AZZAM MOURAD

A Selective Dynamic Compiler for Embedded Java Virtual Machine Targeting ARM Processors

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise en informatique
pour l'obtention du grade de Maître ès Sciences (M.Sc.)

FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL
QUÉBEC

Mars 2005

©Azzam Mourad, 2005

Je dédie ce travail à ma famille

Remerciements

Je tiens à remercier sincèrement ma directrice et mon co-directeur de recherche, les docteurs Nadia Tawbi et Mourad Debbabi pour leur encadrement, leur encouragement et leur grande compréhension. Leur conseils prodigués, leur appui constant et leur rigueur scientifique manifestée tout au long de la durée de cette recherche m'ont sans cesse incité à persévérer dans mes travaux. Ils ont su comment me motiver et me guider dans l'avancement de mes travaux. Je leur remercie pour leur gentillesse, leur support et la qualité de la relation humaine qui existe entre nous.

Je désire remercier Dr. Mohamad Mejri pour avoir accepté d'être mon examinateur et pour la qualité de son évaluation et de ses remarques.

Je remercie mes collègues, avec qui j'ai travaillé dans une ambiance saine et chaleureuse, pour leur support et leur aide.

Je remercie tous mes amis pour être des vrai amis et frères.

Je tiens à exprimer ma gratitude et adresser mes remerciements les plus sincères à mes parents, mon frère et mes soeurs pour leur amour, leur encouragement, leur conseils précieux et leur support moral.

Enfin, je remercie tous ceux qui ont contribué, de prés et de loin, à la réalisation de ce travail et à mon succès.

Résumé

Ce travail présente une nouvelle technique de compilation dynamique sélective pour les systèmes embarqués avec processeurs ARM. Ce compilateur a été intégré dans la plateforme J2ME/CLDC (Java 2 Micro Edition for Connected Limited Device Configuration). L'objectif principal de notre travail est d'obtenir une machine virtuelle accélérée, légère et compacte prête pour l'exécution sur les systèmes embarqués. Cela est atteint par l'implémentation d'un compilateur dynamique sélectif pour l'architecture ARM dans la Kilo machine virtuelle de Sun (KVM). Ce compilateur est appelé Armed E-Bunny. Premièrement, on présente la plateforme Java, le Java 2 Micro Edition(J2ME) pour les systèmes embarqués et les composants de la machine virtuelle Java. Ensuite, on discute les différentes techniques d'accélération pour la machine virtuelle Java et on détaille le principe de la compilation dynamique. Enfin, on illustre l'architecture, le design (la conception), l'implémentation et les résultats expérimentaux de notre compilateur dynamique sélective Armed E-Bunny. La version modifiée de KVM a été portée sur un ordinateur de poche (PDA) et a été testée en utilisant un benchmark standard de J2ME. Les résultats expérimentaux de la performance montrent une accélération de 360 % par rapport à la dernière version de la KVM de Sun avec un espace mémoire additionnel qui n'excède pas 119 kilobytes.

Abstract

This work presents a new selective dynamic compilation technique targeting ARM 16/32-bit embedded system processors. This compiler is built inside the J2ME/CLDC (Java 2 Micro Edition for Connected Limited Device Configuration) platform. The primary objective of our work is to come up with an efficient, lightweight and low-footprint accelerated Java virtual machine ready to be executed on embedded machines. This is achieved by implementing a selective ARM dynamic compiler called Armed E-Bunny into Sun's Kilobyte Virtual Machine (KVM). We first present the Java platform, Java 2 Micro Edition (J2ME) for embedded systems and Java virtual machine components. Then, we discuss the different acceleration techniques for Java virtual machine and we detail the principle of dynamic compilation. After that we illustrate the architecture, design, implementation and experimental results of our selective dynamic compiler Armed E-Bunny. The modified KVM is ported on a handheld PDA and is tested using standard J2ME benchmarks. The experimental results on its performance demonstrate that a speedup of 360% over the last version of Sun's KVM is accomplished with a footprint overhead that does not exceed 119 kilobytes.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivations

The use of wireless systems such as PDAs, cell phones, pagers, etc. becomes a need in our everyday life. In this context, the platform Java, and in particular J2ME/CLDC (Java 2 Micro Edition for Connected Limited Devices Configuration) is now recognized as the standard execution environment for these types of wireless devices due to its security, portability, mobility and network features. The deployment of Java enabled wireless devices reached nearly 15 million units in 2001 and will likely exceed 100 million in 2002. This trend is expected to continue at a nearly exponential pace in the next few years. The scope of the underlying platforms covered by Java goes from powerful systems such as servers, desktop, etc. to resources-limited devices such that PDAs, cell phones, pagers and house appliances. In order to cope with the different requirements of this large range of platforms, Sun Microsystems offers three adequate platforms: J2EE (Java 2 Enterprise Edition) for servers, J2SE (Java 2 Standard Edition) for desktop workstations and J2ME for embedded devices.

In this work we are concerned with the platform J2ME used in embedded systems, which are called also resources-limited systems due to their limitations in terms of memory and power. The Java virtual machine built inside J2ME is called KVM (Kilo Virtual Machine). It is traditionally interpreter-based and designed especially to deal with the special characteristics and limitations of embedded devices. The interpreter of KVM emulates the execution of Java bytecodes on a specific platform. While the main advantages of the interpretation mechanism are the simplicity and portability, its severe drawback remains definitely its poor performance. Although the problem of performance is applicable on all the Java platforms (i.e. J2EE, J2SE and J2ME), the

lack of memory available in embedded systems adds more difficulties to find acceleration solutions for J2ME or to apply already existing ones in other Java platforms.

At the same time, the ARM architecture is becoming the industry's leading 16/32 bit embedded system processor solution due to its performance and RISC (Reduced Instruction Set Computer) features. ARM powered microprocessors are being routinely designed into a wider range of products than any other 32-bit processor. This wide applicability is made possible by the ARM architecture, resulting in optimal system solutions at the crossroads of high performance, small memory size and low power consumption. All these factors make the combinations of the ARM architecture and the J2ME/CLDC an interesting domain for research in terms of acceleration. Many people have been interesting to enhance the performance of the Java virtual machine and lot of techniques have been proposed. These techniques are divided into two main approaches: hardware and software acceleration.

Regarding hardware acceleration, a significant speedup in term of virtual machine performance is achieved. However, the high power consumption and the cost of these acceleration technologies encourage researchers to deviate to software acceleration of embedded Java virtual machine. This energy issue is really damaging especially in the case of low end mobile devices. As examples of these hardware acceleration techniques, many companies such as Zucotto Wireless, Nazomi, etc. have proposed Java processors that execute in silicon Java bytecodes. For software acceleration, general optimizations, ahead-of-time (AOT) optimizations, just-in-time (JIT) compilation and selective dynamic compilation are in general the four categories of software acceleration techniques. General and ahead-of-time optimization can lead to reasonable acceleration, however these techniques are not competent to JIT and selective dynamic compilation which can reach a speedup that exceeds 200%.

JIT techniques can dramatically increase the execution speed of Java programs, however, they are inappropriate in the context of embedded systems owing to their large code size. The compilation process also implements sophisticated flow analysis and register allocation algorithm to generate optimized and high quality code. Even though, almost all standard Java virtual machine such as HotSpot VM [35], IBM Jalapeño [2], IBM JDK [33], Intel ORP [2], IBM Mixed-Mode-Compiler [34], Latte [43], OpenJIT [24] and Kaffe [41] are endowed with such dynamic compilers. Researches proved that the JIT is more practical for desktop Java virtual machine and the selective dynamic compilation is the best optimization technique for the embedded Java virtual machine built inside resources-limited systems. Selective dynamic compilation optimizes programs at runtime, based on information available only at runtime, thus offering the potential for greater performance and less memory use. It deviates from JIT compi-

lation by selecting and compiling only those java code fragments that are frequently executed. This technique can significantly accelerates the virtual machine and at the same time reduces the memory overhead.

