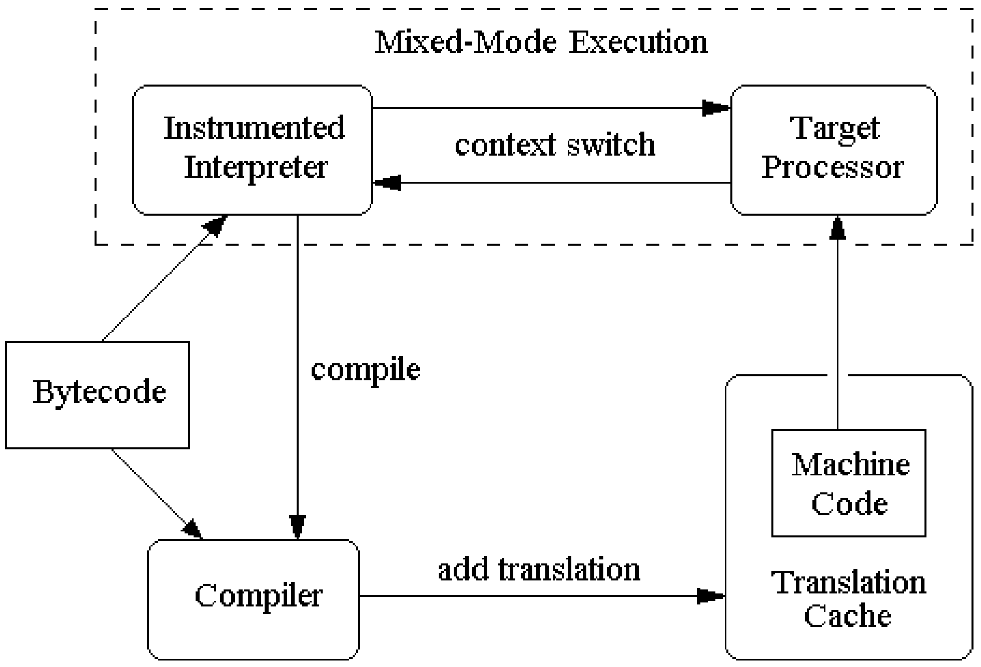


Figure 3.1: The system architecture.

3.2 Chosen Levels of Granularity

Figure 3.1 does not say anything about the levels of granularity at which profiling, compilation and mixed-mode execution happens. However, this fundamental issue must be resolved next, since the remainder of the architecture depends on it. The major implications associated with the choice of level of granularity at each end of the spectrum were discussed in section 2.4. The primary level of granularity selected for this system is the *basic block*. As the discussion below will indicate, the basic block appears to be a level that holds a lot of promise when low memory footprint is a central concern. Furthermore, the typical granularity level in the most closely related systems is the method [ea05, Sha02]. By choosing a different level, a set of new and interesting possibilities and challenges emerge. The following subsections highlight these for each mechanism.

3.2.1 Profiling

Basic blocks are more numerous than methods, putting more stress on the profiling mechanism. In particular, for a counter-based profiling scheme, profiling overhead is now associated with the entry of a basic block, rather than the entry of a method. However, the finer granularity means that high-frequency execution of intra-method sections can be detected. For instance, imagine a typical method that spends most of its time in a loop; the method consists of a one-time initialization that sets up the loop, followed by the loop body that enjoys repeated execution, possibly followed by some epilogue code. In this case, updating the profiling counter only at the entry of the method would not give an accurate picture of how much time is actually spent in the performance-critical part of the method – the loop. Since a loop is composed of basic blocks, moving profiling to the basic block level catches such situations by design.

3.2.2 Compilation

The ability to compile individual basic blocks, rather than an entire method, means that only the performance-critical parts of a method need to be compiled. Continuing with the loop example in the preceding section, only the basic blocks that constitute the "hot" path of the loop body would be selected for compilation; the prologue and epilogue, on the other hand, are not found to be executed frequently and thus continue to be interpreted. On a related note, compiling at the basic block level supports on-demand compilation of exception handlers [LYK^{+}00] by design.

Compared to a method-based compiler, then, a basic block-based compiler has less input bytecode to work with, requiring less memory for temporary data structures maintained during compilation, and less time to perform the compilation. This also means that if a basic block's native representation should be replaced in the translation cache (due to limited cache memory), it can quickly be regenerated at a later time if necessary.

By requiring that a basic block has been interpreted at least once before it is compiled, a lot of complexity is avoided in compilation – we like to call this principle "Interpret once, compile anytime". In particular, when interpretation of a basic block is known to have been performed, the compiler can be sure that any classes referenced by the basic block's bytecode have been initialized. The JVM specification [LY99] dictates that such initialization should not occur before the class is referenced the first time, and it is very important that an implementation adheres to these semantics because initializing a class may involve the implicit invocation of a static class initializer method with arbitrary side effects. By letting the interpreter handle this complexity, there is no need to generate native code that does the checking and non-trivial initialization process. The result is more compact code that doesn't do any redundant work in this regard. A method-based compiler might be forced to compile such checks, since not all basic blocks in the method are guaranteed to have been interpreted, and the bytecode in those blocks may reference one or more classes that have not yet been initialized at the time of compilation.

The principal drawback of compiling at the basic block level is that it inhibits the possibility of inter-block optimizations; thus, the potential code quality that can be achieved is limited compared to that of a method-based compiler. As a consequence, the compilation subsystem is not very scalable; even if more memory became available to the compiler, it wouldn't be able to put it to use, since it doesn't have the control- and data-flow information necessary to perform sophisticated optimizations. However, this is a very conscious choice in our system; memory concerns have higher priority than scalability. As established target systems increase their memory budget over time, they can change to a more general-purpose virtual machine, while this system will hopefully still find its place in new generations of constrained devices.

3.2.3 Mixed-Mode Execution

At the basic block level, the context switch between interpretation and native execution will be more frequent than mixed-mode execution at the method level. In order to avoid that context switching eats up the potential acceleration, it must be possible to perform the context switches very quickly. The architecture can facilitate efficient implementations of the context switch by having the interpreter and native code share as much state as possible.

3.3 Bytecode Execution Engine

The new execution scheme must be integrated into the heart of KVM, which is its bytecode interpretation loop; it is precisely here that on-line profiling, selective compilation and mixed-mode execution will happen.

Figure 3.2 shows the original bytecode execution loop of KVM, configured to only perform thread rescheduling on basic block boundaries. KVM implements a *green threading* scheme, which means that multithreading is achieved strictly through user-level software (no operating system dependencies) and is completely synchronous; between basic blocks, the system checks to see if another thread should be swapped in.

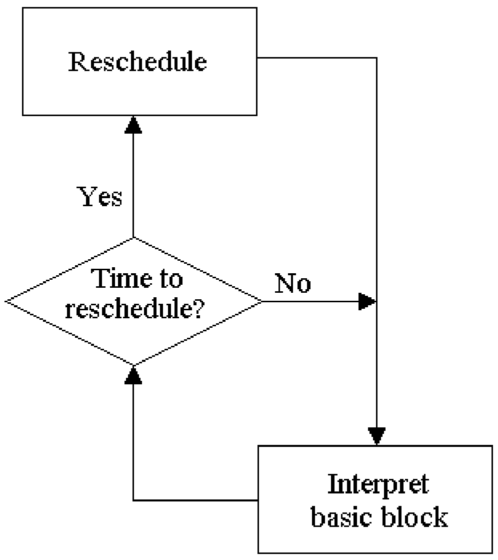


Figure 3.2: The main loop of the original KVM interpreter.

Figure 3.3 shows a modified version of KVM's bytecode execution loop that facilitates the new bytecode execution scheme.

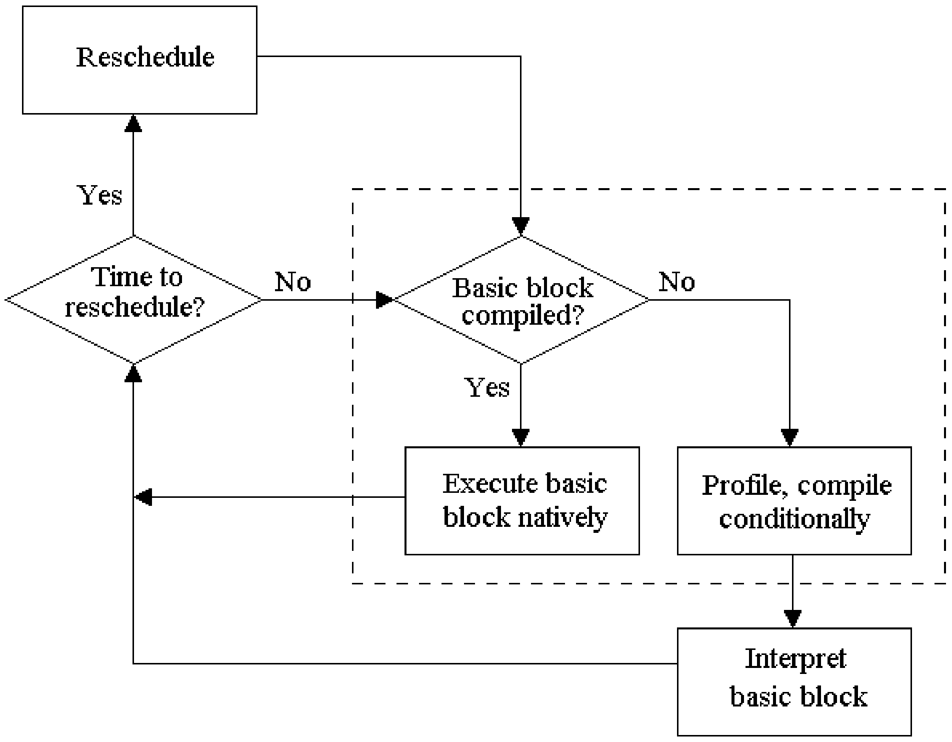


Figure 3.3: The main loop of the JIT-enabled KVM. The new dotted box "intercepts" the original arrow that went straight to "Interpret basic block" in the original version (see figure 3.2 on the preceding page).

Basically, decision logic has been inserted to select between native execution and interpretation on a per-basic block basis. If a basic block is already compiled, it is immediately executed natively; if it is not compiled, profiling is performed, possibly triggering compilation, before the basic block is interpreted in KVM's original fashion. Note that thread rescheduling still happens between execution of basic blocks, remaining completely unaffected by the new execution scheme.

The following subsections decompose the contents of the dotted box in figure 3.3, describing its internal behaviour in more detail.

3.3.1 On-Line Profiling and Selective Compilation

The purpose of on-line profiling is to monitor the execution of basic blocks, so that the selective compilation logic can make well-informed decisions on which basic blocks to compile. Newer virtual machines for client and server systems use very sophisticated selective compilation techniques, such as the Adaptive Optimization System (AOS) in Jalapeño [AFG$^+$00], which consists of three subsystems encompassing several threads of control working together to perform feedback-directed optimization [AFG$^+$04]; these techniques were discussed in section 2.3.4. Such advanced schemes are clearly too heavyweight for our architecture; instead, a traditional, low-overhead technique based on one advocated by Schilling [Sch03] is used. Schilling's results indicated that more advanced profiling schemes did not lead to performance improvements that justified the additional complexity over simpler schemes, and that the simple schemes have high relevance when overhead is a concern.

There are two principal profiling schemes to choose from: *Sampling-based* and *counter-based* [AFG$^+$04]. In sampling-based profiling, bytecode execution is interrupted at regular intervals and the call stack is sampled to determine the methods in which an application spends most of its time, the general idea being that methods that run often will often be on the call stack at the time of sampling. Sampling-based profiling was not considered as a viable alternative because, in our system, on-line profiling is to occur at the basic block level, and the call stack gives an image of the execution state at the method level, necessitating additional processing if basic block locations are to be obtained. Unless sampling occurs extremely frequently (bringing with it much overhead), sampling-based profiling will give incomplete and too coarse-grained data to facilitate decent compilation choices at the basic block level.

For counter-based profiling, there are two variants: Counting the number of times a basic block is entered (executed), which we may call *frequency-based*, or counting the number of processor ticks (time) that is spent executing a basic block, which we may call *measuring-based*. Counting processor ticks means that processing must be done both at the entry (read the processor clock, store result) *and* exit (read the processor clock again, calculate difference, update counter) of the basic block. Since the basic block is a low level of granularity, profiling will occur relatively frequently, and the extra overhead of this scheme compared to a frequency-based one might be significant. Furthermore, it makes more sense to apply such a scheme at the method level, since it can help detect that a method spends much time executing a loop (even if the method itself is invoked relatively infrequently, the time spent in the loop will be reflected in the overall execution time of the method), whereas frequently executed loops are detected by a simple frequency counter at the basic block level.

Therefore, a frequency-based profiling scheme shall be used, in which a counter is associated with each basic block. The counter is initially set to some *treshold*. The treshold might be based on some heuristic involving e.g. the length of the basic block [Sch03] or the type of instructions in the block [MK00], or the treshold might simply be a value that has been determined a priori [ea05]; this is up to the implementation. In any case, each time the basic block is executed, the counter is decremented. Compilation is triggered when the counter reaches zero. Such a scheme affords simple, efficient implementations and is completely deterministic, easing integration with KVM.

The normal selective compilation logic can be overridden for basic blocks that have somehow been inferred to be poor candidates for compilation. For instance, if a basic block is very small (that is, it does very little work), it is possible that the overhead of the mixed-mode context switches associated with the native execution will be greater than the interpreter overhead

[Sch03]; in such cases, the system can refrain from compilation – that is, *inhibit* the compiler – and let the interpreter continue to handle the basic block in question.

3.3.2 Basic Block Descriptors

Since each basic block is to be profiled and possibly compiled, some way of storing *meta-data* for the basic block – that is, data additional to the static bytecodes that compose the basic block – is needed. Furthermore, since these data will be accessed in the critical loop of the bytecode execution engine, they must be simple to manipulate. For this purpose, a new data structure called the *basic block descriptor* is introduced. This data structure is expected to be key to fast, space-efficient implementations of the architecture. The intent of the basic block descriptor is to facilitate the management of the parts of an application being profiled, and those that have been compiled. To this end, a descriptor contains the following four fields:

Start offset The offset of the basic block's first instruction within the method's bytecode array (never changes).

Counter The number of times left to interpret the block before compilation should be triggered (decremented each time the block is interpreted).

Native code pointer Pointer to the compiled version of the basic block (if it has been compiled).

Inhibit flag Setting this flag instructs the selective compilation logic not to consider the basic block for compilation, even when its counter reaches zero. The criteria for setting this flag are implementation-dependent; an implementation might opt to never set it.

Figure 3.4 on page 46 shows a table of basic block descriptors that have been constructed from a method's bytecodes.

3.3.3 Algorithmic Description

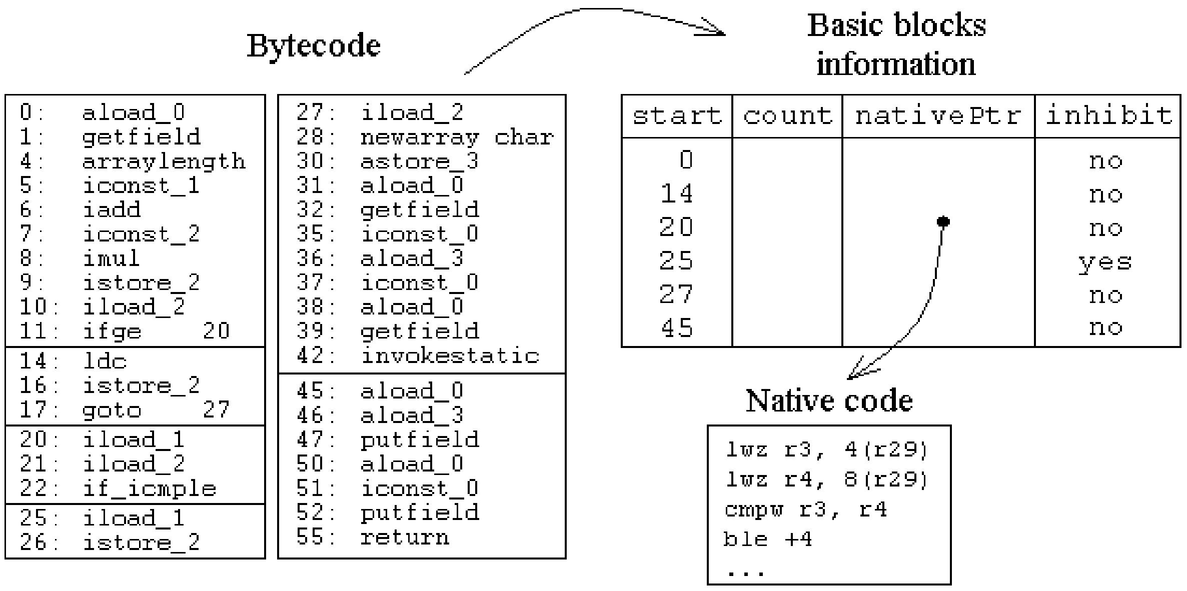
Algorithm 1 on the facing page describes the new way of executing bytecodes in more detail, centered around frequency-based profiling and treshold-based selective compilation.

The difference from KVM's original interpreter is a set of checks that are performed prior to the execution of each basic block. In essence, the algorithm "falls through" to interpretation only when a native version of the basic block is not available; and, when the fall-through has happened a certain number of times (equal to the treshold for the particular basic block), the basic block is compiled, in anticipation of many more future executions of the basic block.

The algorithm starts by checking if it's time to swap in another thread; this step is identical to the original KVM. Next, the basic block descriptor, which describes the block's current JIT-related

Algorithm 1 A complete profiling-guided mixed-mode execution scheme. Ties together per-basic block profiling, selective compilation, native execution and interpretation.

```
reschedulePoint: {we go here at beginning of execution of every basic block}
if time to reschedule then
   reschedule {makes another thread current}
end if
{the following (nested) if statement is the difference from original KVM interpreter!}
look up basic block descriptor for current Instruction Pointer
if basic block has been compiled then
   switch to native execution mode
   execute native version of basic block atomically
   switch back to interpreter mode {combined effects of native execution should equal those
   which interpretation would yield}
   goto reschedulePoint {End of basic block, start over from the top}
else
   decrement basic block counter
   if execution counter reached zero then
      if inhibit compilation of basic block then
         reset counter {compiler has been instructed not to compile this block}
      else
         compile basic block
      end if
   end if
end if
interpretPoint: {If we get here, basic block should be interpreted}
interpret one bytecode
if bytecode was control flow instruction then
   goto reschedulePoint {End of basic block, start over from the top}
else
   goto interpretPoint {Interpret another bytecode}
end if
```



```
0:   aload_0
1:   getfield
4:   arraylength
5:   iconst_1
6:   iadd
7:   iconst_2
8:   imul
9:   istore_2
10:  iload_2
11:  ifge     20
----------------
14:  ldc
16:  istore_2
17:  goto     27
----------------
20:  iload_1
21:  iload_2
22:  if_icmple
----------------
25:  iload_1
26:  istore_2
```

```
27:  iload_2
28:  newarray char
30:  astore_3
31:  aload_0
32:  getfield
35:  iconst_0
36:  aload_3
37:  iconst_0
38:  aload_0
39:  getfield
42:  invokestatic
----------------
45:  aload_0
46:  aload_3
47:  putfield
50:  aload_0
51:  iconst_0
52:  putfield
55:  return
```

start	count	nativePtr	inhibit
0			no
14			no
20		●	no
25			yes
27			no
45			no

```
lwz r3, 4(r29)
lwz r4, 8(r29)
cmpw r3, r4
ble +4
...
```

Figure 3.4: The bytecode of a method divided into basic blocks. A table of corresponding basic block descriptors facilitate JIT execution.

state, is looked up, using the address of the basic block's first instruction as a unique search key. If the basic block is found to be compiled, the basic block's native translation is executed, the interpreter's state is updated correspondingly, and we return to the top of the loop again. On the other hand, if the basic block has not been compiled, then it will be interpreted. However, before doing so, the frequency-based counter associated with the basic block is decremented. If the counter just hit zero, the basic block is compiled (unless, for some reason, the compiler has been inhibited from compiling that particular block). Whether the basic block was compiled or not, interpretation of the block then begins [2]. The interpreter interprets one instruction at a time, in a dedicated loop. Only when a control flow instruction (a conditional or unconditional goto, or a method invocation) has been interpreted, do we return to the top of the main loop, and the procedure is repeated.

When execution flows from native execution back to the interpreter, the native code must ensure that the interpreter's state is fully up-to-date, so that execution may gracefully proceed. Control is returned to the interpreter either as a result of successful native execution of a block, or as a result of an exceptional condition detected by the native code. The process of updating the interpreter's state is referred to as a native-to-interpreter context switch, and includes updating internal KVM registers used to manage execution, and the local variables and operand stack of the methods that have been affected by native execution. The details of this process are very implementation-dependent.

[2]If the basic block was compiled, we might wisely choose to execute it natively right away rather than interpret it one last time; however, this is assumed to have negligible effects on performance.

3.4 Compilation and Code Management

Once a basic block has been selected for compilation by the profiling-enabled interpreter, an entirely separate module is relied upon to perform actual compilation. The compiler takes a basic block descriptor as input and generates machine code from the basic block's bytecode. The location of the resulting code is stored in the basic block descriptor, for quick retrieval the next time the basic block is executed.

To the rest of the system, the compiler can be regarded as a black box. All that is expected of the compiler is that it generates native code that, following execution, leaves the virtual machine in the same state as interpretation of the original basic block would end up in. The compiler is free to use any number of optimizations as long as the preceding requirement is satisfied.

When a basic block is compiled, the resulting machine code must be stored somewhere. Since memory is scarce, the machine code for all the basic blocks that are compiled might not altogether fit in memory. It is the job of the *translation cache* module to manage the memory used for holding translated basic blocks. Exactly how this is to be done is up to the implementation.

Obviously, the compiler is heavily dependent on the target processor for which it generates machine code. Indeed, deciding on a target processor will be the fundamental decision when crafting an implementation.

3.5 Introspective Profiling

After testing the new system against a set of Java applications, the results will tell the hard facts of performance improvement (or lack thereof) relative to the outside world. However, whether the results are perceived as good or bad, they don't give detailed information as to where research and refinement can (and should) go from there. Specifically, before a problem can be fixed, it must be understood; and before it can be understood, it must be found.

Thus, in order to get a better understanding of how the new bytecode execution technique is behaving, an implementation should allow optional monitoring and quantification of JIT-related events – something we call *introspective profiling* (to differentiate this type of profiling from the on-line profiling perfomed by the JIT system itself). The presence of such facilities effectively transforms the system from a black to a white box, facilitating a deeper understanding of results observed at the application level. In early stages of development, introspective profiling can help reveal fundamental weaknesses in an implementation or the architecture itself; in more mature stages, it can drive the continuous refinement of the system, directing the focus to those areas that hold the most promise of improvement.

The following is *some* of the information that may be of interest to a researcher or developer wanting to further his understanding of JIT-enabled execution:

- How many basic blocks have been compiled and executed natively versus interpreted.

This quantifies the division of labour between the native execution mechanism and the interpreter.

- The translation reuse factor [RVJ$^+$01] – how many times the native translation of a basic block has been executed. The concept of JIT execution is to amortize the compilation costs over several subsequent executions, so a high reuse factor is strived for.
- The total number of bytecodes compiled, and the average number of bytecodes per compiled basic block. This says something about how much actual work is done during native execution; the more bytecodes per basic block, the better.
- The total size of the generated native code, and the average code expansion (ratio of native code size to bytecode size). Large code expansion means that more memory will be needed to hold bytecode translations, or that less translations can be held at one time.
- The number of context switches from interpreter mode to native mode. If the number of context switches is extremely high, performance will suffer, since a context switch in itself does not further the execution of an application.

These figures may be used in a number of ways. For instance, an excessive amount of compiled basic blocks reveals that the treshold for compilation may be set too low, or that the treshold for compiler inhibition is set too high. The relevant system variables must then be changed, and the application execution repeated. This gives a new set of performance results and profiling quantities to be interpreted by the researcher; the process may then have to be repeated again, and so on. For many parameters, this process is rather ad hoc [AFG$^+$04], and there is no unique “best combination” across all applications. However, it is a procedure that can be used to find a “reasonable” parameter vector, that produces decent results overall.

Although crucial to implement, the presence of introspective profiling does not have any ramifications that require special support from the architecture; an implementation is free to monitor JIT-related events as it sees fit.

Looking ahead, introspective profiling is the foundation that enables a deeper discussion of the experimental results that are the topic of chapter 5.

Chapter 4

Implementation

This chapter describes *one* possible implementation of the selective compilation framework presented in the previous chapter. The initial prototype leverages Sun's K Virtual Machine (KVM) version 1.1, and targets mixed-mode execution on the PowerPC processor in particular. Taken together, the material may give the reader a more tangible view of the scope of the project. However, as the emphasis is on very real solutions rather than abstract ideas, the bulk of the chapter operates at a rather low level of abstraction. The casual reader may therefore feel free to skip the more technical material without loss of continuity.

The implementation may be logically partitioned into three parts:

- The profile-guided compilation and mixed-mode execution scheme, which has been integrated with the KVM source code. The modifications effectively transform KVM from an interpreter-only execution model to a mixed-mode model, where decisions are made at the basic block-level on whether to interpret a basic block or compile and run it natively. While the changes and additions to the KVM core are not huge in terms of lines of code, they fall on the critical path of execution. Therefore, simplicity and efficiency of the selective compilation logic and the mixed-mode context switch is crucial in order to maintain hope of performance gains over the interpreter-only mode.
- The bytecode-to-native compiler, which has been written from scratch. The compiler takes a basic block of bytecodes as input and produces a corresponding sequence of native instructions for some target processor – in the case of the initial prototype, a 32-bit PowerPC – as output. Measured in lines of code, the compiler clearly represents the biggest implementation effort.
- Debugging and profiling of the JIT-related mechanisms themselves. Debugging-wise, the ability to trace basic block computation, compilation and mixed-mode context switching was implemented. This made it easier to verify that execution proceeded in a sane fashion. Profiling-wise, the JIT subsystems were instrumented to deliver statistics of their execution, such as the number of basic blocks profiled, the number of basic blocks compiled,

the size of the generated native code, the number of context switches, and so on. This enables the understanding of the actual behaviour that lies behind the raw performance measures.

The rest of this chapter is laid out as follows. Section 4.1 gives an overview of the project source code, showing how individual source files contribute to the components of the system, and how the new source code fits into the existing framework on which it is built. Section 4.2 discusses implementation details pertaining to on-line profiling and selective compilation, including the process of building basic block descriptor tables and how to represent the contents of basic block descriptors compactly in memory. Section 4.3 introduces a specific target processor for which compilation and mixed-mode execution is implemented. Section 4.4 concerns the approach to compilation and code generation. Section 4.5 details how the translation cache manages bytecode translations produced by the compiler. Section 4.6 discusses the implementation of introspective profiling facilities. Finally, section 4.7 concludes the chapter with some information on how the system was tested and debugged.

4.1 Overview of Source Code

This section gives an overview of the project's source code, which is based on KVM's source distribution. This section considers how KVM is implemented, in terms of source code and programming conventions; also, the major additions that constitute the implementation of extensions described in the previous chapter are presented, and their place in the KVM framework is established.

4.1.1 Source Code Structure

KVM is a very literal encoding of the JVM specification, which is reflected in a source code that is generally easy to follow. Furthermore, the source code is well-organized and well-commented; this, combined with the straightforward implementation techniques, makes KVM a system that is easy to understand, manipulate and extend. KVM therefore afforded the type of implementation and integration that was required of this project, without the author having to learn every KVM subsystem intimately.

The KVM source distribution comprises roughly 35,000 lines of C code, organized at the top level into five directories as shown in table 4.1 on the next page. The bulk of the code is in the `VmCommon` directory, which is completely platform-independent.

In an effort to cleanly separate our extensions from the existing system, a new directory called `VmJIT` has been added to the source tree structure. This directory contains all the JIT-specific files. Consequently, apart from some files in the `VmCommon` directory that had to be edited in order to bring JIT support into the bytecode execution core, all new code has been collected in `VmJIT`, totaling about 5,000 lines.

Directory	Description
`VmCommon`	Platform-independent code common to all incarnations of KVM.
`VmExtra`	Additional functionality not available on all platforms, but desired on many: JAR file reading, basic networking and debugger interface.
`VmUnix`	Code specific to UNIX / Linux port.
`VmWin`	Code specific to Windows port.
`VmWinCE`	Code specific to Windows CE port.
`VmJIT`	New directory in our solution; contains JIT-related code.

Table 4.1: The top-level structure of the KVM source distribution. VmJIT is a new directory added for the purpose of encapsulating JIT-related code.

4.1.2 Virtual Machine Components

Figure 4.1 on the following page shows an overview of how the KVM source files contribute to the individual components of the virtual machine, and how these components are related in the overall system. As can be seen, the new JIT subsystem is directly related to three parts of the existing system: Bytecode execution, classfile representation and manipulation, and memory management (allocation, to be specific). The exact nature of these relations will be explained in the subsequent sections of the chapter.

4.1.3 Compile-Time Flags

KVM uses a set of *compile-time flags* that control the inclusion of many optional features in the KVM build. The important flags that had to be familiarized during this particular project are summarized in table 4.2 on page 53.

The practice of using compile-time flags has been extended to encompass the new JIT features. Specifically, a flag called `ENABLEJIT` controls the conditional compilation of JIT support; that is, the actual integration of the JIT into KVM. By disabling this flag, the result is a JVM identical to the original KVM. When the flag is enabled, the system is compiled to use the JIT-enhanced execution mode.

4.1.4 Exception Handling Framework

KVM provides support for C++/Java-like exception handling mechanisms (`try`, `catch` and `throw`) in the actual virtual machine implementation through a set of macros that use the standard C library functions for non-local gotos, `setjmp()` and `longjmp()`. When an exception occurs inside the virtual machine (for example, as a result of a failed memory allocation), it is caught by the innermost dynamically nested `try-catch` statement; the outermost

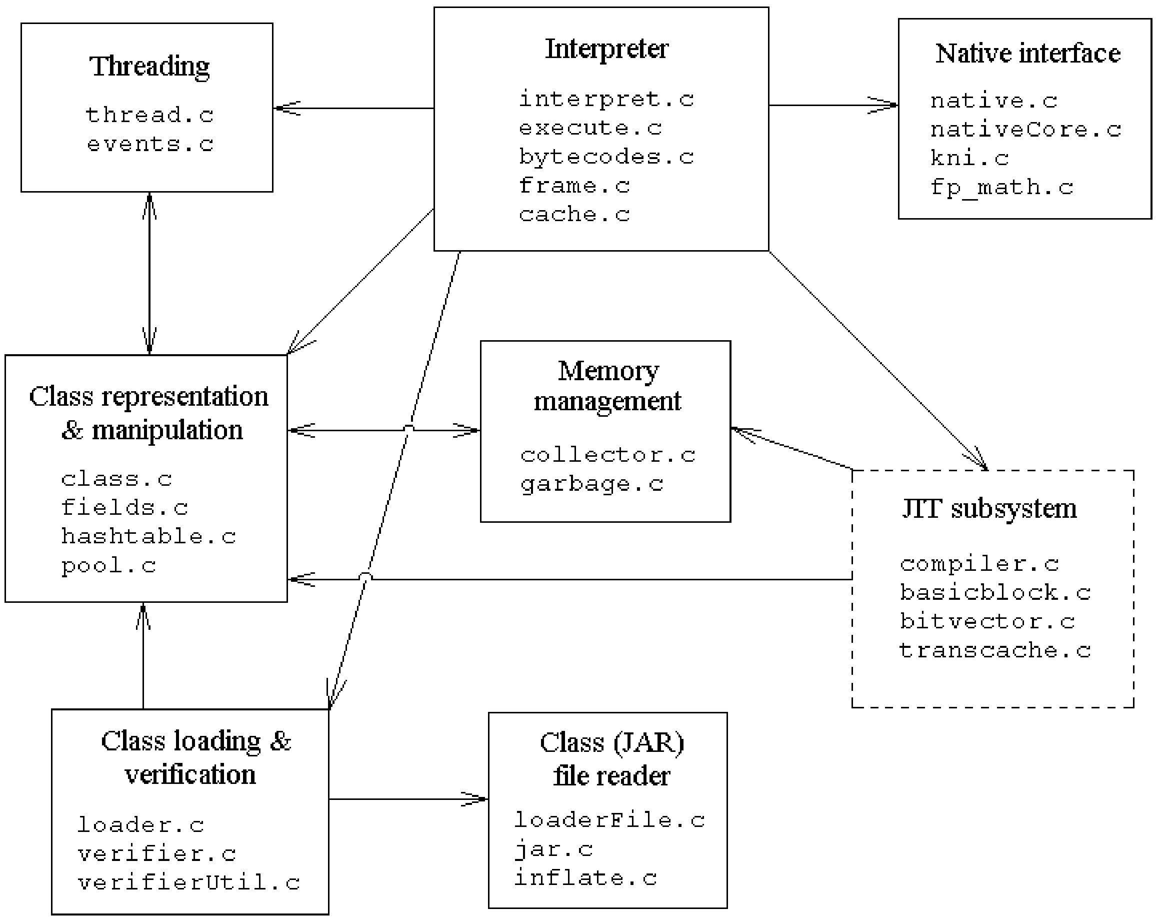


Figure 4.1: Overview of the KVM source code, and how the new JIT subsystem code fits into this framework. Note that the figure only shows functional dependencies, which do not necessarily imply source code changes. For instance, the JIT translation cache relies on the existing memory management system to allocate storage for native code, but this did not necessitate any changes to the memory manager – only knowledge of its interface and semantics.

`try-catch` statement is located in the interpreter core, and turns the exception into an actual Java application-level exception since the virtual machine couldn't handle it internally.

The compiler relies on the existing exception handling facilities to ease the detection and handling of compiler-specific exceptional events, such as attempts to add native code to a full translation cache.

4.2 On-Line Profiling and Selective Compilation

This section looks at the implementation details pertaining to the profiling and selective compilation of basic blocks, required to support the behaviour as prescribed in section 3.3.

Compile-time flag	Description
ROMIZING	Compiles and links Java system classes directly into the KVM binary, which can significantly reduce JVM startup latencies.
ENABLEFASTBYTECODES	Interpreter optimization.
RESCHEDULEATBRANCH	Instructs interpreter to only perform thread rescheduling at control branches, as opposed to every bytecode.
SPLITINFREQUENTBYTECODES	Interpreter optimization.
PADTABLE	Interpreter optimization.
LOCALVMREGISTERS	Interpreter optimization.
INCLUDEDEBUGCODE	Inclusion of debug code (tracing).
ENABLEPROFILING	Profiling of the virtual machine.

Table 4.2: Summary of some important compile-time flags that control features of KVM.

4.2.1 Basic Block Descriptor Table Representation

The table of basic block descriptors for the current method needs to be accessed and searched at the beginning of each iteration of the bytecode execution loop, as implied by algorithm 1 on page 45. Since it will happen very frequently, it is extremely important that this procedure can be done quickly. The following subsections details how the implementation achieves fast table access and lookup.

Ensuring Fast Access

In order to facilitate fast access to the table itself, the table is made part of KVM's internal data structure used to represent methods. This choice ensures that the table can be directly accessed through the method pointer in the current activation record. Storing the table elsewhere would require some type of lookup to access the table (e.g. keeping the basic block tables in a hash map where tables are hashed on method name), which would introduce overhead both in space and time. Simply adding the basic block table to the existing method structure does not have any consequences for existing code that manipulates the structure, so the goal of fast access makes this solution the preferred choice.

Ensuring Fast Lookup

In order to facilitate fast lookup and space-efficient storage of the table, the table shall be stored as a *linear array*. The lookup procedure can now be implemented as a fast search in logarithmic time, since the basic blocks are ordered by increasing bytecode offset in the table. Using a linear array means that the basic blocks of a method must be precomputed before profiling can commence, but this is a small one-time cost that will be amortized over many (fast) subsequent lookups.

An alternative that has been briefly considered is to create individual basic block descriptors lazily, the first time a basic block is executed. The basic block table could then be represented as a linked list. The advantage of this method is that no memory is used to hold descriptors for blocks that are never executed. The list could use a Most Recently Used scheme to keep the "hot" basic blocks near the head of the list, which would perhaps result in equally fast lookup times as performing a logarithmic search of a linear array. However, there can be many basic blocks in a method, and each basic block descriptor is intended to occupy very little memory. Using a linked list can lead to highly fragmented memory due to the allocation of many tiny chunks of memory, eventually necessitating expensive compacting garbage collection. Additionally, there is an overhead of one word associated with each memory allocation in KVM, which is needed by the memory subsystem to keep track of the memory chunk. Since the descriptors themselves are so small and so many, this overhead would be significant relative to the actual data that are stored. By allocating all these tiny descriptors in a single contiguous array, the fixed memory management overhead is amortized over several descriptors, and fragmentation is mitigated. For these reasons, the linear array solution is the preferred choice.

4.2.2 Lazy Creation of Basic Block Descriptor Tables

Initially, when a method structure is built by the class loader, the method's table of basic block descriptors is *nil*. Only when the instrumented interpreter detects that the method has been invoked *twice*, is the memory required to hold the table allocated and the data structure initialized. This choice is motivated by the desire to avoid spending valuable time and space on generating basic blocks meta-data for one-time initialization methods, including static class initializers.

4.2.3 Instrumentation of Method Invocation

In order to support the lazy creation policy for basic block tables, additional processing must be performed in KVM's handling of method invocation. There are four ways for a JVM bytecode program to invoke a method, corresponding to the four bytecodes `invokevirtual`, `invokespecial`, `invokestatic` and `invokeinterface`. An important observation is that the only real difference between these four operations is the underlying algorithm that determines the actual method to invoke at runtime, based on a symbolic method reference in the caller class; once the correct method has been determined, the further steps are identical for all four invocation kinds. KVM exploits this fact by merging the actual invocation process (pushing a new activation record, and so on) following method lookup. In the context of basic block discovery, this means that the logic for deciding when to compute the basic blocks of a method can be put in a single place.

Algorithm 2 on the next page shows how KVM's joint processing of method invocation has been instrumented. Specifically, embedded in the processing is a check to see if the method's *invoked flag* is set. If it is, this means that the method has already been executed at least once. If, additionally, the basic blocks of the method have not already been computed, then this is exactly the second time the method is invoked, and the computation is performed. If, on the other hand, the invoked flag is not set, then it is done so now. That way, the next time the method is invoked, the aforementioned test will succeed.