This work mainly describes the design and the implementation of Armed E-Bunny, a lightweight selective dynamic compiler targeting ARM processors. Based on a selection technique, Armed E-Bunny overcomes all the drawbacks stated above about integrating such compilers inside an embedded Virtual Machine and comes up with a speedup of 360% over the last version of Sun's KVM with a footprint overhead that does not exceed 119KB.

1.2 Objectives

The following are the main objectives of this work:

- Show the role of Java platform and ARM architecture in the advent of embedded technology.
- Present the actual acceleration techniques for Java platforms and in particular for J2ME/CLDC.
- Study the problem found in some virtual machines endowed by dynamic compilers.
- Demonstrate that the selective dynamic compilation is the best solution for embedded Java virtual machine.
- Design and implement a selective dynamic compiler into the KVM.
- Port and execute our system on embedded devices.
- Accomplish an efficient speedup over Sun's virtual machine with the minimum footprint overhead.

1.3 Contributions

The following are the contributions of this work:

- Our dynamic compiler is very efficient in terms of performance while the memory footprint overhead does not exceed 119KB. Results tested on an Embedded-Linux Handheld demonstrate that the modified virtual machine is 3.6 time faster than the last version of Sun's Java virtual machine.
- Our system is the first academic work that targets CLDC-based embedded Java virtual machine by dynamic compilation and one of few commercial systems that target ARM microprocessors.
- Our solution covers also the different issues of integrating a dynamic compiler into a Java virtual machine such as exception handling, garbage collection, threads, switching mechanism between the compiler and the interpreter, etc.

1.4 Document Structure

This document is composed of five chapters. Java platform, Java 2 Micro Edition (J2ME) for embedded systems and the different Java virtual machine components are described in Chapter 2. Chapter 3 discusses the different acceleration techniques for Java virtual machine and details the principle of dynamic compilation. Chapter 4 highlights the architecture of ARM platform as well as the architecture, design and implementation of our selective dynamic compiler Armed E-Bunny. All the detailed information of our system, the difficulties we faced and the experimental results are also stated in this chapter. Finally, the conclusion and the future work are presented in Chapter 5.

Chapter 2

Java platform

2.1 Introduction

The Java technology allows the development of efficient and secure cross-platform software and the Java virtual machine (JVM) is the cornerstone of this technology. The JVM relies on an interpretation mechanism which emulates the execution of Java bytecodes on a specific platform. Although this technique provides portability and platform independency, its main severe drawback remains definitely its poor performance. In this chapter, we discuss first the main components of the Java virtual machine. Then we focus on the embedded Java platform and we describe its architecture, specifications and limitations. Finally, we present the Kilo Java virtual machine and the conclusion.

2.2 Java Platforms

The scope of platforms covered by Java goes from powerful systems such as servers and desktop to resources-limited devices such as PDAs, cell phones, pagers, etc. In order to fulfill the requirements of all these machines, Sun microsystems offers three adequate platforms: J2EE (Java 2 Enterprise Edition) for servers [30], J2SE (Java 2 Standard Edition) for desktop workstations [40] and J2ME (Java 2 Micro Edition) for embedded devices [36]. The JVM is the main component of a Java platform [21]. Some JVM specifications differ with respect to the platform built inside, even though, all of them are based on an interpreter which translates the Java bytecodes into specific machine code. For reason of portability, a Java program is first compiled to machine-independent

codes called bytecodes. The bytecodes generated are then interpreted by the virtual machine to machine-dependent code to be executed on a specific processor.

2.2.1 Java Compilation

Java is a platform independent high level programming language. Although Java is kept very close in terms of syntax and structure to other programming language such as C and C++ , the compilation process is performed differently. Compiling a Java source code provides a list of intermediary form code called bytecodes, which are translated by the Java virtual machine to a specific platform machine code. The generated bytecodes are stack-based and machine-independent. They are stored in a *.class* file that has a special format defining the way of storing Java classes in a platform independent form. Additional information regarding the class access flags, constant pool, list of fields, list of methods, usage of stack and local variables are also stored in the *.class* file. Indeed, this process ensures the portability of Java and permits a Java *.class* file to be run on any machine that has a Java virtual machine built inside.

2.2.2 Java Bytecodes

Java Bytecodes are code generated by a Java compiler and are defined in the Java virtual machine as stack-based instructions [4]. It is a low level language in which several categories of instructions can be distinguished. When a method is invoked, the set of bytecodes that represents it in the *.class* file is interpreted by the virtual machine and then the resulting machine code is executed by the corresponding machine processors. Each bytecode is composed of one byte opcode and zero or more operands. The operands represent a bytecode's parameters and give additional information needed for execution. Specifically, Java bytecode provides instructions for performing a different kind of register and stack operations. This includes pushing and popping values onto/from the stack, manipulating the register content and performing logical and arithmetic operations. As for transfer control instructions, Java bytecodes support both conditional and unconditional jump. There are also some high level bytecodes that are Java specific and that allow for array/object field access, object creation, method invocation as well as type casting and function return. JVM distinguishes 200 standards, 25 quick and 3 reserved bytecodes. The quick ones are quick versions of some standard ones and are created only at runtime to take advantage of previous work done and enhance the execution performance. The reserved ones are special purpose bytecodes and are used only internally by the JVM. The fast invoke, get/set fields and

new are examples of those bytecodes that have quick versions.

2.2.3 Java Execution

The Java virtual machine is the software responsible of executing a Java program on a machine. The implementation of the virtual machine differs with respect to the target machine processors and specifications. In this context, there are four ways through which a Java program is executed: an interpreter, a static compiler, a dynamic compiler and a Java processor. After compiling a Java program by a Java compiler, a command line typed by the user loads the resulting *.class* file together with the system classes and superclasses in the Java virtual machine for execution. Once the loading and initialization mechanisms are completed, a verification process is applied in order to check the structure and the well-typing of the loaded classes. After that, one of the ways mentioned above is performed in order to interpret or compile the bytecodes representing each method saved in the *.class* file and emulates their execution on a specific platform. Indeed, some functions in the virtual machine are responsible of gathering the references and parameters of each method, saving and restoring contexts and preparing the stack for execution.

When a Java method is called, a frame is created. Each method frame contains references to:

- The frame pointer (FP).
- The local variables (Locals Pointer: LP) of the called method.
- The runtime constant pool (CP) of the called method.
- The top of the stack of the called method (Stack Pointer: SP).
- The instruction's pointer following the method's call instruction (Instruction pointer: IP).
- The top of the stack of the calling method.
- The frame pointer of the calling method.

Once a return from the called method is performed, the frame of the called method is destructed and the execution continues at the instruction where IP points. The FP and SP will be set to the values of the calling method's FP and SP which are saved in the context of the called method as previous FP and SP.

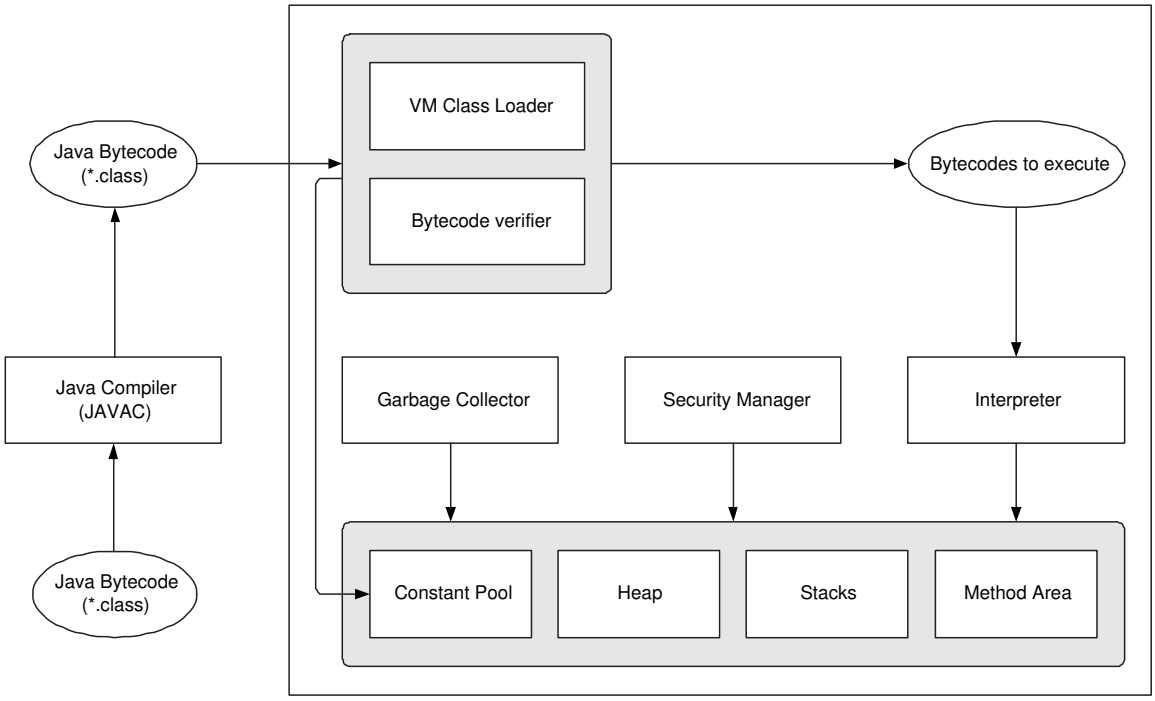

Figure 2.1: JVM Architecture

2.3 Java Virtual Machine

With the advent of Internet and mobile technology, Java gained more popularity due to its portability. Moreover, Java is object oriented programming language, supports multi-threading, includes garbage collector and offers high level security. All these features are provided mainly by its virtual machine. The Java virtual machine (JVM) is a runtime environment consisting of several components that provide Java with platform independency, security and mobility. JVM is an an abstract computer that can load and execute Java programs. In this section we detailed the role and features of the main JVM components responsible of Java bytecodes execution. Figure 2.1 shows these components.

2.3.1 Class Loader

Class loaders are special Java runtime objects. They are used to achieve the loading of classes into the JVM. We distinguish mainly the following:

- System or primordial class loader: This loads system classes from the CLASS-

PATH location.

- Applet class loader: This loads the applet and all the referenced classes.
- RMI class loader: This loads classes for the purpose of remote method invocation.
- User-defined class loader (not trusted): This is a customized class loading that is application dependent.

Class loader security is paramount for the defense against malicious code. It is meant to protect Java classes, and in particular the standard Java library from spoofing attacks. This is done by a delegation of the loading requests to the primordial loader. A class loader defines a distinct namespace for the classes it loads. Therefore multiple code units could be loaded from different sources without a risk of class name collision, and hence could not interfere. This aims basically at ensuring mutual protection among these mobile code units.

2.3.2 Bytecode Verifier

The bytecode verifier can be considered as the low level security layer in the JVM security architecture. The objective of the verifier is to guarantee the basic safety properties. These include stack/registers safety, type safety, object safety and control flow safety. The bytecode verifier achieves the verification process after the loading of a class. This process includes a verification of the syntax and semantics of the *.class* file. The bytecode verifier inspects the bytecodes of the methods to check their validity. It also achieves a data flow analysis to check the type safety property.

2.3.3 JVM Runtime Heap and Structures

Every Java virtual machine must implement some structures and use particular memory space in order to execute a Java program. The constant pool, the method area, the JVM stacks and the method frames are the JVM runtime structures implemented in the Java virtual machine. The heap is the name of the memory space reserved for saving these structures and other virtual machine running purposes.

Heap

The heap is a memory space in which all the JVM runtime structures and allocated objects are saved during a Java program execution. The heap is divided into two sectors: permanent space and garbage collected space. The permanent space is not scanned by the JVM garbage collector and then it cannot be freed once it is full. The permanent space is used to store some structures such as Method area. On the other hand, the garbage collected space is used to allocate JVM stacks and objects. When this space is full, the garbage collector passes through it and frees all the unreferenced objects.

Constant Pool

During the Java code execution, different Java runtime structures are maintained. Function names, constant valued, class descriptions and method descriptions are instances of such structures. All these data are maintained by JVM in a special area called Constant Pool. These data are used by almost every Java runtime component.

Method Area

This area contains all the bytecodes of the methods. During execution, the program counter always points to the next bytecode in the method area to be executed. The program counter is usually saved into a reserved machine register.

JVM Stacks

The execution of a Java program by the Java virtual machine is stack-based. All the data operations and local variables are carried out through a stack created in the heap and called JVM stack. Since mutltithreading execution is supported, the virtual machine uses a different stack for each thread.

Method frames

A method frame is a part of a thread JVM stack created for each method at the moment of its invocation. Each method frame contains the method local variables,

method arguments, calling method context registers and free spaces reserved for the method bytecodes execution. At any given moment, only one frame is active and there are some machine registers specified to point to the corresponding active frame. These machine registers differ from platform to another and are updated each time another method is called or a return to the calling method is performed.

2.3.4 Interpreter

The interpreter is the Java virtual machine component responsible of translating a Java program bytecodes to executable machine code. It offers a high degree of hardware abstraction and provides portability for the JVM. It can be recompiled very easily for any architecture with almost no changes. In fact, the interpretation is a very popular technique used in programming language implementation. Its is very simple, easy to implement, does not require large memory space and provides high portability. However, the main drawback of applying such approach is the poor performance it causes to the JVM at runtime. The interpreter is 5 to 20 times slower than the native code execution. In this context, many approaches have been proposed in order to enhance its performance. Fast-Interpreter is one of these approaches. Such approaches translate the bytecodes to threaded code and then interpret it using a generated interpreter particularly designed for threaded code. However, results shows that optimizing only the interpreter does not lead to high execution speedup, from which was the need to propose other techniques. Some of these techniques replace the interpreter, others run in parallel with it. Dynamic compilation is an instance of these techniques. A dynamic compiler can decrease the running time of a Java virtual machine by a factor of 20.

The interpreter is a software emulation of the processor. It is in general implemented by a main loop that iterates on the called method bytecodes and performs their functions in order. Inside the loop, a switch case is able to dispatch the sequence of bytecodes and differentiate among their execution. Each case value implements one Java bytecode. For each called method, a frame is created and is used as a space of execution. This frame contains all information and date needed to execute a method properly.

2.3.5 Security Manager

The security manager represents the last level in JVM security architecture. The role of the security manager is to guard the application security policies. It ensures the high-level security properties such that some specific files should never be erased or an

internet connection towards some hosts should never be established. The Java API queries, via the Java virtual machine, the security manager before performing any potential security sensitive action. The security manager implements the appropriate check methods that enforce the security policy. The security manager is then able to stop a non-permitted operation by throwing a security exception.

2.3.6 Garbage Collector

A key feature of Java is its garbage collection mechanism, which takes care of freeing dynamically allocated memory that is no longer referenced. The JVM heap stores all the objects created at runtime during a Java program execution. Whenever an allocation is needed and no more heap space is available, the garbage collector is called. Furthermore, a garbage collector may also combat heap fragmentation. Some free blocks of heap are left in between blocks occupied by live objects. Request to allocate new objects may have to be filled by extending the size and changing the fragmentation of the heap. This is performed whenever there is no contiguous free heap space that fits with the size of the new object.