Algorithm 2 The algorithm for handling method invocation in KVM's interpreter, augmented to compute a method's basic blocks the second time it is invoked.

```
... {the following are excerpts from the KVM interpreter's unmodified opcode handlers}
if opcode is INVOKEVIRTUAL then
   do virtual method lookup
   goto invokeMethod_generic
else if opcode is INVOKESPECIAL then
   do 'special' method lookup
   goto invokeMethod_generic
else if opcode is INVOKESTATIC then
   do static method lookup
   goto invokeMethod_generic
else if opcode is INVOKEINTERFACE then
   do interface method lookup
   goto invokeMethod_generic
end if
... {other opcode handlers not shown}
invokeMethod_generic:
{the following if statement has been added}
if invoked flag of method set and basic blocks not computed then
   build basic blocks data structure {used in further profiling}
else
   set the method's invoked flag
end if
... {continue with original KVM invocation processing; push frame, pass arguments, etc.}
```

The invoked flag itself is stored in the existing `accessFlags` member of the KVM method structure; luckily, this variable has several unused bits we can use. (The definition of internal access flags that can represent implementation-specific boolean information beyond the official JVM access flags is a "trick" used in other parts of KVM also.)

4.2.4 Building the Basic Block Descriptor Table

To find the bytecode offsets of basic blocks in the method, the algorithm by Aho et al [ASU86] is used. It only needs a single, fast pass over the bytecodes, where all but the control flow instructions are skipped. A temporary bit vector is used to build the set of basic block headers discovered by the algorithm, only requiring *number-of-instructions-in-basic-block* bits of dynamic memory (the vector data fits in a single word for most basic blocks). Following the discovery of basic block offsets, a basic block table of sufficient length is allocated, and the fields of the basic block descriptors are initialized as follows:

Counter A global, predefined treshold is used to decide on when a basic block should be compiled. Specifically, the profiling counter is initially set to the compile-time constant `COMPILER_TRESHOLD`. This scheme is the simplest to implement, and will be used until empirical evidence is able to prove its inadequacy, justifying the implementation of a more sophisticated per-basic block treshold based on some heuristic, such as the length of the block.

Native Code Pointer Set to the address of the native-to-interpreter context switch code, ensuring a graceful exit to the interpreter even if this code is jumped to before the basic block has been compiled.

Inhibit Flag Set to 1 only if the number of instructions in the basic block falls below the compile-time constant `INHIBIT_TRESHOLD`.

4.2.5 Basic Block Descriptor Representation

Since a method can potentially contain a large number of basic blocks, it is important that their meta data are represented in a space-efficient manner. To make the basic block descriptors as compact as possible, a bit packing [NW01] / bit stealing [BFG02] scheme is used. Since a Java method's implementation can be maximum 65536 bytes long [LY99], the basic block offset only requires 16 bits. Furthermore, it is expected that a 16-bit treshold for the execution count will be sufficient; thus, the offset and execution count can be stored together in one word (32 bits). The native code pointer is currently a direct memory pointer, and so requires one word. However, by ensuring that the native code is at least halfword-aligned, the least significant bit is an implicit 0; this bit can now be used to store the inhibit flag. Thus, each basic block descriptor is two words long.

Bringing the total descriptor size down to just one word would indeed be possible by using techniques analogous to those of Bacon et al's [BFG02] Java object model; specifically by storing a translation cache *index* rather than the code pointer itself. However, this would have to be done at the expense of the maximum allowed method size and execution count treshold, and might lead to slightly reduced performance due to extra levels of indirection when fetching the code pointer. Rather than eagerly introducing this scheme, we choose the simple route and determine later if refinements are indeed necessary.

4.3 Adopting a Target Processor

The prototype targets a 32-bit PowerPC [IBM03]. The PowerPC was chosen because it represents a mature, neutral RISC architecture that implements a sequential execution model, making it a target that is easy to generate machine code for. For a RISC, the PowerPC has a rather rich instruction set; this means that a simple first-generation compiler can implement a straightforward translation of bytecode by mapping JVM instructions to the most general sequence of PowerPC instructions, while more sophisticated compilers can take advantage of more specific features of the PowerPC instruction set (such as *machine code idioms* [CO02]) to implement the semantics of bytecode in more optimal (shorter) sequences of machine code instructions.

4.3.1 Register Usage

The first major challenge in adopting the PowerPC processor is to decide on how to manage its large amount of registers during execution of bytecode translations. The principal PowerPC registers are shown in figure 4.2.

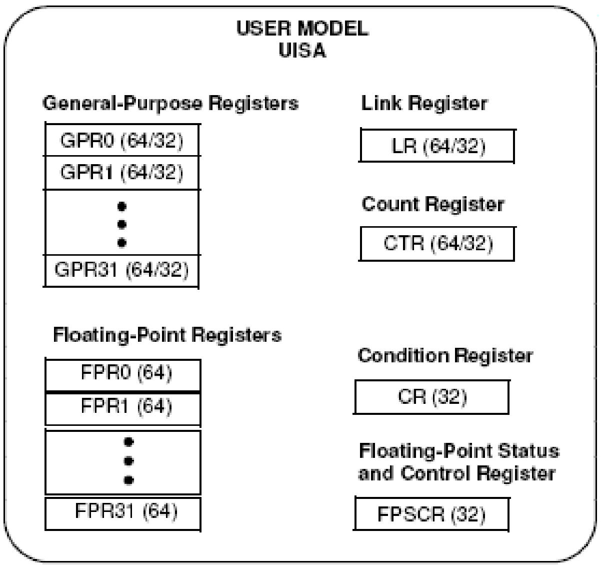


Figure 4.2: Principal PowerPC registers. (Adopted from [IBM03].)

The usage of PowerPC registers in execution of bytecode translations is shown in table 4.3 on the next page. It conforms to the register conventions used by the GNU C Compiler (GCC), which roughly matches the AIX [IBM96] specification. This ensures fast context switches (by being aware of *exactly* which registers must be saved and restored), and the ability to easily call C functions from native code, a feature which is relied on by the code that implements the more complex bytecode operations. This seamless interaction between interpretation and native execution comes at the expense of the project being tied to GCC (something which the

PowerPC register	**Description**
`GPR0`	Scratch register used in calculations.
`GPR1`	C stack pointer (managed by host compiler).
`GPR2`	Table of Contents (TOC) pointer (managed by host compiler).
`GPR3`	Scratch register used in calculations. 1st argument word. 1st word of function return value.
`GPR4`	Scratch register used in calculations. 2nd argument word. 2nd word of function return value.
`GPR5`	Scratch register used in calculations. 3rd argument word.
`GPR6`	Scratch register used in calculations. 4th argument word.
`GPR7`	Scratch register used in calculations. 5th argument word.
`GPR8`	Scratch register used in calculations. 6th argument word.
`GPR9`	Scratch register used in calculations. 7th argument word.
`GPR10`	Scratch register used in calculations. 8th argument word.
`GPR27`	VM Instruction Pointer (IP).
`GPR28`	VM Stack Pointer (SP).
`GPR29`	VM Locals Pointer (LP).
`GPR30`	VM Frame Pointer (FP).
`FPR0`	Scratch register used in calculations
`FPR1`	Scratch register used in calculations. 1st float argument word. Float function return value.
`LR`	Target address of indirect branches and function calls.
`CR`	Used in conditional branching.

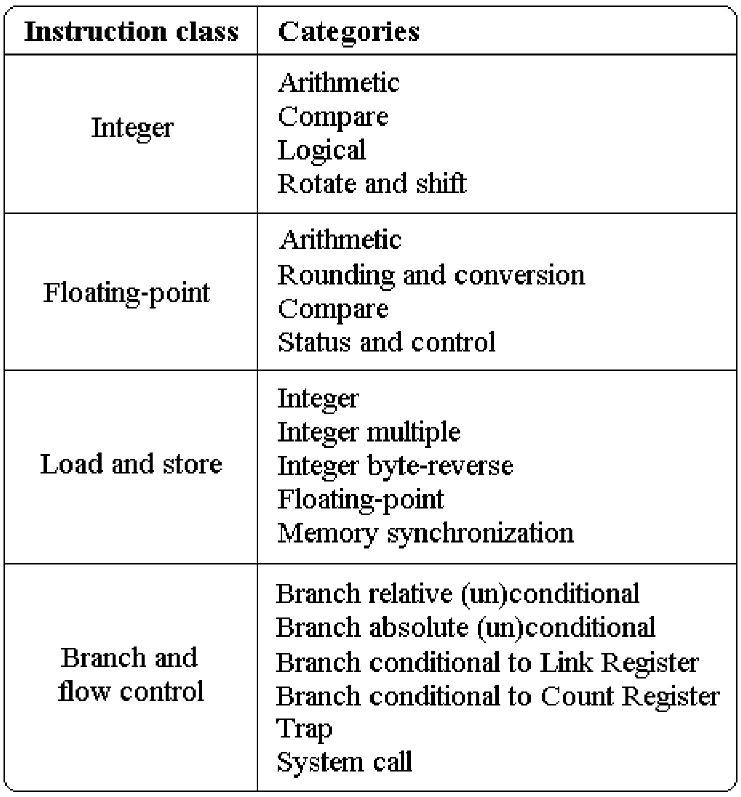
Table 4.3: Usage of PowerPC registers in native execution. The registers not listed are currently unused.

original KVM is not), but since the context switch is so performance-critical when mixed-mode execution occurs at the basic block level, we feel this choice is well justified.

The fact that the PowerPC has so many registers has been used to map frequently accessed parts of the execution state statically; specifically, the four KVM registers – IP, FP, SP and LP – are always kept in the same registers throughout native execution. This implies that, in the context switch from the interpreter to native code, the KVM registers are moved to native registers. In the context switch from native code back to the interpreter, the possibly “dirty” contents of the native registers are written back to their respective KVM registers, ensuring that they are up-to-date when interpretation continues. The transfer of registers is the *only* processing done in the context switch; specifically, it is up to the code for each basic block to ensure that the frame locals and operand stack are up-to-date in memory before the context switch is initiated.

4.3.2 Inherent Instruction Mapping Capabilities

How effectively the JVM instruction set can be mapped onto a target processor depends on the instructions that the processor provides. An overview of the categories of PowerPC instructions is shown in figure 4.3.



Instruction class	Categories
Integer	Arithmetic Compare Logical Rotate and shift
Floating-point	Arithmetic Rounding and conversion Compare Status and control
Load and store	Integer Integer multiple Integer byte-reverse Floating-point Memory synchronization
Branch and flow control	Branch relative (un)conditional Branch absolute (un)conditional Branch conditional to Link Register Branch conditional to Count Register Trap System call

Figure 4.3: Overview of PowerPC instruction classes.

All JVM arithmetic and comparison operations that take `int`, `float` and `double` arguments can be mapped one-to-one to an equivalent PowerPC instruction. Conditional control flow JVM operations can also be mapped one-to-one. Translation of operations that take `long` (64-bit) arguments is more involved, because the targeted PowerPC can only perform 32-bit operations involving integers (the architecture does define 64-bit extensions, however, these are not yet in widespread use). Loads and stores of local variables and object fields, where the offset relative to an activation record or object is a constant known at compile time, can usually be translated compactly; the PowerPC has a set of load and store instructions that use an immediate addressing mode, where a 16-bit signed offset can be encoded directly in the instruction word.

The PowerPC lacks instructions that convert from `int` (`long`) to `float` (`double`). The only way to achieve this is either through a rather long sequence of instructions (provided in [IBM96]) or by calling a library function. A related problem is that a PowerPC floating-point register cannot be moved directly to a general purpose register or vice versa; the register must be stored to memory, then loaded back from memory into the proper register. These limitations were a bit disappointing, considering that similar architectures, like the MIPS, provide instructions that the JVM conversion operations can be mapped directly onto, and where the data path between floating-point and integer registers is point-to-point.

The PowerPC can perform indirect branching through the Count Register and Link Register. This procedure is a bit involved because the target address must first be loaded into a general purpose register, then moved to the Count or Link Register, before a branch can be performed, requiring at least three PowerPC instructions. Thus, translation of virtual method dispatch, a frequent operation in Java, is not facilitated in an optimal way.

For the interested reader who wishes to learn the exact mappings of JVM opcodes to PowerPC instructions, the code listing in section A.1.2 on page 94 may be perused.

4.4 Compilation

This section describes the strategies used by the compiler to generate native code for a basic block of bytecodes.

The compiler translates the bytecodes in a single pass, generating code for bytecode instructions one at a time, without using any intermediate representations. A backpatching scheme is used to deal with control flow instructions. Each opcode is translated in a very straightforward manner, by generating code corresponding to a sequence of *micro operations* which implements the exact behaviour of the particular opcode. For instance, algorithm 3 shows the sequence of operations that implement the `iadd` opcode. Corresponding sequences for other opcodes can be derived quite easily from the opcode descriptions in the JVM specification [LY99], and by studying the implementation of the KVM interpreter. No auxiliary data structures, such as native stacks, are used by compiled code; the compiler generates code that operates on the same data structures and memory locations as the interpreter.

Algorithm 3 Sequence of micro operations that implement the `iadd` opcode.

```
pop second operand from stack
pop first operand from stack
add first and second operand
push result on stack
```

4.4.1 Lazy Update of VM State

The compiler strives to only update the native registers that cache VM state when it is strictly necessary. To this end, the compiler keeps track of the values that the VM registers *ought* to have as code is generated, rather than generating code that actually updates the registers. Specifically, lazy updates are performed as follows:

IP Only updated when the end of a basic block's translation has been reached, or prior to checking for an exception, or prior to calling a C function from native code that might throw an exception.

SP Same as for IP. Additionally, during translation of the block, a relative stack offset is maintained, and is encoded in PowerPC instructions that fetch stack words; this implements pushing and popping behaviour without actually updating the SP register.

FP, LP Only updated in the translation of `return` opcodes.

4.4.2 Dealing With Exceptions

Exceptions are handled exclusively by the interpreter. However, the native code must still check for exceptional conditions, and transfer control the interpreter if one is detected.

In order to avoid redundancy and embedded exception handling code in the handling of opcodes that might throw exceptions (but usually don't), generic exception handling code is installed in the translation cache upon virtual machine startup. An opcode handler then merely has to check for the exceptional condition and conditionally branch to the relevant exception handling code located elsewhere. The exceptional handling code takes care of flushing the VM state to memory and actually raising the exception.

4.4.3 Translating Conditional Branches

A conditional branch signifies the end of a basic block. However, exiting back to the interpreter would be highly inefficient in this case, since the successor basic block can easily be determined directly by the native code. Indeed, the compiler generates code that performs the conditional branch and fetches the native code pointer from the successor basic block's descriptor. The native code then jumps to this pointer. If the successor basic block has been translated, execution of the block continues directly. If the block has not been translated, its descriptor's native code pointer must have pointed to the native-to-interpreter context switch code, causing the context switch to be initiated. The net effect is that the native code only exits back to the interpreter when the successor block has not been translated. This is referred to as *chaining* basic block translations.

4.4.4 Translating Complex Opcodes

Compared to a RISC architecture, the JVM instruction set has instructions that operate at vastly different levels of abstractions; this means that a single bytecode operation often cannot map directly to a single RISC CPU instruction [CD03]. For instance, whereas the `iadd` instruction adds two integers, the `new` instruction is required to resolve a symbolic class reference, initialize the corresponding class if it has not already been initialized, and allocate memory for an instance of the class, which might trigger garbage collection. Furthermore, if the `new` instruction fails at any stage, it must throw an appropriate exception. In order to deal with this category of complex instructions, the compiler generates native code that calls a proper C function to do the work. Fortunately, most of the required C functions could be leveraged directly from the original KVM implementation. The native code simply places the arguments to the function in the proper native registers (according to the GCC calling convention), calls the C function, and processes the return value. Prior to calling the function, the cached VM registers are flushed to memory, so that if an exception should occur, the interpreter's state will be up-to-date when it starts processing the exception.

4.4.5 Optimizations

A small set of simple optimizations are currently applied; they are all focused at generating more compact machine code. The optimizations are described below.

Avoiding Invocation of Empty Methods

If the compiler detects an invocation of a method that is known to be empty (i.e., contains only a `return` opcode), code for the invocation is not generated. This can be detected at compile time only when static invocation semantics are used (`invokespecial` or `invokestatic`) or the target method is declared as `final`. For example, the constructor of the class at the top of the class hierarchy, `Object`, is empty, and every constructor of every subclass must call the `Object` constructor; eliminating this call will lead to faster object initialization.

Inlining of Getter and Setter Methods

By using simple pattern matching, the compiler detects the invocation of getter (setter) methods – methods that get (set) an object's field, and nothing else. The use of such methods is considered good programming practice in object-oriented programming, since it achieves encapsulation of a class's implementation through an interface. Inlining these simple methods results in faster, more compact native code by eliminating the unnecessary (from an implementation standpoint) method call. Figure 4.4 on the next page shows the premise of this optimization. Like the elimination of calls to empty methods described above, this optimization is only safe when the invocation has static semantics or the target method is declared as `final`.

Java source	Bytecode

`int getVariable() {`
`    return this.variable;`
`}`
	`ALOAD_0`
	`GETFIELD variable`
	`IRETURN`

`void setVariable(int value) {`
`    this.variable = value;`
`}`
	`ALOAD_0`
	`ILOAD_1`
	`PUTFIELD variable`
	`RETURN`

Figure 4.4: Getter and setter methods can be detected by simple pattern matching and are simple to inline, since the object that the method is invoked on is at the top of the stack at the point of invocation.

Merging Translation of Common Bytecode Sequences

Some particular sequences of bytecodes occur a lot more frequently than others. O'Donoghue et al [OLPW02] did research on *bigrams* – pairs of bytecodes, where one bytecode is always executed after the other. They list the 10 most frequent bigrams over 12 benchmarks. The most common pair by far was found to be (`aload_0`, `getfield`). Applying this knowledge to translation, the compiler "peeks" at the next bytecode whenever the first in a frequent pair of bytecodes is encountered (e.g. `aload_0`), and if it matches the second bigram component (e.g. `getfield`), the translation of the two bytecodes are merged for better code quality. This technique is analogous to the simple method inlining described above, in that both techniques are based on simple pattern matching.

Peephole Optimization

Peephole optimization [ASU86] is performed on the machine code following normal code generation. The optimizer considers a small window of instructions, trying to recognize redundant operations or operations that can be performed more efficiently using different instructions. In particular, the peephole optimizer looks for unnecessary consecutive loads and stores, and thus removes a lot of the redundancy due to the way JVM stack opcodes are currently translated. Peephole optimization is expected to only be a temporary solution, however; it can be eliminated once a more sophisticated translation scheme for stack operations is in place.

4.5 Native Code Management

The translation cache is implemented as a static, "flat" buffer. This decision was motivated by Shaylor's work [Sha02], where overall performance did not appear to suffer when using a similar scheme. A command line switch, `-cachesize`, can be used to specify the size of the translation cache (analogous to how the virtual machine's heap size can be specified through the `-heapsize` switch); 64KB is the default size. When the JIT subsystem is initialized, a buffer of the specified size is allocated. The translation cache keeps an internal index into this buffer, which is incremented as the compiler adds instructions to it. Figure 4.5 illustrates this scheme. When the buffer is full, the internal index is reset, effectively clearing the buffer. Additionally, the descriptors of basic blocks that have been compiled are updated to reflect the fact that their translations are no longer valid. Upon subsequent execution of such blocks, profiling will start over.

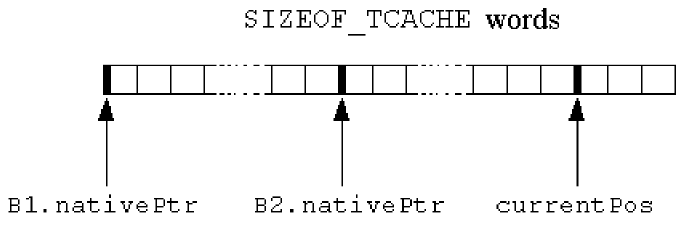


Figure 4.5: The translations of basic blocks are stored in a shared static array of words. An internal marker (`currentPos`) is used to record the position in the array where the next instruction will be stored.

This is the simplest possible code management scheme to implement. If empirical evidence should suggest that discarding the entire contents of the cache is hopelessly inefficient, the code management scheme must be revisited; for instance, a Least Recently Used (LRU) replacement policy could be used, like in E-Bunny [ea05]. However, like Shaylor [Sha02], we hypothesize that the discarded code can be quickly regenerated if necessary, because compilation of basic blocks is very fast. In fact, unless the translation cache is so small that the machine code corresponding to the working set of the application won't fit (which would cause heavy thrashing), clearing the cache regularly might even be an effective way of getting rid of code that has gone from "hot" to "cold" over a period of time, or code that shouldn't have been compiled in the first place (due to inadequate decision logic). Given that compilation is extremely cheap, any "hot" basic blocks that are discarded as a side-effect will soon be recompiled at little cost. In this case, the overall impact on performance should be small, and allows us to choose a smaller cache size than what would be possible if recompilation were expensive.

4.6 Introspective Profiling

As explained in section 3.5, the purpose of introspective profiling is to quantify the behaviour of the JIT system in order to facilitate a deeper understanding of the interplay between interpretation, selective compilation and mixed-mode execution. KVM already has a framework for profiling, and by following the conventions of the existing framework, implementing similar capabilities for the new JIT system was straightforward. A compile-time flag, `ENABLEPROFILING`, controls the conditional compilation of profiling-related code. Profiling the virtual machine slows down execution slightly, therefore it is convenient to have the option to statically disable profiling for production builds of the system.

The profiling is implemented as a set of counters that are incremented in response to various events. Specifically, the counters are:

CompileCounter The total number of basic blocks compiled; incremented each time the compiler is invoked.

CompiledBytecodesCounter The total number of bytecode instructions compiled; incremented each time the compiler translates a single instruction.

CompiledBytecodeSize The total size of bytecode compiled, measured in bytes (used to compute code expansion).

CompiledNativeCodeSize The total size of native code generated, measured in bytes (used to compute code expansion).

NativeExecCounter The total number of interpreter-to-native context switches; incremented each time native code is entered.

InhibitCompileCounter The total number of times the compiler has been inhibited from compiling a basic block; incremented each time the basic block's execution count reaches the treshold for compilation *and* the basic block's inhibit flag is set.

TcacheExhaustedCounter The total number of times the translation cache has been cleared to make room for new translations; incremented each time an attempt is made to add an instruction to the translation cache when it is full.

After execution has ended, the values of the above counters are written to the JVM's standard output (either a console window or a file).

4.7 Testing and Debugging

In order to test and debug the operation of the system, a full-system simulation tool called Simics [FHH$^+$02] has been used. The tool simulates a complete embedded target platform (MontaVista) upon which the virtual machine and compiler can be run. Such a simulated setup offers many benefits during low-level systems software development, including deterministic execution and high system visibility.

Debugging-wise, Simics *magic breakpoint* feature was invaluable. A "magic instruction" is a special instruction not normally executed by target applications; Simics can be instructed to halt simulation when such an instruction is encountered. By having the compiler insert magic instructions in the code generated for a bytecode operation, the resulting code could quickly be stepped through and verified in Simics's integrated debugger.

Finding code generation bugs still proved to be a challenge, though, even for the rather simple code generation strategy employed by the prototype. There were two causes. First, the virtual machine executes a tremendous number of bytecodes, and the bug might not do its damage until far into execution. Second, the bug might not actually *manifest* – as in ultimately causing an exception to be thrown, for example – until long after the actual damage has been done. Indeed, there is no guarantee that the bug will manifest at all. The solution was to painstakingly write various small Java applications that generated particular (sequences of) bytecode instructions, so that the full set of instructions could eventually be verified. This was not a very rigid approach to testing, but no free set of complete virtual machine tests could be found. (Interestingly, the KVM `Makefile` contains instructions to run an application called `RegressionTest`, presumably used to test the virtual machine, but this application is not included in the public source distribution.)

Even when the machine code generated for each bytecode instruction was verified to be correct, understanding the highly dynamic interactions of the system components was a challenge. The reflective profiling mechanisms capture the application-level behaviour, but during development, something more fine-grained was needed to reduce the sense of working in the dark. The approach taken was to extend the existing *tracing facilities* of KVM to include the textual tracing of JIT-related activities. Specifically, the following activities can be traced in debug builds:

- The discovery of a method's basic blocks.
- The compilation of basic blocks.
- The mixed-mode context switches.

Combining these traces with the existing KVM trace generators for bytecode interpretation and method execution gave a very detailed history of the progression of the mixed-mode execution, and aided the discovery of a few bugs that would otherwise have been very hard to track down.

Chapter 5

Experimental Results

This chapter presents the results obtained from running a set of Java applications on the initial prototype of the JIT-enhanced virtual machine. The purpose of this presentation is to show how the performance compares to the original, interpreter-based KVM, and to provide a discussion of what can be drawn from these results. We also look at how changing parameters associated with the JIT subsystem affects performance. In the end, these results serve as the main indicator of whether the project goal of creating a lightweight acceleration mechanism has been achieved, and will be used to guide the future work on the system.

The rest of this chapter is laid out as follows. Section 5.1 discusses the approach to selecting the benchmarks used to evaluate the system, and introduces the selected benchmarks. Section 5.2 presents the performance improvements over KVM, starting with the overall improvements, before offering more detailed presentations and discussions of the results obtained for each benchmark. In the following three sections, the introspective profiling facilities of the system are relied upon to further the understanding of the system's behaviour. First, in section 5.3, the code expansion due to bytecode being compiled to machine code is considered; code expansion determines how many bytecode translations can fit in a fixed-size translation cache. Next, sections 5.4 and 5.5 consider the effects of changing the treshold for triggering compilation and the size of the translation cache, respectively. Any trends that emerge from changing these parameters are identified and their impact is discussed. Section 5.6 summarizes the memory footprint of the implementation. Section 5.7 concludes the chapter with an informal comparison of this system to a related system, E-Bunny [ea05].

5.1 Benchmarks

The system that has been developed is not a general-purpose Java virtual machine. Rather, it is intended to operate in particular domains, where the common characteristic is that memory is scarce. Selecting applications that are representative of such domains is difficult; in particular, there is a risk of choosing applications that are either too narrow (don't have relevance for a suitable range of embedded systems) or too wide (don't have relevance for most embedded systems at all).

Initially, standard benchmarks used on desktop systems were pursued. SPEC JVM98 [(SP] is a popular commercial benchmark suite that tests many aspects of JVM operation and performance. However, SPEC JVM98 requires at least 48MB of memory, which means it has low relevance for most embedded applications [Per]. Furthermore, SPEC JVM98 requires that the full Java client libraries are installed, while we are targeting CLDC, which only includes a few core libraries. JavaGrande [EPP] is another popular benchmark, but it addresses the performance of so called "Grande" applications – applications which have large requirements for any or all of memory, bandwidth and processing power. Thus, JavaGrande, like SPEC JVM98, seems too heavyweight for testing our system.

Instead, a set of lightweight benchmarks have been selected. The main criteria for these benchmarks have been

- The benchmark must have memory requirements that are realistic in an embedded environment, which means a few megabytes, at most. The benchmark should not constantly allocate objects, because this would cause the garbage collector to run frequently; we want to isolate the effects of JIT execution as much as possible.
- Simplicity of the benchmark takes presedence over real-world relevance. The results obtained from a simple application are easier to interpret than those due to a complex one, and may help reveal fundamental shortcomings of (specific parts of) the architecture or implementation. Use of real-world applications is considered future work.
- The benchmark must only require the core libraries available in a CLDC configuration; this effectively excludes all benchmarks which test graphics capabilities or other functionality that falls outside the areas addressed by CLDC.
- The benchmark implementation itself must be relatively small, reflecting the fact that real-world applications for systems with limited memory must be small.

Three benchmarks fulfilling these criteria comprise the set of applications used during testing. Each benchmark is briefly presented in the sequel.

CaffeineMark Embedded

From the CaffeineMark homepage [Corc]:

> The CaffeineMark is a series of tests that measure the speed of Java programs running in various hardware and software configurations. CaffeineMark scores roughly correlate with the number of Java instructions executed per second, and do not depend significantly on the the amount of memory in the system or on the speed of a computers disk drives or internet connection.

Name of test	Description
Sieve	The classic sieve of eratosthenes finds prime numbers.
Loop	The loop test uses sorting and sequence generation as to measure compiler optimization of loops.
Logic	Tests the speed with which the virtual machine executes decision-making instructions.
Method	The Method test executes recursive function calls to see how well the VM handles method calls.
Float	Simulates a 3D rotation of objects around a point.

Table 5.1: The tests included in the CaffeineMark Embedded benchmark.

CaffeineMark is a microbenchmark that exercises core JVM functionality in an isolated fashion – each test focuses on one category of bytecode instructions. This makes it a very useful benchmark especially in early phases of performance evaluation and tuning, since it can help reveal fundamental bottlenecks. CaffeineMark was also used to test a related system, E-Bunny [ea05].

Table 5.1 gives an overview of the tests that are part of CaffeineMark.

jBYTEmark

jBYTEmark is a Java port of the BYTEmark benchmark, originally written in C. From the BYTEmark homepage [Mag]:

> The BYTEmark benchmark test suite is used to determine how the processor, its caches and coprocessors influence overall system performance. Its measurements can indicate problems with the processor subsystem and (since the processor is a major influence on overall system performance) give us an idea of how well a given system will perform.

jBYTEmark is a more complex bytemark than CaffeineMark, but the algorithms that are implemented are well-known, facilitating understanding of its implementation and of the JVM functionality that is affected by it.

Table 5.2 on the following page gives an overview of the tests that are part of jBYTEmark.

SciMark

From the SciMark homepage [PM]:

Name of test	**Description**
Numeric sort	Sorts an array of 32-bit integers.
String sort	Sorts an array of strings of arbitrary length.
Bitfield	Executes a variety of bit manipulation functions.
Emulated floating-point	A small software floating-point package.
Fourier coefficients	A numerical analysis routine for calculating series approximations of waveforms.
Assignment algorithm	A well-known task allocation algorithm.
Huffman compression	A well-known text and graphics compression algorithm.
IDEA encryption	A relatively new block cipher algorithm.
Neural Net	A small but functional back-propagation network simulator.
LU Decomposition	A robust algorithm for solving linear equations.

Table 5.2: The tests included in the jBYTEmark benchmark.

Name of test	**Description**
FFT	Fast Fourier Transform
SOR	Jacobi Successive Over-relaxation
Monte Carlo	Monte Carlo integration
Sparse matmult	Sparse matrix multiply
LU	Dense LU matrix factorization

Table 5.3: The tests included in the SciMark benchmark.

> SciMark is a composite Java benchmark measuring the performance of numerical codes occurring in scientific and engineering applications. It consists of five computational kernels [...] chosen to provide an indication of how well the underlying JVM/JITs perform on applications utilizing these types of algorithms. The problems sizes are purposely chosen to be small in order to isolate the effects of memory hierarchy and focus on internal JVM/JIT and CPU issues.

As stated, the problem sizes used in SciMark's tests are small, which makes the benchmark relevant for systems with limited memory. Like jBYTEmark, the tests implement well-known algorithms, many of which are found in embedded devices, such as the Fast Fourier Transform, which is often used in signal processing equipment.

Table 5.3 gives an overview of the tests that are part of SciMark.

5.2 Performance

In this section, the performance improvements over the original KVM are quantified for each of the three benchmarks under study. The JIT has been configured with a compilation treshold of 1K[1] – that is, a basic block is only compiled after 1K interpretations – and a translation cache of 64KB, which is the standard size used by two highly related systems [ea05, Sha02]. Inhibiting compilation of certain basic blocks has been disabled; every basic block whose counter reaches the treshold is compiled. The original KVM has been configured with all significant optimizations enabled.

The benchmarks all utilize an absolute score system for their tests, where the higher the score, the better. In order to make it easier to compare results across benchmarks, the scores have been normalized relative to the scores achieved with the original KVM. The *normalized* score, or *speedup*[2], of the JIT relative to KVM can be expressed as

$S = \frac{Score(JIT)}{Score(KVM)}$

That is, $S = 1$ is identical score (no improvement), while $S = 2$ means the JIT achieved twice the score of KVM (100% improvement).

5.2.1 Overall Improvements

Each benchmark consists of several tests, which are scored individually. Figure 5.1 on the following page shows the *average* scores relative to KVM for all three benchmarks, which is the sum of scores for each benchmark divided by the number of tests in that benchmark. Also shown are the scores of the individual benchmark tests that achieved the highest and lowest scores relative to KVM, to give an indicator of the range of results within each benchmark.

Overall, the JIT system achieved a speedup of 4.0 for jBYTEmark, 4.7 for CaffeineMark and 7.7 for SciMark. No individual test achieved a score that was lower than that achieved with KVM; the lowest speedup achieved was 1.4. For one of SciMark's tests the score was more than an order of magnitude larger (10.2).

Perrier [Per] states that sophisticated JIT systems may achieve as much as 30 times the performance over an interpreter for CaffeineMark in particular. However, two circumstances make our results encouraging. First, little time has been invested in tuning the system so far; the fact that improvements are still relatively high indicates that there was a lot of inherent potential for performance improvement in the architecture from the start. Second, the initial prototype translates bytecode in a very naïve way, and there are certainly vast opportunities for improvement in just the compiler implementation without introducing excessive overhead. It would appear

[1]The choice of treshold was rather arbitrary, since little effort has been spent on tuning the system yet. Section 5.4 considers the effects of changing the treshold.

[2]Speedup is normally associated with running time, not scores, so we use the term loosely; however, we hope it is clear from the description what is meant.

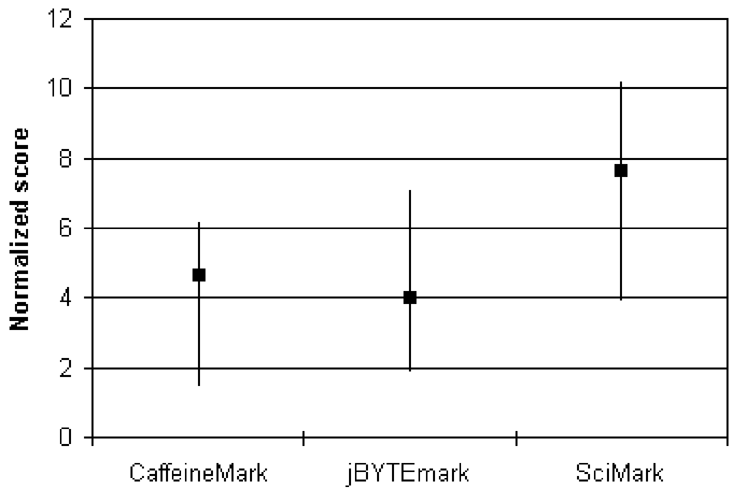


Figure 5.1: Average scores relative to KVM for all three benchmarks.

that the system is off to a good start, and so far it is just that – a start, with still a lot of room for implementations to grow.

In the sequel, the results from each benchmark will be discussed in further detail.

5.2.2 CaffeineMark

Figure 5.2 shows the scores relative to KVM for CaffeineMark.

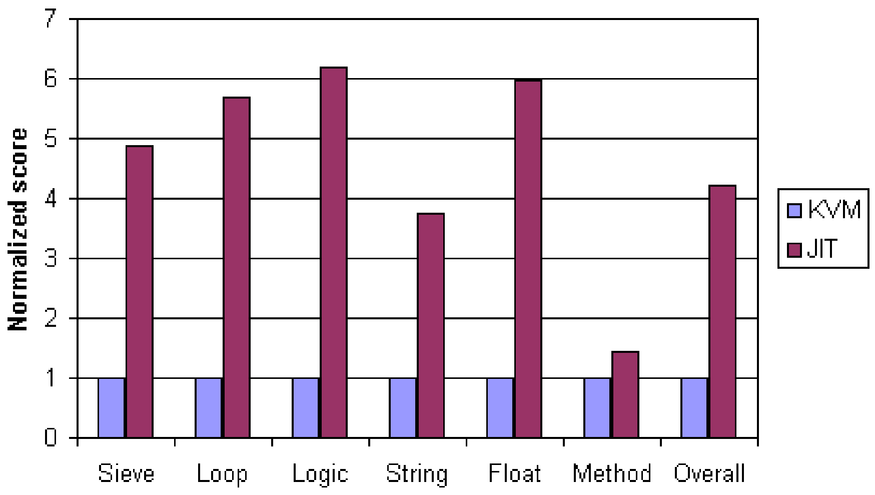


Figure 5.2: Scores relative to KVM for the CaffeineMark Embedded benchmark.

Generally, results are seen to be consistent, with four of six tests in the range 4.9 to 6.2, and only one test, `Method`, falling below three. For the `Method` test, however, which tests how fast the virtual machine can handle dynamic method invocation, JIT execution only gives a speedup of 1.4.

One possible explanation for the weak result of `Method` is the rather inefficient way in which native code handles the `invokevirtual` opcode. Resolving the target method of a virtual method invocation is a relatively complex procedure, so the compiler generates code that calls a C function to do the actual work. Thus, this lookup procedure is done every time the method is invoked. The interpreter, on the other hand, can be configured to use a caching technique to avoid the costly lookup as often as possible. It appears that this mechanism is fast enough to "cancel out" most of the overall performance gains resulting from native execution.

Another possible explanation is that the very frequent method invocations in that particular test cause a very large amount of context switches to take place, resulting in a lot of work being done that isn't really useful; more time is spent switching between native execution and interpretation than actually executing bytecode.

The `Float` tests performs so well because it consists of a loop with one long basic block that does all the work, meaning that a long sequence of bytecode is executed natively at a time, avoiding almost all interaction with the interpreter once the basic block has been compiled. Since the profiling scheme is inherently sensitive to loops, compilation happens early, so a lot of opportunity for native execution is captured. `Loop` benefits from the same behaviour, while `Logic`, which is basically a sequence of nested `if` statements, is so fast due to the chaining of basic blocks within a method (see section 4.4.5 on page 62), again avoiding frequent context switches back to the interpreter.

5.2.3 jBYTEmark

Figure 5.3 on the next page shows the scores relative to KVM for jBYTEmark.

All except one test, `Fourier`, achieve a score more than twice that of KVM, and even that test isn't far behind, with a speedup of 1.9. Half of the tests surpassed the four mark. Due to time constraints and the relative complexity of jBYTEmark compared to CaffeineMark, a further investigation of the reasons for variations in the results could not be pursued.

5.2.4 SciMark

Figure 5.4 on page 75 shows the scores relative to KVM for SciMark.

SciMark achieves the best results of the three benchmarks considered, with four of five tests achieving a speedup of seven or more. Even the test with the lowest relative score, `Monte Carlo`, achieves a score that is four times that of KVM.

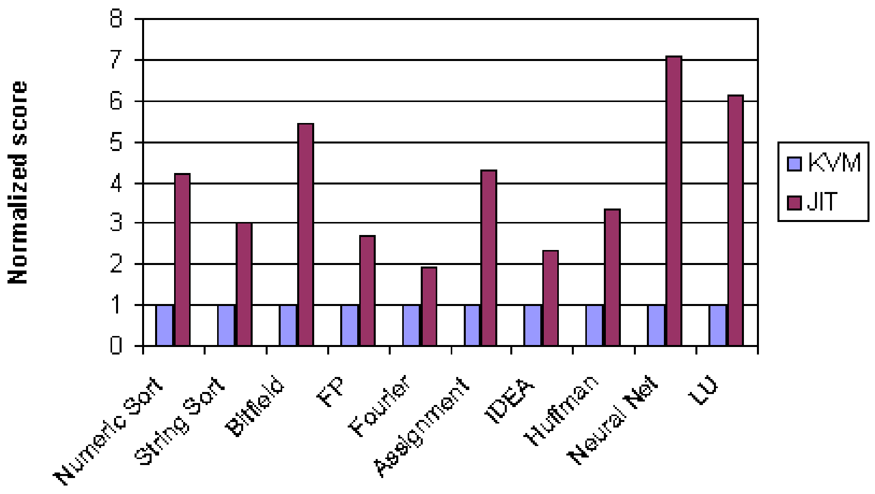


Figure 5.3: Scores relative to KVM for the jBYTEmark benchmark.

Benchmark	Code expansion
CaffeineMark	12.3
jBYTEmark	12.4
SciMark	11.8
Average	12.2

Table 5.4: Code expansion for the set of benchmarks.

5.3 Code Expansion

Code expansion measures the relative size of machine code compared to the size of the original bytecode. Table 5.4 shows the code expansion factors for the benchmarks that have been studied.

The code expansion factors are relatively high. This was expected, however. In the initial prototype, fast and simple code generation came at the expense of larger machine code. The original Java bytecode is a very compact representation due to its stack-based computation model, where operands are implicit; the average bytecode instruction length has been shown to be 1.8 bytes [RVJ$^+$01]. The target processor, on the other hand, is a RISC architecture with fixed-length instructions (each four bytes long), and uses a register-based computation model with explicit operands. The initial prototype compiler's naïve mapping of bytecode to RISC, without utilizing the register file in an efficient way, is the main reason for the large size of compiled code. The spatial effects of naïve code generation would not have been so severe if the target were a CISC architecture with variable-length instructions, such as Intel x86, where many one- and two-byte instructions are available and stack-based computation is supported in hardware.

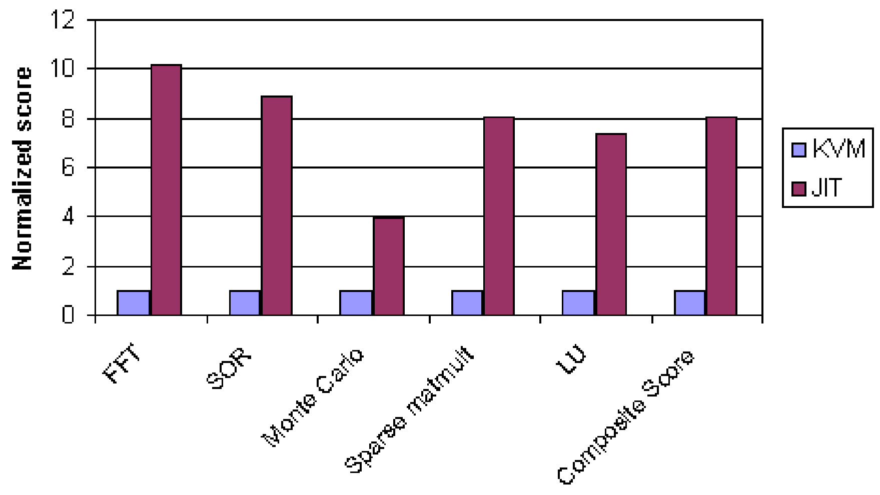


Figure 5.4: Scores relative to KVM for the SciMark benchmark.

Another reason for the increase in code size was due to the unfortunate memory layout of the Linux process running the virtual machine. Many of the translations for complex bytecodes rely on calling C functions to do work that would be too involved to implement in machine code (e.g. virtual machine services for memory allocation or synchronization); also, this practice enables the JIT to leverage much of the existing KVM framework, sharing the code with the interpreter. Whereas the translation cache resides in the process's data segment, the C functions reside in the code segment. Unfortunately, these were located so far apart in virtual memory that the distance could not be encoded in a PowerPC relative branch instruction. Thus, the native code must perform an absolute branch, which requires four instructions rather than one (which not only increases code size, but affects performance as well).

By introducing some simple register allocation techniques in the compiler, code expansion due to translation of stack-based computation could be more than halved; this is one of the top priorities for future work, since bringing down code expansion has synergetic effects for systems with little memory: Less memory is needed to hold translations, memory traffic is reduced, code locality is increased and power consumption is reduced (due to both fewer memory accesses and faster execution).

5.4 Effects of Selective Compilation

A predefined treshold was used to decide how many times a basic block should be interpreted before compilation of the block should be initiated. In section 5.2, the treshold used was 1K. Figure 5.5 on the following page shows the results of changing the compilation treshold for CaffeineMark.

The most noticeable result is due to the `Logic` test, whose performance drops drastically when

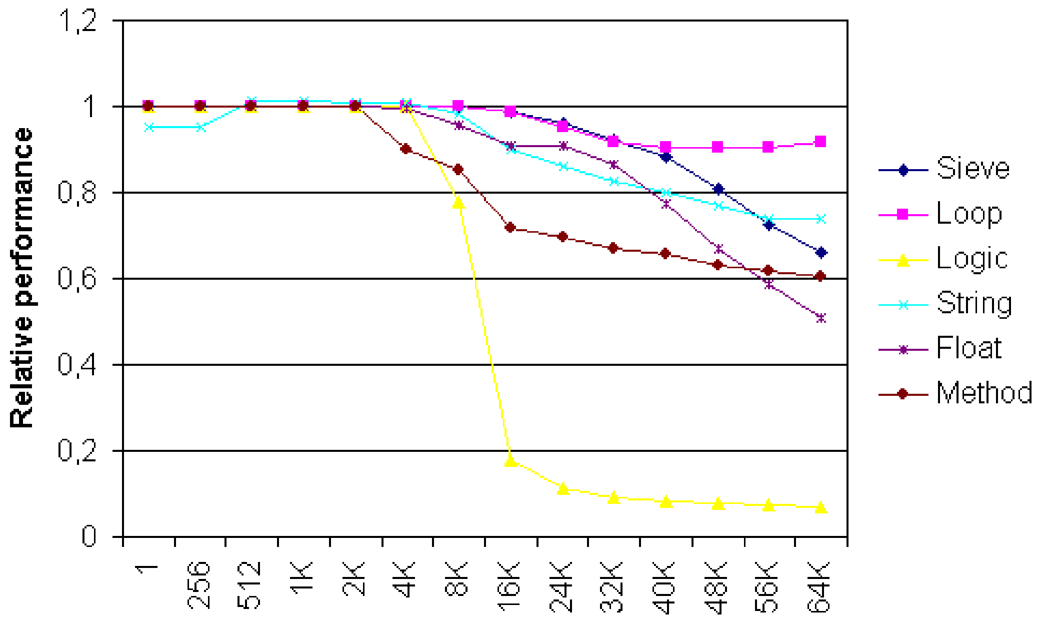


Figure 5.5: Effects of changing the compilation treshold for CaffeineMark.

the treshold exceeds 8K. This is most likely due to the treshold becoming so large that no code is compiled at all, losing all ability for speedup. `String` is the only test whose performance actually improved when the treshold was increased, but only for a rather small region (512-2048).

On the whole, for the applications tested, the performance usually degraded as the treshold was increased, indicating that eager compilation is preferred. This might be because the profiling scheme itself was not sophisticated enough. However, it might also be because the applications were all small, meaning that the selective compilation logic simply didn't have much code to choose from in the first place, lessening the impact of its decisions. According to Radhakrishnan et al [RVJ$^+$01], selective translation using good heuristics can improve performance, but the saving is only 10-15 percent at best. The attainable savings appear to be even smaller when the application being profiled consists of only tens of classes, as argued by Shaylor [Sha02]. Our results back up this claim.

When compilation is expensive, selective compilation can help reduce startup latencies and provide better interactive response [Sha02]. However, our compiler only compiles one basic block at a time, and it does so very quickly. Thus, invoking the compiler frequently is not as big a source of latency as it would be if the compilation process were more time-consuming.

The issue of compilation treshold should be revisited once real-world applications are selected for testing.

5.5 Effects of Limited Translation Cache

The results presented in section 5.2 were obtained using a 64KB translation cache (machine code buffer). Translations are added to the cache as basic blocks are compiled one after the other. When the cache becomes full, it is simply cleared to make room for new translations. With a large cache size, this event is relatively rare, avoiding much impact on performance due to "hot" basic blocks having to be recompiled. In order to investigate the impact on performance when the cache is cleared more frequently, the measurements of section 5.2 were repeated with gradually smaller translation caches.

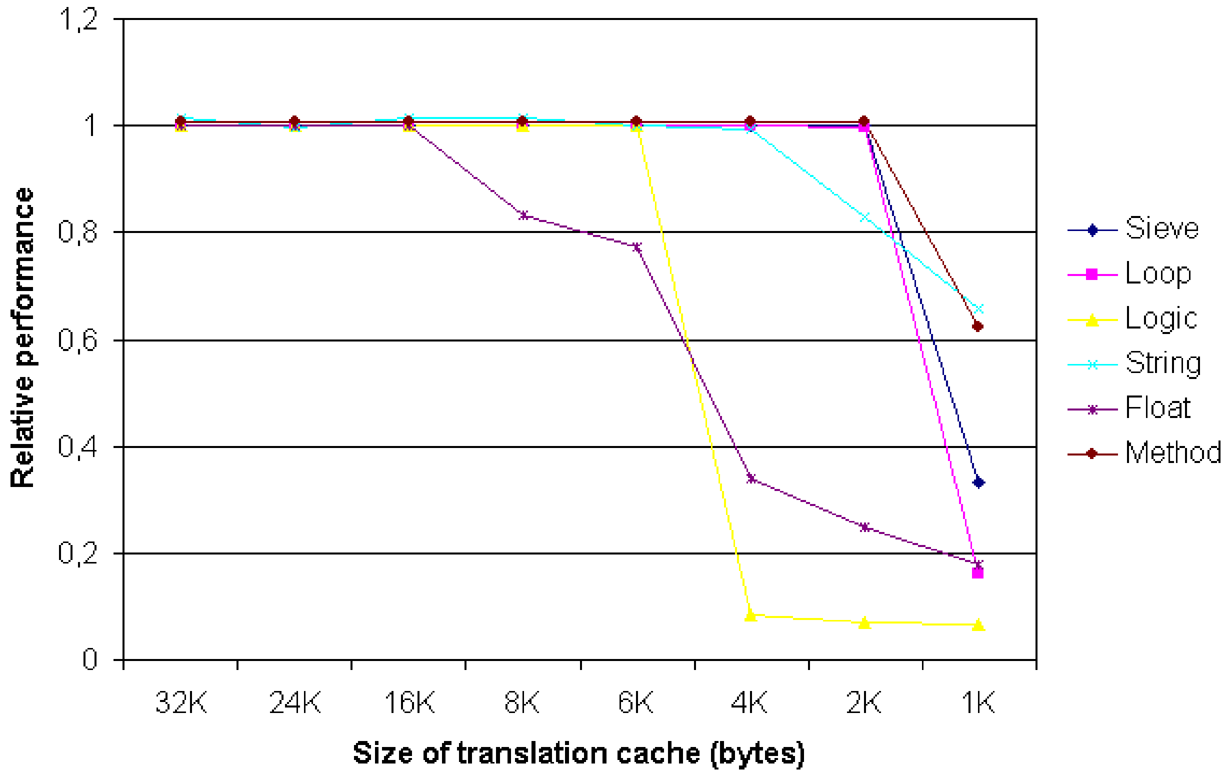


Figure 5.6: Performance of CaffeineMark (relative to using a 64KB translation cache) as the translation cache is made smaller.

Figure 5.6 shows the resulting slowdown for CaffeineMark. As can be seen, the size of the translation cache can be reduced from 64KB to 16KB without much impact on performance. At that point, the difference in working set size (and code expansion, possibly) of each test begins to take effect. While `Float` enjoys a rather controlled decline, `Logic` experiences a very sharp drop in performance when the translation cache is reduced from 6KB to 4KB. 4KB is also seen to be the treshold after which `String`'s performance starts to decline. Finally, at 1KB, the remaining three tests – `Sieve`, `Loop` and `Method` – are affected as well.

It is interesting to note that the performance of the `String` test actually improves (albeit marginally – almost indiscernible in figure 5.6) when reducing the size of the translation cache from 64KB to 32KB. This indicates that there are some parts of the application that are compiled that ideally shouldn't have been; that is, running these parts natively gives worse performance than interpretation does. When the translation cache is made smaller, the inefficient translations are occasionally disposed of, causing the interpreter to execute the relevant bytecode instead (at least until the next time they are compiled), with better performance. Similar behaviour is indicated in the results from Shaylor's system [Sha02], but in a more marked way.

The ability to use a translation cache as small as 16KB without affecting performance must of course be seen in relation to CaffeineMark itself. CaffeineMark's implementation is small; the combined size of the bytecode for CaffeineMark's classes is only a couple of kilobytes. Thus, when code expansion is taken into account, 16KB seems fairly reasonable.

Figure 5.7 shows the corresponding slowdown for jBYTEmark. jBYTEmark contains more elaborate tests than CaffeineMark, resulting in more bytecode to translate. This can explain why, already at 32KB, some tests begin to slow down, and at 8KB, half the tests experience performance degradation of 20% or more.

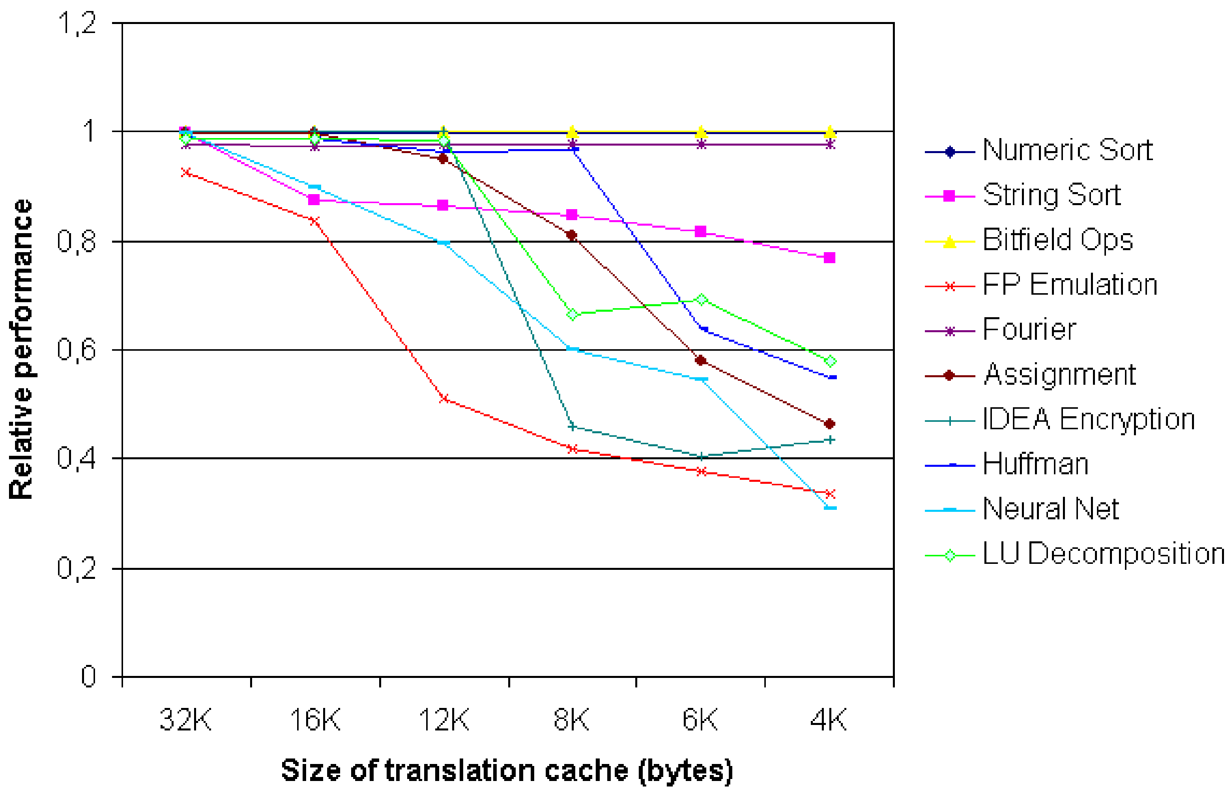


Figure 5.7: Performance of jBYTEmark (relative to using a 64KB translation cache) as the translation cache is made smaller.

Measurements were also made as the size of the translation cache was increased beyond 64KB, but this did not make any difference for any of the tested applications. In any case, increasing the size further would defeat the overall purpose of the system, which is to achieve performance improvement while incurring only a *small* footprint.

5.6 Memory Footprint

The static footprint of the JIT subsystem is 30KB of PowerPC instructions; the original KVM is 200KB, which means an increase of 15%.

The additional dynamic footprint is due to two things: The tables of basic block descriptors, and the translation cache. The basic block descriptors are allocated lazily, and have a very compact representation; the total contribution due to them has been 1 or 2KB of dynamic memory at most for the applications tested. The choices made regarding table and descriptor representations in section 4.2 therefore proved adequate. The translation cache is allocated at JVM startup and has a default size of 64KB (which can be changed by end users through a command line switch), although in the practical experiments conducted so far 32KB was found to be sufficient.

Thus, the combined memory footprint due to the acceleration technique does not exceed 100KB. This keeps the system well within the limits of the CLDC specification [Mic03], meaning that the resulting system can be used in environments similar to those targeted by the original KVM, a major goal from the outset.

5.7 Comparison

E-Bunny [ea05] was described in section 2.5.3 on page 34; it too uses KVM as its infrastructure. Figure 5.8 on the following page compares our system's speedup for the CaffeineMark benchmark to the corresponding speedup achieved by E-Bunny. Although the figures aren't directly comparable – for one thing, E-Bunny targets the Intel x86 processor while our prototype targets the PowerPC – they give an indicator of where some of the future improvement potential lies.

`Sieve` and `Loop` achieved significantly higher speedup on this system than on E-Bunny, while the remaining three tests – `Logic`, `String` and `Method` – achieved higher speedup on E-Bunny. Overall, E-Bunny's scores are 6.6% higher than those of our system, largely due to E-Bunny's much larger relative score on the `Method` test. The results are definitely encouraging, however; even though our system is in its infancy, it appears to be competitive with other solutions.

An explanation of the high score for the `Loop` test on our system relative to E-Bunny might be that E-Bunny uses method-level profiling counters; this makes it less sensitive to methods that execute loops, causing the method containing the loop(s) to be interpreted many times before it is eventually compiled. Our system, on the other hand, uses basic block-level profiling, which

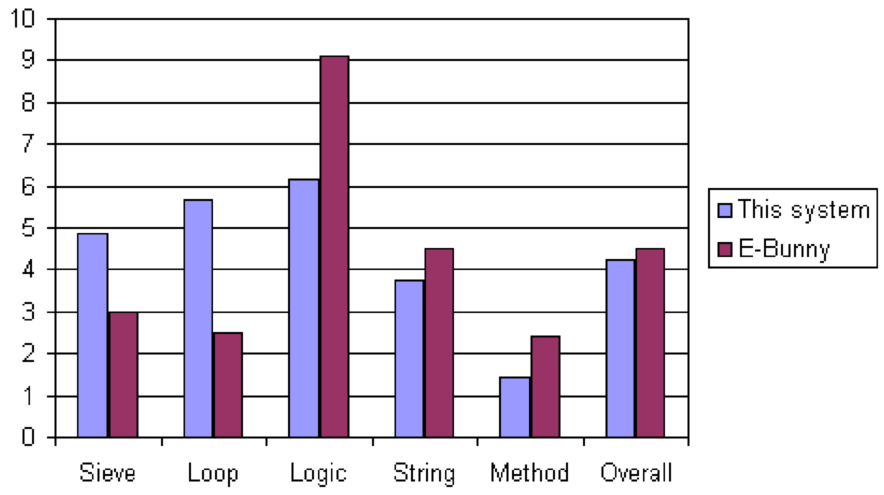


Figure 5.8: Relative scores of this system versus E-Bunny for the CaffeineMark benchmark.

quickly catches repeated execution of loops, and results in the switch to a more efficient native execution occuring much sooner.

Results of the `Logic` test are particularly interesting. Relative to the other tests, `Logic` got the best result on our system when compared to the original KVM, indicating that logic instructions are handled efficiently. However, E-Bunny achieved a significantly higher relative score (almost 50% above ours), indicating that there is still much potential in this area.

The `Method` test showed relatively little improvement on our system. It was also the test with smallest gains on E-Bunny; still, it achieved more than twice the score of KVM. One explanation for E-Bunny's faster handling of method invocation is that E-Bunny compiles all the methods that the top-level compiled method might invoke. This means that there is no context switch from native execution to interpretation when a compiled method invokes another method, and, when a method returns, there is no context switch unless the method returned to is not compiled. Our system, on the other hand, currently has to perform a context switch both on invocation and return, leading to more work being done at method boundaries, possibly inhibiting performance gains.

E-Bunny increases the static memory footprint by 45% over the original KVM, while the corresponding increase due to our solution is 15%. E-Bunny needs 74KB of dynamic memory for managing compilation and translation, while ours has a user-adjustable footprint; 32KB has been shown to be sufficient for the applications tested thus far. When using a 64KB translation cache, the ratio of speedup to footprint is 1.4% higher for our system than the corresponding ratio for E-Bunny.

All in all, the comparison shows that the implementation of control flow and method invocation is something that can favourably be addressed in our system; more than likely, addressing these two fundamental classes of JVM instructions will have synergetic effects that can lead to boosts in the score of other tests as well (such as the `String` test).

Chapter 6

Conclusions and Future Work

In this text, a performance-enhancing bytecode execution scheme suitable for JVMs that operate in resource-constrained environments was presented. The performance gain was to be achieved by introducing JIT compilation into the bytecode execution engine of the JVM. The initial prototype, despite the fact that the compiler uses extremely simplistic compilation techniques, and that little time has yet been invested in tuning the critical system parameters, shows consistently encouraging results for all benchmarks that have been used to evaluate the system, achieving peak speedup of 10.2 over the original bytecode interpretation scheme, and an average speedup of 5.5. This increase in performance comes at a cost of 15% (30KB) increase in the static memory footprint of the virtual machine, and an adjustable, highly predictable increase in additional dynamic memory footprint (32KB was sufficient for the applications tested).

Key factors to the success of the system were:

- Leveraging an existing virtual machine implementation targeting resource-constrained environments. This significantly reduced implementation efforts and provided an infrastructure where memory concerns were already a key consideration, affording further space-efficient extensions;
- Compiling only small sections (basic blocks) of a method at a time, requiring little dynamic memory and giving fast compilation times;
- Designing small data structures that afford fast manipulation and small space requirements even when on-line profiling happens at a fine level of granularity (the basic block);
- Using a single Java execution stack, so that a minimum of processing must be done when switching between interpretation and native execution, allowing this switch to be done quickly; this choice also meant that additional complexity associated with managing two separate stacks did not have to be introduced in the garbage collector and thread subsystems; and

- Chaining the execution of several successive translations in native mode, so that the switch between interpretation and native execution need not be done as often.

A novel part of the system presented was to use the basic block as the level of granularity, as opposed to the method level, which would be a more conventional choice. The results so far indicate that this is an approach that can indeed be very effective in resource-constrained environments; the theories put forth in section 3.2 regarding low, predictable memory footprint and fast compilation times associated with basic blocks have largely been confirmed by empirical data. On-line profiling did not result in significant overhead either, even at this fine level of granularity, and results indicated that the ability to automatically detect loops made the compilation system more responsive to hot regions. Fine-grained mixed-mode execution performed well despite resulting in far more frequent context switches than method-level mixed-mode execution would. However, one issue that still needs attention is code expansion, which was found to be larger than for method-level compilers. This can largely be attributed to the simplistic compiler of the prototype, but whether or not code expansion can be reduced to a level comparable to method-level compilation – and whether such a reduction is even critical for further success – is still an open question; there are smaller costs associated with compiling and discarding code at the basic block level compared to the method level. Implementing a more sophisticated compiler will be the first step towards resolving this issue.

Overall, the results achieved so far have been very encouraging. Even so, there is still much work that can be done. The major possible directions are outlined in the next section.

6.1 Future Work

There are two major avenues that may be pursued in the immediate future:

- *Performance analysis*: Examining the behaviour that underlies the initial prototype's performance in more detail, in order to understand what the bottlenecks are and direct attention to those parts.

- *Compilation*: Improving the quality of code produced by the compiler (without introducing noticeable overhead).

Interesting tasks within these categories are suggested and outlined in the following sections.

6.1.1 Performance Analysis

Initial performance results show an overall performance improvement over the original KVM system. However, more detailed analyses that examine the relative time distribution between interpretation, profiling, compilation, mixed-mode context switching and native execution, could help focus efforts towards the parts of the system that are likely to give the most performance gains for the least amount of work and additional complexity. A first step towards this understanding might be to run the benchmarks under an instrumented interpreter that records the relative frequencies of each bytecode instruction, or category of instructions; this way, perhaps interesting correlations with the results presented in section 5.2 can be discovered.

There are still aspects of the system that can be tuned. For instance, the effects of compilation treshold and inhibiting compilation of small basic blocks should be studied in further detail.

Lastly, testing should be performed using applications that are more representative of real-world applications. A natural next step would be to use the Richards and DeltaBlue benchmarks [Wol]; Shaylor [Sha02] used DeltaBlue to benchmark KJIT, so this would allow a comparison to his system.

6.1.2 Compilation

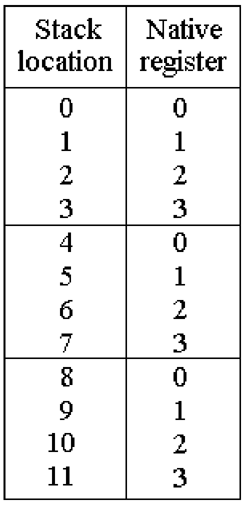
The compiler that is part of the prototype implementation generates code in the simplest possible way; as such, it does not utilize the native processor capabilities (such as the register file and instruction set) in an efficient manner. The compiler also generates many unnecessary load and store instructions because of the naïve way in which JVM stack operations are translated. This situation can be improved in a number of ways, without introducing much complexity into the compiler; improving code quality would lead to faster, more compact code, reducing translation cache memory requirements and improving code locality. Two of the possible improvements to code generation are outlined below. These were in fact intended for the initial prototype, but had to be sacrificed due to time constraints; therefore, they are explained in some detail in order reflect the amount of planning that has been done.

Shadowing the Operand Stack in Registers

The intent of this technique is to keep operand stack words in native registers as long as possible, rather than writing them back to memory after every operation; in other words, it is a lazy write-back strategy. The idea is to dedicate a set of N consecutive native registers to stack shadowing, and then map operand stack location I to native register $ImodN$. With $N = 4$ registers, the mappings will be as shown in figure 6.1 on the next page.

An array of register descriptors must be maintained during code generation to keep track of the actual state of a register – what stack location's value it contains, if any. The value only has to be written back in case of a conflict with another stack location that needs to be brought into the same register. In addition, all registers must be flushed to memory prior to the switch back to interpreter mode.

This technique increases the utilization of the processor register file, and eliminates the need for a subsequent peephole optimizer pass, since quality code is generated straight away. However, it requires a bit more effort to implement.



Stack location	Native register
0	0
1	1
2	2
3	3
4	0
5	1
6	2
7	3
8	0
9	1
10	2
11	3

Figure 6.1: Mapping of operand stack locations to native registers. In this example, four native registers are dedicated to stack shadowing, so every fourth stack location shares native register.

Since register selection is done in a single modulo operation, taking constant time, it is expected to be a very fast scheme. The only memory overhead due to the technique is the static N words of memory needed to hold the array of register-to-stack-location mappings.

Caching and Inlining Techniques for Complex Opcodes

The basic idea of this technique is to cache the results of opcodes that perform a relatively costly dynamic lookup procedure, such as `invokevirtual` and `checkcast`. By having the compiler generate code that inlines the cache validity check, the costly lookup procedure only needs to be performed when the contents of the cache are found to be invalid. Algorithm 4 shows the general procedure. The KVM interpreter already uses such a technique in the execution of "quick" versions of some opcodes, and by introducing similar behaviour in the compiler, performance might improve noticeably.

Algorithm 4 An algorithm that tries to avoid subsequent calls to a costly lookup procedure by caching the latest result.

```
if dynamic class of object matches cache then
    result is contents of cache
else
    result is result of regular (slow) lookup procedure {e.g. lookupVirtualMethod}
    update cache {e.g. store the tuple (dynamic class, method)}
end if
... {continue with normal processing using result}
```

Similarly, in the handling of `monitorenter` and `monitorexit`, the compiler could inline the simple cases of acquiring and releasing a monitor, only calling the general handlers when it is detected that more complex processing must be performed.

Bibliography

[AFG$^+$00] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.

[AFG$^+$04] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. IBM Research Report RC23143 (W0312-097), IBM Research Division, Thomas J. Watson Research Center, May 2004.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[BFG02] David F. Bacon, Stephen J. Fink, and David Grove. Space- and Time-Efficient Implementation of the Java Object Model. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 111–132, 2002.

[CD03] Lynn Comp and Tim Dobbing. Runtime Abstractions in the Wireless and Handheld Space. *Intel Technology Journal*, 7(1):68–76, February 2003.

[CKV$^+$02] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Wolf. Energy Savings Through Compression in Embedded Java Environments. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 163–168, 2002.

[CO02] Michael Chen and Kunle Olukotun. Targeting Dynamic Compilation for Embedded Environments. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium (JVM '02)*, pages 151–164, August 2002.

[Cora] Aplix Corporation. JBlend homepage. http://www.aplix.co.jp/en/jblend/.

[Corb] Intel Corporation. Open Runtime Platform (ORP) homepage. http://orp.sourceforge.net/.

[Corc] Pendragon Software Corporation. CaffeineMark 3.0 homepage. http://www.benchmarkhq.ru/cm30/.

[CSKO02] A. Corsaro, D. Schmidt, R. Klefstad, and C. O'Ryan. Virtual Component: A Design Pattern for Memory-Constrained Embedded Applications. In *Submitted to the 9 Annual Conference on the Pattern Languages of Programs, (Monticello, Illinois)*, September 2002.

[D.M98] D.Mulchandani. Java for Embedded Systems. *IEEE Internet Computing*, 2(3):30–39, May 1998.

[DSCS03] Paul Drews, Doug Sommer, Roger Chandler, and Terry Smith. Managed Runtime Environments for Next-Generation Mobile Devices. *Intel Technology Journal*, 7(1):77–81, February 2003.

[ea05] Mourad Debbabi et al. E-Bunny: A Dynamic Compiler for Embedded Java Virtual Machines. *Journal of Object Technology*, 4(1):81–106, January 2005.

[EPP] EPPC. JavaGrande benchmark homepage. http://www.epcc.ed.ac.uk/javagrande/.

[FHH$^+$02] Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[GH03] E. Gagnon and L. Hendren. Effective inline-threaded interpretation of java bytecode using preparation sequences. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 170–184, April 2003.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[IBM] IBM. Jikes RVM (Research Virtual Machine) homepage. http://jikesrvm.sourceforge.net/.

[IBM96] IBM. *The PowerPC Compiler Writer's Guide*, January 1996.

[IBM03] IBM. *PowerPC Microprocessor Family: Programming Environments Manual for 64 and 32-Bit Microprocessors*, June 2003. Version 2.0.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[LYK$^+$00] SeungIl Lee, Byung-Sun Yang, Suhyun Kim, Seongbae Park, Soo-Mook Moon, Kemal Ebcioglu, and Erik R. Altman. Efficient Java exception handling in just-in-time compilation. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 1–8, June 2000.

[Mag] BYTE Magazine. jBYTEmark homepage. http://www.byte.com/bmark/bmark.htm.

[Mica] Sun Microsystems. Connected, Limited Device Configuration (CLDC) homepage. http://java.sun.com/products/cldc/.

[Micb] Sun Microsystems. Java Card Technology homepage. http://java.sun.com/products/javacard/.

[Mic00] Sun Microsystems. J2ME Building Blocks for Mobile Devices, May 2000. Whitepaper.

[Mic03] Sun Microsystems. *Connected, Limited Device Configuration*, March 2003. Specification Version 1.1.

[Mic05] Sun Microsystems. CLDC HotSpot Implementation Virtual Machine, February 2005. Whitepaper.

[MK00] Geetha Manjunath and Venkatesh Krishnan. A Small Hybrid JIT for Embedded Systems. *ACM SIGPLAN Notices*, 35(4):44–50, April 2000.

[MO98] Harlan McGhan and Mike O'Connor. PicoJava: A Direct Execution Engine For Java Bytecode. *IEEE Computer*, 31(10):22–30, October 1998.

[NW01] James Noble and Charles Weir. *Small Memory Software: Patterns for Systems with Limited Memory*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[OLPW02] Diarmuid O'Donoghue, Aine Leddy, James Power, and John Waldron. Bigram Analysis of Java Bytecode Sequences. In *Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002*, pages 187–192, 2002.

[OT97] J. Michael O'Connor and Marc Tremblay. picoJava-I: The Java Virtual Machine in Hardware. *IEEE Micro*, 17(2):45–53, March 1997.

[PC97] Michael P. Plezbert and Ron K. Cytron. Does "Just in Time" = "Better Late than Never"? In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 120–131, 1997.

[Per] Vincent Perrier. Getting a Handle on Java Performance. http://www.techonline.com/community/tech_topic/java/20447.

[PM] Roldan Pozo and Bruce Miller. SciMark 2.0 homepage. http://math.nist.gov/scimark2/.

[PR98] Ian Piumarta and Fabian Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, June 1998.

[Raj02] Anand S. Rajan. A Study of Cache Performance in Java Virtual Machines. Master's thesis, The University of Texas at Austin, 2002.

[RVJ$^+$01] R. Radhakrishnan, N. Vijaykrishnan, L.K. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java runtime systems: Characterization and architectural implications. *IEEE Transactions on Computers*, 50(2):131–146, February 2001.

[SBCK03] Ulrik Pagh Schultz, Kim Burgaard, Flemming Gram Christensen, and Jørgen Lindskov Knudsen. Compiling Java for Low-End Embedded Systems. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 42–50, 2003.

[Sch03] Jonathan L. Schilling. The Simplest Heuristics May Be the Best in Java JIT Compilers. *ACM SIGPLAN Notices*, 38(2):36–46, February 2003.

[Sha02] Nik Shaylor. A Just-In-Time Compiler for Memory Constrained Low-Power Devices. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium (JVM '02)*, pages 119–126, August 2002.

[SMB04] Narendran Sachindran, J. Eliot B. Moss, and Emery D. Berger. MC2: high-performance garbage collection for memory-constrained environments. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 81–98, 2004.

[(SP] The Standard Performance Evaluation Corporation (SPEC). SPEC JVM98 benchmark homepage. http://www.spec.org/jvm98/.

[Sun03] Sun Microsystems. *KVM Porting Guide*, March 2003. CLDC, Version 1.1.

[SYK$^+$01] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-in-Time Compiler. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 180–194, October 2001.

[TBS99] Antero Taivalsaari, Bill Bush, and Doug Simon. The Spotless System: Implementing a Java System for the Palm Connected Organizer, February 1999.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[VKK$^+$01] N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin. Energy behavior of Java applications from the memory perspective. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, April 2001.

[VMK02] K. S. Venugopal, Geetha Manjunath, and Venkatesh Krishnan. sEc: A Portable Interpreter Optimizing Technique for Embedded Java Virtual Machine. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium (JVM '02)*, pages 127–138, August 2002.

[Wol] Mario Wolczko. Benchmarking Java with Richards and DeltaBlue. http://research.sun.com/people/mario/java_benchmarking/.

Appendix A

Code

This appendix contains listings for the source code that has been written for this project.