Because the heap is garbage collected, Java programmers don't have to explicitly free allocated memory. Moreover, knowing when to explicitly free allocated heap can be very tricky, hence, giving this job to the JVM has several advantages. First, programmers do not have to spend time on chasing elusive memory problems. This helps them to be more productive when programming in Java than any other programming language. Second, garbage collection ensures a program integrity. It is an important part of Java security that prevents the JVM crashes caused by improper memory freeing. However, a remarkable disadvantage of a garbage collected heap is is the overhead time needed to scan the heap, free unreferenced objects and even change fragmentation. This affects the performance of the Java virtual machine at runtime. Fortunately, a panoply of garbage collection algorithms have been developed in order to deal with this drawback. Although many different techniques have been applied on different Java platforms virtual machines, all of them must do two basic things. First, they have to detect no longer referenced objects. Second, they have to eliminate the unused objects in the heap and make it available for eventual storage. Fragmentation change may also be applied in some cases. In this section, we distinguish first the reference counting and tracing collectors which are two approaches used to select objects to be eliminated by the garbage. Then, we describe the compacting and copying collectors, which are responsible of freeing unreferenced objects and combatting heap fragmentation [16, 42].

Reference Counting Collector

Reference counting collector is an old garbage collection strategy which distinguishes live objects from garbage ones, by checking the value of the counter of each object in the heap. Each object has a reference count that is set to one at the object creation moment. This counter is incremented by one each time the object reference is used during a program execution. On the other hand, this counter is decremented by one when the object reference goes out of scope or is assigned a new value. The garbage collector selects and eliminates the objects with counter values equal to zero. The main advantage of this strategy is the small chunks of time required to trigger the garbage collector. It is relevant for programs that cannot be interrupted for long time. However, this type of collectors does not decrement the counter's values of the objects that refer to one another, which means some object counters will never reach zero even if they are no longer reached by the program.

Tracing Collector

Tracing collector is the garbage collection algorithm mostly used by the Java virtual machines. It is known also as mark and sweep algorithm. During the first phase of this type of garbage collection process, the garbage collector scans the graph of object references and marks all the objects encountered by setting some flags in a particular data structure. The second phase, which is called sweep, consists of removing the unmarked objects and freeing heap spaces. In the Java virtual machine implementations, the sweep phase includes also other steps to combat heap fragmentation in case no more heap space is available to free. Compacting and copying collectors are two different strategies used to perform this work.

Compacting and Copying Collectors

Once the garbage collection is called and no more free heap space is available to free for the new allocation, the garbage collector combats heap fragmentation in order to find relevant heap space that fits with the size of the new allocated object. In this context, there are two techniques used in order to move objects on the fly to reduce heap fragmentation. The fist one is called compacting collector, while the second one is called copying collector.

The Compacting collector eliminates unused free spaces between allocated objects by

pushing all referenced objects over one side of the heap. This process fills an entire side of the heap with non-garbage references, while the other side becomes empty and ready for new allocation. All the object references are updated by the new heap locations. The main disadvantage of compacting collector is the overhead time needed to change the heap fragmentation.

The Copying collector divides the heap into two space area. Only one space area is used at once, and a switch between the two area is applied when the current used one is full. Moving all live objects to the other area eliminates also the empty unused spaces between allocated objects, which means there is no need to combat heap fragmentation. At the switch moment, the program execution is stopped until the copy process is accomplished. A common copying collector is called Stop and Copy. The main drawback of this strategy is the double heap size required during execution because only one half of the heap is available at once for allocation and storage.

2.3.7 Exception Handling

The Java programming language provides the exception handling mechanism to help programs report and handle errors. This mechanism is very convenient for developers to detect the errors that occur when the Java semantics are violated. When an error occurs at runtime, the program or the virtual machine throws an exception. If this particular type of errors is handled, the exception is catched by a block of code called exception handler found in the program. An exception handler can handle one or many types of errors. Once an exception is handled, the program continues its execution at the first catch that fills this condition. Otherwise, an interrupt occurs and the program exists abnormally.

The statement used for throwing and handling the exception is a try/catch/finally, try/catch or try/finally. Table 2.1 shows an example of this process. The try statement identifies a block of statements within which an exception might be thrown. The catch statement must be associated with a try statement and identifies a block of statements that can handle a particular type of exception. The statements are executed if an exception of a particular type occurs within the try block. The finally statement must be associated with a try statement and identifies a block of statements that are executed regardless of whether or not an error occurs within the try block.

```
try
{
    statement (s)
}

catch (exceptiontype name)
{
    statement (s)
}

finally
{
    statement (s)
}
```

Table 2.1: Exception Handling

2.3.8 Threads

One of the feature of the Java language is the multithreading support. Java allows programmers to create many threads as part of an application or applet. All threads within a Java program can execute within a shared memory, can share access to objects and can send notifications to each other. Multithreading is an essential part of the Java virtual machine specification. It is handled at software level. This support enhances program portability and also provides for a significant increase in programmer productivity over a linked library approach.

Java virtual machine can have several threads of execution at the same time. Each thread has a structure representing its state and a stack for its execution. At start-up, the main thread is first created, then additional threads can be spawn as desired. An interesting aspect of Java multithreading is the technique performed to schedule threads switching and control the access to the shared objects. This technique applied consists of two processes called threads switching and threads synchronization.

The Java virtual machine schedules between threads to access the machine processor. Each thread is assigned a time-slice counter which specifies its time to execute its code. The technique used is priority based. The thread with high priority has advantage over the other ones to run and the new created thread is assigned the same priority as its creator. Whenever a thread time-slice counter reaches zero, the Java virtual machine switches automatically to another thread with lower priority. In case there are many threads with the same priority, the FIFO ordering technique is applied. The switch mechanism requires saving the context of the current method in the thread structure (i.e fp, ip and sp values) and loading another method context of the thread to which

the execution switches. This context saving permits a thread to eventually continue its execution from the same point it stopped before the switch.

Furthermore, shared objects cannot be accessed by two or more threads at the same time. For this purpose, the Java virtual machine provides a lock for each object in order to synchronize between threads, so each thread has to lock the object before accessing it or executing any of its methods, and other threads are prevented from having access to the same object until the lock owner releases it. The threads who need to access a locked object are put in the waiting list of the object lock. It is a queue list used to keep order in accessing the corresponding object. Sometimes a notification is performed by the virtual machine to awake blocked objects.

2.4 Java for Embedded Systems

With the advent of Internet, connected intelligent information appliances such as cell phones, pagers, personal organizers, PDAs, etc. are becoming more popular and important in our everyday lives. The number of these embedded devices are increasing rapidly. In this context, the Java language is a good choice for many embedded industries. For instance, according to Sun microsystems statistics, more than 400 millions of Java phones were deployed in the market in 2003 and more than 600 millions are expected during 2004 [39].

In fact, an important improvement in embedded technology has been accomplished. They are now more connected to the Internet by either wire or wireless connection. Embedded devices allow now to browse and download new services and applications online such as interactive games, banking and ticketing applications, wireless collaboration and so on. All these features require a relevant platform for application development available for these devices. Java has met the demand of this kind of machines and has extended its scope with the introduction of Java 2 Micro Edition (J2ME) technology. Java is now the main platform of a huge number of embedded devices of different types and this number is increasing day after day. It is enabling the development of many new and powerful information appliances products and allows users, service providers and device manufacturers to profit of a rich collection of service applications delivered to user devices on demand.

2.4.1 Embedded Systems

The use of embedded devices such as cell phones, pagers, screen phones, PDAs, etc. increases day after day. The services provided by such devices become more efficient and relevant to help people in their everyday business and private lives. Embedded

devices have limited resources in term of memory, processor and power. These restrictions results in many problems for software that will run on such machines. These software should have small size executable code which should execute operations that do not require lot of memory, processor and power consumption. Java has deployed the Java 2 micro edition platform which is designed specifically to be relevant for embedded devices. However, the execution performance is the main drawback in this platform. Many researches have been done in this domain to solve this problem and many approaches have been proposed.

An embedded devices is a very small machine. Its memory is a precious resource. Such type of devices has an available memory less than 512 kilobytes to handle the entire Java runtime environment. The minimum requirement is 300 kilobytes of RAM and 1 megabyte of flash and ROM. The available memory space is used to store heap data, subroutines and stack information. So, the memory footprint of the Kilo virtual machine (KVM) of J2ME should have the minimum size possible. Regarding the battery life, the design of the virtual machine should preserve the maximum power in order to extend the mobility duration. An important enhancement has been accomplished in this context in order to enlarge the memory space and battery life, however, the small size of these machines and their connectionless feature are still the main problems for developers.

2.4.2 Java 2 Micro Edition

To fulfil the need of all the products market, and depending on the resources available and services needed, Sun has grouped its Java technology into three platforms categories as follows:

- J2EE used for servers and services providers workstations.
- J2SE used for destop computers.
- J2ME used for embedded devices.

Figure 2.2 illustrates the different Java platforms together with the machines using them. In fact, Sun has recognized that the first two edition J2EE and J2SE cannot be deployed into embedded devices due to the resources they require to be run. For this reason, J2ME is particularly addressed to take into account the embedded systems characteristics and has reached, since its deployment in 2002, a good success in the market of restricted resources devices. J2ME does not need high resources machines to be executed. In this context, some other features of other Java platforms, such as the floating point, the Java native interface (JNI) and the security manager of the Java

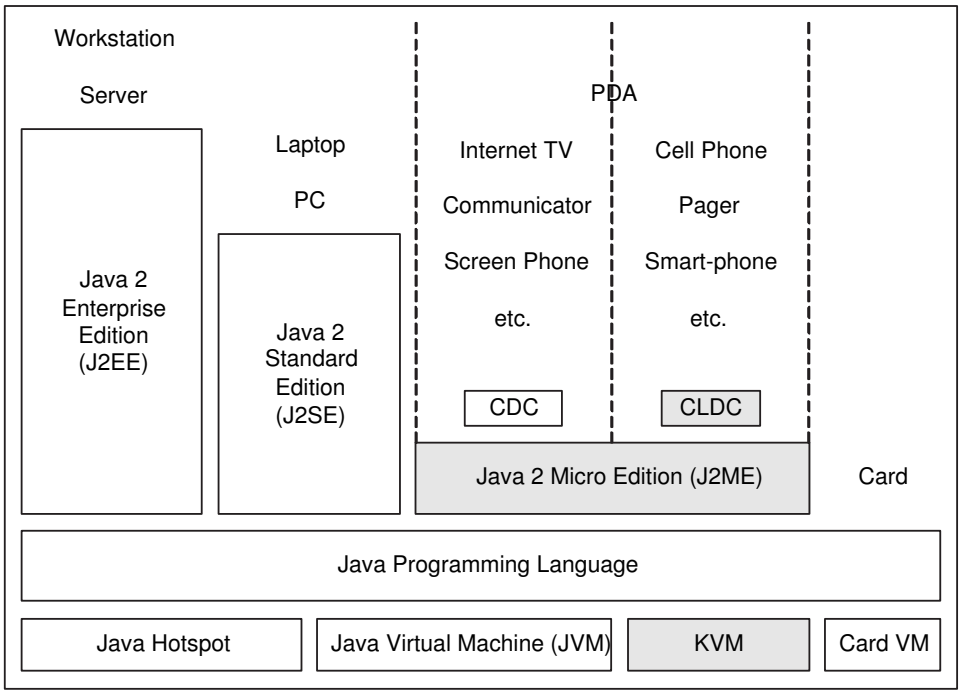

Figure 2.2: Java Platforms

virtual machine, do not exist in the J2ME edition. However, J2ME is still so powerful in terms of high level object oriented programming language, portability and security and has maintained its compatibility with the other Java platforms J2SE and J2EE.

J2ME is presently the execution environment platform of many types of devices. These devices are divided into two categories of products as follows:

- J2ME/CDC: This category of products include TV set-top boxes, Internet TVs, Internet-enabled screen phones, high-end communicators and automobile entertainenment/navigation systems. CDC means Connected Device Configuration. These devices have more resources than the small embedded devices. These resources consist of large user interface capabilities, 2 to 16 megabytes of memory space and high bandwidth network connection.

- J2ME/CLDC: This category of products include cell phones, pagers, personal organizers, etc. They are known as CLDC (Connected, Limited Device Configuration) devices. These embedded devices have very restricted resources which consist of simple user interface, memory size less than 512 kilobytes to handle runtime environment and low bandwidth connection.

Indeed, all the J2ME machines are more likely converting to use wireless connection, hence the differences between the above two categories are becoming mainly defined in

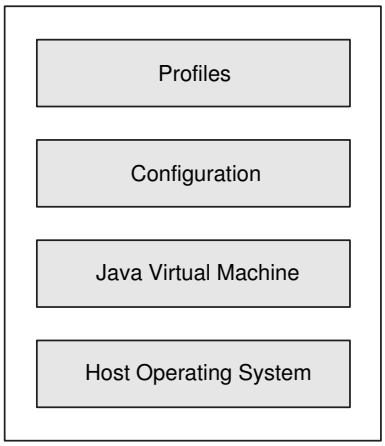

Figure 2.3: J2ME Architecture

terms of memory size, battery life durability, bandwidth consideration and physical screen size.

The flexibility of J2ME is maintained thanks to an architecture designed to be modular and scalable. The J2ME architecture consists of three related software layers built upon the operating system of the target machine. The three layers that are shown in figure 2.3 are: Java virtual machine, configuration and profiles. The first layer above the operation system is the Java virtual machine. It is an implementation of a Java virtual machine that targets a family of devices supporting a particular J2ME configuration. The implementation of the virtual machine differs with respect to the J2ME configuration. For instance, the J2ME/CDC has the classic Java virtual machine implementation built inside, while the J2ME/CLDC has the Kilo virtual machine which has different implementation addressed to limited resources devices. The second upper layer is the configuration layer. Its placement is in the middle in order to relate the virtual machine layer to the profiles layer. The configuration layer defines the set of Java virtual machine features and the Java class libraries supported by a set of devices having same characteristics. In J2ME, Sun distinguishes two types of configuration: CDC and CLDC. The third layer is the profiles. It is the only layer visible to users that provides the execution environment to run applications. The profiles layer consists of a set of Application Programming Interfaces (APIs) built to target a particular family of devices. Some devices can support many profiles types, while others such as embedded devices support only one which is the MIDP (Mobile Information Device Profile).

J2ME Profiles

The J2ME profiles layer is the most visible layer to users layered on top of the configuration [38]. It is defined as a collection of Java APIs and class libraries specific for a

family of devices. A profile is a Java application execution environment addressed to a set of devices having similar features. In fact, a profile provides the portability to the applications written and deployed by different manufacturers in order to be run on all devices supporting this profile. To provide portability, the profile provides a complete toolkit to implement application for devices supporting J2ME such as cell phones, pagers, set-top box, washing machine, or interactive electronic toy. Furthermore, the profile is created to support and execute a group of application addressed to several categories of devices, even if these devices do not have the same features and resources. For instance, a cell phone and a washing machine have totally different machine characteristics and resources, even though, they can have the same profile built inside to run some applications relevant for both of them. Presently, MIDP is the only profile existing designed for cell phones and related devices.