A.1 Source Files

A.1.1 execute.c

Modification of KVM interpreter loop. For brevity, only the code that implements algorithm 1 on page 45 is included here.

```
reschedulePoint:
    RESCHEDULE

#if ENABLEJIT
   /*
    * The next piece of code may be considered the heart of the JIT.
    * - The basic block about to be executed is looked up
    * - If the block is compiled, it (and possibly one or more
    *   successors) is executed natively
    * - Otherwise, we increase a "hit" counter and determine if
    *   it's time to compile the block; if so, it is compiled now; and
    * - The basic block is interpreted
    */
    {
       BASICBLOCK bb;
       /* Check if we have profiling/native info for this method. */
       if (fp->thisMethod->u.java.basicBlocks != NIL) {
           /* Get the basic block about to be executed. */
           bb = basicBlockForIp(fp->thisMethod, ip);
```

```
            if (bb->counter == 0) {
                /* BB has been compiled. */
#if INCLUDEDEBUGCODE
                if (tracenativeexec) {
                    nativeExecutionTrace(fp->thisMethod, bb);
                }
#endif    /* INCLUDEDEBUGCODE */
#if ENABLEPROFILING
                NativeExecCounter++;
#endif    /* ENABLEPROFILING */
    /*
     * Initiate native execution of basic block.
     * The ENTER_JIT_CODE macro contains target processor-specific
     * assembly instructions that copies VM registers from GlobalState
     * struct into native registers as appropriate, then jumps to
     * bb->nativeCode.
     */
                ENTER_JIT_CODE(bb);
            }
            else {
                /* Decrease counter and compile if we hit zero. */
                bb->counter--;
                if (bb->counter == 0) {
#if ENABLEINHIBITION
                    if (INHIBIT_COMPILER(bb)) {
                        /* Shouldn't compile this basic block. */
                        bb->counter = 65535;
#if ENABLEPROFILING
                        InhibitCompileCounter++;
#endif    /* ENABLEPROFILING */
                    }
                    else
#endif    /* ENABLEINHIBITION */
                    {
                        /* Go ahead and compile. */
                        compileBasicBlock(fp->thisMethod, bb,
                            &&resumeInterpreterFromNative);
                    }
                }
            }
        }
    }
#endif /* ENABLEJIT */

    goto next0; /* Interpret */
```

```
#if ENABLEJIT
    /*
     * This is where we get back to after executing one or more basic
     * blocks natively.
     * The LEAVE_JIT_CODE macro contains target processor-specific
     * assembly instructions that write any native register-cached
     * VM registers back to the interpreter’s GlobalState struct.
     */
resumeInterpreterFromNative:
    LEAVE_JIT_CODE
    goto reschedulePoint;
#endif /* ENABLEJIT */
```

A.1.2 compiler.c

Compiler implementation. Since the translation cache implementation is very simple its implementation is also contained in this file.

```
/*=================================================================
 * SYSTEM:    KVM compiler
 * SUBSYSTEM: Basic block compiler
 * FILE:      compiler.c
 * OVERVIEW:  Functions for compiling individual basic blocks.
 *
 * AUTHOR:    Kent M. Hansen
 * NOTE:
 *===============================================================*/

/*=================================================================
 * Include files
 *===============================================================*/

#include <global.h>
#include <math.h>
#include <ppcinstr.h>

#if ENABLEJIT

/*=================================================================
 * Definitions and declarations
 *===============================================================*/

/* Requested translation cache size
   when starting the VM from command line */
long RequestedTranslationCacheSize;

/* The native registers used to hold VM registers during native
   code execution. */
#define GS_REG 26    /* &GlobalState */
#define IP_REG 27    /* ip_global */
#define SP_REG 28    /* sp_global */
#define LP_REG 29    /* lp_global */
#define FP_REG 30    /* fp_global */

#define methodCode(meth) (meth)->u.java.code

/* Takes an offset in cells, calculates byte offset in object */
#define FIELD_BYTE_OFFSET(_off) (((_off)+2)*4)
```

```
/* GNU C Library-specific */
long long __muldi3(long long, long long);
long long __divdi3(long long, long long);
long long __moddi3(long long, long long);
long long __ashldi3(long long, int);
long long __ashrdi3(long long, int);
long long __lshrdi3(long long, int);
int __cmpdi2(long long, long long);
long long __fixsfdi(float);
long long __fixdfdi(float);
float __floatdisf(long long);
double __floatdidf(long long);

/* The buffer where generated native code is stored. */
static int *tcache;

/* Current position in tcache.
   It is incremented each time a word is added to the buffer. */
static int tcachePos;

/* Relative stack pointer. Instead of incrementing/decrementing
   the stack pointer explicitly at every push/pop, we encode the
   displacement relative to the value of SP_REG in the native
   load/store instructions. */
static long relSp;

/* IP of instruction being compiled */
static BYTE *thisIp;

/* IP last written to mem */
static BYTE *savedIp;

/* At the beginning of the tcache, we place common code that
   handles exceptions, similar to how it's done in the
   interpreter. This way, we just have to generate a
   conditional branch to the proper handler in the actual
   bytecode translation.
   Below follow the addresses of the exception handlers,
   relative to the start of the tcache. */
static void *handleNullPointerException;
static void *handleArrayIndexOutOfBoundsException;
static void *handleClassCastException;
static void *handleArithmeticException;
```

```
/* The address that we goto to return to the interpreter. */
static void *interpResume;

/*==================================================================
 * Static methods (only used in this file)
 *=================================================================*/

void *tcacheAddress();
int tcacheDistance(void *);
void tcacheAppend(int);
void backpatchConditionalBranch(int);

static void
recordBackpatchLocation(int *pos)
{
    *pos = tcachePos;
}

/*==================================================================
 * FUNCTION:      peepholeOptimize()
 * TYPE:          Peephole optimizer
 * OVERVIEW:      Simple peephole optimizer that gets rid of
 *                redundant load after store.
 *
 * INTERFACE:
 *   parameters:  start First instruction
 *                last After last instruction
 *   returns:     Number of instructions disposed of
 *=================================================================*/

static int
peepholeOptimize(cell *start, cell *end)
{
    int i, j, k, l;
    int count = end - start - 1;
    for (i=0, j=0; i<count; i++) {
        k = start[i];
        start[j++] = k;
        l = start[i+1];
        /* If this instruction is STW and next instruction is LWZ */
        if ( ((k & 0xFC000000) == 0x90000000) &&
             ((l & 0xFC000000) == 0x80000000) ) {
            /* If arguments match */
            if ( (k & 0x03FFFFFF) == (l & 0x03FFFFFF) ) {
                i++; /* Skip redundant LWZ */
            }
        }
        /* If this instruction is STFS and next instruction is LFS */
        else if ( ((k & 0xFC000000) == 0xD0000000) &&
                  ((l & 0xFC000000) == 0xC0000000) ) {
            if ( (k & 0x03FFFFFF) == (l & 0x03FFFFFF) ) {
                i++; /* Skip redundant LFS */
            }
        }
        /* If this instruction is STFD and next instruction is LFD */
        else if ( ((k & 0xFC000000) == 0xD8000000) &&
                  ((l & 0xFC000000) == 0xC8000000) ) {
            if ( (k & 0x03FFFFFF) == (l & 0x03FFFFFF) ) {
                i++; /* Skip redundant LFD */
            }
        }
    }
    start[j++] = start[count];
    return (end - &start[j]);
}

/*=================================================================
 * FUNCTION:      isEmptyMethod()
 * TYPE:          Helper function
 * OVERVIEW:      Tests if a method is empty. A method is "empty"
 *                if it consists of the single bytecode RETURN;
 *                that is, the method doesn't do any real work
 *                whatsoever.
 *
 * INTERFACE:
 *   parameters:  method: Method to test for emptyness
 *   returns:     TRUE if empty, FALSE if not
 *===============================================================*/

static bool_t
isEmptyMethod(METHOD method)
{
    if (method->accessFlags & (ACC_NATIVE | ACC_ABSTRACT) ) {
        return FALSE;
    }
    if ( (method->u.java.codeLength == 1) &&
         (method->u.java.code[0] == RETURN) ) {
        return TRUE;
    }
    return FALSE;
```

```
}

/*==================================================================
 * FUNCTION:      isWrapperMethod()
 * TYPE:          Helper function
 * OVERVIEW:      Tests if a method is a wrapper. A "wrapper
 *                method" is a method that does nothing but
 *                invoke another method, then returns.
 *
 * INTERFACE:
 *   parameters:  method: Method to test for wrapperness
 *   returns:     TRUE if wrapper, FALSE if not
 *================================================================*/

static bool_t
isWrapperMethod(METHOD method)
{
    /* Not implemented */
    return FALSE;
}

/*==================================================================
 * FUNCTION:      isGetterMethod()
 * TYPE:          Helper function
 * OVERVIEW:      Tests if a method is a getter method. A
 *                "getter method" is a method that consists of
 *                the three bytecodes:
 *                ALOAD_0 - GETFIELD - A/D/F/I/LRETURN
 *                for instance methods, and
 *                GETSTATIC - A/D/F/I/LRETURN
 *                for class (static) methods.
 *
 * INTERFACE:
 *   parameters:  method: Method to test for getterness
 *   returns:     TRUE if getter method, FALSE if not
 *================================================================*/

static bool_t
isGetterMethod(METHOD method)
{
    if (method->accessFlags & (ACC_NATIVE | ACC_ABSTRACT) ) {
        return FALSE;
    }
    if (method->accessFlags & ACC_STATIC) {
        /* Static method:
           Recognize pattern GETSTATIC I/L/F/D/ARETURN */
        if (method->u.java.codeLength == 4) {
            BYTE op = method->u.java.code[0];
            if ( (op == GETSTATIC) ||
                (op == GETSTATIC_FAST) ||
                (op == GETSTATICP_FAST) ||
                (op == GETSTATIC2_FAST)
            ) {
                op = method->u.java.code[3];
                if ( (op == IRETURN) ||
                    (op == LRETURN) ||
                    (op == FRETURN) ||
                    (op == DRETURN) ||
                    (op == ARETURN)
                ) {
                    return TRUE;
                }
            }
        }
    }
    else {
        if (method->u.java.codeLength == 5) {
            BYTE op = method->u.java.code[0];
            if (op == ALOAD_0) {
                op = method->u.java.code[1];
                if ( (op == GETFIELD) ||
                    (op == GETFIELD_FAST) ||
                    (op == GETFIELDP_FAST) ||
                    (op == GETFIELD2_FAST)
                ) {
                    op = method->u.java.code[4];
                    if ( (op == IRETURN) ||
                        (op == LRETURN) ||
                        (op == FRETURN) ||
                        (op == DRETURN) ||
                        (op == ARETURN)
                    ) {
                        return TRUE;
                    }
                }
            }
        }
    }
    return FALSE;
}
```

```
/*================================================================
 * FUNCTION:      isSetterMethod()
 * TYPE:          Helper function
 * OVERVIEW:      Tests if a method is a setter method. A
 *                "setter method" is
 *                a method that consists of the three bytecodes:
 *                ALOAD_0 - I/L/F/D/ALOAD_1 - PUTFIELD - RETURN
 *                for instance methods, and
 *                I/L/F/D/ALOAD_0 - PUTSTATIC - RETURN
 *                for class (static) methods.
 *
 * INTERFACE:
 *   parameters:  method: Method to test for setterness
 *   returns:     TRUE if getter method, FALSE if not
 *===============================================================*/

static bool_t
isSetterMethod(METHOD method)
{
    if (method->accessFlags & (ACC_NATIVE | ACC_ABSTRACT) ) {
        return FALSE;
    }
    if (method->accessFlags & ACC_STATIC) {
        if (method->u.java.codeLength == 5) {
            BYTE op = method->u.java.code[0];
            if ( (op == ILOAD_0) ||
                 (op == LLOAD_0) ||
                 (op == FLOAD_0) ||
                 (op == DLOAD_0) ||
                 (op == ALOAD_0)
            ) {
                op = method->u.java.code[1];
                if ( (op == PUTSTATIC) ||
                     (op == PUTSTATIC_FAST) ||
                     (op == PUTSTATIC2_FAST)
                ) {
                    op = method->u.java.code[4];
                    if (op == RETURN) {
                        return TRUE;
                    }
                }
            }
        }
    }
    else {
        if (method->u.java.codeLength == 6) {
            unsigned char op = method->u.java.code[0];
            if (op == ALOAD_0) {
                op = method->u.java.code[1];
                if ( (op == ILOAD_1) ||
                     (op == LLOAD_1) ||
                     (op == FLOAD_1) ||
                     (op == DLOAD_1) ||
                     (op == ALOAD_1)
                ) {
                    op = method->u.java.code[2];
                    if ( (op == PUTFIELD) ||
                         (op == PUTFIELD_FAST) ||
                         (op == PUTFIELD2_FAST)
                    ) {
                        op = method->u.java.code[5];
                        if (op == RETURN) {
                            return TRUE;
                        }
                    }
                }
            }
        }
    }
    return FALSE;
}

/*=================================================================
 * FUNCTION:      exitTheMonitorPlease()
 * TYPE:          Native code helper function
 * OVERVIEW:      Called by native code to exit the monitor for
 *                an object.
 *
 * INTERFACE:
 *   parameters:  object: The object whose monitor to exit
 *   returns:     <nothing>
 *===============================================================*/

static void
exitTheMonitorPlease(OBJECT object)
{
    char *exitFailure;
    if (monitorExit(object, &exitFailure) == MonitorStatusError) {
        raiseException(exitFailure);
    }
}

/*=================================================================
 * FUNCTION:      lookupSwitch()
 * TYPE:          Native code helper function
 * OVERVIEW:      Called by native code to get the target basic
 *                block for a LOOKUPSWITCH operation.
 *                The reason for not inlining this code is that
 *                it would be very bulky.
 *                The function is almost identical to the
 *                interpreter's handling of LOOKUPSWITCH.
 *
 * INTERFACE:
 *   parameters:  method, key, ip
 *   returns:     BASICBLOCK to transfer control to
 *=================================================================*/

static BASICBLOCK
lookupSwitch(METHOD method, long key, BYTE *ip)
{
    struct pair { cell value; cell offset; };
    cell *base = (cell *)(((long)ip + 4) & ~3);
    long numberOfPairs = getAlignedCell(base + 1);
    long delta = numberOfPairs - 1;

    struct pair *firstPair = (struct pair *)(base + 2);
    for (;;) {
        /* ASSERT:  The item we are looking for, if it is in the
           table, is in between firstPair and firstPair + delta,
           inclusive.
         */
        if (delta < 0) {
            /* Item not found, use default at base */
            break;
        } else {
            long halfDelta = (delta >> 1);
            struct pair *middlePair = firstPair + halfDelta;
            /* Don't use cell.  It's unsigned */
            long middleValue = getAlignedCell(&middlePair->value);
            if (middleValue < key) {
                /* Item is second half of the table */
                firstPair = middlePair + 1;
                delta = delta - halfDelta - 1;
            } else if (middleValue > key) {
                /* Item is at the beginning of the table. */
                delta = halfDelta - 1;
            } else {
                /* We have it.  Set base to point to the location
                   of the offset.
                 */
                base = &middlePair->offset;
                break;
            }
        }
    }
    /* base points at the offset by which we increment the ip */
    return basicBlockForIp(method, ip + getAlignedCell(base) );
}

/*=================================================================
 * FUNCTION:      invokeMethod()
 * TYPE:          Native code helper function
 * OVERVIEW:      Called by native code to set up the execution
 *                environment for an invoked method.
 *
 * INTERFACE:
 *   parameters:  thisMethod
 *   returns:     Native address to resume execution from
 *===============================================================*/

extern OBJECT thisObjectGCSafe;

static void *
invokeMethod(METHOD thisMethod, OBJECT thisObject)
{
    /* Check if the method is a native method */
    if (thisMethod->accessFlags & ACC_NATIVE) {
        invokeNativeFunction(thisMethod);
        return interpResume;
    }

    /* Check if this is an abstract method */
    if (thisMethod->accessFlags & ACC_ABSTRACT) {
        raiseExceptionWithMessage(
            AbstractMethodError, methodName(thisMethod));
    }

    /* Create an execution frame for executing a Java method */
    thisObjectGCSafe = thisObject;
    pushFrame(thisMethod);

    /* Check if this is a synchronized method */
    if (thisMethod->accessFlags & ACC_SYNCHRONIZED) {
        monitorEnter(thisObjectGCSafe);
        fp_global->syncObject = thisObjectGCSafe;
    }

    thisObjectGCSafe = NULL;

    if (thisMethod->u.java.basicBlocks == NULL) {
        return interpResume;
    }
    return thisMethod->u.java.basicBlocks->basicBlocks[0].nativeCode;
}

/* Invoke a method that is known to be native */
static void *
invokeNativeMethod(METHOD thisMethod)
{
    invokeNativeFunction(thisMethod);
}

/* Invoke a method that is known to be special or static */
static void *
invokeSpecialOrStaticMethod(METHOD thisMethod, OBJECT thisObject)
{
    /* Create an execution frame for executing a Java method */
    thisObjectGCSafe = thisObject;
    pushFrame(thisMethod);

    thisObjectGCSafe = NULL;

/*    if (thisMethod->u.java.basicBlocks == NULL) {
        return interpResume;
    } */
    return thisMethod->u.java.basicBlocks->basicBlocks[0].nativeCode;
}

/* Invoke a method that is known to be synchronized special
   or static */
static void *
invokeSyncSpecialOrStaticMethod(METHOD thisMethod, OBJECT thisObject)
{
    /* Create an execution frame for executing a Java method */
    thisObjectGCSafe = thisObject;
    pushFrame(thisMethod);

    monitorEnter(thisObjectGCSafe);
    fp_global->syncObject = thisObjectGCSafe;

    thisObjectGCSafe = NULL;

/*    if (thisMethod->u.java.basicBlocks == NULL) {
        return interpResume;
    } */
    return thisMethod->u.java.basicBlocks->basicBlocks[0].nativeCode;
}

/*=================================================================
 * FUNCTION:      tcacheAppend()
 * TYPE:          Native code generation
 * OVERVIEW:      Appends one word to tcache.
 *
 * INTERFACE:
 *   parameters:  value: Word value to append
 *   returns:     Nothing, but throws exception if buffer
 *                overflows
 *===============================================================*/

void tcacheAppend(int value)
{
    if (tcachePos == RequestedTranslationCacheSize) {
        /* Buffer overflow */
        THROW(OutOfMemoryObject);
    }
    tcache[tcachePos++] = value;
}

/*=================================================================
 * FUNCTION:      tcacheAddress()
 * TYPE:          Native code generation
 * OVERVIEW:      Gets the absolute address of (pointer to) the
 *                current position in the tcache.
 *
 * INTERFACE:
 *   parameters:  none
 *   returns:     Address of current position
 *===============================================================*/
```

```
void *tcacheAddress()
{
    return &tcache[tcachePos];
}

/*=================================================================
 * FUNCTION:      tcacheDistance()
 * TYPE:          Native code generation
 * OVERVIEW:      Gets the distance (in number of words) between
 *                the given address and the current tcache
 *                address.
 *
 * INTERFACE:
 *   parameters:  address: Address whose distance to calculate
 *   returns:     Distance to address (in words)
 *================================================================*/

int tcacheDistance(void *address)
{
    return ((address - tcacheAddress()) >> 2);
}

/*=================================================================
 * FUNCTION:      backpatchConditionalBranch()
 * TYPE:          Native code generation
 * OVERVIEW:      Backpatches a preceding conditional branch
 *                instruction. The instruction word is modified
 *                so that the instruction describes a jump from
 *                that instruction's address to the current
 *                address.
 *
 * INTERFACE:
 *   parameters:  pos: Position in tcache of instruction to patch
 *   returns:     <nothing>
 *================================================================*/

void backpatchConditionalBranch(int pos)
{
    int displ;
    displ = tcachePos - pos;
    tcache[pos] |= (displ & 0x3FFF) << 2;
}

void backpatchRelativeBranch(int pos)
{
    int displ;
    displ = tcachePos - pos;
    tcache[pos] |= (displ & 0xFFFFFF) << 2;
}

/*=================================================================
 * Low-level native instruction generators.
 * These are 1-to-1 with the target ISA (PowerPC, in this case).
 * The instruction word is encoded and added to the tcache.
 *===============================================================*/

#define compileWord(value) tcacheAppend((long)value)

#define compileMr(dest, src) \
    compileWord(MR(dest, src))
#define compileMtctr(src) \
    compileWord(MTCTR(src))
#define compileBctr() \
    compileWord(BCTR())
#define compileLbzx(dest, base, index) \
    compileWord(LBZX(dest, base, index))
#define compileLhzx(dest, base, index) \
    compileWord(LHZX(dest, base, index))
#define compileLhax(dest, base, index) \
    compileWord(LHAX(dest, base, index))
#define compileLwzx(dest, base, index) \
    compileWord(LWZX(dest, base, index))
#define compileExtsb(dest, src) \
    compileWord(EXTSB(dest, src))
#define compileExtsh(dest, src) \
    compileWord(EXTSH(dest, src))
#define compileLi(dest, value) \
    compileWord(LI(dest, value))
#define compileLis(dest, value) \
    compileWord(LIS(dest, value))
#define compileAdd(dest, src1, src2) \
    compileWord(ADD(dest, src1, src2))
#define compileAddc(dest, src1, src2) \
    compileWord(ADDC(dest, src1, src2))
#define compileAdde(dest, src1, src2) \
    compileWord(ADDE(dest, src1, src2))
#define compileAddze(dest, src) \
    compileWord(ADDZE(dest, src))
#define compileSub(dest, src1, src2) \
    compileWord(SUB(dest, src1, src2))
```

```
#define compileSubc(dest, src1, src2) \
    compileWord(SUBC(dest, src1, src2))
#define compileSube(dest, src1, src2) \
    compileWord(SUBE(dest, src1, src2))
#define compileMullw(dest, src1, src2) \
    compileWord(MULLW(dest, src1, src2))
#define compileDivw(dest, src1, src2) \
    compileWord(DIVW(dest, src1, src2))
#define compileSlw(dest, src1, src2) \
    compileWord(SLW(dest, src1, src2))
#define compileSrw(dest, src1, src2) \
    compileWord(SRW(dest, src1, src2))
#define compileSraw(dest, src1, src2) \
    compileWord(SRAW(dest, src1, src2))
#define compileAnd(dest, src1, src2) \
    compileWord(AND(dest, src1, src2))
#define compileOr(dest, src1, src2) \
    compileWord(OR(dest, src1, src2))
#define compileXor(dest, src1, src2) \
    compileWord(XOR(dest, src1, src2))
#define compileNeg(dest, src) \
    compileWord(NEG(dest, src))
#define compileNot(dest, src) \
    compileWord(NOT(dest, src))
#define compileSlwi(dest, src, count) \
    compileWord(SLWI(dest, src, count))
#define compileSrawi(dest, src, count) \
    compileWord(SRAWI(dest, src, count))
#define compileFadd(dest, src1, src2) \
    compileWord(FADD(dest, src1, src2))
#define compileFsub(dest, src1, src2) \
    compileWord(FSUB(dest, src1, src2))
#define compileFmul(dest, src1, src2) \
    compileWord(FMUL(dest, src1, src2))
#define compileFdiv(dest, src1, src2) \
    compileWord(FDIV(dest, src1, src2))
#define compileFadds(dest, src1, src2) \
    compileWord(FADDS(dest, src1, src2))
#define compileFsubs(dest, src1, src2) \
    compileWord(FSUBS(dest, src1, src2))
#define compileFmuls(dest, src1, src2) \
    compileWord(FMULS(dest, src1, src2))
#define compileFdivs(dest, src1, src2) \
    compileWord(FDIVS(dest, src1, src2))
#define compileFneg(dest, src) \
    compileWord(FNEG(dest, src))
#define compileFrsp(dest, src) \
    compileWord(FRSP(dest, src))
#define compileFctiwz(dest, src) \
    compileWord(FCTIWZ(dest, src))
#define compileFcmpo(src1, src2) \
    compileWord(FCMPO(0, src1, src2))
#define compileOri(dest, src, value) \
    compileWord(ORI(dest, src, value))
#define compileAndi(dest, src, value) \
    compileWord(ANDI_(dest, src, value))
#define compileLhz(dest, offset, base) \
    compileWord(LHZ(dest, offset, base))
#define compileLwz(dest, offset, base) \
    compileWord(LWZ(dest, offset, base))
#define compileStw(dest, offset, base) \
    compileWord(STW(dest, offset, base))
#define compileLfs(dest, offset, base) \
    compileWord(LFS(dest, offset, base))
#define compileStfs(dest, offset, base) \
    compileWord(STFS(dest, offset, base))
#define compileLfd(dest, offset, base) \
    compileWord(LFD(dest, offset, base))
#define compileStfd(dest, offset, base) \
    compileWord(STFD(dest, offset, base))
#define compileStbx(dest, base, index) \
    compileWord(STBX(dest, base, index))
#define compileSthx(dest, base, index) \
    compileWord(STHX(dest, base, index))
#define compileStwx(dest, base, index) \
    compileWord(STWX(dest, base, index))
#define compileAddi(dest, src, value) \
    compileWord(ADDI(dest, src, value))
#define compileAddic(dest, src, value) \
    compileWord(ADDIC(dest, src, value))
#define compileCmpwi(src, value) \
    compileWord(CMPWI(src, value))
#define compileCmplwi(src, value) \
    compileWord(CMPLWI(src, value))
#define compileCmpw(src1, src2) \
    compileWord(CMPW(src1, src2))
#define compileBeq(offset) \
    compileWord(BEQ(offset))
#define compileBne(offset) \
    compileWord(BNE(offset))
```

```
#define compileBgt(offset) \
    compileWord(BGT(offset))
#define compileBlt(offset) \
    compileWord(BLT(offset))
#define compileBge(offset) \
    compileWord(BGE(offset))
#define compileBle(offset) \
    compileWord(BLE(offset))
#define compileBun(offset) \
    compileWord(BUN(offset))
#define compileB(offset) \
    compileWord(B(offset))
#define compileBl(offset) \
    compileWord(BL(offset))
#define compileBlr() \
    compileWord(BLR())
#define compileBlrl() \
    compileWord(BLRL())
#define compileMtlr(src) \
    compileWord(MTLR(src))
#define compileMflr(src) \
    compileWord(MFLR(src))

/*=================================================================
 * Medium-level native code generators.
 *================================================================*/

#define IS_SHORT(s) ( ((s) <= 32767) && ((s) > -32768) )

#define IS_USHORT(s) ( ((s) <= 65535) && ((s) > 0) )

/* Moves a 32-bit constant into a register in least
   expensive way. */
static void
compileWordToReg(long value, int reg)
{
    if (IS_SHORT(value)) {
        compileLi(reg, value);
    }
    else if ((value & 0xFFFF) == 0) {
        compileLis(reg, value >> 16);
    }
    else {
        compileLis(reg, value >> 16);
        compileOri(reg, reg, value);
    }
}

/* Compares a 32-bit constant to a register in least
   expensive way. */
static void
compileCompareWord(int src, long value)
{
    if (IS_SHORT(value)) {
        compileCmpwi(src, value);
    }
    else if (IS_USHORT(value)) {
        compileCmplwi(src, value);
    }
    else {
        compileWordToReg(value, 0);
        compileCmpw(src, 0);
    }
}

/* Pushes int register on stack */
static void
compilePushReg(int src)
{
    compileStw(src, ++relSp*4, SP_REG);
}

/* Moves a pointer to a register */
#define compilePtrToReg(ptr, dest) \
    compileWordToReg((long)(ptr), dest)

/* Pops stack word into int register */
static void
compilePopReg(dest)
{
    compileLwz(dest, relSp--*4, SP_REG);
}

/* Pushes 32-bit constant on stack */
static void
compilePushWord(long value)
{
    compileWordToReg(value, 3);
    compilePushReg(3);
}
```

```
/* Pushes float register on stack (single precision) */
static void
compilePushFregs(int src)
{
    compileStfs(src, ++relSp*4, SP_REG);
}

/* Pops stack word into float register (single precision) */
static void
compilePopFregs(int dest)
{
    compileLfs(dest, relSp--*4, SP_REG);
}

/* Pushes float register on stack (double precision) */
static void
compilePushFregd(int src)
{
    compileStfd(src, ++relSp*4, SP_REG); relSp++;
}

/* Pops stack word into float register (double precision) */
static void
compilePopFregd(int dest)
{
    relSp--; compileLfd(dest, relSp--*4, SP_REG);
}

/* Loads a local into a register */
static void
compileLocalToReg(int index, int dest)
{
    compileLwz(dest, (index)*4, LP_REG);
}

/* Pushes a local onto stack */
static void
compilePushLocal(int index)
{
    compileLocalToReg(index, 3);
    compilePushReg(3);
}

/* Stores register to local */
```

```
static void
compileRegToLocal(int src, int index)
{
    compileStw(src, (index)*4, LP_REG);
}

/* Pops stack word into local */
static void
compilePopLocal(int index)
{
    compilePopReg(3);
    compileRegToLocal(3, index);
}

/* Moving stack word to register */
static void
compileStackToReg(int index, int dest)
{
    compileLwz(dest, (relSp-(index))*4, SP_REG);
}

#define compileTopStackToReg(dest) \
    compileStackToReg(0, dest)
#define compileSecondStackToReg(dest) \
    compileStackToReg(1, dest)
#define compileThirdStackToReg(dest) \
    compileStackToReg(2, dest)
#define compileFourthStackToReg(dest) \
    compileStackToReg(3, dest)

/* Moving register to stack word */
static void
compileRegToStack(int src, int index)
{
    compileStw(src, (relSp-(index))*4, SP_REG);
}

#define compileRegToTopStack(src) \
    compileRegToStack(src, 0)
#define compileRegToSecondStack(src) \
    compileRegToStack(src, 1)
#define compileRegToThirdStack(src) \
    compileRegToStack(src, 2)
#define compileRegToFourthStack(src) \
    compileRegToStack(src, 3)
```

```
/* Makes sure the IP native register contains the
   correct value */
static bool_t
compileUpdateIp(BYTE *newIp)
{
    int displ = newIp - savedIp;
    bool_t result = (displ != 0);
    if (result) {
        if (IS_SHORT(displ)) {
            compileAddi(IP_REG, IP_REG, displ);
        }
        else {
            compilePtrToReg(newIp, IP_REG);
        }
        savedIp = newIp;
    }
    return result;
}

/* Updates the stack pointer if it's changed */
static bool_t
compileUpdateSp()
{
    bool_t result = (relSp != 0);
    if (result) {
        compileAddi(SP_REG, SP_REG, relSp*4);
        relSp = 0;
    }
    return result;
}

#define compileGetClass(destReg, objectReg) \
    compileLwz(destReg, 0, objectReg)  /* object->ofClass */

#define compileGetFpWord(destReg, index) \
    compileLwz(destReg, index*4, FP_REG)
#define compileGetPreviousFp(destReg) \
    compileGetFpWord(destReg, 0)
#define compileGetPreviousIp(destReg) \
    compileGetFpWord(destReg, 1)
#define compileGetPreviousSp(destReg) \
    compileGetFpWord(destReg, 2)
#define compileGetThisMethod(destReg) \
    compileGetFpWord(destReg, 3)
```

```
#define compileGetSyncObject(destReg) \
    compileGetFpWord(destReg, 5)

#define compileGetArrayLength(destReg, arrayReg) \
    compileLwz(destReg, 8, arrayReg)  /* array->length */

/*=================================================================
 * FUNCTION:      compileUpdateVM()
 * TYPE:          Native code generation
 * OVERVIEW:      Makes sure that the VM registers are
 *                up-to-date. This should be called before native
 *                code that branches to handleException*, to
 *                ensure that the global VM state will be correct
 *                in case an exception does indeed occur.
 *
 * INTERFACE:
 *   parameters:  none
 *   returns:     <nothing>
 *================================================================*/

static void
compileUpdateVM()
{
    compileUpdateIp(thisIp);
    compileUpdateSp();
}

static void
compileSaveVM()
{
    compileUpdateVM();
    compileStw(IP_REG, 0, GS_REG);     /* GlobalState.gs_ip */
    compileStw(SP_REG, 4, GS_REG);     /* GlobalState.gs_sp */
    /* We assume FP_REG and LP_REG saved in method prolog */
}

static void
compileRestoreVM()
{
    compileLwz(IP_REG, 0, GS_REG);     /* GlobalState.gs_ip */
    compileLwz(SP_REG, 4, GS_REG);     /* GlobalState.gs_sp */
    compileLwz(LP_REG, 8, GS_REG);     /* GlobalState.gs_lp */
    compileLwz(FP_REG, 12, GS_REG);    /* GlobalState.gs_fp */
}
```

```
/*=================================================================
 * FUNCTION:      compileBranch()
 * TYPE:          Native code generation
 * OVERVIEW:      Compiles native code that branches to a given
 *                address.
 *
 * INTERFACE:
 *   parameters:  addr: Absolute address to branch to
 *                link: TRUE if branch and link, FALSE otherwise
 *   returns:     <nothing>
 *=================================================================*/

void compileBranch(void *addr, bool_t link)
{
    int displ;
    /* Calculate displacement from there to here */
    displ = ((int)addr - (int)tcacheAddress()) >> 2;
    /* See if it fits in 24 bits */
    if ( (displ >= -8388608) && (displ < 8388608) ) {
        /* Then we can do relative branch */
        if (link) {
            compileBl(displ);
        }
        else {
            compileB(displ);
        }
    }
    else {
        /* We must do full 32-bit branch */
        /* Chances are this will happen when the called function
           is located in the program's code segment and not the
           data segment (since these may be placed at virtual
           addresses far apart) */
        /* Transfer address via temp reg R0 to LR */
        compilePtrToReg(addr, 0);
        compileMtlr(0);
        if (link) {
            compileBlrl();
        }
        else {
            compileBlr();
        }
    }
}
```

```
/*=================================================================
 * FUNCTION:      compileCall()
 * TYPE:          Native code generation
 * OVERVIEW:      Compiles native code that calls a function.
 *
 * INTERFACE:
 *   parameters:  function: Absolute address of function to call
 *   returns:     <nothing>
 *===============================================================*/

static void
compileCall(void *function)
{
    compileBranch(function, TRUE);
}

/*=================================================================
 * FUNCTION:      compileNullCheck()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code that branches to null
 *                exception handler if given register is NULL.
 *
 * INTERFACE:
 *   parameters:  addressReg: Register to check for NULL value
 *   returns:     <nothing>
 *===============================================================*/

static void
compileNullCheck(int addressReg)
{
    compileCmpwi(addressReg, 0);
    compileBeq(tcacheDistance(handleNullPointerException));
}

/*=================================================================
 * FUNCTION:      compileIndexBoundsCheck()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code that checks that an array
 *                index is within bounds, and branches to
 *                exception handler if not.
 *
 * INTERFACE:
 *   parameters:  arrayReg: Register containing pointer to array
 *                indexReg: Register containing array index
 *   returns:     <nothing>
 *=============================================================*/

static void
compileIndexBoundsCheck(int arrayReg, int indexReg)
{
    /* Low bounds (0) */
    compileCmpwi(indexReg, 0);
    compileBlt(
        tcacheDistance(handleArrayIndexOutOfBoundsException));
    /* High bounds (arraylength) */
    compileGetArrayLength(0, arrayReg);
    compileCmpw(indexReg, 0);
    compileBge(
        tcacheDistance(handleArrayIndexOutOfBoundsException));
}

/*=============================================================
 * FUNCTION:      compileElementOffset()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code that computes the byte
 *                offset of an array element.
 *
 * INTERFACE:
 *   parameters:  destReg:  Where to store computed offset
 *                indexReg: Register containing array index
 *                shift:    How much to shift the index left
 *                          (0, 1, 2, 3)
 *   returns:     <nothing>
 *=============================================================*/

static void
compileElementOffset(int destReg, int indexReg, int shift)
{
    if (shift) {
        compileSlwi(destReg, indexReg, shift);
        /* Zeroeth element @ offset 12 */
        compileAddi(destReg, destReg, 12);
    }
    else {
        /* Zeroeth element @ offset 12 */
        compileAddi(destReg, indexReg, 12);
    }
}

/*=============================================================
 * FUNCTION:      compileArrayLoadProlog()
 * TYPE:          Native code generation
 * OVERVIEW:      Compiles code shared by all array load
 *                operations.
 *                Specifically:
 *                - Index and array reference are popped from
 *                  stack
 *                - Array reference is checked for null
 *                - Index is checked to be within bounds
 *                - Byte offset of element is computed
 *
 * INTERFACE:
 *   parameters:  shift Index shift amount (0, 1, 2, 3)
 *   returns:     <nothing>
 *=============================================================*/

static void
compileArrayLoadProlog(int arrayReg, int indexReg,
    int offsetReg, int shift)
{
    compilePopReg(indexReg);
    compilePopReg(arrayReg);
    compileUpdateVM();
    compileNullCheck(arrayReg);
    compileIndexBoundsCheck(arrayReg, indexReg);
    compileElementOffset(offsetReg, indexReg, shift);
}

/*=============================================================
 * FUNCTION:      compileArrayStoreProlog()
 * TYPE:          Native code generation
 * OVERVIEW:      Compiles code shared by all array store
 *                operations.
 *                Identical to compileArrayLoadProlog, except the
 *                value to be stored is popped first.
 *
 * INTERFACE:
 *   parameters:  shift Index shift amount (0, 1, 2, 3)
 *   returns:     <nothing>
 *=============================================================*/

static void
compileArrayStoreProlog(int arrayReg, int indexReg,
    int offsetReg, int valueReg, int shift)
{
    compilePopReg(valueReg);
    compileArrayLoadProlog(arrayReg, indexReg, offsetReg, shift);
}

/*=================================================================
 * FUNCTION:      compileBBDispatch()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code that performs an indirect
 *                jump to the "nativeCode" field of a BASICBLOCK.
 *
 * INTERFACE:
 *   parameters:  bb: BASICBLOCK to jump to
 *   returns:     <nothing>
 *===============================================================*/

static void
compileBBDynamicDispatch(METHOD method, int bbReg)
{
    compilePtrToReg(methodCode(method), IP_REG);
    compileLhz(0, 4, bbReg);     /* field startIp */
    compileAdd(IP_REG, IP_REG, 0);
    compileLwz(0, 0, bbReg);     /* field "nativeCode" */
    compileMtlr(0);
    compileBlr();
}

static void
compileBBStaticDispatch(METHOD method, BYTE *ip)
{
    BASICBLOCK bb;
    bb = basicBlockForIp(method, ip);
    compileUpdateIp(bb->startIp + methodCode(method));
    compileLis(3, ((int)bb + 0x8000) >> 16);
    compileLwz(0, (int)bb & 0xFFFF, 3); /* field "nativeCode" */
    compileMtlr(0);
    compileBlr();
}

/*=================================================================
 * FUNCTION:      compileIfProlog()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code that's common to
 *                beginning of all
 *                IF-opcodes. Specifically,
 *                - The value to compare to 0 is popped from
 *                  operand stack
 *                - The in-register stack pointer is updated
 *                - The value is compared to 0
 *
 * INTERFACE:
 *   parameters:  none
 *   returns:     Position in tcache after code generation.
 *                It is expected that the next instruction
 *                generated will be a BEQ, BNE etc., and so the
 *                buffer position must be used to backpatch the
 *                branch later.
 *=============================================================*/

static void
compileIfProlog(int valueReg, int *bpatLoc)
{
    compilePopReg(valueReg);
    compileUpdateSp();
    compileCmpwi(valueReg, 0);
    recordBackpatchLocation(bpatLoc);
}

/*================================================================
 * FUNCTION:      compileIf_icmpProlog()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code that's common to
 *                beginning of all IF_ICMP-opcodes. Specifically,
 *                - The two values to compare are popped from
 *                  operand stack
 *                - The in-register stack pointer is updated
 *                - The values are compared
 *
 * INTERFACE:
 *   parameters:  none
 *   returns:     Position in tcache after code generation.
 *                It is expected that the next instruction
 *                generated will be a BEQ, BNE etc., and so the
 *                buffer position must be used to backpatch the
 *                branch later.
 *=============================================================*/

static void
compileIf_icmpProlog(int src1Reg, int src2Reg, int *bpatLoc)
{
    compilePopReg(src1Reg);
    compilePopReg(src2Reg);
    compileUpdateSp();
    compileCmpw(src2Reg, src1Reg);
    recordBackpatchLocation(bpatLoc);
}

/*=================================================================
 * FUNCTION:      compileIfEpilog()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code that's common to end of
 *                all IF-opcodes.
 *
 * INTERFACE:
 *   parameters:  method Method being compiled
 *                backpatchPos Position of relative branch
 *                instruction
 *   returns:     <nothing>
 *===============================================================*/

static void
compileIfEpilog(METHOD method, int backpatchPos)
{
    BYTE *targetIp;
    BYTE *prevSavedIp = savedIp;
    /* Fall-through */
    compileBBStaticDispatch(method, thisIp+3);
    /* Target */
    backpatchConditionalBranch(backpatchPos);
    savedIp = prevSavedIp;
    targetIp = thisIp + getShort(thisIp+1);
    compileBBStaticDispatch(method, targetIp);
}

/*=================================================================
 * FUNCTION:      compileResumeInterpreterReschedule()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native that resumes interpretation at
 *                the reschedule point (see execute.c).
 *
 * INTERFACE:
 *   parameters:  none
 *   returns:     <nothing>
 *===============================================================*/

static void
```

```
compileResumeInterpreterReschedule()
{
    compileBranch(interpResume, FALSE);
}

/*=================================================================
 * FUNCTION:      compileReturn()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for RETURN opcodes.
 *
 * INTERFACE:
 *   parameters:  method Method being compiled
 *                cellsReturned 0, 1 or 2
 *   returns:     <nothing>
 *=================================================================*/

static void
compileReturn(METHOD method, int cellsReturned)
{
        /* Straightforward solution: goto next0 */
        if (method->accessFlags & ACC_SYNCHRONIZED) {
            compileSaveVM();
            compileGetSyncObject(3);    /* Arg 1 (OBJECT) */
            compileCall(exitTheMonitorPlease);
        }
        if (cellsReturned > 1) {
            compilePopReg(4);
        }
        if (cellsReturned > 0) {
            compilePopReg(3);
        }
        /* Restore previous instruction pointer */
        compileGetPreviousIp(IP_REG);
        /* Restore previous stack pointer */
        compileGetPreviousSp(SP_REG);
        /* Restore previous frame pointer */
        compileGetPreviousFp(FP_REG);
        /* fp->thisMethod */
        compileGetThisMethod(5);
        /* thisMethod->frameSize */
        compileLhz(0, (SIZEOF_METHOD << log2CELL)-4, 5);
        compileSlwi(0, 0, log2CELL);
        /* Restore previous locals pointer */
        compileSub(LP_REG, FP_REG, 0);
        relSp = 0;
        if (cellsReturned > 0) {
            compilePushReg(3);
        }
        if (cellsReturned > 1) {
            compilePushReg(4);
        }
        compileUpdateSp();
        /* Return to interpreter */
        compileResumeInterpreterReschedule();
#if 0
        /* Or: BB pointer could be stored in frame */
        compileLwz(3, 20, FP_REG);      /* fp->nextBB */
        compileBBDispatch(3);
#endif
}

/*=================================================================
 * FUNCTION:      compileGetField()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for GETFIELD opcode.
 *
 * INTERFACE:
 *   parameters:  ofClass Class containing field
 *                cpIndex Constant pool index of symbolic field
 *                        reference
 *   returns:     <nothing>
 *===============================================================*/

static void
compileGetField(INSTANCE_CLASS ofClass, unsigned int cpIndex)
{
    FIELD field;
    int offset;

    /* Resolve constant pool reference */
    field = resolveFieldReference(
        ofClass->constPool, cpIndex,
        FALSE, GETFIELD, ofClass);

    /* Pop instance */
    compilePopReg(3);
    compileUpdateVM();
    compileNullCheck(3);

    /* Load either one or two cells depending on field type */
    offset = FIELD_BYTE_OFFSET(field->u.offset);
    if (field->accessFlags & ACC_DOUBLE) {
        compileLwz(0, offset, 3);
        compilePushReg(0);
        compileLwz(3, offset+4, 3);
        compilePushReg(3);
    }
    else {
        compileLwz(3, offset, 3);
        compilePushReg(3);
    }
}

/*=================================================================
 * FUNCTION:      compilePutField()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for PUTFIELD opcode.
 *
 * INTERFACE:
 *   parameters:  ofClass Class containing field
 *                cpIndex Constant pool index of symbolic field
 *                        reference
 *   returns:     <nothing>
 *=================================================================*/

static void
compilePutField(INSTANCE_CLASS ofClass, unsigned int cpIndex)
{
    FIELD field;
    int offset;

    /* Resolve constant pool reference */
    field = resolveFieldReference(
        ofClass->constPool, cpIndex,
        FALSE, PUTFIELD, ofClass);

    /* Store either one or two cells depending on field type */
    offset = FIELD_BYTE_OFFSET(field->u.offset);
    if (field->accessFlags & ACC_DOUBLE) {
        compilePopReg(3);    /* Low word */
        compilePopReg(4);    /* High word */
        compilePopReg(5);    /* Object */
        compileUpdateVM();
        compileNullCheck(5);
        compileStw(4, offset, 5);    /* High word */
        compileStw(3, offset+4, 5);  /* Low word */
    } else {
        compilePopReg(3);
        compilePopReg(4);   /* Object */
        compileUpdateVM();
        compileNullCheck(4);
        compileStw(3, offset, 4);
    }
}

/*=================================================================
 * FUNCTION:      compileGetFieldFast()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for GETFIELD_FAST opcodes.
 *
 * INTERFACE:
 *   parameters:  index Index of cell in object
 *                opcode Opcode
 *   returns:     <nothing>
 *=================================================================*/

static void
compileGetFieldFast(unsigned int index, BYTE opcode)
{
    unsigned int offset;
    offset = FIELD_BYTE_OFFSET(index);

    /* Pop instance */
    compilePopReg(3);
    compileUpdateVM();
    compileNullCheck(3);

    /* Load word(s) onto the operand stack */
    if (opcode == GETFIELD2_FAST) {
        compileLwz(0, offset, 3);    /* High word */
        compilePushReg(0);
        compileLwz(3, offset+4, 3);  /* Low word */
        compilePushReg(3);
    }
    else {
        compileLwz(3, offset, 3);
        compilePushReg(3);
    }
}
```

```
/*================================================================
 * FUNCTION:      compilePutFieldFast()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for PUTFIELD_FAST opcodes.
 *
 * INTERFACE:
 *   parameters:  index Index of cell in object
 *                opcode Opcode
 *   returns:     <nothing>
 *================================================================*/

static void
compilePutFieldFast(unsigned int index, BYTE opcode)
{
    unsigned int offset;
    offset = FIELD_BYTE_OFFSET(index);

    /* Pop value */
    compilePopReg(3);
    if (opcode == PUTFIELD2_FAST) {
        compilePopReg(4);
    }
    /* Pop instance */
    compilePopReg(5);
    compileUpdateVM();
    compileNullCheck(5);

    /* Store word(s) from the operand stack */
    if (opcode == PUTFIELD2_FAST) {
        compileStw(4, offset, 5);    /* High word */
        compileStw(3, offset+4, 5);  /* Low word */
    }
    else {
        compileStw(3, offset, 5);
    }
}

/*================================================================
 * FUNCTION:      compileGetStatic()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for GETSTATIC opcode.
 *
 * INTERFACE:
 *   parameters:  ofClass Class containing field
 *                cpIndex Constant pool index of symbolic field
 *                        reference
 *   returns:     <nothing>
 *=============================================================*/

static void
compileGetStatic(INSTANCE_CLASS ofClass, unsigned int cpIndex)
{
    FIELD field;
    cell *location;

    /* Resolve constant pool reference */
    field = resolveFieldReference(
        ofClass->constPool, cpIndex,
        TRUE, GETSTATIC, ofClass);

    /* Move address of field to register */
    location = (cell *)field->u.staticAddress;
    compileLis(3, ((int)location + 0x8000) >> 16);

    /* Load either one or two cells depending on field type */
    if (field->accessFlags & ACC_DOUBLE) {
        compileLwz(0, (int)location & 0xFFFF, 3);
        compilePushReg(0);
        location++;
        compileLis(3, ((int)location + 0x8000) >> 16);
        compileLwz(3, (int)location & 0xFFFF, 3);
        compilePushReg(3);
    }
    else {
        compileLwz(3, (int)location & 0xFFFF, 3);
        compilePushReg(3);
    }
}

/*=============================================================
 * FUNCTION:      compilePutStatic()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for PUTSTATIC opcode.
 *
 * INTERFACE:
 *   parameters:  ofClass Class containing field
 *                cpIndex Constant pool index of symbolic field
 *                        reference
 *   returns:     <nothing>
 *=============================================================*/
```

```
static void
compilePutStatic(INSTANCE_CLASS ofClass, unsigned int cpIndex)
{
    FIELD field;
    cell *location;

    /* Resolve constant pool reference */
    field = resolveFieldReference(
        ofClass->constPool, cpIndex,
        TRUE, PUTSTATIC, ofClass);

    /* Move address of field to register */
    location = (cell *)field->u.staticAddress;
    compileLis(4, ((int)location + 0x8000) >> 16);

    /* Store either one or two cells depending on field type */
    if (field->accessFlags & ACC_DOUBLE) {
        compilePopReg(3);
        compileStw(3, (int)location & 0xFFFF, 4);
        location++;
        compileLis(4, ((int)location + 0x8000) >> 16);
        compilePopReg(3);
        compileStw(3, (int)location & 0xFFFF, 4);
    }
    else {
        compilePopReg(3);
        compileStw(3, (int)location & 0xFFFF, 4);
    }
}

/*=================================================================
 * FUNCTION:      compileGetStaticFast()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for GETSTATIC_FAST opcodes.
 *
 * INTERFACE:
 *   parameters:  ofClass Class containing field
 *                cpIndex Constant pool index of symbolic field
 *                        reference
                  opcode Opcode
 *   returns:     <nothing>
 *=================================================================*/

static void
```

```
compileGetStaticFast(INSTANCE_CLASS ofClass,
    unsigned int cpIndex, BYTE opcode)
{
    cell *location;
    FIELD field;
    field = (FIELD)ofClass->constPool->entries[cpIndex].cache;

    /* Move address of field to register */
    location = (cell *)field->u.staticAddress;
    compileLis(3, ((int)location + 0x8000) >> 16);

    /* Push contents of the field onto the operand stack */
    if (opcode == GETSTATIC2_FAST) {
        compileLwz(0, (int)location & 0xFFFF, 3); /* High word */
        compilePushReg(0);
        location++;
        compileLis(3, ((int)location + 0x8000) >> 16);
        compileLwz(3, (int)location & 0xFFFF, 3); /* Low word */
        compilePushReg(3);
    }
    else {
        compileLwz(3, (int)location & 0xFFFF, 3);
        compilePushReg(3);
    }
}

/*=================================================================
 * FUNCTION:      compilePutStaticFast()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for PUTSTATIC_FAST
 *                opcodes.
 *
 * INTERFACE:
 *   parameters:  ofClass Class containing field
 *                cpIndex Constant pool index of symbolic field
 *                        reference
                  opcode Opcode
 *   returns:     <nothing>
 *=================================================================*/

static void
compilePutStaticFast(INSTANCE_CLASS ofClass,
    unsigned int cpIndex, BYTE opcode)
{
    cell *location;
    FIELD field;
    field = (FIELD)ofClass->constPool->entries[cpIndex].cache;

    /* Move address of field to register */
    location = (cell *)field->u.staticAddress;
    compileLis(4, ((int)location + 0x8000) >> 16);

    /* Store contents of the operand stack into field */
    if (opcode == PUTSTATIC2_FAST) {
        compilePopReg(3);
        compileStw(3, (int)location & 0xFFFF, 4);
        location++;
        compileLis(4, ((int)location + 0x8000) >> 16);
        compilePopReg(3);
        compileStw(3, (int)location & 0xFFFF, 4);
    }
    else {
        compilePopReg(3);
        compileStw(3, (int)location & 0xFFFF, 4);
    }
}

/*=================================================================
 * FUNCTION:      compileAload_0()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for ALOAD_0 opcodes.
 *                Merges translation of ALOAD_0 - GETFIELD.
 *
 * INTERFACE:
 *   parameters:  none
 *   returns:     <nothing>
 *=================================================================*/

static int
compileAload_0()
{
    BYTE nextOp = thisIp[1];
    if ( (nextOp == GETFIELD_FAST) ||
         (nextOp == GETFIELDP_FAST) ) {
        /* Combine translation of ALOAD_0 - GETFIELD */
        unsigned int offset;
        offset = FIELD_BYTE_OFFSET( getUShort(thisIp+2) );
        compileLocalToReg(0, 3);
        thisIp++;
        compileUpdateVM();
        compileNullCheck(3);
        compileLwz(3, offset, 3);
        compilePushReg(3);
        return 1;
    }
    else {
        compilePushLocal(0);
        return 0;
    }
}

/*=================================================================
 * FUNCTION:      compileInvokeVirtual()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for INVOKEVIRTUAL opcodes.
 *
 * INTERFACE:
 *   parameters:  cpMethod Symbolic method to invoke
 *   returns:     <nothing>
 *===============================================================*/

static void
compileInvokeVirtual(METHOD cpMethod)
{
    int argCount;
    /* Calculate the number of parameters from signature */
    argCount = cpMethod->argCount;

    /* get this */
    compileStackToReg(argCount-1, 3);
    compileUpdateVM();
    compileNullCheck(3);

    /* Get the dynamic class of the object */
    compileGetClass(3, 3);

    /* Find the actual method */
    compilePtrToReg(cpMethod, 4);   /* 2nd argument */
    compileCall(lookupDynamicMethod);

    /* Setup environment for method */
    compileStackToReg(argCount-1, 4);    /* this */
    thisIp += 3;
    compileSaveVM();
    compileCall(invokeMethod);
    compileRestoreVM();
    compileMtlr(3);
    compileBlr();
}
#if 0
using cache to optimize virtual dispatch:
compileBl(2);
compileWord(0);    /* reserve space for class cache */
compileMflr(4);
compileLwz(5, 0, 4);    /* cached class */
compileCmpw(3, 5);
compileBeq(0);    /* Skip lookup if the same class */
compilePtrToReg(cpMethod, 4);   /* 2nd argument */
compileCall(lookupDynamicMethod);
compileStw(...);    // don’t have address of cache anymore
#endif

/*=================================================================
 * FUNCTION:      compileInvokeSpecial()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for INVOKESPECIAL opcodes.
 *
 * INTERFACE:
 *   parameters:  cpMethod Symbolic method to invoke
 *   returns:     TRUE if the method was inlined, FALSE if not
 *=================================================================*/

static bool_t
compileInvokeSpecial(METHOD cpMethod)
{
    int argCount;
    /* Calculate the number of parameters from signature */
    argCount = cpMethod->argCount;

    /* get this */
    compileStackToReg(argCount-1, 4);
    compileUpdateVM();
    compileNullCheck(4);

    if (isEmptyMethod(cpMethod)) {
        /*
            * The target method doesn’t do anything except
            * RETURN.
            * We therefore don’t bother calling it; instead we
            * just pop the arguments.
        */
        relSp -= argCount;
        return TRUE;
    }
    else if (isGetterMethod(cpMethod)) {
        /* Inline the get */
        BYTE *c0de = methodCode(cpMethod);
        BYTE op = c0de[1];
        unsigned short index = getUShort(c0de + 2);
        if (op == GETFIELD) {
            compileGetField(cpMethod->ofClass, index);
        }
        else {
            /* "Fast" version */
            compileGetFieldFast(index, op);
        }
        return TRUE;
    }
    else if (isSetterMethod(cpMethod)) {
        /* Inline the set */
        BYTE *c0de = methodCode(cpMethod);
        BYTE op = c0de[2];
        unsigned short index = getUShort(c0de + 3);
        if (op == PUTFIELD) {
            compilePutField(cpMethod->ofClass, index);
        }
        else {
            /* "Fast" version */
            compilePutFieldFast(index, op);
        }
        return TRUE;
    }
    else {
        compilePtrToReg(cpMethod, 3);
        thisIp += 3;
        compileSaveVM();
        if (cpMethod->accessFlags & ACC_NATIVE) {
            compileCall(invokeNativeMethod);
            compileRestoreVM();
            compileResumeInterpreterReschedule();
        }
        else {
            if (cpMethod->accessFlags & ACC_SYNCHRONIZED) {
                compileCall(invokeSyncSpecialOrStaticMethod);
            }
            else {
                compileCall(invokeSpecialOrStaticMethod);
            }
            compileRestoreVM();
            compileMtlr(3);
            compileBlr();
        }
        return FALSE;
    }
}

/*=================================================================
 * FUNCTION:      compileInvokeStatic()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for INVOKESTATIC opcodes.
 *
 * INTERFACE:
 *   parameters:  cpMethod Symbolic method to invoke
 *   returns:     TRUE if the method was inlined, FALSE if not
 *===============================================================*/

static bool_t
compileInvokeStatic(METHOD cpMethod)
{
    if (isEmptyMethod(cpMethod)) {
        /*
        * The target method doesn’t do anything except RETURN.
        * We therefore don’t bother calling it; instead we just
        * pop the arguments.
        */
        relSp -= cpMethod->argCount;
        return TRUE;
    }
    else if (isGetterMethod(cpMethod)) {
        /* Inline the get */
        BYTE *c0de = methodCode(cpMethod);
        BYTE op = c0de[0];
        unsigned short index = getUShort(c0de + 1);
        if (op == GETSTATIC) {
            compileGetStatic(cpMethod->ofClass, index);
        }
        else {
            /* "Fast" version */
            compileGetStaticFast(cpMethod->ofClass, index, op);
        }
        return TRUE;
    }
    else if (isSetterMethod(cpMethod)) {
        /* Inline the set */
        BYTE *c0de = methodCode(cpMethod);
        BYTE op = c0de[1];
        unsigned short index = getUShort(c0de + 2);
        if (op == PUTSTATIC) {
            compilePutStatic(cpMethod->ofClass, index);
        }
        else {
            /* "Fast" version */
            compilePutStaticFast(cpMethod->ofClass, index, op);
        }
        return TRUE;
    }
    else {
        compilePtrToReg(cpMethod, 3);
        thisIp += 3;
        compileSaveVM();
        if (cpMethod->accessFlags & ACC_NATIVE) {
            compileCall(invokeNativeMethod);
            compileRestoreVM();
            compileResumeInterpreterReschedule();
        }
        else {
            compilePtrToReg(cpMethod->ofClass, 4);
            if (cpMethod->accessFlags & ACC_SYNCHRONIZED) {
                compileCall(invokeSyncSpecialOrStaticMethod);
            }
            else {
                compileCall(invokeSpecialOrStaticMethod);
            }
            compileRestoreVM();
            compileMtlr(3);
            compileBlr();
        }
        return FALSE;
    }
}

/*=================================================================
 * FUNCTION:      compileInvokeInterface()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for INVOKEINTERFACE
 *                 opcodes.
 *
 * INTERFACE:
 *   parameters:  cpMethod Symbolic method to invoke
 *   returns:     <nothing>
 *=============================================================*/

static void
compileInvokeInterface(METHOD cpMethod,
    INSTANCE_CLASS ofClass, int argCount)
{
    /* get this */
    compileStackToReg(argCount-1, 3);
    compileUpdateVM();
    compileNullCheck(3);

    /* Get method table entry based on dynamic class */
    /* with given method name and signature */
    compileGetClass(3, 3);
    /* The next line of code is OK, since key is
       read-only in lookupMethod */
    /* (Normally, we would create a copy of
       cpMethod->nameTypeKey on the stack, and pass a pointer
       to that instead. That's how GCC implements
       pass-structure-as-value) */
    compilePtrToReg(&cpMethod->nameTypeKey, 4);
    compilePtrToReg(ofClass, 5);
    compileCall(lookupMethod);

    /* Setup environment for method */
    compileStackToReg(argCount-1, 4);    /* this */
    thisIp += 5;
    compileSaveVM();
    compileCall(invokeMethod);
    compileRestoreVM();
    compileMtlr(3);
    compileBlr();
#if 0
Might use the same cache-style optimization for invokeinterface
#endif
}

/*=============================================================
 * FUNCTION:      compileCheckCast()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for CHECKCAST opcodes.
 *
 * INTERFACE:
 *   parameters:  thisClass Class to check against
 *   returns:     <nothing>
 *=============================================================*/

static void
compileCheckCast(CLASS thisClass)
{
    int pos;
    /* If object reference is NULL, or if the object */
    /* is a proper subclass of the cpIndexed class,  */
    /* the operand stack remains unchanged; otherwise */
    /* throw exception */
    compileTopStackToReg(3);
    compileCmpwi(3, 0);     /* is it NULL? */
    recordBackpatchLocation(&pos);
    compileBeq(0);          /* If so, skip */
    compileGetClass(3, 3);
    compilePtrToReg(thisClass, 4);
    compileCall(isAssignableTo);
    /* Goto handleClassCastException if result == 0 */
    compileCmpwi(3, 0);
    compileBeq(tcacheDistance(handleClassCastException));
    /* Backpatch */
    backpatchConditionalBranch(pos);
#if 0
Cache-style optimization:
Save latest (dynamic class, result) tuple (2 words)
Check if dynamic class is the same
if yes: reuse the previous result
otherwise: store dynamic class, call isAssignableTo(), store result
#endif
}

/*=============================================================
 * FUNCTION:      compileInstanceOf()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for INSTANCEOF opcodes.
 *
 * INTERFACE:
 *   parameters:  thisClass Class to check against
 *   returns:     <nothing>
 *=============================================================*/
```

```
static void
compileInstanceOf(CLASS thisClass)
{
    int pos;
    /* Check if assignable */
    compilePopReg(3);
    compileCmpwi(3, 0);
    recordBackpatchLocation(&pos);
    compileBeq(0);  /* Skip if null */
    compileGetClass(3, 3);
    compilePtrToReg(thisClass, 4);
    compileCall(isAssignableTo);
    /* Backpatch */
    backpatchConditionalBranch(pos);
    /* Push result */
    compilePushReg(3);
#if 0
Cache-style optimization: Same as checkcast
#endif
}

/*=================================================================
 * FUNCTION:      compileNew()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for NEW opcodes.
 *
 * INTERFACE:
 *   parameters:  thisClass Class to create instance of
 *   returns:     <nothing>
 *===============================================================*/

static void
compileNew(INSTANCE_CLASS thisClass)
{
    /* Create an object */
    compilePtrToReg(thisClass, 3);
    compileSaveVM();
    compileCall(instantiate);

    /* Push new object on stack */
    compilePushReg(3);
}

/*=================================================================
 * FUNCTION:      compileNew()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for NEWARRAY opcodes.
 *
 * INTERFACE:
 *   parameters:  type Array class
 *   returns:     <nothing>
 *=================================================================*/

static void
compileNewArray(ARRAY_CLASS type)
{
    /* Create an array */
    compilePtrToReg(type, 3);
    compilePopReg(4);   /* arrayLength */
    compileSaveVM();
    compileCall(instantiateArray);

    /* Push new array on stack */
    compilePushReg(3);
}

/*==================================================================
 * FUNCTION:      compileMultiANewArray()
 * TYPE:          Native code generation
 * OVERVIEW:      Generates native code for MULTIANEWARRAY opcode.
 *
 * INTERFACE:
 *   parameters:  thisClass Array class
 *                dimensions Number of dimensions
 *   returns:     <nothing>
 *=================================================================*/

static void
compileMultiANewArray(ARRAY_CLASS thisClass, int dimensions)
{
    /* Call C function to do the creation */
    compilePtrToReg(thisClass, 3);
    compileSaveVM();
    compileAddi(4, SP_REG, -(dimensions-1)*4);
    compileWordToReg(dimensions, 5);
    compileCall(instantiateMultiArray);

    /* Push new array on stack */
    compileAddi(4, SP_REG, -dimensions*4);
    compilePushReg(3);
}

/*=================================================================
 * FUNCTION:      tcacheReset()
 * TYPE:          Native code generation
 * OVERVIEW:      Resets the tcache.
 *
 * INTERFACE:
 *   parameters:  none
 *   returns:     <nothing>
 *================================================================*/

void tcacheReset()
{
    void *handleException;
    tcachePos = 0;
    /* Need to reset the links to native code */
    FOR_ALL_CLASSES(clazze)
        if (!IS_ARRAY_CLASS(clazze)) {
            FOR_EACH_METHOD(
                meth, ((INSTANCE_CLASS)clazze)->methodTable)
            if (meth->u.java.basicBlocks != NIL) {
                int i;
                BASICBLOCK bb;
                for (i=0; i<meth->u.java.basicBlocks->length; i++) {
                    bb = &meth->u.java.basicBlocks->basicBlocks[i];
                    bb->nativeCode = interpResume;
                    bb->counter = COMPILER_TRESHOLD(bb);
                }
            }
            END_FOR_EACH_METHOD
        }
    END_FOR_ALL_CLASSES

    /* Compile common code that handles exceptions */
    handleException = tcacheAddress();
    compileStw(IP_REG, 0, GS_REG);    /* GlobalState.gs_ip */
    compileStw(SP_REG, 4, GS_REG);    /* GlobalState.gs_sp */
    compileStw(LP_REG, 8, GS_REG);    /* GlobalState.gs_lp */
    compileStw(FP_REG, 12, GS_REG);   /* GlobalState.gs_fp */
    /* Argument (const char *) must be in GPR3 already */
    compileCall(raiseException);

    handleNullPointerException = tcacheAddress();
    compilePtrToReg(NullPointerException, 3);
    compileBranch(handleException, FALSE);

    handleArrayIndexOutOfBoundsException = tcacheAddress();
    compilePtrToReg(ArrayIndexOutOfBoundsException, 3);
    compileBranch(handleException, FALSE);

    handleClassCastException = tcacheAddress();
    compilePtrToReg(ClassCastException, 3);
    compileBranch(handleException, FALSE);

    handleArithmeticException = tcacheAddress();
    compilePtrToReg(ArithmeticException, 3);
    compileBranch(handleException, FALSE);
}

/*================================================================
 * FUNCTION:      dumpNativeCode()
 * TYPE:          Debugging
 * OVERVIEW:      Dumps the generated native code to a file.
 *
 * INTERFACE:
 *   parameters:  filename: File to dump to
 *   returns:     <nothing>
 *===============================================================*/

void
dumpNativeCode(const char *filename)
{
    FILE *fp = fopen(filename, "wb");
    fwrite(tcache, sizeof(int), tcachePos, fp);
    fclose(fp);
}

/*================================================================
 *===============================================================*/