J2ME Configuration

Java configuration defines a Java platform for a particular category of devices with similar requirements, resources and hardware capabilities. It links a J2ME profile to the relevant Java virtual machine. To be more specific, a configuration specifies the Java programming language features supported, the Java virtual machine features supported and the basic Java libraries and APIs supported. For instance, the devices having restricted resources will be grouped together and will have a specific configuration, while the others with high resources will have a different configuration. Moreover, a profiler should be compatible with the configuration and the virtual machine at the same time. This compatibility assures the portability feature of the J2ME platform. Since all the features are automatically included in the profiler and each configuration specifies the virtual machine features, so any applications implementer can assume that an application addressed for a particular profile is also addressed to its corresponding configuration and Java virtual machine. Hence, it can be executed on all the devices that support this profile. For example, an application addressed to the profile of the embedded systems (i.e MIDP) can be executed on all the embedded devices that support the MIDP profile. Sun Microsystems has defined two standard J2ME configurations: J2ME/CLDC and J2ME/CDC. This distinction is mainly based on the resources and capabilities of the J2ME devices. Figure 2.4 shows the relations between CLDC, CDC and the Classical Java platform J2SE in terms of Java classes libraries and features supported. This relation is more detailed in the following two paragraphs.

CLDC CLDC (Connected Limited Device Configuration) is the configuration built for embedded devices such as personal, mobile, connected information devices [39]. Such devices have simple user interface, small memory size and low bandwidth connection.

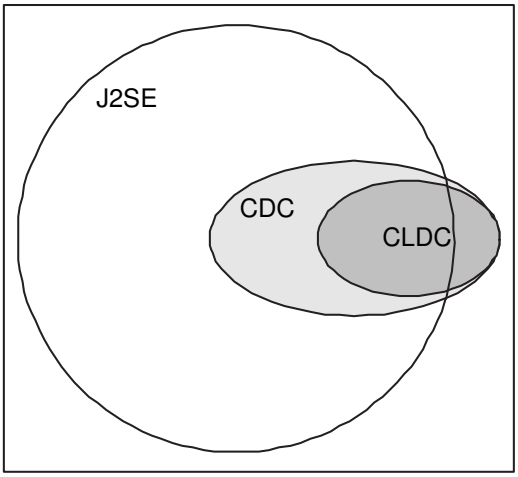

Figure 2.4: CDC, CLDC and J2SE

The majority of the Java classes of this configuration are either the same classes or subclasses of the classes of the J2SE platform. Moreover, this configuration includes also some new classes designed particularly for small-footprint devices. In fact, the reason, why some features in J2SE are not included in J2ME/CLDC, is the limitation of the devices supporting J2ME/CLDC. For instance, some features need large memory space which is not available in embedded devices. To deal with that, Sun Microsystems has omitted some features and re-implemented others that do not fit in CLCD in order to respect the hardware capabilities of J2ME/CLDC machines. The Java virtual machine supported in J2ME/CLDC is called the Kilo virtual machine (KVM). Many features that exist in the classical Java virtual machine are not implemented nor supported inside it. A detailed description of the KVM is presented later in this chapter.

The Java programming language supported in CLDC supports all the specifications of the classical Java language except:

- The Object.finalize() which finalize a class instance.
- The float and double data types.
- Some subclasses of Java.lang.error. The absence of these subclass restricts the error handling capabilities.

The Kilo Java virtual machine supports all the specifications and features of the classical virtual machine except:

- The security management performed by the security manager. Indeed the security manager component of the classical JVM does not exist in the virtual machine of J2ME/CLDC.

- The user defined Java level class loaders.
- The float bytecodes.
- The thread groups and daemon threads.
- The Java native interface (JNI).
- The Weak references.
- The Reflection.

CDC CDC (Connected Device Configuration) is the configuration built for devices such as shared, fixed, connected information devices. Such devices have more resources than the small embedded devices. These resources consist of large user interface capabilities, 2 to 16 megabytes of memory space and high bandwidth network connection. As shown in figure 2.4, CDC is considered as a superset of CLDC and a subset of J2SE. Like CLDC, the majority of CDC classes are the same classes or subclasses of the classes of J2SE. Furthermore, CDC includes also some specific new classes that do not exist in J2SE nor J2ME/CLDC. These classes are designed particularly to fit the need of devices such as TV set-top boxes, Internet TVs, Internet-enabled screen phones, high-end communicators and automobile entertainenment/navigation systems. The Java programming language and Java virtual machine of the classical J2SE are entirely supported by J2ME/CDC. The Java virtual machine used in CDC is called CVM. CVM supports all the specifications and features of the classical virtual machine such as security management, Java native interface, weak references, reflection, thread groups, daemon threads, float bytecodes, user defined Java level class loaders, etc. The only difference distinguishing the two Java virtual machines is that the CVM has relatively small footprint ranged between 2 and 16 megabytes.

J2ME Virtual Machine

J2ME has two types of virtual machine that can be built inside. The first one is called CVM which has the same specification and features of the J2SE virtual machine. CVM is relevant for devices with memory size ranged between 2 and 16 megabytes. The second one is called KVM (kilo virtual machine). KVM is addressed to embedded devices with restricted resources and memory size less than 512 kilobytes. KVM is described in detail in the next section.

2.4.3 Java Kilo Virtual Machine

The kilo virtual machine (KVM) is the Java virtual machine for J2ME/CLDC platform [37, 28]. With respect to Sun documentation, the KVM supports only the CLDC configuration and the CLDC runs only on top of the KVM. It is expected that this situation will change and KVM may eventually support other configuration. The KVM is compact and portable specifically designed for resource-constrained embedded devices. It is relevant for the CLDC configuration and does not support all the classical Java virtual machine features. The design goal of the KVM was to create the smallest possible complete Java virtual machine that includes all the main aspects of the Java programming language and can run on limited resources devices with memory availability less than 512 kilobytes. It is called Kilo virtual machine because its memory budget is measured in kilobytes, while other Java virtual machines memory budgets are measured in megabytes. This optimization in footprint is due to the deletion of some of the classical Java virtual machine features. To be more specific, the following are the main aspects of KVM :

- KVM has small size. Its static memory footprint is ranged between 40 and 80 kilobytes depending on the compilation options and the target platforms.
- KVM's implementation is clean, well-commented, and highly portable.
- KVM is modular and customizable.
- KVM maintains the main aspects of the Java virtual machine.

Indeed, these features make the KVM the optimal solution for embedded devices having 16/32 bit RISC/CISC microprocessors with a total memory space that does not exceed a few hundred of kilobytes. Cellular phones, pagers, personal organizers and small payment terminals are instances of such embedded devices. The heap space needed for running a Java program can be less than 120 kilobytes and the size of the memory, occupied by the class libraries that are defined by the configuration, is typically less than 100 kilobytes. A more typical KVM implementation requires a total memory budget in the range of 256 to 512 kilobytes, half of it reserved as heap space, 40 to 80 kilobytes reserved for the virtual machine itself and the rest reserved for configuration and profile class libraries. The partition of volatile and non volatile memory space changes with respect to the implementation, the device, the configuration and the profile.

KVM Implementation

The KVM is implemented in the C programming language and designed to be built with any C compiler. Its source code consists of 24000 lines of C code, located in almost 60 files. The majority of the KVM source code are machine-independent, and the remaining small parts of code that are machine-dependent are isolated in small number of files. Furthermore, KVM defines special purpose compile-time flags and options in order to facilitate porting the KVM to many target platforms. Data alignment, long (64 bit) integers, big and little endian, memory allocation, garbage collection, interpreter options and optimizations, debugging and tracing options, etc. are instances of such flags and options. Although the KVM implementation and design are based on the classical virtual machine specifications, its performance is 30% to 80% less than the performance of the JDK1.1 release.