#if INCLUDEDEBUGCODE

static void
compilationTrace(METHOD method, BASICBLOCK bb)
{
        char className[256];
        char methodSignature[256];
        char *methodName = methodName(method);
        getClassName_inBuffer((CLASS)method->ofClass, className);
        change_Key_to_MethodSignature_inBuffer(
            method->nameTypeKey.nt.typeKey,
            methodSignature);

        /* Print name of the class, method name & signature */
        fprintf(stdout, "compiling: %s.%s%s %d\n", className,
            methodName, methodSignature, bb->startIp);
        fflush(stdout);
}

static const char* const byteCodeNames[] = BYTE_CODE_NAMES;

#endif    /* INCLUDEDEBUGCODE */

/*=================================================================
 * FUNCTION:      compileBasicBlock()
 * TYPE:
 * OVERVIEW:      Compiles a basic block into native code.
 *                This is the top-level compiler function that is
 *                exported.
 *
 * INTERFACE:
 *   parameters:  method; Method that basic block is a part of
 *                basicBlock: Basic block to compile
 *                resume: Where native code should goto to get
 *                        back to interpreter
 *   returns:     Sets basicBlock->nativeCode to address of
 *                native code.
 *===============================================================*/

void
compileBasicBlock(METHOD method, BASICBLOCK basicBlock,
    void *resume)
{
    BYTE op;
    BYTE *nextIp;
    BYTE *targetIp;
    float floatConst;
    double doubleConst;
    bool_t notDone;

#if ENABLEPROFILING
    /* Prepare to count the number of instructions compiled */
    int bcCounter = 0;
#endif /* ENABLEPROFILING */

#if INCLUDEDEBUGCODE
    if (tracecompilation || tracecompilationverbose) {
        compilationTrace(method, basicBlock);
    }
#endif /* INCLUDEDEBUGCODE */

    /* Initialize some variables */
    interpResume = resume;    /* NB: Global variable */
    relSp = 0;                /* NB: Global variable */
    nextIp = methodCode(method) + basicBlock->startIp;
    savedIp = nextIp;
    notDone = TRUE;
    /* Record native code address for basic block */
    basicBlock->nativeCode = tcacheAddress();
    /* Translate all instructions in block */
    TRY
    while (notDone) {
        /* Get pointer to next bytecode to compile */
        thisIp = nextIp;
#if ENABLEPROFILING
        bcCounter++;
#endif /* ENABLEPROFILING */
#if INCLUDEDEBUGCODE
        if (tracecompilationverbose) {
            fprintf(stdout, "  %d: %s\n",
           thisIp - method->u.java.code, byteCodeNames[*thisIp]);
            fflush(stdout);
        }
#endif /* INCLUDEDEBUGCODE */
        switch (op = *thisIp) {
/* ------------------------------------------------------------ */

        case NOP:   /* Do nothing */
        break;

/* ------------------------------------------------------------ */

        /* Push null reference onto the operand stack */
        case ACONST_NULL:
        compilePushWord(0);
        break;
```

```
/* ------------------------------------------------------------ */

        /* Push int constant -1 onto the operand stack */
        case ICONST_M1:
        compilePushWord(-1);
        break;

/* ------------------------------------------------------------ */

        /* Push int constant 0 onto the operand stack */
        case ICONST_0:
        compilePushWord(0);
        break;

/* ------------------------------------------------------------ */

        /* Push int constant 1 onto the operand stack */
        case ICONST_1:
        compilePushWord(1);
        break;

/* ------------------------------------------------------------ */

        /* Push int constant 2 onto the operand stack */
        case ICONST_2:
        compilePushWord(2);
        break;

/* ------------------------------------------------------------ */

        /* Push int constant 3 onto the operand stack */
        case ICONST_3:
        compilePushWord(3);
        break;

/* ------------------------------------------------------------ */

        /* Push int constant 4 onto the operand stack */
        case ICONST_4:
        compilePushWord(4);
        break;

/* ------------------------------------------------------------ */

        /* Push int constant 5 onto the operand stack */
        case ICONST_5:
        compilePushWord(5);
        break;

/* ------------------------------------------------------------ */

        /* Push long constant 0 onto the operand stack */
        case LCONST_0:
        compilePushWord(0);
        compilePushWord(0);
        break;

/* ------------------------------------------------------------ */

        /* Push long constant 1 onto the operand stack */
        case LCONST_1:
        compilePushWord(0);
        compilePushWord(1);
        break;

/* ------------------------------------------------------------ */

        /* Push float constant 0 onto the operand stack */
        case FCONST_0:
        floatConst = 0.0f;
        compilePushWord(*(int *)&floatConst);
        break;

/* ------------------------------------------------------------ */

        /* Push float constant 1 onto the operand stack */
        case FCONST_1:
        floatConst = 1.0f;
        compilePushWord(*(int *)&floatConst);
        break;

/* ------------------------------------------------------------ */

        /* Push float constant 2 onto the operand stack */
        case FCONST_2:
        floatConst = 2.0f;
        compilePushWord(*(int *)&floatConst);
        break;

/* ------------------------------------------------------------ */
```

```
        /* Push double constant 0 onto the operand stack */
        case DCONST_0:
        doubleConst = 0.0;
        compilePushWord(((int *)&doubleConst)[0]);
        compilePushWord(((int *)&doubleConst)[1]);
        break;

/* ------------------------------------------------------------ */

        /* Push double constant 1 onto the operand stack */
        case DCONST_1:
        doubleConst = 1.0;
        compilePushWord(((int *)&doubleConst)[0]);
        compilePushWord(((int *)&doubleConst)[1]);
        break;

/* ------------------------------------------------------------ */

        /* Push byte constant onto the operand stack */
        case BIPUSH:
        compilePushWord(((signed char *)thisIp)[1]);
        break;

/* ------------------------------------------------------------ */

        /* Push short constant onto the operand stack */
        case SIPUSH:
        compilePushWord(getShort(thisIp+1));
        break;

/* ------------------------------------------------------------ */

        /* Push constant pool item onto the operand stack */
        case LDC:
        {
        unsigned int cpIndex;
        CONSTANTPOOL_ENTRY thisEntry;
        cpIndex = thisIp[1];
        thisEntry = &method->ofClass->constPool->entries[cpIndex];
        compilePushWord(thisEntry->integer);
        }
        break;

/* ------------------------------------------------------------ */
```

```
        /* Push constant pool item onto the operand stack */
        case LDC_W:
        {
        unsigned int cpIndex;
        CONSTANTPOOL_ENTRY thisEntry;
        cpIndex = getUShort(thisIp+1);
        thisEntry = &method->ofClass->constPool->entries[cpIndex];
        compilePushWord(thisEntry->integer);
        }
        break;

/* ----------------------------------------------------------- */

        /* Push constant pool item onto the operand stack */
        case LDC2_W:
        {
        unsigned int cpIndex;
        CONSTANTPOOL_ENTRY thisEntry;
        cpIndex = getUShort(thisIp+1);
        thisEntry = &method->ofClass->constPool->entries[cpIndex];
        compilePushWord(thisEntry[0].integer);
        compilePushWord(thisEntry[1].integer);
        }
        break;

/* ----------------------------------------------------------- */

        case ILOAD:     /* Load integer from local variable */
        compilePushLocal(thisIp[1]);
        break;

/* ----------------------------------------------------------- */

        case LLOAD:     /* Load long from local variable */
        compilePushLocal(thisIp[1]);
        compilePushLocal(thisIp[1]+1);
        break;

/* ----------------------------------------------------------- */

        case FLOAD:     /* Load float from local variable */
        compilePushLocal(thisIp[1]);
        break;
```

```
/* ---------------------------------------------------------- */

        case DLOAD:     /* Load double from local variable */
        compilePushLocal(thisIp[1]);
        compilePushLocal(thisIp[1]+1);
        break;

/* ---------------------------------------------------------- */

        case ALOAD:     /* Load reference from local variable */
        compilePushLocal(thisIp[1]);
        break;

/* ---------------------------------------------------------- */

        case ILOAD_0:   /* Load integer from local variable 0 */
        compilePushLocal(0);
        break;

/* ---------------------------------------------------------- */

        case ILOAD_1:   /* Load integer from local variable 1 */
        compilePushLocal(1);
        break;

/* ---------------------------------------------------------- */

        case ILOAD_2:   /* Load integer from local variable 2 */
        compilePushLocal(2);
        break;

/* ---------------------------------------------------------- */

        case ILOAD_3:   /* Load integer from local variable 3 */
        compilePushLocal(3);
        break;

/* ---------------------------------------------------------- */

        case LLOAD_0:   /* Load long from local variable 0 */
        compilePushLocal(0);
        compilePushLocal(1);
        break;

/* ---------------------------------------------------------- */
```

```
        case LLOAD_1:   /* Load long from local variable 1 */
        compilePushLocal(1);
        compilePushLocal(2);
        break;

/* ------------------------------------------------------------ */

        case LLOAD_2:   /* Load long from local variable 2 */
        compilePushLocal(2);
        compilePushLocal(3);
        break;

/* ------------------------------------------------------------ */

        case LLOAD_3:   /* Load long from local variable 3 */
        compilePushLocal(3);
        compilePushLocal(4);
        break;

/* ------------------------------------------------------------ */

        case FLOAD_0:   /* Load float from local variable 0 */
        compilePushLocal(0);
        break;

/* ------------------------------------------------------------ */

        case FLOAD_1:   /* Load float from local variable 1 */
        compilePushLocal(1);
        break;

/* ------------------------------------------------------------ */

        case FLOAD_2:   /* Load float from local variable 2 */
        compilePushLocal(2);
        break;

/* ------------------------------------------------------------ */

        case FLOAD_3:   /* Load float from local variable 3 */
        compilePushLocal(3);
        break;

/* ------------------------------------------------------------ */
```

```
        case DLOAD_0:   /* Load double from local variable 0 */
        compilePushLocal(0);
        compilePushLocal(1);
        break;

/* ------------------------------------------------------------ */

        case DLOAD_1:   /* Load double from local variable 1 */
        compilePushLocal(1);
        compilePushLocal(2);
        break;

/* ------------------------------------------------------------ */

        case DLOAD_2:   /* Load double from local variable 2 */
        compilePushLocal(2);
        compilePushLocal(3);
        break;

/* ------------------------------------------------------------ */

        case DLOAD_3:   /* Load double from local variable 3 */
        compilePushLocal(3);
        compilePushLocal(4);
        break;

/* ------------------------------------------------------------ */

        case ALOAD_0:   /* Load reference from local variable 0 */
        nextIp += compileAload_0();
        break;

/* ------------------------------------------------------------ */

        case ALOAD_1:   /* Load reference from local variable 1 */
        compilePushLocal(1);
        break;

/* ------------------------------------------------------------ */

        case ALOAD_2:   /* Load reference from local variable 2 */
        compilePushLocal(2);
        break;
```

```
/* ------------------------------------------------------------ */

        case ALOAD_3:   /* Load reference from local variable 3 */
        compilePushLocal(3);
        break;

/* ------------------------------------------------------------ */

        case IALOAD:    /* Load integer from array */
        compileArrayLoadProlog(4, 3, 3, 2);
        compileLwzx(3, 4, 3);   /* Load the int element */
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case LALOAD:    /* Load long from array */
        compileArrayLoadProlog(4, 3, 3, 3);
        compileLwzx(0, 4, 3);   /* Load the high word of element */
        compilePushReg(0);
        compileAddi(3, 3, 4);
        compileLwzx(3, 4, 3);   /* Load the low word of element */
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case FALOAD:    /* Load float from array */
        compileArrayLoadProlog(4, 3, 3, 2);
        compileLwzx(3, 4, 3);   /* Load the float element */
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case DALOAD:    /* Load double from array */
        compileArrayLoadProlog(4, 3, 3, 3);
        compileLwzx(0, 4, 3);   /* Load the high word of element */
        compilePushReg(0);
        compileAddi(3, 3, 4);
        compileLwzx(3, 4, 3);   /* Load the low word of element */
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */
```

```
        case AALOAD:    /* Load reference from array */
        compileArrayLoadProlog(4, 3, 3, 2);
        compileLwzx(3, 4, 3);   /* Load the reference element */
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case BALOAD:    /* Load byte from array */
        compileArrayLoadProlog(4, 3, 3, 0);
        compileLbzx(3, 4, 3);   /* Load the byte element */
        compileExtsb(3, 3);     /* Sign-extend */
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case CALOAD:    /* Load char from array */
        compileArrayLoadProlog(4, 3, 3, 1);
        compileLhzx(3, 4, 3);   /* Load the char element */
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case SALOAD:    /* Load short from array */
        compileArrayLoadProlog(4, 3, 3, 1);
        compileLhax(3, 4, 3);   /* Load the short element */
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case ISTORE:    /* Store integer into local variable */
        compilePopLocal(thisIp[1]);
        break;

/* ------------------------------------------------------------ */

        case LSTORE:    /* Store long into local variable */
        compilePopLocal(thisIp[1]+1);
        compilePopLocal(thisIp[1]);
        break;
```

```
/* ------------------------------------------------------------ */

        case FSTORE:    /* Store float into local variable */
        compilePopLocal(thisIp[1]);
        break;

/* ------------------------------------------------------------ */

        case DSTORE:    /* Store double into local variable */
        compilePopLocal(thisIp[1]+1);
        compilePopLocal(thisIp[1]);
        break;

/* ------------------------------------------------------------ */

        case ASTORE:    /* Store reference into local variable */
        compilePopLocal(thisIp[1]);
        break;

/* ------------------------------------------------------------ */

        case ISTORE_0:  /* Store integer into local variable 0 */
        compilePopLocal(0);
        break;

/* ------------------------------------------------------------ */

        case ISTORE_1:  /* Store integer into local variable 1 */
        compilePopLocal(1);
        break;

/* ------------------------------------------------------------ */

        case ISTORE_2:  /* Store integer into local variable 2 */
        compilePopLocal(2);
        break;

/* ------------------------------------------------------------ */

        case ISTORE_3:  /* Store integer into local variable 3 */
        compilePopLocal(3);
        break;

/* ------------------------------------------------------------ */
```

```
        case LSTORE_0:  /* Store long into local variable 0 */
        compilePopLocal(1);
        compilePopLocal(0);
        break;

/* ------------------------------------------------------------ */

        case LSTORE_1:  /* Store long into local variable 1 */
        compilePopLocal(2);
        compilePopLocal(1);
        break;

/* ------------------------------------------------------------ */

        case LSTORE_2:  /* Store long into local variable 2 */
        compilePopLocal(3);
        compilePopLocal(2);
        break;

/* ------------------------------------------------------------ */

        case LSTORE_3:  /* Store long into local variable 3 */
        compilePopLocal(4);
        compilePopLocal(3);
        break;

/* ------------------------------------------------------------ */

        case FSTORE_0:  /* Store float into local variable 0 */
        compilePopLocal(0);
        break;

/* ------------------------------------------------------------ */

        case FSTORE_1:  /* Store float into local variable 1 */
        compilePopLocal(1);
        break;

/* ------------------------------------------------------------ */

        case FSTORE_2:  /* Store float into local variable 2 */
        compilePopLocal(2);
        break;

/* ------------------------------------------------------------ */
```

```
        case FSTORE_3:  /* Store float into local variable 3 */
        compilePopLocal(3);
        break;

/* ------------------------------------------------------------ */

        case DSTORE_0:  /* Store double into local variable 0 */
        compilePopLocal(1);
        compilePopLocal(0);
        break;

/* ------------------------------------------------------------ */

        case DSTORE_1:  /* Store double into local variable 1 */
        compilePopLocal(2);
        compilePopLocal(1);
        break;

/* ------------------------------------------------------------ */

        case DSTORE_2:  /* Store double into local variable 2 */
        compilePopLocal(3);
        compilePopLocal(2);
        break;

/* ------------------------------------------------------------ */

        case DSTORE_3:  /* Store double into local variable 3 */
        compilePopLocal(4);
        compilePopLocal(3);
        break;

/* ------------------------------------------------------------ */

        case ASTORE_0:  /* Store reference into local variable 0 */
        compilePopLocal(0);
        break;

/* ------------------------------------------------------------ */

        case ASTORE_1:  /* Store reference into local variable 1 */
        compilePopLocal(1);
        break;
```

```
/* ---------------------------------------------------------- */

        case ASTORE_2:  /* Store reference into local variable 2 */
        compilePopLocal(2);
        break;

/* ---------------------------------------------------------- */

        case ASTORE_3:  /* Store reference into local variable 3 */
        compilePopLocal(3);
        break;

/* ---------------------------------------------------------- */

        case IASTORE:   /* Store into integer array */
        compileArrayStoreProlog(5, 4, 4, 3, 2);
        compileStwx(3, 5, 4);   /* Store the element */
        break;

/* ---------------------------------------------------------- */

        case LASTORE:   /* Store into long array */
        compilePopReg(3);
        compileArrayStoreProlog(6, 5, 5, 4, 3);
        compileStwx(4, 6, 5);   /* Store the high word */
        compileAddi(5, 5, 4);
        compileStwx(3, 6, 5);   /* Store the low word */
        break;

/* ---------------------------------------------------------- */

        case FASTORE:   /* Store into float array */
        compileArrayStoreProlog(5, 4, 4, 3, 2);
        compileStwx(3, 5, 4);   /* Store the element */
        break;

/* ---------------------------------------------------------- */

        case DASTORE:   /* Store into double array */
        compilePopReg(3);
        compileArrayStoreProlog(6, 5, 5, 4, 3);
        compileStwx(4, 6, 5);   /* Store the high word */
        compileAddi(5, 5, 4);
        compileStwx(3, 6, 5);   /* Store the low word */
        break;
```

```
/* -------------------------------------------------------------- */

       case AASTORE:   /* Store into reference array */
       compileArrayStoreProlog(5, 4, 4, 3, 2);
       compileStwx(3, 5, 4);   /* Store the element */
       break;

/* -------------------------------------------------------------- */

       case BASTORE:   /* Store into byte array */
       compileArrayStoreProlog(5, 4, 4, 3, 0);
       compileStbx(3, 5, 4);   /* Store the element */
       break;

/* -------------------------------------------------------------- */

       case CASTORE:   /* Store into char array */
       compileArrayStoreProlog(5, 4, 4, 3, 1);
       compileSthx(3, 5, 4);   /* Store the element */
       break;

/* -------------------------------------------------------------- */

       case SASTORE:   /* Store into short array */
       compileArrayStoreProlog(5, 4, 4, 3, 1);
       compileSthx(3, 5, 4);   /* Store the element */
       break;

/* -------------------------------------------------------------- */

       case POP:       /* Pop top operand stack word */
       relSp--;
       break;

/* -------------------------------------------------------------- */

       case POP2:      /* Pop two top operand stack words */
       relSp--;
       relSp--;
       break;

/* -------------------------------------------------------------- */

       case DUP:       /* Duplicate top operand stack word */
```

```
        compileTopStackToReg(3);
        compilePushReg(3);
        break;

/* ----------------------------------------------------------- */

        case DUP_X1:    /* Duplicate top operand stack word, put two down  */
        compileTopStackToReg(3);
        compileSecondStackToReg(4);
        compileRegToTopStack(4);
        compileRegToSecondStack(3);
        compilePushReg(3);
        break;

/* ----------------------------------------------------------- */

        case DUP_X2:    /* Duplicate top operand stack word, put three down */
        compileTopStackToReg(3);
        compileSecondStackToReg(4);
        compileThirdStackToReg(5);
        compileRegToThirdStack(3);
        compileRegToSecondStack(5);
        compileRegToTopStack(4);
        compilePushReg(3);
        break;

/* ----------------------------------------------------------- */

        case DUP2:      /* Duplicate top two operand stack words */
        compileSecondStackToReg(3);
        compilePushReg(3);
        compileSecondStackToReg(3);
        compilePushReg(3);
        break;

/* ----------------------------------------------------------- */

        case DUP2_X1:   /* Duplicate top 2 operand stack words, put 3 down */
        compileTopStackToReg(3);
        compileSecondStackToReg(4);
        compileThirdStackToReg(5);
        compileRegToThirdStack(4);
        compileRegToSecondStack(3);
        compileRegToTopStack(5);
        compilePushReg(4);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case DUP2_X2:   /* Duplicate top 2 operand stack words, put 4 down *
        compileTopStackToReg(3);
        compileSecondStackToReg(4);
        compileThirdStackToReg(5);
        compileFourthStackToReg(6);
        compileRegToFourthStack(4);
        compileRegToThirdStack(3);
        compileRegToSecondStack(6);
        compileRegToTopStack(5);
        compilePushReg(4);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case SWAP:      /* Swap top two operand stack words */
        compileTopStackToReg(3);
        compileSecondStackToReg(4);
        compileRegToTopStack(4);
        compileRegToSecondStack(3);
        break;

/* ------------------------------------------------------------ */

        case IADD:      /* Add integer */
        compilePopReg(3);
        compilePopReg(4);
        compileAdd(3, 4, 3);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case LADD:      /* Add long */
        compilePopReg(3);
        compilePopReg(4);
        compilePopReg(5);
        compilePopReg(6);
        compileAddc(3, 5, 3);
        compileAdde(4, 6, 4);
        compilePushReg(4);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case FADD:      /* Add float */
        compilePopFregs(1);
        compilePopFregs(0);
        compileFadds(1, 0, 1);
        compilePushFregs(1);
        break;

/* ------------------------------------------------------------ */

        case DADD:      /* Add double */
        compilePopFregd(1);
        compilePopFregd(0);
        compileFadd(1, 0, 1);
        compilePushFregd(1);
        break;

/* ------------------------------------------------------------ */

        case ISUB:      /* Subtract integer */
        compilePopReg(3);
        compilePopReg(4);
        compileSub(3, 4, 3);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case LSUB:      /* Subtract long */
        compilePopReg(3);
        compilePopReg(4);
        compilePopReg(5);
        compilePopReg(6);
        compileSubc(3, 5, 3);
        compileSube(4, 6, 4);
        compilePushReg(4);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */
```

```
        case FSUB:      /* Subtract float */
        compilePopFregs(1);
        compilePopFregs(0);
        compileFsubs(1, 0, 1);
        compilePushFregs(1);
        break;

/* ------------------------------------------------------------ */

        case DSUB:      /* Subtract double */
        compilePopFregd(1);
        compilePopFregd(0);
        compileFsub(1, 0, 1);
        compilePushFregd(1);
        break;

/* ------------------------------------------------------------ */

        case IMUL:      /* Multiply integer */
        compilePopReg(3);
        compilePopReg(4);
        compileMullw(3, 4, 3);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case LMUL:      /* Multiply long */
        compilePopReg(6);
        compilePopReg(5);    /* Arg 2 in 5:6 */
        compilePopReg(4);
        compilePopReg(3);    /* Arg 1 in 3:4 */
        compileCall(__muldi3);    /* Returns result in 3:4 */
        compilePushReg(3);
        compilePushReg(4);
        break;

/* ------------------------------------------------------------ */

        case FMUL:      /* Multiply float */
        compilePopFregs(1);
        compilePopFregs(0);
        compileFmuls(1, 0, 1);
        compilePushFregs(1);
        break;

/* ------------------------------------------------------------ */

        case DMUL:      /* Multiply double */
        compilePopFregd(1);
        compilePopFregd(0);
        compileFmul(1, 0, 1);
        compilePushFregd(1);
        break;

/* ------------------------------------------------------------ */

        case IDIV:      /* Divide integer */
        compilePopReg(3);
        compileUpdateVM();
        compileCmpwi(3, 0);
        compileBeq(tcacheDistance(handleArithmeticException));
        compilePopReg(4);
        compileDivw(3, 4, 3);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case LDIV:      /* Divide long */
        compilePopReg(6);
        compilePopReg(5);    /* Arg 2 in 5:6 */
        compileUpdateVM();
        compileOr(0, 5, 6);
        compileCmpwi(0, 0);
        compileBeq(tcacheDistance(handleArithmeticException));
        compilePopReg(4);
        compilePopReg(3);    /* Arg 1 in 3:4 */
        compileCall(__divdi3);    /* Returns result in 3:4 */
        compilePushReg(3);
        compilePushReg(4);
        break;

/* ------------------------------------------------------------ */

        case FDIV:      /* Divide float */
        compilePopFregs(1);
        compilePopFregs(0);
        compileFdivs(1, 0, 1);
        compilePushFregs(1);
        break;

/* ----------------------------------------------------------- */

        case DDIV:      /* Divide double */
        compilePopFregd(1);
        compilePopFregd(0);
        compileFdiv(1, 0, 1);
        compilePushFregd(1);
        break;

/* ----------------------------------------------------------- */

        case IREM:      /* Remainder integer */
        compilePopReg(3);
        compileUpdateVM();
        compileCmpwi(3, 0);
        compileBeq(tcacheDistance(handleArithmeticException));
        compilePopReg(4);
        compileDivw(0, 4, 3);
        compileMullw(0, 0, 3);
        compileSub(3, 4, 0);
        compilePushReg(3);
        break;

/* ----------------------------------------------------------- */

        case LREM:      /* Remainder long */
        compilePopReg(6);
        compilePopReg(5);    /* Arg 2 in 5:6 */
        compileUpdateVM();
        compileOr(0, 5, 6);
        compileCmpwi(0, 0);
        compileBeq(tcacheDistance(handleArithmeticException));
        compilePopReg(4);
        compilePopReg(3);    /* Arg 1 in 3:4 */
        compileCall(__moddi3);    /* Returns result in 3:4 */
        compilePushReg(3);
        compilePushReg(4);
        break;

/* ----------------------------------------------------------- */

        case FREM:      /* Remainder float */
        compilePopFregs(1);     /* Arg 1 */
        compileCall(fmod);     /* Returns result in 1 */
        compileFrsp(1, 1);
        compilePushFregs(1);
        break;

/* ----------------------------------------------------------- */

        case DREM:      /* Remainder double */
        compilePopFregd(1);     /* Arg 1 */
        compileCall(fmod);     /* Returns result in 1 */
        compilePushFregd(1);
        break;

/* ----------------------------------------------------------- */

        case INEG:      /* Negate integer */
        compilePopReg(3);
        compileNeg(3, 3);
        compilePushReg(3);
        break;

/* ----------------------------------------------------------- */

        case LNEG:      /* Negate long */
        compilePopReg(3);
        compilePopReg(4);
        compileNot(3, 3);
        compileAddic(3, 3, 1);
        compileNot(4, 4);
        compileAddze(4, 4);
        compilePushReg(4);
        compilePushReg(3);
        break;

/* ----------------------------------------------------------- */

        case FNEG:      /* Negate float */
        compilePopFregs(1);
        compileFneg(1, 1);
        compilePushFregs(1);
        break;

/* ----------------------------------------------------------- */
```

```
        case DNEG:      /* Negate double */
        compilePopFregd(1);
        compileFneg(1, 1);
        compilePushFregd(1);
        break;

/* ------------------------------------------------------------ */

        case ISHL:      /* Shift left integer */
        compilePopReg(3);
        compilePopReg(4);
        compileAndi(3, 3, 31);
        compileSlw(3, 4, 3);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case LSHL:      /* Shift left long */
        compilePopReg(5);    /* Arg 2 in 5 */
        compileAndi(5, 5, 63);
        compilePopReg(4);
        compilePopReg(3);    /* Arg 1 in 3:4 */
        compileCall(__ashldi3);    /* Returns result in 3:4 */
        compilePushReg(3);
        compilePushReg(4);
        break;

/* ------------------------------------------------------------ */

        case ISHR:      /* Shift right integer */
        compilePopReg(3);
        compilePopReg(4);
        compileAndi(3, 3, 31);
        compileSraw(3, 4, 3);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case LSHR:      /* Shift right long */
        compilePopReg(5);    /* Arg 2 in 5 */
        compileAndi(5, 5, 63);
        compilePopReg(4);
        compilePopReg(3);    /* Arg 1 in 3:4 */
        compileCall(__ashrdi3);    /* Returns result in 3:4 */
        compilePushReg(3);
        compilePushReg(4);
        break;

/* ------------------------------------------------------------ */

        case IUSHR:     /* Shift right unsigned integer */
        compilePopReg(3);
        compilePopReg(4);
        compileAndi(3, 3, 31);
        compileSrw(3, 4, 3);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case LUSHR:     /* Shift right unsigned long */
        compilePopReg(5);    /* Arg 2 in 5 */
        compileAndi(5, 5, 63);
        compilePopReg(4);
        compilePopReg(3);    /* Arg 1 in 3:4 */
        compileCall(__lshrdi3);    /* Returns result in 3:4 */
        compilePushReg(3);
        compilePushReg(4);
        break;

/* ------------------------------------------------------------ */

        case IAND:      /* Logical AND integer */
        compilePopReg(3);
        compilePopReg(4);
        compileAnd(3, 4, 3);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case LAND:      /* Logical AND long */
        compilePopReg(3);
        compilePopReg(4);
        compilePopReg(5);
        compilePopReg(6);
        compileAnd(3, 3, 5);
        compileAnd(4, 4, 6);
        compilePushReg(4);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case IOR:       /* Logical OR integer */
        compilePopReg(3);
        compilePopReg(4);
        compileOr(3, 4, 3);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case LOR:       /* Logical OR long */
        compilePopReg(3);
        compilePopReg(4);
        compilePopReg(5);
        compilePopReg(6);
        compileOr(3, 3, 5);
        compileOr(4, 4, 6);
        compilePushReg(4);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case IXOR:      /* Logical XOR integer */
        compilePopReg(3);
        compilePopReg(4);
        compileXor(3, 4, 3);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case LXOR:      /* Logical XOR long */
        compilePopReg(3);
        compilePopReg(4);
        compilePopReg(5);
        compilePopReg(6);
        compileXor(3, 3, 5);
        compileXor(4, 4, 6);
        compilePushReg(4);
        compilePushReg(3);
        break;

/* ----------------------------------------------------------- */

        case IINC:      /* Increment local variable */
        compileLocalToReg(thisIp[1], 3);
        compileAddi(3, 3, (signed char)thisIp[2]);
        compileRegToLocal(3, thisIp[1]);
        break;

/* ----------------------------------------------------------- */

        case I2L:       /* Convert integer to long */
        compileTopStackToReg(3);
        compilePushReg(3);
        compileSrawi(3, 3, 31);
        compileRegToSecondStack(3);
        break;

/* ----------------------------------------------------------- */

        case I2F:       /* Convert integer to float */
        compilePopReg(3);
        compileWordToReg(0x43300000, 0);
        compileStw(0, (relSp+1)*4, SP_REG);
        compileWordToReg(0x80000000, 0);
        compileStw(0, (relSp+2)*4, SP_REG);
        compileLfd(0, (relSp+1)*4, SP_REG);
        compileXor(3, 3, 0);
        compileStw(3, (relSp+2)*4, SP_REG);
        compileLfd(1, (relSp+1)*4, SP_REG);
        compileFsub(1, 1, 0);
        compileFrsp(1, 1);
        compilePushFregs(1);
        break;

/* ----------------------------------------------------------- */

        case I2D:       /* Convert integer to double */
        compilePopReg(3);
        compileWordToReg(0x43300000, 0);
        compileStw(0, (relSp+1)*4, SP_REG);
        compileWordToReg(0x80000000, 0);
        compileStw(0, (relSp+2)*4, SP_REG);
        compileLfd(0, (relSp+1)*4, SP_REG);
        compileXor(3, 3, 0);
        compileStw(3, (relSp+2)*4, SP_REG);
        compileLfd(1, (relSp+1)*4, SP_REG);
        compileFsub(1, 1, 0);
        compilePushFregd(1);
        break;

/* ------------------------------------------------------------ */

        case L2I:       /* Convert long to integer */
        compilePopReg(3);
        --relSp;        /* Ditch high word */
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case L2F:       /* Convert long to float */
        compilePopReg(4);
        compilePopReg(3);    /* Arg 1 in 3:4 */
        compileCall(__floatdisf);    /* Returns result in 1 */
        compilePushFregs(1);
        break;

/* ------------------------------------------------------------ */

        case L2D:       /* Convert long to double */
        compilePopReg(4);
        compilePopReg(3);    /* Arg 1 in 3:4 */
        compileCall(__floatdidf);    /* Returns result in 1 */
        compilePushFregd(1);
        break;

/* ------------------------------------------------------------ */

        case F2I:       /* Convert float to integer */
        compilePopFregs(1);
        compileFctiwz(1, 1);
        compilePushFregd(1);
        compilePopReg(3);
        relSp--;
        compilePushReg(3);
        break;
```

```
/* ------------------------------------------------------------- */

        case F2L:       /* Convert float to long */
        compilePopFregs(1);    /* Arg 1 in 1 */
        compileCall(__fixsfdi);    /* Returns result in 3:4 */
        compilePushReg(3);
        compilePushReg(4);
        break;

/* ------------------------------------------------------------- */

        case F2D:       /* Convert float to double */
        compilePopFregs(1);
        compilePushFregd(1);
        break;

/* ------------------------------------------------------------- */

        case D2I:       /* Convert double to integer */
        compilePopFregd(1);
        compileFctiwz(1, 1);
        compilePushFregd(1);
        compilePopReg(3);
        relSp--;
        compilePushReg(3);
        break;

/* ------------------------------------------------------------- */

        case D2L:       /* Convert double to long */
        compilePopFregd(1);    /* Arg 1 in 1 */
        compileCall(__fixdfdi);    /* Returns result in 3:4 */
        compilePushReg(3);
        compilePushReg(4);
        break;

/* ------------------------------------------------------------- */

        case D2F:       /* Convert double to float */
        compilePopFregd(1);
        compileFrsp(1, 1);
        compilePushFregs(1);
        break;

/* ------------------------------------------------------------- */
```

```
        case I2B:       /* Convert integer to byte */
        compileTopStackToReg(3);
        compileExtsb(3, 3);
        compileRegToTopStack(3);
        break;

/* ------------------------------------------------------------ */

        case I2C:       /* Convert integer to char */
        compileTopStackToReg(3);
        compileAndi(3, 3, 0xFFFF);
        compileRegToTopStack(3);
        break;

/* ------------------------------------------------------------ */

        case I2S:       /* Convert integer to short */
        compileTopStackToReg(3);
        compileExtsh(3, 3);
        compileRegToTopStack(3);
        break;

/* ------------------------------------------------------------ */

        case LCMP:      /* Compare long */
        compilePopReg(6);
        compilePopReg(5);    /* Arg 2 in 5:6 */
        compilePopReg(4);
        compilePopReg(3);    /* Arg 1 in 3:4 */
        compileCall(__cmpdi2);    /* Returns result in 3 */
        compileAddi(3, 3, -1);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case FCMPL:     /* Compare float */
        compilePopFregs(1);
        compilePopFregs(0);
        compileLi(3, 1);
        compileFcmpo(0, 1);
        compileBgt(4);
        compileAddi(3, 3, -1);
        compileBeq(2);
        compileAddi(3, 3, -1);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case FCMPG:     /* Compare float */
        compilePopFregs(1);
        compilePopFregs(0);
        compileLi(3, 1);
        compileFcmpo(0, 1);
        compileBun(5);
        compileBgt(4);
        compileAddi(3, 3, -1);
        compileBeq(2);
        compileAddi(3, 3, -1);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case DCMPL:     /* Compare double */
        compilePopFregd(1);
        compilePopFregd(0);
        compileLi(3, 1);
        compileFcmpo(0, 1);
        compileBgt(4);
        compileAddi(3, 3, -1);
        compileBeq(2);
        compileAddi(3, 3, -1);
        compilePushReg(3);
        break;

/* ------------------------------------------------------------ */

        case DCMPG:     /* Compare double */
        compilePopFregd(1);
        compilePopFregd(0);
        compileLi(3, 1);
        compileFcmpo(0, 1);
        compileBun(5);
        compileBgt(4);
        compileAddi(3, 3, -1);
        compileBeq(2);
        compileAddi(3, 3, -1);
        compilePushReg(3);
        break;

/* -------------------------------------------------------------- */

        case IFEQ:      /* Branch if equal to zero */
        {
        int pos;
        compileIfProlog(3, &pos);
        compileBeq(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;

/* -------------------------------------------------------------- */

        case IFNE:      /* Branch if not equal to zero */
        {
        int pos;
        compileIfProlog(3, &pos);
        compileBne(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;

/* -------------------------------------------------------------- */

        case IFLT:      /* Branch if less than zero */
        {
        int pos;
        compileIfProlog(3, &pos);
        compileBlt(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;

/* -------------------------------------------------------------- */

        case IFGE:      /* Branch if greater or equal to zero */
        {
        int pos;
        compileIfProlog(3, &pos);
        compileBge(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case IFGT:      /* Branch if greater than zero */
        {
        int pos;
        compileIfProlog(3, &pos);
        compileBgt(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case IFLE:      /* Branch if less or equal zero */
        {
        int pos;
        compileIfProlog(3, &pos);
        compileBle(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case IF_ICMPEQ: /* Branch if equal */
        {
        int pos;
        compileIf_icmpProlog(3, 4, &pos);
        compileBeq(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case IF_ICMPNE: /* Branch if not equal */
        {
        int pos;
        compileIf_icmpProlog(3, 4, &pos);
        compileBne(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case IF_ICMPLT: /* Branch if less than */
        {
        int pos;
        compileIf_icmpProlog(3, 4, &pos);
        compileBlt(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case IF_ICMPGE: /* Branch if greater or equal */
        {
        int pos;
        compileIf_icmpProlog(3, 4, &pos);
        compileBge(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case IF_ICMPGT: /* Branch if greater than */
        {
        int pos;
        compileIf_icmpProlog(3, 4, &pos);
        compileBgt(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;
```

```
/* ------------------------------------------------------------ */

        case IF_ICMPLE: /* Branch if less or equal */
        {
        int pos;
        compileIf_icmpProlog(3, 4, &pos);
        compileBle(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case IF_ACMPEQ: /* Branch if references equal */
        {
        int pos;
        compileIf_icmpProlog(3, 4, &pos);
        compileBeq(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case IF_ACMPNE: /* Branch if references not equal */
        {
        int pos;
        compileIf_icmpProlog(3, 4, &pos);
        compileBne(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case GOTO:      /* Go to address */
        compileUpdateSp();
        targetIp = thisIp + getShort(thisIp + 1);
        compileBBStaticDispatch(method, targetIp);
        notDone = FALSE;
        break;
```

```
/* ------------------------------------------------------------ */

        case JSR:       /* Jump to subroutine */
        /* Not implemented */
        break;

/* ------------------------------------------------------------ */

        case RET:       /* Return from subroutine */
        /* Not implemented */
        break;

/* ------------------------------------------------------------ */

        case TABLESWITCH:    /* Access jump table by index and jump */
        {
        int j;
        long lowValue;
        long highValue;
        long numberOfCases;
        int pos1, pos2, pos3;
        BYTE *defaultIp;
        cell *base = (cell *)(((long)thisIp + 4) & ~3);
        /* Get default target. */
        defaultIp = thisIp + getAlignedCell(base);
        /* Get ranges. */
        lowValue = getAlignedCell(base+1);
        highValue = getAlignedCell(base+2);
        /* Pop value and range check. */
        compilePopReg(3);
        compileUpdateSp();
        compileCompareWord(3, lowValue);
        recordBackpatchLocation(&pos1);
        compileBlt(0);
        compileCompareWord(3, highValue);
        recordBackpatchLocation(&pos2);
        compileBgt(0);
        /* Skip table of BB pointers, table address will be in LR */
        recordBackpatchLocation(&pos3);
        compileBl(0);
        /* Create table of target basic blocks */
        numberOfCases = highValue-lowValue+1;
        for (j=0; j<numberOfCases; j++) {
            targetIp = thisIp + getAlignedCell(base+3+j);
            compileWord(basicBlockForIp(method, targetIp));
        }
        /* Backpatch the BL above */
        backpatchRelativeBranch(pos3);
        /* Make the index 0-based by subtracting lowValue */
        if (lowValue != 0) {
            if (IS_SHORT(-lowValue)) {
                compileAddi(3, 3, -lowValue);
            }
            else {
                compileWordToReg(-lowValue, 0);
                compileAdd(3, 3, 0);
            }
        }
        /* Fetch entry from table */
        compileMflr(0);        /* Table address */
        compileSlwi(3, 3, 2);  /* offset = index*4 */
        compileLwzx(3, 3, 0);  /* nativeCode */
        /* Dispatch */
        compileBBDynamicDispatch(method, 3);
        /* We come here if default address is used */
        backpatchConditionalBranch(pos2);
        backpatchConditionalBranch(pos1);
        compileBBStaticDispatch(method, defaultIp);
        }
        notDone = FALSE;
        break;

/* ---------------------------------------------------------- */

        /* Access jump table by key match and jump */
        case LOOKUPSWITCH:
        {
            /* Call C function that does the search
            in logarithmic time */
            compileGetThisMethod(3);    /* Arg 1 (METHOD) */
            compilePopReg(4);           /* Arg 2 (key) */
            compileUpdateSp();
            compilePtrToReg(thisIp, 5); /* Arg 3 (ip) */
            compileCall(lookupSwitch);  /* Returns BASICBLOCK in 3 */
            compileBBDynamicDispatch(method, 3);
        }
        notDone = FALSE;
        break;

/* ---------------------------------------------------------- */
```

```
        case IRETURN:   /* Return integer */
        compileReturn(method, 1);
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case LRETURN:   /* Return long */
        compileReturn(method, 2);
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case FRETURN:   /* Return float */
        compileReturn(method, 1);
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case DRETURN:   /* Return double */
        compileReturn(method, 2);
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case ARETURN:   /* Return reference */
        compileReturn(method, 1);
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case RETURN:    /* Return void */
        compileReturn(method, 0);
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case GETSTATIC: /* Load class field */
        compileGetStatic(method->ofClass, getUShort(thisIp + 1));
        break;

/* ------------------------------------------------------------ */

        case PUTSTATIC: /* Store class field */
        compilePutStatic(method->ofClass, getUShort(thisIp + 1));
        break;

/* ------------------------------------------------------------ */

        case GETFIELD:  /* Load instance field */
        compileGetField(method->ofClass, getUShort(thisIp + 1));
        break;

/* ------------------------------------------------------------ */

        case PUTFIELD:  /* Store instance field */
        compilePutField(method->ofClass, getUShort(thisIp + 1));
        break;

/* ------------------------------------------------------------ */

        case INVOKEVIRTUAL:
        {
        /* Invoke instance method; dispatch based on dynamic class */
        unsigned int cpIndex;
        METHOD cpMethod;

        /* Get the constant pool index */
        cpIndex = getUShort(thisIp + 1);

        /* Resolve constant pool reference */
        cpMethod = resolveMethodReference(
            method->ofClass->constPool, cpIndex, FALSE, method->ofClass);

        compileInvokeVirtual(cpMethod);
        notDone = FALSE;
        }
        break;

/* ------------------------------------------------------------ */

        case INVOKESPECIAL:
        {
        /* Invoke instance method; special handling for superclass, */
        /* private and instance initialization method invocations */
        unsigned int cpIndex;
        METHOD cpMethod;

        /* Get the constant pool index */
        cpIndex = getUShort(thisIp + 1);

        /* Resolve constant pool reference */
        cpMethod = resolveMethodReference(
            method->ofClass->constPool, cpIndex, FALSE, method->ofClass);

        notDone = compileInvokeSpecial(cpMethod);
        }
        break;

/* ------------------------------------------------------------ */

        case INVOKESTATIC:
        {
        /* Invoke class (static) method */
        unsigned int cpIndex;
        METHOD cpMethod;

        /* Get the constant pool index */
        cpIndex = getUShort(thisIp + 1);

        /* Resolve constant pool reference */
        cpMethod = resolveMethodReference(
            method->ofClass->constPool, cpIndex, TRUE, method->ofClass);

        notDone = compileInvokeStatic(cpMethod);
        }
        break;

/* ------------------------------------------------------------ */

        case INVOKEINTERFACE:
        {
        /* Invoke interface method */
        unsigned int cpIndex;
        METHOD cpMethod;
        int argCount;

        /* Get the constant pool index */
        cpIndex = getUShort(thisIp + 1);
        /* Resolve constant pool reference */
        cpMethod = resolveMethodReference(
            method->ofClass->constPool, cpIndex, FALSE, method->ofClass);

        /* Get the argument count (specific to INVOKEINTERFACE bytecode) */
        argCount = thisIp[3];

        compileInvokeInterface(cpMethod, method->ofClass, argCount);
        notDone = FALSE;
        }
        break;

/* ------------------------------------------------------------ */

        case UNUSED_BA:
        break;

/* ------------------------------------------------------------ */

        case NEW:       /* Create new object */
        {
        INSTANCE_CLASS thisClass;
        unsigned int cpIndex;
        /* Get the CONSTANT_Class index */
        cpIndex = getUShort(thisIp + 1);

        /* Get the corresponding class pointer */
        thisClass = (INSTANCE_CLASS)resolveClassReference(
            method->ofClass->constPool, cpIndex, method->ofClass);

        compileNew(thisClass);
        }
        break;

/* ------------------------------------------------------------ */

        case NEWARRAY:  /* Create new array object */
        {
        ARRAY_CLASS type;
        unsigned int arrayType = thisIp[1];
        type = PrimitiveArrayClasses[arrayType];

        compileNewArray(type);
        }
        break;

/* --------------------------------------------------------- */

        case ANEWARRAY: /* Create new array of reference type */
        {
        CLASS elemClass;
        ARRAY_CLASS thisClass;
        /* Get the CONSTANT_Class index */
        unsigned int cpIndex = getUShort(thisIp + 1);

        /* Get the corresponding class pointer */
        elemClass =
           resolveClassReference(
               method->ofClass->constPool, cpIndex, method->ofClass);
        thisClass = getObjectArrayClass(elemClass);

        compileNewArray(thisClass);
        }
        break;

/* --------------------------------------------------------- */

        case ARRAYLENGTH:   /* Get length of array */
        compilePopReg(3);   /* Array */
        compileUpdateVM();
        compileNullCheck(3);
        compileGetArrayLength(3, 3);
        compilePushReg(3);
        break;

/* --------------------------------------------------------- */

        case ATHROW:    /* Throw exception or error */
        compilePopReg(3);    /* THROWABLE_INSTANCE */
        compileUpdateVM();
        compileNullCheck(3);
        compilePtrToReg(thisObjectGCSafe, 4);
        compileStw(3, 0, 4);
        compileCall(throwException);
        notDone = FALSE;
        break;

/* --------------------------------------------------------- */
```

```
        case CHECKCAST: /* Check whether object is of given type */
        {
        CLASS thisClass;
        /* Get the CONSTANT_Class index */
        unsigned int cpIndex = getUShort(thisIp + 1);

        /* Get the corresponding class pointer */
        thisClass = resolveClassReference(
            method->ofClass->constPool, cpIndex, method->ofClass);

        compileCheckCast(thisClass);
        }
        break;

/* ---------------------------------------------------------- */

        case INSTANCEOF:    /* Check whether object is of given type */
        {
        CLASS thisClass;
        /* Get the CONSTANT_Class index */
        unsigned int cpIndex = getUShort(thisIp + 1);

        /* Get the corresponding class pointer */
        thisClass = resolveClassReference(
            method->ofClass->constPool, cpIndex, method->ofClass);

        compileInstanceOf(thisClass);
        }
        break;

/* ---------------------------------------------------------- */

        case MONITORENTER:  /* Enter a monitor */
        compilePopReg(3);
        compileUpdateVM();
        compileNullCheck(3);
        thisIp++;            /* Note monitorEnter can context switch */
        compileSaveVM();
        compileCall(monitorEnter);
        compileRestoreVM();
        /* Return to interpreter */
        compileResumeInterpreterReschedule();
        notDone = FALSE;
        break;
```

```
/* ------------------------------------------------------------ */

        case MONITOREXIT:   /* Exit a monitor */
        compilePopReg(3);
        compileUpdateVM();
        compileNullCheck(3);
        thisIp++;    /* In case we context switch in monitorExit one day */
        compileSaveVM();
        compileCall(exitTheMonitorPlease);
        compileRestoreVM();
        /* Return to interpreter */
        compileResumeInterpreterReschedule();
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case WIDE:
        switch (*(thisIp+1)) {

        case ILOAD:     /* Load integer from local variable (wide) */
        compilePushLocal(getUShort(thisIp+1));
        break;

        case LLOAD:     /* Load long from local variable (wide) */
        compilePushLocal(getUShort(thisIp+1));
        compilePushLocal(getUShort(thisIp+1)+1);
        break;

        case FLOAD:     /* Load float from local variable (wide) */
        compilePushLocal(getUShort(thisIp+1));
        break;

        case DLOAD:     /* Load double from local variable (wide) */
        compilePushLocal(getUShort(thisIp+1));
        compilePushLocal(getUShort(thisIp+1)+1);
        break;

        case ALOAD:     /* Load reference from local variable (wide) */
        compilePushLocal(getUShort(thisIp+1));
        break;

        case ISTORE:    /* Store integer into local variable (wide) */
        compilePopLocal(getUShort(thisIp+1));
        break;
        case LSTORE:    /* Store long into local variable (wide) */
        compilePopLocal(getUShort(thisIp+1)+1);
        compilePopLocal(getUShort(thisIp+1));
        break;

        case FSTORE:    /* Store float into local variable (wide) */
        compilePopLocal(getUShort(thisIp+1));
        break;

        case DSTORE:    /* Store double into local variable (wide) */
        compilePopLocal(getUShort(thisIp+1)+1);
        compilePopLocal(getUShort(thisIp+1));
        break;

        case ASTORE:    /* Store reference into local variable (wide) */
        compilePopLocal(getUShort(thisIp+1));
        break;

        case RET:       /* Return from subroutine (wide) */
        /* Not implemented */
        break;

        case IINC:      /* Increment local variable (wide) */
        compileLocalToReg(getUShort(thisIp+1), 3);
        compileAddi(3, 3, getShort(thisIp+3));
        compileRegToLocal(3, getUShort(thisIp+1));
        break;

        }    /* WIDE */
        break;

/* ----------------------------------------------------------- */

        case MULTIANEWARRAY:    /* Create new multidimensional array */
        {
        unsigned int cpIndex;
        int dimensions;
        ARRAY_CLASS thisClass;

        /* Get the CONSTANT_Class index */
        cpIndex = getUShort(thisIp + 1);

        /* Get the corresponding class pointer */
        thisClass = (ARRAY_CLASS)
```

```
            resolveClassReference(
                method->ofClass->constPool, cpIndex, method->ofClass);

        /* Get the requested array dimensionality */
        dimensions = thisIp[3];

        compileMultiANewArray(thisClass, dimensions);
        }
        break;

/* ------------------------------------------------------------ */

        case IFNULL:    /* Branch if reference is null */
        {
        int pos;
        compileIfProlog(3, &pos);
        compileBeq(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case IFNONNULL: /* Branch if reference is not null */
        {
        int pos;
        compileIfProlog(3, &pos);
        compileBne(0);
        compileIfEpilog(method, pos);
        }
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case GOTO_W:    /* Go to address (wide) */
        targetIp = thisIp + getCell(thisIp+1);
        compileBBStaticDispatch(method, targetIp);
        notDone = FALSE;
        break;

/* ------------------------------------------------------------ */

        case JSR_W:     /* Jump to subroutine (wide) */
        /* Not implemented */
        break;

/* ------------------------------------------------------------ */

        case BREAKPOINT:    /* Breakpoint trap */
        break;

/*=============================================================
* Fast bytecodes (used internally by the system
*                 only if FASTBYTECODES flag is on)
*=============================================================*/

        case GETFIELD_FAST: /* Get single-word field (fast version) */
        case GETFIELDP_FAST:    /* Get pointer field */
        compileGetFieldFast(getUShort(thisIp + 1), op);
        break;

/* ------------------------------------------------------------ */

        case GETFIELD2_FAST:    /* Get double-word field (fast version) */
        compileGetFieldFast(getUShort(thisIp + 1), op);
        break;

/* ------------------------------------------------------------ */

        case PUTFIELD_FAST: /* Store single-word field (fast version) */
        compilePutFieldFast(getUShort(thisIp + 1), op);
        break;

/* ------------------------------------------------------------ */

        case PUTFIELD2_FAST:    /* Store double-word field (fast version) */
        compilePutFieldFast(getUShort(thisIp + 1), op);
        break;

/* ------------------------------------------------------------ */

        case GETSTATIC_FAST:
        case GETSTATICP_FAST:
        compileGetStaticFast(method->ofClass, getUShort(thisIp + 1), op);
        break;

/* ------------------------------------------------------------ */
```

```
        case GETSTATIC2_FAST:
        compileGetStaticFast(method->ofClass, getUShort(thisIp + 1), op);
        break;

/* ---------------------------------------------------------- */

        case PUTSTATIC_FAST:
        compilePutStaticFast(method->ofClass, getUShort(thisIp + 1), op);
        break;

/* ---------------------------------------------------------- */

        case PUTSTATIC2_FAST:
        compilePutStaticFast(method->ofClass, getUShort(thisIp + 1), op);
        break;

/* ---------------------------------------------------------- */

        case UNUSED_D5:
        break;

/* ---------------------------------------------------------- */

        case INVOKEVIRTUAL_FAST:
        {
        /* Invoke instance method; dispatch based on dynamic class */
        /* (fast version) */
        unsigned int iCacheIndex;
        ICACHE   thisICache;
        unsigned int cpIndex;
        METHOD cpMethod;

        /* Get the inline cache index */
        iCacheIndex = getUShort(thisIp + 1);

        /* Get the inline cache entry */
        thisICache = GETINLINECACHE(iCacheIndex);

        /* Get the constant pool index */
        cpIndex = thisICache->origParam;

        /* Resolve constant pool reference */
        cpMethod = resolveMethodReference(
            method->ofClass->constPool, cpIndex, FALSE, method->ofClass);
        compileInvokeVirtual(cpMethod);
        notDone = FALSE;
        }
        break;

/* ------------------------------------------------------------ */

        case INVOKESPECIAL_FAST:
        /* Invoke instance method; special handling for superclass, */
        /* private and instance initialization method invocations */
        /* (fast version) */
        {
        unsigned int cpIndex;
        METHOD cpMethod;

        cpIndex = getUShort(thisIp + 1);

        cpMethod = (METHOD)(method->ofClass->constPool->entries[cpIndex].cache);

        notDone = compileInvokeSpecial(cpMethod);
        }
        break;

/* ------------------------------------------------------------ */

        case INVOKESTATIC_FAST:
        /* Invoke a class (static) method (fast version) */
        {
        unsigned int cpIndex;
        METHOD cpMethod;
        cpIndex = getUShort(thisIp + 1);
        cpMethod = (METHOD)method->ofClass->constPool->entries[cpIndex].cache;

        notDone = compileInvokeStatic(cpMethod);
        }
        break;

/* ------------------------------------------------------------ */

        case INVOKEINTERFACE_FAST:
        {
        unsigned int cpIndex;
        METHOD cpMethod;
        unsigned int iCacheIndex;
        ICACHE   thisICache;
        unsigned int   argCount;

        /* Get the inline cache index */
        iCacheIndex = getUShort(thisIp + 1);

        /* Get the inline cache entry */
        thisICache = GETINLINECACHE(iCacheIndex);

        /* Get the constant pool index */
        cpIndex = thisICache->origParam;

        /* Resolve constant pool reference */
        cpMethod = resolveMethodReference(
            method->ofClass->constPool, cpIndex, FALSE, method->ofClass);

        /* Get the argument count (specific to INVOKEINTERFACE bytecode) */
        argCount = thisIp[3];

        compileInvokeInterface(cpMethod, method->ofClass, argCount);
        notDone = FALSE;
        }
        break;

/* ------------------------------------------------------------ */

        case NEW_FAST:  /* Create new object (fast version) */
        {
        INSTANCE_CLASS thisClass;
        /* Get the inline cache index */
        unsigned int cpIndex = getUShort(thisIp + 1);

        /* Get the class pointer */
        thisClass = (INSTANCE_CLASS)
            (method->ofClass->constPool->entries[cpIndex].clazz);

        compileNew(thisClass);
        }
        break;

/* ------------------------------------------------------------ */

        case ANEWARRAY_FAST:    /* Create reference array (fast version) */
        {
        /* Get the inline cache index */
        unsigned int iCacheIndex = getUShort(thisIp + 1);
        /* Get the inline cache entry */
        ICACHE thisICache = GETINLINECACHE(iCacheIndex);
        /* Get the class pointer stored in cache */
        ARRAY_CLASS thisClass = (ARRAY_CLASS)thisICache->contents;

        compileNewArray(thisClass);
        }
        break;

/* ------------------------------------------------------------ */

        case MULTIANEWARRAY_FAST:   /* Create multi-dimensional array */
        {
        /* Get the inline cache index */
        unsigned int cpIndex;
        int dimensions;
        ARRAY_CLASS thisClass;

        /* Get the inline cache index */
        cpIndex = getUShort(thisIp + 1);

        /* Get the class pointer stored in cache */
        thisClass = (ARRAY_CLASS)
            method->ofClass->constPool->entries[cpIndex].clazz;

        /* Get the requested array dimensionality */
        dimensions = thisIp[3];

        compileMultiANewArray(thisClass, dimensions);
        }
        break;

/* ------------------------------------------------------------ */

        case CHECKCAST_FAST:    /* Check object type (fast version) */
        /* Check whether object is of given type (fast version) */
        {
        unsigned int cpIndex;
        CLASS thisClass;

        cpIndex = getUShort(thisIp + 1);
        thisClass =
            method->ofClass->constPool->entries[cpIndex].clazz;
        compileCheckCast(thisClass);
        }
        break;

/* -------------------------------------------------------------- */

        case INSTANCEOF_FAST:   /* Check object type (fast version) */
        {
        unsigned int cpIndex = getUShort(thisIp + 1);
        CLASS thisClass =
            method->ofClass->constPool->entries[cpIndex].clazz;

        compileInstanceOf(thisClass);
        }
        break;

        default:
        fatalError("JIT: Invalid opcode");
        break;

        }
        nextInstruction(&nextIp);
    }
    CATCH(exc)
    /* We ran out of tcache space. Clear it and try to recompile. */
        /* printf("ATTENTION: no more tcache space\n"); */
        tcacheReset();
#if ENABLEPROFILING
        TcacheExhaustedCounter++;
#endif /* ENABLEPROFILING */
        TRY
        compileBasicBlock(method, basicBlock, resume);
        CATCH(excExc)
        /* Not even space for this single basic block. Bail out! */
        fatalError("JIT: tcache too small to fit a single basic block");
        END_CATCH
    END_CATCH
/*    fgetc(stdin); */

    tcachePos -= peepholeOptimize(
        basicBlock->nativeCode, (cell*)tcacheAddress() );

#if ENABLEPROFILING
    CompileCounter++;
    CompiledBytecodesCounter += bcCounter;
    CompiledBytecodeSize +=
        nextIp - (methodCode(method) + basicBlock->startIp);
    CompiledNativeCodeSize +=
        (BYTE *)tcacheAddress() - (BYTE *)basicBlock->nativeCode;
#endif /* ENABLEPROFILING */

}

/*=================================================================
 * FUNCTION:      initializeCompiler()
 * TYPE:          Initialization
 * OVERVIEW:      Initializes the compiler.
 *
 * INTERFACE:
 *   parameters:  none
 *   returns:     <nothing>
 *=================================================================*/

void
InitializeCompiler()
{
    /* Until we figure out how to interact with GC (compaction etc.),
       allocate static code buffer. */
    tcache = (int *)callocPermanentObject(
        RequestedTranslationCacheSize);
    tcacheReset();
}

/*=================================================================
 * FUNCTION:      finalizeCompiler()
 * TYPE:          Finalization
 * OVERVIEW:      Finalizes the compiler.
 *
 * INTERFACE:
 *   parameters:  none
 *   returns:     <nothing>
 *=================================================================*/

void
FinalizeCompiler()
{
    /* dumpNativeCode("code.ppc"); */
}

#endif    /* ENABLEJIT */
```