At runtime, many threads can be executed by the virtual machine and only one of them can access the processor at a given moment. Each thread has its own Java stack and a counter that defines its time-slice of execution. In fact, the KVM schedules the execution of threads by itself and switches between them depending on the need, priority and processor access time of each one of them. The KVM defines also five virtual machine registers which are IP (Instruction Pointer), SP (Stack Pointer), FP (Frame Pointer), LP (Locals Pointer) and CP (Constant Pool Pointer). These registers form the context of the current running thread. Switching from one thread to another requires to save the context of the old thread and load the context of the new thread. The saved context of a particular thread is restored whenever a switch is performed to it. In the KVM, only one heap contains all the structures including classes, objects, threads, stacks and internal data structures.

Furthermore, the KVM includes a garbage collection based on a simple Mark-Sweep-Compact algorithm. When the heap is full, the garbage collector searches all the referenced objets and mark them. Once finished, all the unmarked objects are freed from the heap. In case there is no more unreferenced objects to free and there is still no space available for new allocation, a compact mechanism is performed in order to move all the objects to one end of the heap and free the other end. Figure 2.5 describes this algorithm. Moreover, KVM supports the Java-Code-Compact utility known also as Romizer. The Romizer permit the link of all the Java classes in virtual machine in order to save some VM startup time. In fact, this mechanism combines Java class files and produces a C file that can be compiled and linked by the Java virtual machine. Referencing to Sun documentation, activation the Romizer produces a speedup of 5%.

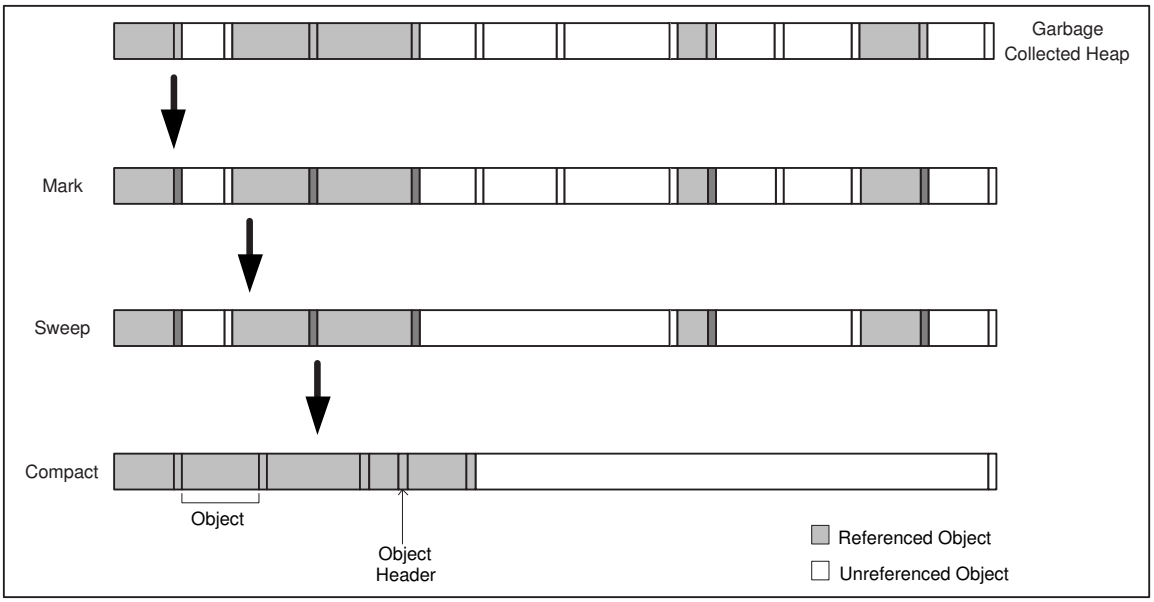

Figure 2.5: Garbage Collection Algorithm

2.5 Conclusion

While the main advantages of the Java virtual machine is its portability, its severe drawback remains definitely its poor performance. Although the problem of performance is applicable on all the Java platforms (i.e. J2EE, J2SE and J2ME), the lack of memory available in embedded systems adds more difficulties to find acceleration solutions for J2ME or to apply already existing ones for other Java platforms. In fact, these factors make the Java virtual machine an interesting domain for research in term of acceleration. Many people have been interested in enhancing the performance of the Java virtual machine and lot of techniques have been proposed. An overview of these techniques will be presented in the next chapter. In this chapter, we discussed first the main components of the Java virtual machine. Then, we focussed on the embedded Java platform and we described its architecture, specifications and limitations. Finally, we presented the Kilo Java virtual machine and we showed its poor execution performance.

Chapter 3

Acceleration of Java Virtual Machine

3.1 Introduction

The Java virtual Machine (JVM) is the cornerstone of the Java technology. While the main advantage of JVM is the portability it provides, its severe drawback remains definitely its poor performance. Enhancing the Java virtual machine performance is a very active research area. Different approaches were and are being considered. Moreover, embedding a Java virtual machine into resource-constrained devices or systems poses very challenging, but interesting problems in terms of footprint, computation and energy consumption. These three factors stand in the way of the acceleration techniques requiring huge data structures and complex computations, and hence increasing energy consumption. In fact, the limitation of resources of embedded devices makes many acceleration techniques, which are relevant for the classical Java virtual machine (JVM), not applicable in the context of embedded systems and then not relevant for the embedded Java virtual machine. In this chapter, we present first the optimization techniques for the Java virtual machine, and particularly we focus on the software techniques. Then, we discuss the dynamic compilation and we show its performance efficiency over other techniques. Finally, we detail dynamic compilation into embedded context and we present some virtual machines endowed with dynamic compilers.

3.2 Optimization Techniques for Java Virtual Machine

Many people were interested in the acceleration of the Java virtual machine and many techniques have been proposed. These techniques are divided into two main approaches: hardware and software acceleration.

Regarding hardware acceleration, a significant speedup in term of virtual machine performance is achieved. However, the high power consumption and the cost of these acceleration technologies encourage researchers to deviate to software acceleration of embedded Java virtual machine. This energy issue is really damaging especially in the case of low end mobile devices. As examples of these hardware acceleration techniques, many companies such as Zucotto Wireless [6], Nazomi [25], etc. have proposed Java processors that execute in silicon Java bytecodes.

General optimizations, static compilation and dynamic compilation are in general the four categories of software acceleration techniques [2, 5, 8, 14, 15, 35, 27]. General optimizations consist in designing and implementing more efficient virtual machine components such as better garbage collector, fast threading system, accelerated lookups, etc.). Static compilation consist in using extensive static analysis to optimize program before execution. Flow analysis, annotated type analysis, abstract interpretation, etc. are examples of static analysis. Some results demonstrate that general and static compilation optimization can lead to a reasonable acceleration. However, these techniques are not competent to other techniques such as dynamic compilation that can reach a big speedup (more than 200%) [39]. Dynamic compilation is the technique used in most modern desktop JVM implementation. A dynamic compiler can dramatically increase the execution speed of Java programs. However, it is inappropriate in the context of embedded systems owing to its large code size. The compilation process also implements sophisticated flow analysis and register allocation algorithm to generate optimized and high quality native code.

Other approaches based on dynamic compilation were also proposed in order to be relevant for embedded Java virtual machine. These techniques optimize programs at runtime, based on information available only at runtime, thus offering the potential for greater performance and less memory use. Selective dynamic compilation is one of these approaches that usually base their optimizations on run-time-computed values of particular variables and data structures called run-time constants. This technique deviates from other dynamic compilation techniques by selecting and compiling only those java code fragments that are frequently executed. An example of Java code fragment can be a method. This technique can significantly accelerate the Java virtual machine and at the same time reduce the memory overhead.

3.3 Hardware Optimizations

Many hardware acceleration techniques were proposed in order to accelerate Java execution. Hardware solutions try to perform all the complex JVM functions in the processors. In this context, dedicated Java processors, running in parallel with native processors, are used to emulate the execution of all or some of Java bytecodes on a specific platform. This technique, used to translate the platform-independent bytecodes into platform-dependent binaries, is very efficient in terms of performance. The resulting machine-dependent binary code is then transferred by the Java processors to the host processors for execution.