A.1.3 basicblock.c

Basic block detector implementation.

```
/*=========================================================================
 * SYSTEM:    KVM compiler
 * SUBSYSTEM: Internal basic block locator
 * FILE:      basicblock.c
 * OVERVIEW:  Utility functions for managing basic blocks.
 *
 * AUTHOR:    Kent M. Hansen
 * NOTE:
 *=======================================================================*/

/*=========================================================================
 * Include files
 *=======================================================================*/

#include <global.h>

#if ENABLEJIT

/*=========================================================================
 * Global variables and definitions
 *=======================================================================*/

/*
 * Basic blocks that are less than INHIBIT_TRESHOLD
 * instructions will have their inhibit flag set, instructing
 * the selective compilation logic to not compile the basic
 * block even when the execution count has reached the treshold.
 */
#define INHIBIT_TRESHOLD 1

#define BASIC_BLOCK_AT(_tab_, _idx_) &(_tab_)->basicBlocks[_idx_]

/*=========================================================================
 * Internal iterator macros
 *=======================================================================*/

#define FOR_ALL_INSTRUCTIONS(__ip__, __index__, __method__) \
{ \
    BYTE *__ip__ = __method__->u.java.code; \
    BYTE *__endIp__ = __ip__ + __method__->u.java.codeLength; \
    __index__ = 0; \
    for (; __ip__ < __endIp__; \
    nextInstruction(&__ip__), __index__++ ) {

#define END_FOR_ALL_INSTRUCTIONS } }

/*=================================================================
 * FUNCTION:      nextInstruction()
 * TYPE:          Internal utility
 * OVERVIEW:      Advances instruction pointer to next
 *                instruction.
 *
 * INTERFACE:
 *   parameters:  thisIp: Instruction pointer
 *   returns:     <nothing>
 *===============================================================*/

void
nextInstruction(BYTE **thisIp)
{
    switch (*((*thisIp)++)) {
        case NOP:
        case ACONST_NULL:
        case ICONST_M1:
        case ICONST_0:
        case ICONST_1:
        case ICONST_2:
        case ICONST_3:
        case ICONST_4:

        case ICONST_5:
        case LCONST_0:
        case LCONST_1:
        case FCONST_0:
        case FCONST_1:
        case FCONST_2:
        case DCONST_0:
        case DCONST_1:  break;

        case BIPUSH:    *thisIp += 1;   break;
        case SIPUSH:    *thisIp += 2;   break;
        case LDC:       *thisIp += 1;   break;
        case LDC_W:
        case LDC2_W:    *thisIp += 2;   break;
        case ILOAD:
        case LLOAD:
```

```
case FLOAD:

case DLOAD:
case ALOAD:     *thisIp += 1;   break;
case ILOAD_0:
case ILOAD_1:
case ILOAD_2:
case ILOAD_3:
case LLOAD_0:
case LLOAD_1:

case LLOAD_2:
case LLOAD_3:
case FLOAD_0:
case FLOAD_1:
case FLOAD_2:
case FLOAD_3:
case DLOAD_0:
case DLOAD_1:

case DLOAD_2:
case DLOAD_3:
case ALOAD_0:
case ALOAD_1:
case ALOAD_2:
case ALOAD_3:
case IALOAD:
case LALOAD:

case FALOAD:
case DALOAD:
case AALOAD:
case BALOAD:
case CALOAD:
case SALOAD:    break;
case ISTORE:
case LSTORE:

case FSTORE:
case DSTORE:
case ASTORE:    *thisIp += 1;   break;
case ISTORE_0:
case ISTORE_1:
case ISTORE_2:
case ISTORE_3:
```

```
case LSTORE_0:

case LSTORE_1:
case LSTORE_2:
case LSTORE_3:
case FSTORE_0:
case FSTORE_1:
case FSTORE_2:
case FSTORE_3:
case DSTORE_0:

case DSTORE_1:
case DSTORE_2:
case DSTORE_3:
case ASTORE_0:
case ASTORE_1:
case ASTORE_2:
case ASTORE_3:
case IASTORE:

case LASTORE:
case FASTORE:
case DASTORE:
case AASTORE:
case BASTORE:
case CASTORE:
case SASTORE:
case POP:

case POP2:
case DUP:
case DUP_X1:
case DUP_X2:
case DUP2:
case DUP2_X1:
case DUP2_X2:
case SWAP:

case IADD:
case LADD:
case FADD:
case DADD:
case ISUB:
case LSUB:
case FSUB:
```

```
case DSUB:

case IMUL:
case LMUL:
case FMUL:
case DMUL:
case IDIV:
case LDIV:
case FDIV:
case DDIV:

case IREM:
case LREM:
case FREM:
case DREM:
case INEG:
case LNEG:
case FNEG:
case DNEG:

case ISHL:
case LSHL:
case ISHR:
case LSHR:
case IUSHR:
case LUSHR:
case IAND:
case LAND:

case IOR:
case LOR:
case IXOR:
case LXOR:  break;
case IINC:  *thisIp += 2;   break;
case I2L:
case I2F:
case I2D:

case L2I:
case L2F:
case L2D:
case F2I:
case F2L:
case F2D:
case D2I:
```

```
case D2L:

case D2F:
case I2B:
case I2C:
case I2S:
case LCMP:
case FCMPL:
case FCMPG:
case DCMPL:

case DCMPG:     break;
case IFEQ:
case IFNE:
case IFLT:
case IFGE:
case IFGT:
case IFLE:
case IF_ICMPEQ:

case IF_ICMPNE:
case IF_ICMPLT:
case IF_ICMPGE:
case IF_ICMPGT:
case IF_ICMPLE:
case IF_ACMPEQ:
case IF_ACMPNE:
case GOTO:

case JSR:   *thisIp += 2;   break;
case RET:   *thisIp += 1;   break;
case TABLESWITCH:
{
    long lowValue;
    long highValue;
    long numberOfCases;
    *thisIp += (4 - (long)(*thisIp)) & 3;
    lowValue = getAlignedCell((*thisIp) + 4);
    highValue = getAlignedCell((*thisIp) + 8);
    numberOfCases = highValue-lowValue+1;
    *thisIp += 12 + numberOfCases*4;
}
break;
case LOOKUPSWITCH:
{
    long numberOfPairs;
    *thisIp += (4 - (long)(*thisIp)) & 3;
    numberOfPairs = getAlignedCell((*thisIp) + 4);
    *thisIp += 8 + numberOfPairs*8;
}
break;
case IRETURN:
case LRETURN:
case FRETURN:
case DRETURN:

case ARETURN:
case RETURN:    break;
case GETSTATIC:
case PUTSTATIC:
case GETFIELD:
case PUTFIELD:
case INVOKEVIRTUAL:
case INVOKESPECIAL:

case INVOKESTATIC:      *thisIp += 2;   break;
case INVOKEINTERFACE:   *thisIp += 4;   break;
case UNUSED_BA: break;
case NEW:               *thisIp += 2;   break;
case NEWARRAY:          *thisIp += 1;   break;
case ANEWARRAY:         *thisIp += 2;   break;
case ARRAYLENGTH:
case ATHROW:    break;

case CHECKCAST:
case INSTANCEOF:        *thisIp += 2;   break;
case MONITORENTER:
case MONITOREXIT:   break;
case WIDE:
switch (*((*thisIp)++)) {
    case ILOAD:
    case LLOAD:
    case FLOAD:
    case DLOAD:
    case ALOAD:
    case ISTORE:
    case LSTORE:
    case FSTORE:
    case DSTORE:
    case ASTORE:
            case RET:        *thisIp += 2;   break;
            case IINC:       *thisIp += 4;   break;
        }
        break;
        case MULTIANEWARRAY:    *thisIp += 3;   break;
        case IFNULL:
        case IFNONNULL:     *thisIp += 2;   break;

        case GOTO_W:
        case JSR_W:         *thisIp += 4;   break;
        case BREAKPOINT:    break;

/*==============================================================
 * Fast bytecodes (used internally by the system
 *                 only if FASTBYTECODES flag is on)
 *=============================================================*/

        case GETFIELD_FAST:
        case GETFIELDP_FAST:
        case GETFIELD2_FAST:
        case PUTFIELD_FAST:
        case PUTFIELD2_FAST:
        case GETSTATIC_FAST:
        case GETSTATICP_FAST:
        case GETSTATIC2_FAST:
        case PUTSTATIC_FAST:
        case PUTSTATIC2_FAST:   *thisIp += 2;   break;
        case UNUSED_D5:     break;
        case INVOKEVIRTUAL_FAST:
        case INVOKESPECIAL_FAST:

        case INVOKESTATIC_FAST: *thisIp += 2;   break;
        case INVOKEINTERFACE_FAST:  *thisIp += 4;   break;
        case NEW_FAST:          *thisIp += 2;   break;
        case ANEWARRAY_FAST:    *thisIp += 2;   break;
        case MULTIANEWARRAY_FAST:
        case CHECKCAST_FAST:
        case INSTANCEOF_FAST:   *thisIp += 2;   break;

        default:
        fatalError("nextInstruction(): Invalid opcode");
        break;
    }
}
```

```
/*===============================================================
 * FUNCTION:      countInstructions()
 * TYPE:          Internal utility
 * OVERVIEW:      Counts the number of instructions of a method.
 *
 * INTERFACE:
 *   parameters:  thisMethod: Method whose instructions to count
 *   returns:     Number of instructions
 *=============================================================*/

static long
countInstructions(METHOD thisMethod)
{
    long index;
    long count = 0;
    FOR_ALL_INSTRUCTIONS(dummyIp, index, thisMethod)
        count++;
    END_FOR_ALL_INSTRUCTIONS
    return count;
}

/*===============================================================
 * FUNCTION:      indexOfInstruction()
 * TYPE:          Internal utility
 * OVERVIEW:      Gets the index of an instruction in method.
 *
 * INTERFACE:
 *   parameters:  thisMethod: Method where instruction belongs
 *   returns:     Index of the instruction
 *=============================================================*/

static long
indexOfInstruction(METHOD thisMethod, BYTE *theIp)
{
    long index;
    FOR_ALL_INSTRUCTIONS(instrIp, index, thisMethod)
        if (instrIp == theIp) {
            return index;
        }
    END_FOR_ALL_INSTRUCTIONS
    return -1;
}

/*===============================================================
 * FUNCTION:      isLastInstruction()
 * TYPE:           Internal utility
 * OVERVIEW:       Tests if an instruction is the last in the
 *                 method.
 *
 * INTERFACE:
 *   parameters:  thisMethod: Method where instruction belongs
 *   returns:     TRUE if last instruction, FALSE otherwise
 *=============================================================*/

static bool_t
isLastInstruction(METHOD thisMethod, BYTE *theIp)
{
    BYTE *startIp = thisMethod->u.java.code;
    BYTE *endIp = startIp + thisMethod->u.java.codeLength;
    nextInstruction(&theIp);
    return (theIp == endIp) ? TRUE : FALSE;
}

/*==============================================================
 * FUNCTION:      computeLeaders()
 * TYPE:          Internal utility
 * OVERVIEW:      Computes the leaders of the method.
 *
 * INTERFACE:
 *   parameters:  thisMethod: Method whose leaders to compute
 *                leaders:    Bit vector to enter leaders in
 *   returns:     <nothing>
 *=============================================================*/

/* static const char* const byteCodeNames[] = BYTE_CODE_NAMES; */

static void
computeLeaders(METHOD thisMethod, BITVECTOR leaders)
{
    long index;
    BYTE *targetIp;
    /* First instruction is leader. */
    bitVectorAdd(leaders, 0);
    /* Figure out remaining leaders. */
    FOR_ALL_INSTRUCTIONS(thisIp, index, thisMethod)
/*      printf("%d: %s\n",
        thisIp - thisMethod->u.java.code,
        byteCodeNames[*thisIp]); */
        switch (*thisIp) {
            case IFEQ:
```

```
case IFNE:
case IFLT:
case IFGE:
case IFGT:
case IFLE:
case IF_ICMPEQ:
case IF_ICMPNE:
case IF_ICMPLT:
case IF_ICMPGE:
case IF_ICMPGT:
case IF_ICMPLE:
case IF_ACMPEQ:
case IF_ACMPNE:
case GOTO:
/* Target is leader. */
targetIp = thisIp + getShort(thisIp + 1);
bitVectorAdd(
    leaders,
    indexOfInstruction(thisMethod, targetIp) );
/* Next instruction is leader. */
if (!isLastInstruction(thisMethod, thisIp)) {
    bitVectorAdd(leaders, index+1 );
}
break;

case JSR:
/* Target is leader. */
targetIp = thisIp + getShort(thisIp + 1);
bitVectorAdd(
    leaders,
    indexOfInstruction(thisMethod, targetIp) );
/* Next instruction is leader. */
bitVectorAdd(leaders, index+1 );
break;

case RET:
/* Next instruction is leader. */
if (!isLastInstruction(thisMethod, thisIp)) {
    bitVectorAdd(leaders, index+1 );
}
break;

case TABLESWITCH:
{
    int j;
```

```
        long lowValue;
        long highValue;
        long numberOfCases;
        cell *base = (cell *)(((long)thisIp + 4) & ~3);
        /* Default is leader. */
        targetIp = thisIp + getAlignedCell(base);
        bitVectorAdd(
            leaders,
            indexOfInstruction(thisMethod, targetIp) );
        /* All jump targets are leaders. */
        lowValue = getAlignedCell(base+1);
        highValue = getAlignedCell(base+2);
        numberOfCases = highValue-lowValue+1;
        for (j=0; j<numberOfCases; j++) {
            targetIp = thisIp + getAlignedCell(base+3+j);
            bitVectorAdd(
                leaders,
                indexOfInstruction(
                    thisMethod, targetIp) );
        }
    }
    /* Next instruction is leader. */
    bitVectorAdd(leaders, index+1 );
    break;

    case LOOKUPSWITCH:
    {
        int j;
        int numberOfPairs;
        cell *base = (cell *)(((long)thisIp + 4) & ~3);
        /* Default is leader. */
        targetIp = thisIp + getAlignedCell(base);
        bitVectorAdd(
            leaders,
            indexOfInstruction(thisMethod, targetIp) );
        /* All targets are leaders. */
        numberOfPairs = getAlignedCell(base+1);
        for (j=0; j<numberOfPairs; j++) {
            targetIp = thisIp +
                getAlignedCell(base+3+j*2);
            bitVectorAdd(
                leaders,
                indexOfInstruction(
                    thisMethod, targetIp) );
        }
```

```
}
/* Next instruction is leader. */
bitVectorAdd(leaders, index+1 );
break;

case IRETURN:
case LRETURN:
case FRETURN:
case DRETURN:
case ARETURN:
case RETURN:
/* Next instruction is leader. */
if (!isLastInstruction(thisMethod, thisIp)) {
    bitVectorAdd(leaders, index+1 );
}
break;

case INVOKEVIRTUAL:
case INVOKESPECIAL:
/* Next instruction is leader. */
bitVectorAdd(leaders, index+1 );
break;

case INVOKESTATIC:
/* This instruction is leader. */
bitVectorAdd(leaders, index );
/* Next instruction is leader. */
bitVectorAdd(leaders, index+1 );
break;

case INVOKEINTERFACE:
/* Next instruction is leader. */
bitVectorAdd(leaders, index+1 );
break;

case NEW:
/* This instruction is leader. */
bitVectorAdd(leaders, index );
break;

/* case ATHROW: */
case MONITORENTER:
case MONITOREXIT:
/* Next instruction is leader. */
bitVectorAdd(leaders, index+1 );
```

```
break;

case WIDE:
if (*(thisIp + 1) == RET) {
    /* Next instruction is leader. */
    if (!isLastInstruction(thisMethod, thisIp)) {
        bitVectorAdd(leaders, index+1 );
    }
}
break;

case IFNULL:
case IFNONNULL:
/* Target is leader. */
targetIp = thisIp + getShort(thisIp + 1);
bitVectorAdd(
    leaders,
    indexOfInstruction(thisMethod, targetIp) );
/* Next instruction is leader. */
bitVectorAdd(leaders, index+1 );
break;

case GOTO_W:
/* Target is leader. */
targetIp = thisIp + getCell(thisIp + 1);
bitVectorAdd(
    leaders,
    indexOfInstruction(thisMethod, targetIp) );
/* Next instruction is leader. */
if (!isLastInstruction(thisMethod, thisIp)) {
    bitVectorAdd(leaders, index+1 );
}
break;

case JSR_W:
/* Target is leader. */
targetIp = thisIp + getCell(thisIp + 1);
bitVectorAdd(
    leaders,
    indexOfInstruction(thisMethod, targetIp) );
/* Next instruction is leader. */
bitVectorAdd(leaders, index+1 );
break;

case INVOKEVIRTUAL_FAST:
```

```
            case INVOKESPECIAL_FAST:
            case INVOKESTATIC_FAST:
            case INVOKEINTERFACE_FAST:
            /* Next instruction is leader. */
            bitVectorAdd(leaders, index+1 );
            break;

            default:
            /* Not a control instruction */
            break;
        }
    END_FOR_ALL_INSTRUCTIONS
}

/*===============================================================
 *==============================================================*/
#if INCLUDEDEBUGCODE

static void
basicBlocksTrace(METHOD method)
{
        int i;
        BASICBLOCK bb;
        char className[256];
        char methodSignature[256];
        char *methodName = methodName(method);

        getClassName_inBuffer(
            (CLASS)method->ofClass, className);
        change_Key_to_MethodSignature_inBuffer(
            method->nameTypeKey.nt.typeKey, methodSignature);

        /* Print name of the class, method name & signature */
        fprintf(stdout, "basic blocks for: %s.%s%s\n",
            className, methodName, methodSignature);

        for (i=0; i<method->u.java.basicBlocks->length; i++) {
            bb = &method->u.java.basicBlocks->basicBlocks[i];
            printf("  %d: ip=%d inhibit=%d\n",
                i, bb->startIp, INHIBIT_COMPILER(bb) );
        }
}

#endif    /* INCLUDEDEBUGCODE */
```

```
#if ENABLEINHIBITION

/*=========================================================
 * FUNCTION:      maybeInhibit()
 * TYPE:          Utility
 * OVERVIEW:      Sets the inhibit flag for a basic block if its
 *                length (in number of instructions) is below a
 *                treshold. When the inhibit flag is set, the basic
 *                block will not be compiled regardless of how many
 *                times it is executed.
 *
 * INTERFACE:
 *   parameters:  bb: Basic block
 *                length: Length of basic block
 *   returns:     Nothing, but sets the inhibit flag
 *                if length < treshold
 *=========================================================*/

static void
maybeInhibit(BASICBLOCK bb, int length)
{
    if (length < INHIBIT_TRESHOLD) {
        /* Use bit 0 of nativeCode field to store inhibt flag.
        This is safe since the nativeCode pointer only uses
        the upper 30 bits (word-aligned address). */
        bb->nativeCode = (cell *)((long)bb->nativeCode | 1);
    }
}

#endif    /* ENABLEINHIBITION */

/*=========================================================
 * FUNCTION:      computeBasicBlocks()
 * TYPE:          Utility
 * OVERVIEW:      Computes the basic blocks of a method.
 *
 * INTERFACE:
 *   parameters:  thisMethod: Method whose basic blocks to
 *                            compute
 *                defaultNative: Default native code address
 *                            of blocks
 *   returns:     Stores the array of basic blocks in method
 *                descriptor
 *=========================================================*/
```

```
void
computeBasicBlocks(METHOD thisMethod, void *defaultNative)
{
    long instructionCount;
    long basicBlockCount;
    int basicBlockTableSize;
    int bitVectorSize;
    int blockIndex;
    int prevLeader;
    int instrIndex;
    BITVECTOR_HANDLE leadersH;
    BASICBLOCKTABLE basicBlockTable;

    /* Count all instructions in method */
    instructionCount = countInstructions(thisMethod);

    bitVectorSize = SIZEOF_BITVECTOR(instructionCount);

    START_TEMPORARY_ROOTS

    /* Allocate temp bitvector that can
        hold <instructionCount> bits */
    DECLARE_TEMPORARY_ROOT(BITVECTOR, leaders,
        (BITVECTOR)callocObject(bitVectorSize, GCT_NOPOINTERS) );
    leaders->capacity = instructionCount;

    /* Compute leaders of method */
    computeLeaders(thisMethod, leaders);

    /* Save leaders handle,
        since we may GC in allocation below */
    leadersH = &leaders;

    /* Allocate table of basic blocks */
    basicBlockCount = bitVectorElementCount(leaders);
    basicBlockTableSize =
        SIZEOF_BASICBLOCKTABLE(basicBlockCount);
    basicBlockTable = (BASICBLOCKTABLE)
/*  callocObject(basicBlockTableSize, GCT_NOPOINTERS); */
        callocPermanentObject(basicBlockTableSize);
    basicBlockTable->length = basicBlockCount;
/*    printf("  # of blocks: %ld\n", basicBlockCount); */

    /* In case of GC above */
    leaders = unhand(leadersH);
    /* Initialize each basic block */
    blockIndex = 0;
    prevLeader = 0;
    FOR_ALL_INSTRUCTIONS(theIp, instrIndex, thisMethod)
        BASICBLOCK bb;
        /* Scan instructions until next leader */
        if (!bitVectorContains(leaders, instrIndex)) continue;

        /* Set basic block fields */
        bb = BASIC_BLOCK_AT(basicBlockTable, blockIndex);
        bb->nativeCode = defaultNative;
        bb->startIp = theIp - thisMethod->u.java.code;
        bb->counter = COMPILER_TRESHOLD(bb);

#if ENABLEINHIBITION
        if (blockIndex > 0) {
        /* Now we know the length of the previous block,
           and can set the inhibit flag if appropriate */
           maybeInhibit(bb-1, instrIndex - prevLeader);
        }
        prevLeader = instrIndex;
#endif    /* ENABLEINHIBITION */

        /* Prepare for next basic block */
        blockIndex++;
    END_FOR_ALL_INSTRUCTIONS

#if ENABLEINHIBITION
    /* Deal with the length of last basic block. */
    maybeInhibit(
        BASIC_BLOCK_AT(basicBlockTable, basicBlockCount-1),
        instrIndex - prevLeader);
#endif    /* ENABLEINHIBITION */

    END_TEMPORARY_ROOTS /* Ditches temp bitvector */

    /* Assign BB table to method */
    thisMethod->u.java.basicBlocks = basicBlockTable;

#if INCLUDEDEBUGCODE
    if (tracebasicblocks) {
        basicBlocksTrace(thisMethod);
    }
#endif /* INCLUDEDEBUGCODE */
```

```
#if ENABLEPROFILING
    BBComputeCounter++;
#endif /* ENABLEPROFILING */

}

/*==============================================================
 * FUNCTION:      basicBlockForIp()
 * TYPE:          Utility
 * OVERVIEW:      Gets the basic block given IP of leader.
 *
 * INTERFACE:
 *   parameters:  thisMethod: Method containing basic block
 *                theIp: IP of basic block's first instruction
 *   returns:     The basic block with given IP
 *=============================================================*/

BASICBLOCK
basicBlockForIp(METHOD thisMethod, BYTE *theIp)
{
    BASICBLOCKTABLE table = thisMethod->u.java.basicBlocks;
    /* Take advantage of the fact that IPs in the table are
       strictly ordered to perform a search in
       O(log2(tableLength)) time. */
    long width = table->length / 2;
    long index = width;
    unsigned short relIp = theIp - thisMethod->u.java.code;
    /* NOTE: This will HANG if the given IP is NOT the IP of
       any basic block in the table, so we better hope it is */
/*    printf("Getting index of block @ %p (%d)\n",
    theIp, relIp); */
    while (1) {
        width = (width+1) / 2;
        if (table->basicBlocks[index].startIp > relIp) {
            /* Eliminate right half */
            index -= width;
            if (index < 0) {
                index = 0;
            }
        }
        else if (table->basicBlocks[index].startIp < relIp) {
            /* Eliminate left half */
            index += width;
            if (index >= table->length) {
                index = table->length-1;
            }
        }
        else {
            /* Found it */
            return &table->basicBlocks[index];
        }
    }
}

#endif    /* ENABLEJIT */
```

A.1.4 bitvector.c

Bit vector implementation.

```
/*==============================================================
 * SYSTEM:    KVM compiler
 * SUBSYSTEM: Internal bitvector
 * FILE:      bitvector.c
 * OVERVIEW:  Implements bitvector representation.
 *
 * AUTHOR:    Kent M. Hansen
 * NOTE:
 *============================================================*/

/*==============================================================
 * Include files
 *============================================================*/

#include <global.h>

/*==============================================================
 * Internal macros
 *============================================================*/

/* Gets the cell that contains the specified bit. */
#define BITVECTOR_CELL(v, b) (v)->data[(b) / CELL_BITS]

/* Creates mask for a single bit. */
#define BITVECTOR_MASK(b) (1 << ((b) & (CELL_BITS-1)))

/*==============================================================
 * FUNCTION:      bitVectorContains()
 * TYPE:
 * OVERVIEW:      Tests if a given bit is contained in a bit
 *                vector.
 *
 * INTERFACE:
 *   parameters:  thisVector: Bit vector
 *                index: Index of bit to test for inclusion
 *   returns:     TRUE if contained, FALSE otherwise
 *============================================================*/

bool_t
bitVectorContains(BITVECTOR thisVector, long index)
{
    return (
      BITVECTOR_CELL(thisVector, index) & BITVECTOR_MASK(index)
           ? TRUE : FALSE );
}

/*==============================================================
 * FUNCTION:      bitVectorAdd()
 * TYPE:
 * OVERVIEW:      Adds a bit to bit vector.
 *
 * INTERFACE:
 *   parameters:  thisVector: Bit vector
 *                index: Index of bit to add
 *   returns:     <nothing>
 *=============================================================*/

void
bitVectorAdd(BITVECTOR thisVector, long index)
{
    BITVECTOR_CELL(thisVector, index) |= BITVECTOR_MASK(index);
}

/*==============================================================
 * FUNCTION:      bitVectorElementCount()
 * TYPE:
 * OVERVIEW:      Counts the number of elements (1-bits) in a
 *                bit vector.
 *
 * INTERFACE:
 *   parameters:  thisVector: Bit vector whose elements to count
 *   returns:     Number of elements
 *=============================================================*/

int
bitVectorElementCount(BITVECTOR thisVector)
{
    long index;
    int count;
    count = 0;
    for (index=0; index<thisVector->capacity; index++) {
        if (bitVectorContains(thisVector, index)) {
            count++;
        }
    }
    return count;
```

```
}
```

A.2 Header Files

A.2.1 compiler.h

Compiler interface.

```
#ifndef COMPILER_H
#define COMPILER_H

/*=============================================================
 * Functions
 *===========================================================*/

void InitializeCompiler();
void FinalizeCompiler();
void compileBasicBlock(METHOD, BASICBLOCK, void *);
void dumpNativeCode(const char *);

#endif  /* !COMPILER_H */
```

A.2.2 basicblock.h

Basic block detector interface.

```
#ifndef BASICBLOCK_H
#define BASICBLOCK_H

/*==============================================================
 * Include files
 *=============================================================*/

/*==============================================================
 * Definitions and declarations
 *=============================================================*/

/* Flag set the first time a method is invoked. */
#define ACC_INVOKED 0x80000000

/* How many times a basic block must be executed before it will be
   compiled. */
#define COMPILER_TRESHOLD(_bblock) 1024

#define INHIBIT_COMPILER(_bblock) ((long)((_bblock)->nativeCode) & 1)

/*==============================================================
 * COMMENTS:
 * BASICBLOCK is the structure used to hold information about a basic
 * block of a method.
 *=============================================================*/

/*  BASICBLOCK */
struct basicBlockStruct {
    cell *nativeCode;
    unsigned short startIp;
    unsigned short counter;
};

struct basicBlockTableStruct {
    long length;
    struct basicBlockStruct basicBlocks[1];
};

/*==============================================================
 * Sizes of the data structures above
 *=============================================================*/
```

```
#define SIZEOF_BASICBLOCK   StructSizeInCells(basicBlockStruct)

#define SIZEOF_BASICBLOCKTABLE(n)  \
        (StructSizeInCells(basicBlockTableStruct) + \
        (((n) - 1) * SIZEOF_BASICBLOCK))

/*==============================================================
 * Iterator macros
 *=============================================================*/

#define FOR_ALL_BASICBLOCKS(__block__, __table__) \
{ \
    BASICBLOCK __block__ = __table__->basicBlocks[0]; \
    BASICBLOCK __endBlock__ = __block__ + __table__->length; \
    for (; __block__ != __endBlock__; __block__++ ) {

#define END_FOR_ALL_BASICBLOCKS } }

/*==============================================================
 * Functions
 *=============================================================*/

void nextInstruction(BYTE **);
void computeBasicBlocks(METHOD, void *);
BASICBLOCK basicBlockForIp(METHOD, BYTE *);

#endif  /* !BASICBLOCK_H */
```

A.2.3 bitvector.h

Bit vector interface.

```
#ifndef BITVECTOR_H
#define BITVECTOR_H

/*========================================================
 * Definitions and declarations
 *======================================================*/

/* Number of bits in a cell. */
#define CELL_BITS (CELL * 8)

/*========================================================
 * COMMENTS:
 * BITVECTOR is a bit vector representation.
 *======================================================*/

/*  BITVECTOR */
struct bitVectorStruct
{
    long capacity;  /* NB: In bits, not cells */
    cell data[1];
};

/*========================================================
 * Sizes of the data structures above
 *======================================================*/

#define SIZEOF_BITVECTOR(n)  \
        (StructSizeInCells(bitVectorStruct) + \
        (((((n)+CELL_BITS-1)/CELL_BITS) - 1) * CELL))

/*========================================================
 * Functions
 *======================================================*/

bool_t bitVectorContains(BITVECTOR, long);
void bitVectorAdd(BITVECTOR, long);
int bitVectorElementCount(BITVECTOR);

#endif  /* !BITVECTOR_H */
```

A.2.4 ppcinstr.h

PowerPC instruction generators, in the form of macros.

```
/**
 * Macros for generating PowerPC instruction words.
 */
#ifndef PPCINSTR_H
#define PPCINSTR_H

#define INS_OP(Op)  (((Op) & 0x3F) << 26)

/* Generic instruction word encoders */
#define Op_D_A_B_OE_OpOp_Rc(Op, D, A, B, OE, OpOp, Rc)  \
    (unsigned int)( INS_OP(Op) | ((D)  << 21) | ((A)  << 16) | \
    ((B) << 11) | ((OE) << 10) | ((OpOp) << 1) | (Rc) )

#define Op_S_A_B_OpOp_Rc(Op, S, A, B, OpOp, Rc) \
    (unsigned int)( INS_OP(Op) | ((S)  << 21) | ((A)  << 16) | \
    ((B) << 11) | ((OpOp) << 1) | (Rc) )

#define Op_D_A_SIMM(Op, D, A, SIMM) \
    (unsigned int)( INS_OP(Op) | ((D)  << 21) | ((A)  << 16) | \
    ((SIMM) & 0xFFFF) )

#define Op_LI_AA_LK(Op, LI, AA, LK) \
    (unsigned int)( INS_OP(Op) | (((LI) & 0x00FFFFFF) << 2)  | \
    ((AA) << 1)  | (LK) )

#define Op_BO_BI_BD_AA_LK(Op, BO, BI, BD, AA, LK) \
    (unsigned int)( INS_OP(Op) | ((BO) << 21) | ((BI) << 16) | \
    (((BD) & 0x3FFF) << 2) | ((AA) << 1) | (LK) )

#define Op_BO_BI_OpOp_LK(Op, BO, BI, OpOp, LK) \
    (unsigned int)( INS_OP(Op) | ((BO) << 21) | ((BI) << 16) | \
    ((OpOp) << 1) | (LK) )
#define Op_crfD_L_A_B_OpOp(Op, crfD, L, A, B, OpOp) \
    (unsigned int)( INS_OP(Op) | ((crfD) << 23) | ((L) << 21) | \
    ((A) << 16) | ((B) << 11) | ((OpOp) << 1) )

#define Op_crfD_L_A_SIMM(Op, crfD, L, A, SIMM) \
    (unsigned int)( INS_OP(Op) | ((crfD) << 23) | ((L) << 21) | \
    ((A) << 16) | ((SIMM) & 0xFFFF) )

#define Op_D_A_B_C_OpOp_Rc(Op, D, A, B, C, OpOp, Rc) \
    (unsigned int)( INS_OP(Op) | ((D)  << 21) | ((A)  << 16) | \
    ((B) << 11) | ((C) << 6) | ((OpOp) << 1) | (Rc) )

#define Op_D_A_d(Op, D, A, d) \
    (unsigned int)( INS_OP(Op) | ((D)  << 21) | ((A)  << 16) | \
    ((d) & 0xFFFF) )

#define Op_D_A_ds_F(Op, D, A, ds, F) \
    (unsigned int)( INS_OP(Op) | ((D)  << 21) | ((A)  << 16) | \
    ((ds) <<2) | (F) )

#define Op_D_spr_OpOp(Op, D, spr, OpOp) \
    (unsigned int)( INS_OP(Op) | ((D)  << 21) | ((spr) << 11) | \
    ((OpOp) << 1) )

/* Instruction templates */
#define T_ADD(D, A, B, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, OE, 266, Rc)
#define T_ADDC(D, A, B, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, OE,  10, Rc)
#define T_ADDE(D, A, B, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, OE, 138, Rc)
#define T_ADDME(D, A, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, 0, OE, 234, Rc)
#define T_ADDZE(D, A, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, 0, OE, 202, Rc)
#define T_AND(A, S, B, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, B,      28, Rc)
#define T_ANDC(A, S, B, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, B,      60, Rc)
#define T_B(LI, AA, LK) \
    Op_LI_AA_LK(        18, LI, AA, LK)
#define T_BC(BO, BI, BD, AA, LK) \
    Op_BO_BI_BD_AA_LK(  16, BO,  BI, BD, AA, LK)
#define T_BCCTR(BO, BI, LK) \
    Op_BO_BI_OpOp_LK(   19, BO,  BI,     528, LK)
#define T_BCLR(BO, BI, LK) \
    Op_BO_BI_OpOp_LK(   19, BO,  BI,      16, LK)
#define T_CMP(crfD, L, A, B) \
    Op_crfD_L_A_B_OpOp( 31, crfD, L, A, B, 0)
#define T_CMPI(crfD, L, A, SIMM) \
    Op_crfD_L_A_SIMM(   11, crfD, L, A, SIMM)
#define T_CMPL(crfD, L, A, B) \
    Op_crfD_L_A_B_OpOp( 31, crfD, L, A, B, 32)
#define T_CMPLI(crfD, L, A, UIMM) \
    Op_crfD_L_A_SIMM(   10, crfD, L, A, UIMM)
#define T_CNTLZD(S, A, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, 0,       58, Rc)
#define T_CNTLZW(S, A, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, 0,       26, Rc)
#define T_DIVD(D, A, B, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, OE, 489, Rc)
#define T_DIVDU(D, A, B, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, OE, 457, Rc)
#define T_DIVW(D, A, B, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, OE, 491, Rc)
#define T_DIVWU(D, A, B, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, OE, 459, Rc)
#define T_EQV(A, S, B, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, B,      284, Rc)
#define T_EXTSB(A, S, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, 0,      954, Rc)
#define T_EXTSH(A, S, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, 0,      922, Rc)
#define T_EXTSW(A, S, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, 0,      986, Rc)
#define T_FABS(D, B, Rc) \
    Op_S_A_B_OpOp_Rc(   63, D, 0, B,      264, Rc)
#define T_FADD(D, A, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 63, D, A, B, 0,    21, Rc)
#define T_FADDS(D, A, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 59, D, A, B, 0,    21, Rc)
#define T_FCFID(D, B, Rc) \
    Op_S_A_B_OpOp_Rc(   63, D, 0, B,      846, Rc)
#define T_FCTID(D, B, Rc) \
    Op_S_A_B_OpOp_Rc(   63, D, 0, B,      814, Rc)
#define T_FCTIDZ(D, B, Rc) \
    Op_S_A_B_OpOp_Rc(   63, D, 0, B,      815, Rc)
#define T_FCTIW(D, B, Rc) \
    Op_S_A_B_OpOp_Rc(   63, D, 0, B,       14, Rc)
#define T_FCTIWZ(D, B, Rc) \
    Op_S_A_B_OpOp_Rc(   63, D, 0, B,       15, Rc)
#define T_FDIV(D, A, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 63, D, A, B, 0,    18, Rc)
#define T_FDIVS(D, A, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 59, D, A, B, 0,    18, Rc)
#define T_FMADD(D, A, C, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 63, D, A, B, C,    29, Rc)
#define T_FMADDS(D, A, C, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 59, D, A, B, C,    29, Rc)
```

```
#define T_FMR(D, B, Rc) \
    Op_S_A_B_OpOp_Rc(   63, D, 0, B,       72, Rc)
#define T_FMSUB(D, A, C, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 63, D, A, B, C,    28, Rc)
#define T_FMSUBS(D, A, C, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 59, D, A, B, C,    28, Rc)
#define T_FMUL(D, A, C, Rc) \
    Op_D_A_B_C_OpOp_Rc( 63, D, A, 0, C,    25, Rc)
#define T_FMULS(D, A, C, Rc) \
    Op_D_A_B_C_OpOp_Rc( 59, D, A, 0, C,    25, Rc)
#define T_FNABS(D, B, Rc) \
    Op_S_A_B_OpOp_Rc(   63, D, 0, B,      136, Rc)
#define T_FNEG(D, B, Rc) \
    Op_S_A_B_OpOp_Rc(   63, D, 0, B,       40, Rc)
#define T_FNMADD(D, A, C, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 63, D, A, B, C,    31, Rc)
#define T_FNMADDS(D, A, C, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 59, D, A, B, C,    31, Rc)
#define T_FNMSUB(D, A, C, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 63, D, A, B, C,    30, Rc)
#define T_FNMSUBS(D, A, C, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 59, D, A, B, C,    30, Rc)
#define T_FRES(D, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 59, D, 0, B, 0,    24, Rc)
#define T_FRSP(D, B, Rc) \
    Op_S_A_B_OpOp_Rc(   63, D, 0, B,       12, Rc)
#define T_FRSQRTE(D, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 63, D, 0, B, 0,    26, Rc)
#define T_FSEL(D, A, C, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 63, D, A, B, C,    23, Rc)
#define T_FSQRT(D, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 63, D, 0, B, 0,    22, Rc)
#define T_FSQRTS(D, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 59, D, 0, B, 0,    22, Rc)
#define T_FSUB(D, A, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 63, D, A, B, 0,    20, Rc)
#define T_FSUBS(D, A, B, Rc) \
    Op_D_A_B_C_OpOp_Rc( 59, D, A, B, 0,    20, Rc)
#define T_MFFS(D, Rc) \
    Op_S_A_B_OpOp_Rc(   63, D, 0, 0,      583, Rc)
#define T_MTFSFI(crfD, IMM, Rc) \
    Op_S_A_B_OpOp_Rc(   31, (crfD) << 2, 0, (IMM) << 1, 134, Rc)
#define T_MULHD(D, A, B, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, 0,    73, Rc)
#define T_MULHDU(D, A, B, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, 0,    9, Rc)
#define T_MULHW(D, A, B, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, 0,   75, Rc)
#define T_MULHWU(D, A, B, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, 0,   11, Rc)
#define T_MULLD(D, A, B, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, OE, 233, Rc)
#define T_MULLW(D, A, B, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, OE, 235, Rc)
#define T_NAND(A, S, B, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, B,     476, Rc)
#define T_NEG(D, A, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, 0, OE, 104, Rc)
#define T_NOR(A, S, B, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, B,     124, Rc)
#define T_OR(A, S, B, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, B,     444, Rc)
#define T_ORC(A, S, B, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, B,     412, Rc)
#define T_RLWIMI(A, S, SH, MB, ME, Rc) \
    Op_D_A_B_C_OpOp_Rc(20, S, A, SH, MB, ME, Rc)
#define T_RLWINM(A, S, SH, MB, ME, Rc) \
    Op_D_A_B_C_OpOp_Rc(21, S, A, SH, MB, ME, Rc)
#define T_RLWNM(A, S, B, MB, ME, Rc) \
    Op_D_A_B_C_OpOp_Rc(23, S, A, B, MB, ME, Rc)
#define T_SLD(A, S, B, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, B,      27, Rc)
#define T_SLW(A, S, B, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, B,      24, Rc)
#define T_SRAW(A, S, B, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, B,     792, Rc)
#define T_SRAWI(A, S, SH, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, SH,    824, Rc)
#define T_SRD(A, S, B, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, B,     539, Rc)
#define T_SRW(A, S, B, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, B,     536, Rc)
#define T_SUBF(D, A, B, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, OE,  40, Rc)
#define T_SUBFC(D, A, B, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, OE,   8, Rc)
#define T_SUBFE(D, A, B, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, B, OE, 136, Rc)
#define T_SUBFME(D, A, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, 0, OE, 232, Rc)
```

```
#define T_SUBFZE(D, A, OE, Rc) \
    Op_D_A_B_OE_OpOp_Rc(31, D, A, 0, OE, 200, Rc)
#define T_XOR(A, S, B, Rc) \
    Op_S_A_B_OpOp_Rc(   31, S, A, B,     316, Rc)

/* Specific instruction word encoders */
#define ADD(D, A, B)            T_ADD( D, A, B, 0, 0)
#define ADD_(D, A, B)           T_ADD( D, A, B, 0, 1)
#define ADDO(D, A, B)           T_ADD( D, A, B, 1, 0)
#define ADDO_(D, A, B)          T_ADD( D, A, B, 1, 1)
#define ADDC(D, A, B)           T_ADDC(D, A, B, 0, 0)
#define ADDC_(D, A, B)          T_ADDC(D, A, B, 0, 1)
#define ADDCO(D, A, B)          T_ADDC(D, A, B, 1, 0)
#define ADDCO_(D, A, B)         T_ADDC(D, A, B, 1, 1)
#define ADDE(D, A, B)           T_ADDE(D, A, B, 0, 0)
#define ADDE_(D, A, B)          T_ADDE(D, A, B, 0, 1)
#define ADDEO(D, A, B)          T_ADDE(D, A, B, 1, 0)
#define ADDEO_(D, A, B)         T_ADDE(D, A, B, 1, 1)
#define ADDI(D, A, SIMM)        Op_D_A_SIMM(14, D, A, SIMM)
#define ADDIC(D, A, SIMM)       Op_D_A_SIMM(12, D, A, SIMM)
#define ADDIC_(D, A, SIMM)      Op_D_A_SIMM(13, D, A, SIMM)
#define ADDIS(D, A, SIMM)       Op_D_A_SIMM(15, D, A, SIMM)
#define ADDME(D, A)             T_ADDME(D, A, 0, 0)
#define ADDME_(D, A)            T_ADDME(D, A, 0, 1)
#define ADDMEO(D, A)            T_ADDME(D, A, 1, 0)
#define ADDMEO_(D, A)           T_ADDME(D, A, 1, 1)
#define ADDZE(D, A)             T_ADDZE(D, A, 0, 0)
#define ADDZE_(D, A)            T_ADDZE(D, A, 0, 1)
#define ADDZEO(D, A)            T_ADDZE(D, A, 1, 0)
#define ADDZEO_(D, A)           T_ADDZE(D, A, 1, 1)
#define AND(A, S, B)            T_AND(  A, S, B, 0)
#define AND_(A, S, B)           T_AND(  A, S, B, 1)
#define ANDC(A, S, B)           T_AND(  A, S, B, 0)
#define ANDC_(A, S, B)          T_AND(  A, S, B, 1)
#define ANDI_(A, S, UIMM)       Op_D_A_SIMM(28, S, A, UIMM)
#define ANDIS_(A, S, UIMM)      Op_D_A_SIMM(29, S, A, UIMM)
#define B(LI)                   T_B(LI, 0, 0)
#define BL(LI)                  T_B(LI, 0, 1)
#define BA(LI)                  T_B(LI, 1, 0)
#define BLA(LI)                 T_B(LI, 1, 1)
#define BC(BO, BI, BD)          T_BC(BO, BI, BD, 0, 0)
#define BCL(BO, BI, BD)         T_BC(BO, BI, BD, 0, 1)
#define BCA(BO, BI, BD)         T_BC(BO, BI, BD, 1, 0)
#define BCLA(BO, BI, BD)        T_BC(BO, BI, BD, 1, 1)
#define BCCTR(BO, BI)           T_BCCTR(BO, BI, 0)
```

```
#define BCCTRL(BO, BI)          T_BCCTR(BO, BI, 1)
#define BCLR(BO, BI)            T_BCLR(BO, BI, 0)
#define BCLRL(BO, BI)           T_BCLR(BO, BI, 1)
#define CMP(crfD, A, B)         T_CMP( crfD, 0, A, B)
#define CMPI(crfD, A, SIMM)     T_CMPI(crfD, 0, A, SIMM)
#define CMPL(crfD, A, B)        T_CMPL(crfD, 0, A, B)
#define CMPLI(crfD, A, UIMM)    T_CMPLI(crfD, 0, A, UIMM)
#define CNTLZW(S, A)            T_CNTLZW(S, A, 0)
#define CNTLZW_(S, A)           T_CNTLZW(S, A, 1)
#define CRAND(crbD, crbA, crbB) \
    Op_S_A_B_OpOp_Rc(19, crbD, crbA, crbB, 257, 0)
#define CRANDC(crbD, crbA, crbB) \
    Op_S_A_B_OpOp_Rc(19, crbD, crbA, crbB, 129, 0)
#define CREQV(crbD, crbA, crbB) \
    Op_S_A_B_OpOp_Rc(19, crbD, crbA, crbB, 289, 0)
#define CRNAND(crbD, crbA, crbB) \
    Op_S_A_B_OpOp_Rc(19, crbD, crbA, crbB, 225, 0)
#define CRNOR(crbD, crbA, crbB) \
    Op_S_A_B_OpOp_Rc(19, crbD, crbA, crbB,  33, 0)
#define CROR(crbD, crbA, crbB) \
    Op_S_A_B_OpOp_Rc(19, crbD, crbA, crbB, 449, 0)
#define CRORC(crbD, crbA, crbB) \
    Op_S_A_B_OpOp_Rc(19, crbD, crbA, crbB, 417, 0)
#define CRXOR(crbD, crbA, crbB) \
    Op_S_A_B_OpOp_Rc(19, crbD, crbA, crbB, 193, 0)
#define DCBA(A, B) \
    Op_S_A_B_OpOp_Rc(31, 0, A, B,  758, 0)
#define DCBF(A, B) \
    Op_S_A_B_OpOp_Rc(31, 0, A, B,   86, 0)
#define DCBI(A, B) \
    Op_S_A_B_OpOp_Rc(31, 0, A, B,  470, 0)
#define DCBST(A, B) \
    Op_S_A_B_OpOp_Rc(31, 0, A, B,   54, 0)
#define DCBT(A, B) \
    Op_S_A_B_OpOp_Rc(31, 0, A, B,  278, 0)
#define DCBTST(A, B) \
    Op_S_A_B_OpOp_Rc(31, 0, A, B,  246, 0)
#define DCBZ(A, B) \
    Op_S_A_B_OpOp_Rc(31, 0, A, B, 1014, 0)
#define DIVW(D, A, B)           T_DIVW( D, A, B, 0, 0)
#define DIVW_(D, A, B)          T_DIVW( D, A, B, 0, 1)
#define DIVWO(D, A, B)          T_DIVW( D, A, B, 1, 0)
#define DIVWO_(D, A, B)         T_DIVW( D, A, B, 1, 1)
#define DIVWU(D, A, B)          T_DIVWU(D, A, B, 0, 0)
#define DIVWU_(D, A, B)         T_DIVWU(D, A, B, 0, 1)
```

```
#define DIVWUO(D, A, B)         T_DIVWU(D, A, B, 1, 0)
#define DIVWUO_(D, A, B)        T_DIVWU(D, A, B, 1, 1)
#define ECIWX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B, 310, 0)
#define ECOWX(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 438, 0)
#define EIEIO() \
    Op_S_A_B_OpOp_Rc(31, 0, 0, 0, 854, 0)
#define EQV(A, S, B)            T_EQV(  A, S, B, 0)
#define EQV_(A, S, B)           T_EQV(  A, S, B, 1)
#define EXTSB(A, S)             T_EXTSB(A, S, 0)
#define EXTSB_(A, S)            T_EXTSB(A, S, 1)
#define EXTSH(A, S)             T_EXTSH(A, S, 0)
#define EXTSH_(A, S)            T_EXTSH(A, S, 1)
#define FABS(D, B)              T_FABS( D, B, 0)
#define FABS_(D, B)             T_FABS( D, B, 1)
#define FADD(D, A, B)           T_FADD( D, A, B, 0)
#define FADD_(D, A, B)          T_FADD( D, A, B, 1)
#define FADDS(D, A, B)          T_FADDS(D, A, B, 0)
#define FADDS_(D, A, B)         T_FADDS(D, A, B, 1)
#define FCMPO(crfD, A, B) \
    Op_S_A_B_OpOp_Rc(63, (crfD) << 2, A, B, 32, 0)
#define FCMPU(crfD, A, B) \
    Op_S_A_B_OpOp_Rc(63, (crfD) << 2, A, B,  0, 0)
#define FCTIW(D, B)             T_FCTIW( D, B, 0)
#define FCTIW_(D, B)            T_FCTIW( D, B, 1)
#define FCTIWZ(D, B)            T_FCTIWZ(D, B, 0)
#define FCTIWZ_(D, B)           T_FCTIWZ(D, B, 1)
#define FDIV(D, A, B)           T_FDIV(  D, A, B, 0)
#define FDIV_(D, A, B)          T_FDIV(  D, A, B, 1)
#define FDIVS(D, A, B)          T_FDIVS( D, A, B, 0)
#define FDIVS_(D, A, B)         T_FDIVS( D, A, B, 1)
#define FMADD(D, A, C, B)       T_FMADD( D, A, C, B, 0)
#define FMADD_(D, A, C, B)      T_FMADD( D, A, C, B, 1)
#define FMADDS(D, A, C, B)      T_FMADDS(D, A, C, B, 0)
#define FMADDS_(D, A, C, B)     T_FMADDS(D, A, C, B, 1)
#define FMR(D, B)               T_FMR(D, B, 0)
#define FMR_(D, B)              T_FMR(D, B, 1)
#define FMSUB(D, A, C, B)       T_FMSUB(D, A, C, B, 0)
#define FMSUB_(D, A, C, B)      T_FMSUB(D, A, C, B, 1)
#define FMSUBS(D, A, C, B)      T_FMSUBS(D, A, C, B, 0)
#define FMSUBS_(D, A, C, B)     T_FMSUBS(D, A, C, B, 1)
#define FMUL(D, A, C)           T_FMUL( D, A, C, 0)
#define FMUL_(D, A, C)          T_FMUL( D, A, C, 1)
#define FMULS(D, A, C)          T_FMULS(D, A, C, 0)
```

```
#define FMULS_(D, A, C)          T_FMULS(D, A, C, 1)
#define FNABS(D, B)              T_FNABS(D, B, 0)
#define FNABS_(D, B)             T_FNABS(D, B, 1)
#define FNEG(D, B)               T_FNEG( D, B, 0)
#define FNEG_(D, B)              T_FNEG( D, B, 1)
#define FNMADD(D, A, C, B)       T_FNMADD(D, A, C, B, 0)
#define FNMADD_(D, A, C, B)      T_FNMADD(D, A, C, B, 1)
#define FNMADDS(D, A, C, B)      T_FNMADDS(D, A, C, B, 0)
#define FNMADDS_(D, A, C, B)     T_FNMADDS(D, A, C, B, 1)
#define FNMSUB(D, A, C, B)       T_FNMSUB(D, A, C, B, 0)
#define FNMSUB_(D, A, C, B)      T_FNMSUB(D, A, C, B, 1)
#define FNMSUBS(D, A, C, B)      T_FNMSUBS(D, A, C, B, 0)
#define FNMSUBS_(D, A, C, B)     T_FNMSUBS(D, A, C, B, 1)
#define FRES(D, B)               T_FRES(D, B, 0)
#define FRES_(D, B)              T_FRES(D, B, 1)
#define FRSP(D, B)               T_FRSP(D, B, 0)
#define FRSP_(D, B)              T_FRSP(D, B, 1)
#define FSUB(D, A, B)            T_FSUB(D, A, B, 0)
#define FSUB_(D, A, B)           T_FSUB(D, A, B, 1)
#define FSUBS(D, A, B)           T_FSUBS(D, A, B, 0)
#define FSUBS_(D, A, B)          T_FSUBS(D, A, B, 1)
#define ICBI(A, B) \
    Op_S_A_B_OpOp_Rc(31, 0, A, B, 982, 0)
#define ISYNC() \
    Op_S_A_B_OpOp_Rc(19, 0, 0, 0, 150, 0)
#define LBZ(D, d, A)             Op_D_A_d(34, D, A, d)
#define LBZU(D, d, A)            Op_D_A_d(35, D, A, d)
#define LBZUX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B, 119, 0)
#define LBZX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B,  87, 0)
#define LFD(D, d, A)             Op_D_A_d(50, D, A, d)
#define LFDU(D, d, A)            Op_D_A_d(51, D, A, d)
#define LFDUX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B, 631, 0)
#define LFDX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B, 599, 0)
#define LFS(D, d, A)             Op_D_A_d(48, D, A, d)
#define LFSU(D, d, A)            Op_D_A_d(49, D, A, d)
#define LFSUX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B, 567, 0)
#define LFSX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B, 535, 0)
#define LHA(D, d, A)             Op_D_A_d(42, D, A, d)
#define LHAU(D, d, A)            Op_D_A_d(43, D, A, d)
```

```
#define LHAUX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B, 375, 0)
#define LHAX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B, 343, 0)
#define LHBRX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B, 790, 0)
#define LHZ(D, d, A)            Op_D_A_d(40, D, A, d)
#define LHZU(D, d, A)           Op_D_A_d(41, D, A, d)
#define LHZUX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B, 311, 0)
#define LHZX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B, 279, 0)
#define LMW(D, d, A)            Op_D_A_d(46, D, A, d)
#define LSWI(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B, 597, 0)
#define LSWX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B, 533, 0)
#define LWARX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B,  20, 0)
#define LWBRX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B, 534, 0)
#define LWZ(D, d, A)            Op_D_A_d(32, D, A, d)
#define LWZU(D, d, A)           Op_D_A_d(33, D, A, d)
#define LWZUX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B,  55, 0)
#define LWZX(D, A, B) \
    Op_S_A_B_OpOp_Rc(31, D, A, B,  23, 0)
#define MCRF(crfD, crfS) \
    Op_S_A_B_OpOp_Rc(19, crfD, crfS, 0, 0, 0)
#define MCRFS(crfD, crfS) \
    Op_S_A_B_OpOp_Rc(63, crfD, crfS, 0, 64, 0)
#define MCRXR(crfD) \
    Op_S_A_B_OpOp_Rc(31, crfD, 0, 0, 512, 0)
#define MFCR(D) \
    Op_S_A_B_OpOp_Rc(31, D, 0, 0,  19, 0)
#define MFFS(D)                 T_MFFS(D, 0)
#define MFFS_(D)                T_MFFS(D, 1)
#define MFMSR(D) \
    Op_S_A_B_OpOp_Rc(31, D, 0, 0, 83, 0)
#define MFSPR(D, spr) \
    Op_D_spr_OpOp(31, D, (((spr) & 31) << 5) | \
    (((spr) >> 5) & 31), 339)
#define MFTB(D) \
    Op_D_spr_OpOp(31, D, (((268) & 31) << 5) | \
    (((268) >> 5) & 31), 371)
```

```
#define MFTBU(D) \
    Op_D_spr_OpOp(31, D, (((269) & 31) << 5) | \
    (((269) >> 5) & 31), 371)
#define MTCRF(S, CRM) \
    Op_D_spr_OpOp(31, S, (CRM) << 1, 144)
#define MTFSFI(crfD, IMM)        T_MTFSFI(crfD, IMM, 0)
#define MTFSFI_(crfD, IMM)       T_MTFSFI(crfD, IMM, 1)
#define MTMSR(S) \
    Op_S_A_B_OpOp_Rc(31, S, 0, 0, 146, 0)
#define MTSPR(spr, S) \
    Op_D_spr_OpOp(31, S, (((spr) & 31) << 5) | \
    (((spr) >> 5) & 31), 467)
#define MULHW(D, A, B)           T_MULHW( D, A, B, 0)
#define MULHW_(D, A, B)          T_MULHW( D, A, B, 1)
#define MULHWU(D, A, B)          T_MULHWU(D, A, B, 0)
#define MULHWU_(D, A, B)         T_MULHWU(D, A, B, 1)
#define MULLI(D, A, SIMM)        Op_D_A_SIMM(7, D, A, SIMM)
#define MULLW(D, A, B)           T_MULLW( D, A, B, 0, 0)
#define MULLW_(D, A, B)          T_MULLW( D, A, B, 0, 1)
#define MULLWO(D, A, B)          T_MULLW( D, A, B, 1, 0)
#define MULLWO_(D, A, B)         T_MULLW( D, A, B, 1, 1)
#define NAND(A, S, B)            T_NAND(  A, S, B, 0)
#define NAND_(A, S, B)           T_NAND(  A, S, B, 1)
#define NEG(D, A)                T_NEG(   D, A, 0, 0)
#define NEG_(D, A)               T_NEG(   D, A, 0, 1)
#define NEGO(D, A)               T_NEG(   D, A, 1, 0)
#define NEGO_(D, A)              T_NEG(   D, A, 1, 1)
#define NOR(A, S, B)             T_NOR(  A, S, B, 0)
#define NOR_(A, S, B)            T_NOR(  A, S, B, 1)
#define OR(A, S, B)              T_OR(  A, S, B, 0)
#define OR_(A, S, B)             T_OR(  A, S, B, 1)
#define ORC(A, S, B)             T_ORC(  A, S, B, 0)
#define ORC_(A, S, B)            T_ORC(  A, S, B, 1)
#define ORI(A, S, UIMM)          Op_D_A_SIMM(24, S, A, UIMM)
#define ORIS(A, S, UIMM)         Op_D_A_SIMM(25, S, A, UIMM)
#define RFI() \
    Op_S_A_B_OpOp_Rc(19, 0, 0, 0, 50, 0)
#define RFID() \
    Op_S_A_B_OpOp_Rc(19, 0, 0, 0, 18, 0)
#define RLWIMI(A, S, SH, MB, ME) T_RLWIMI(A, S, SH, MB, ME, 0)
#define RLWIMI_(A, S, SH, MB, ME) T_RLWIMI(A, S, SH, MB, ME, 1)
#define RLWINM(A, S, SH, MB, ME) T_RLWINM(A, S, SH, MB, ME, 0)
#define RLWINM_(A, S, SH, MB, ME) T_RLWINM(A, S, SH, MB, ME, 1)
#define RLWNM(A, S, B, MB, ME) T_RLWNM(A, S, B, MB, ME, 0)
#define RLWNM_(A, S, B, MB, ME) T_RLWNM(A, S, B, MB, ME, 1)
```

```
#define SC()                    ((17 << 26) | 2)
#define SLW(A, S, B)            T_SLW(  A, S, B, 0)
#define SLW_(A, S, B)           T_SLW(  A, S, B, 1)
#define SRAW(A, S, B)           T_SRAW( A, S, B, 0)
#define SRAW_(A, S, B)          T_SRAW( A, S, B, 1)
#define SRAWI(A, S, SH)         T_SRAWI(A, S, SH, 0)
#define SRAWI_(A, S, SH)        T_SRAWI(A, S, SH, 1)
#define SRW(A, S, B)            T_SRW(  A, S, B, 0)
#define SRW_(A, S, B)           T_SRW(  A, S, B, 1)
#define STB(S, d, A)            Op_D_A_d(38, S, A, d)
#define STBU(S, d, A)           Op_D_A_d(39, S, A, d)
#define STBUX(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 247, 0)
#define STBX(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 215, 0)
#define STFD(S, d, A) \          Op_D_A_d(54, S, A, d)
#define STFDU(S, d, A)          Op_D_A_d(55, S, A, d)
#define STFDUX(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 759, 0)
#define STFDX(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 727, 0)
#define STFIWX(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 983, 0)
#define STFS(S, d, A)           Op_D_A_d(52, S, A, d)
#define STFSU(S, d, A)          Op_D_A_d(53, S, A, d)
#define STFSUX(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 695, 0)
#define STFSX(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 663, 0)
#define STH(S, d, A)            Op_D_A_d(44, S, A, d)
#define STHBRX(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 918, 0)
#define STHU(S, d, A)           Op_D_A_d(45, S, A, d)
#define STHUX(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 439, 0)
#define STHX(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 407, 0)
#define STMW(S, d, A)           Op_D_A_d(47, S, A, d)
#define STSWI(S, A, NB) \
    Op_S_A_B_OpOp_Rc(31, S, A, NB, 725, 0)
#define STSWX(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 661, 0)
#define STW(S, d, A)            Op_D_A_d(36, S, A, d)
#define STWBRX(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 662, 0)
```

```
#define STWCX_(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 150, 1)
#define STWU(S, d, A)            Op_D_A_d(37, S, A, d)
#define STWUX(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 183, 0)
#define STWX(S, A, B) \
    Op_S_A_B_OpOp_Rc(31, S, A, B, 151, 0)
#define SUBF(D, A, B)            T_SUBF( D, A, B, 0, 0)
#define SUBF_(D, A, B)           T_SUBF( D, A, B, 0, 1)
#define SUBFO(D, A, B)           T_SUBF( D, A, B, 1, 0)
#define SUBFO_(D, A, B)          T_SUBF( D, A, B, 1, 1)
#define SUBFC(D, A, B)           T_SUBFC(D, A, B, 0, 0)
#define SUBFC_(D, A, B)          T_SUBFC(D, A, B, 0, 1)
#define SUBFCO(D, A, B)          T_SUBFC(D, A, B, 1, 0)
#define SUBFCO_(D, A, B)         T_SUBFC(D, A, B, 1, 1)
#define SUBFE(D, A, B)           T_SUBFE(D, A, B, 0, 0)
#define SUBFE_(D, A, B)          T_SUBFE(D, A, B, 0, 1)
#define SUBFEC(D, A, B)          T_SUBFE(D, A, B, 1, 0)
#define SUBFEC_(D, A, B)         T_SUBFE(D, A, B, 1, 1)
#define SUBFIC(D, A, SIMM)       Op_D_A_SIMM(8, D, A, SIMM)
#define SUBFME(D, A)             T_SUBFME(D, A, 0, 0)
#define SUBFME_(D, A)            T_SUBFME(D, A, 0, 1)
#define SUBFMEC(D, A)            T_SUBFME(D, A, 1, 0)
#define SUBFMEC_(D, A)           T_SUBFME(D, A, 1, 1)
#define SUBFZE(D, A)             T_SUBFME(D, A, 0, 0)
#define SUBFZE_(D, A)            T_SUBFME(D, A, 0, 1)
#define SUBFZEC(D, A)            T_SUBFME(D, A, 1, 0)
#define SUBFZEC_(D, A)           T_SUBFME(D, A, 1, 1)
#define SYNC() \
    Op_S_A_B_OpOp_Rc(31, 0, 0, 0, 598, 0)
#define TW(TO, A, B) \
    Op_S_A_B_OpOp_Rc(31, TO, A, B, 4, 0)
#define TWI(TO, A, SIMM)         Op_D_A_SIMM(3, TO, A, SIMM)
#define XOR(A, S, B)             T_XOR(A, S, B, 0)
#define XOR_(A, S, B)            T_XOR(A, S, B, 1)
#define XORI(A, S, UIMM)         Op_D_A_SIMM(26, S, A, UIMM)
#define XORIS(A, S, UIMM)        Op_D_A_SIMM(27, S, A, UIMM)

/* Simplified mnemonics. */
#define LI(rD, value)            ADDI(rD, 0, value)
#define LA(rD, disp, rA)         ADDI(rD, rA, disp)
#define SUBI(rD, rA, value)      ADDI(rD, rA, -value)
#define SUBIC(rD, rA, value)     ADDIC(rD, rA, -value)
#define SUBIC_(rD, rA, value)    ADDIC_(rD, rA, -value)
#define LIS(rD, value)           ADDIS(rD, 0, value)
```

```
#define SUBIS(rD, rA, value)     ADDIS(rD, rA, -value)
#define BDNZ(target)             BC(16, 0, target)
#define BLT(target)              BC(12, 0, target)
#define BGE(target)              BC(4, 0, target)
#define BGT(target)              BC(12, 1, target)
#define BLE(target)              BC(4, 1, target)
#define BEQ(target)              BC(12, 2, target)
#define BNE(target)              BC(4, 2, target)
#define BNL(target)              BGE(target)
#define BNG(target)              BLE(target)
#define BSO(target)              BC(12, 3, target)
#define BNS(target)              BC(4, 3, target)
#define BUN(target)              BSO(target)
#define BNU(target)              BNS(target)
#define BLTCTR()                 BCCTR(12, 0)
#define BCTR()                   BCCTR(20, 0)
#define BLTLR()                  BCLR(12, 0)
#define BLELR()                  BCLR(4, 1)
#define BEQLR()                  BCLR(12, 2)
#define BGELR()                  BCLR(4, 0)
#define BGTLR()                  BCLR(12, 1)
#define BNLLR()                  BCLR(4, 0)
#define BNELR()                  BCLR(4, 2)
#define BNGLR()                  BCLR(4, 1)
#define BSOLR()                  BCLR(12, 3)
#define BNSLR()                  BCLR(4, 3)
#define BUNLR()                  BCLR(12, 3)
#define BNULR()                  BCLR(4, 3)
#define BLR()                    BCLR(20, 0)
#define BLTLRL()                 BCLRL(12, 0)
#define BLELRL()                 BCLRL(4, 1)
#define BEQLRL()                 BCLRL(12, 2)
#define BGELRL()                 BCLRL(4, 0)
#define BGTLRL()                 BCLRL(12, 1)
#define BNLLRL()                 BCLRL(4, 0)
#define BNELRL()                 BCLRL(4, 2)
#define BNGLRL()                 BCLRL(4, 1)
#define BSOLRL()                 BCLRL(12, 3)
#define BNSLRL()                 BCLRL(4, 3)
#define BUNLRL()                 BCLRL(12, 3)
#define BNULRL()                 BCLRL(4, 3)
#define BLRL()                   BCLRL(20, 0)
#define CLRLWI(rA, rS, n)        RLWINM(rA, rS, 0, n, 31)
#define CLRRWI(rA, rS, n)        RLWINM(rA, rS, 0, 0, 31-(n))
#define CLRLSLWI(rA, rS, b, n) \
    RLWINM(rA, rS, n, (b)-(n), 31-(n))
#define CMPW(rA, rB)              CMP(0, rA, rB)
#define CMPWI(rA, value)          CMPI(0, rA, value)
#define CMPLW(rA, rB)             CMPL(0, rA, rB)
#define CMPLWI(rA, value)         CMPLI(0, rA, value)
#define CRNOT(crbD, crbA)         CRNOR(crbD, crbA, crbA)
#define CRMOVE(crbD, crbA)        CROR(crbD, crbA, crbA)
#define CRCLR(crbD)               CRXOR(crbD, crbD, crbD)
#define EXTLWI(rA, rS, n, b)      RLWINM(rA, rS, b, 0, (n)-1)
#define EXTRWI(rA, rS, n, b) \
    RLWINM(rA, rS, (b)+(n), 32-(n), 31)
#define INSLWI(rA, rS, n, b) \
    RLWIMI(rA, rS, 32-(b), b, (b)+(n)-1)
#define INSRWI(rA, rS, n, b) \
    RLWIMI(rA, rS, 32-((b)+(n)), b, (b)+(n)-1)
#define NOT(rA, rS)               NOR(rA, rS, rS)
#define MFXER(rD)                 MFSPR(rD, 1)
#define MFLR(rD)                  MFSPR(rD, 8)
#define MFCTR(rD)                 MFSPR(rD, 9)
#define MR(rA, rS)                OR(rA, rS,  rS)
#define MTXER(rD)                 MTSPR(1, rD)
#define MTLR(rD)                  MTSPR(8, rD)
#define MTCTR(rD)                 MTSPR(9, rD)
#define NOP()                     ORI(0, 0, 0)
#define ROTLWI(rA, rS, n)         RLWINM(rA, rS, n, 0, 31)
#define ROTRWI(rA, rS, n)         RLWINM(rA, rS, 32-(n), 0, 31)
#define ROTLW(rA, rS, rB)         RLWNM(rA, rS, rB, 0, 31)
#define SLWI(rA, rS, n)           RLWINM(rA, rS, n, 0, 31-(n))
#define SRWI(rA, rS, n)           RLWINM(rA, rS, 32-(n), (n), 31)
#define SUB(rD, rA, rB)           SUBF(rD, rB, rA)
#define SUB_(rD, rA, rB)          SUBF_(rD, rB, rA)
#define SUBC(rD, rA, rB)          SUBFC(rD, rB, rA)
#define SUBC_(rD, rA, rB)         SUBFC_(rD, rB, rA)
#define SUBE(rD, rA, rB)          SUBFE(rD, rB, rA)
#define SUBE_(rD, rA, rB)         SUBFE_(rD, rB, rA)
#define TWLT(rA, rB)              TW(16, rA, rB)
#define TWGT(rA, rB)              TW(8, rA, rB)
#define TWEQ(rA, rB)              TW(4, rA, rB)
#define TWLTU(rA, rB)             TW(2, rA, rB)
#define TWGTU(rA, rB)             TW(1, rA, rB)
#define TWGE(rA, rB)              TW(12, rA, rB)
#define TWGEU(rA, rB)             TW(5, rA, rB)
#define TRAP()                    TW(31, 0, 0)
#define TWLTI(rA, value)          TWI(16, rA, value)
#define TWGTI(rA, value)          TWI(8, rA, value)
```

```
#define TWEQI(rA, value)        TWI(4, rA, value)
#define TWLTIU(rA, value)       TWI(2, rA, value)
#define TWGTIU(rA, value)       TWI(1, rA, value)
#define TWGEI(rA, value)        TWI(12, rA, value)
#define TWGEIU(rA, value)       TWI(5, rA, value)

#endif  /* !PPCINSTR_H */
```

Appendix B

Summary of Changes to KVM

Table B.1 gives an overview of the changes that have been made to the KVM source distribution in order to facilitate our extensions.

File	**Description of changes**
`Makefile`	Add files in `VmJIT` to list of files to compile
`VmCommon/src/execute.c`	Add profiling, mixed-mode bytecode execution
`VmCommon/src/bytecodes.c`	No rescheduling after `new` opcode
`VmCommon/src/profiling.c`	Add JIT-related profiling variables
`VmExtra/src/main.c`	Add processing of `-cachesize` switch
`VmCommon/h/global.h`	`#include` JIT-related header files
`VmCommon/h/fields.h`	Add `basicBlockTable` to method structure
`VmCommon/h/profiling.h`	Export JIT-related profiling variables
`VmUnix/h/machine_md.h`	Set compile-time flags, add macros

Table B.1: Summary of changes made to the KVM distribution.