Although running JVM functions in hardware leads to a significant improvement in terms of virtual machine performance, the cost and high power consumption make it worthless, particularly in the context of embedded systems. Moreover, some bytecodes and complex virtual machine functions are not handled in hardware and need to be implemented in software (i.e. JVM). This entails a complex interaction between hardware and software. Many companies such as Nazomi, ARM, inSilicon, etc. are providing a variety of hardware accelerators, most of them are available in the market. In the sequel, we present some of these accelerators.

3.3.1 JSTAR, Nazomi

JSTAR Nazomi is a coprocessor accelerator fully compliant with all Java technology standards and Sun virtual machine implementations [17, 25]. It is now integrated with several leading commercial microprocessors and virtual machines. JSTAR can now run Java technology applications with greatly improved performance and much lower power consumption. It acts as a Java interpreter in silicon, which retrieves Java bytecodes from memory, translates them to platform-dependent machine code, and then executes them in conjunction with the native processor. Figure 3.1 illustrates the JSTAR enhanced system block diagram and shows its interfaces with the system through an SRAM interface. JSTAR handles 159 bytecodes directly in hardware and passes control back to the JVM whenever a bytecode is not handled. The switch mechanism is based on a call back table [25]. The main benefits of this accelerator are:

- Highest Performance.
- Quickest time to market
- Lowest Cost
- Lowest Power

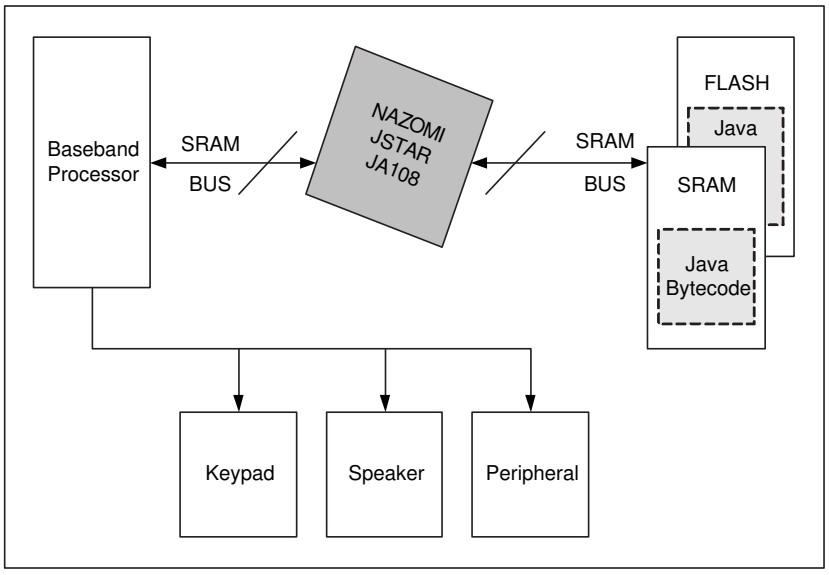

Figure 3.1: NAZOMI JSTAR

- Consumption Adaptable to any Java virtual machine: Sun Microsystems authorized or independently developed

3.3.2 ARM Jazelle

ARM Jazelle technology [32] for Java acceleration gives platform developers the freedom to run Java applications alongside established OS, middleware and application code on a single processor. The single processor solution offers higher performance, lower system cost and lower power than coprocessor solutions. Jazelle was developed to be the ARM solution for executing Java in hardware.

ARM processors have historically supported two instruction sets: the ARM instruction set, in which all instructions are 32-bits long, and the Thumb instruction set, which compresses the most-commonly used instructions into a 16-bit format. Jazelle technology extends this concept by adding a third instruction set - Java Byte Code - to the capability of the processor. Besides, there is an additional instruction set support for entering Java applications, real-time interrupt handling, and debug support for mixed Java/ARM applications. From the programmer's point of view, the processor has a new mode in which it behaves like a Java virtual machine. Once in Java state, the processor fetches and decodes Java bytecodes and maintains the Java operand stack. The processor can switch easily, under operating system control, between Java state and ARM/Thumb state. Jazelle is completely compatible with the ARM interrupt and exception model, hence, giving easy integration with operating systems and applications.

Jazelle interpreter is addressed to ARM architecture, while JSTAR can work with any CPU. This feature differentiates Jazelle from JSTAR, otherwise they resembles in translating bytecodes into native machine code. In Jazelle, 140 bytecodes are supported directly in hardware. Jazelle removes the interpretation loop from the JVM and replaces it with ARM property support code called VMZ, which is not larger than the code taken out. The main objectives of Jazelle are:

- Reducing size and enhancing performance.
- Allowing Java instructions to be restartable and permitting interruption during execution.
- Providing natural way to deal with interrupts.

3.3.3 JVXtreme

JVXtreme works in cooperation and in parallel with the main processor, generating better performance than in-line conversion of Java bytecodes to native processor instructions. JVXtreme, though, reduces instruction fetch time so most Java bytecodes are executed in a single cycle. The Java Virtual Machine is implemented on the native processor, but JVXtreme handles substantial portions. Figure 3.2 shows JVXtreme architecture. This architecture provides the speed and compact code advantages of a native Java processor while working with a native processor.

JVXtreme handles 92 bytecodes directly in hardware and, like Jazelle and JSTAR, it leaves the other bytecodes for software execution on the host CPU. During bytecode execution, JVXtreme passes control back to the host CPU when it encounters an unsupported bytecode. JVXtreme handles also stack overflow and underflow automatically. The JVM uses this feature during a task switch. JVXtreme currently support the ARM7 and ARM9 processors. It has a small footprint and operates at clock speeds up to 200 MHz.

3.3.4 Parthus MachStream

Parthus MachStream platform is a modular, hardware-based intellectual property engine, that dramatically boosts the performance and significantly decreases the battery power requirements for multiple aspects of the wireless applications environment. It implements a hardware-based Just-In-Time compiler (JIT). The Parthus accelerator transforms the software JIT into a silicon implementation. Parths claims that Machstream can save over 95 percent of the battery power consumed by the processor.

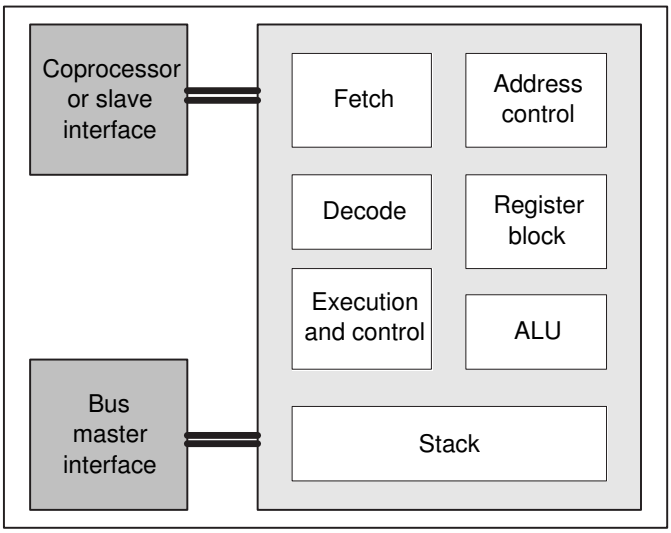

Figure 3.2: JVXtreme Architecture

MachStream supports 148 bytecodes directly in hardware, while the others are executed by the host CPU. Moreover, MachStream contains a silicon-based JIT code for translation and optimization called Mach I, and a hardware-based data preprocessing engine called Mach II. In fact, MachStream is inherently processor independent.

3.3.5 Aurora DeCaf

DeCaf processor is a hardware accelerator introduced by Aurora VLSI [44]. Unlike other hardware accelerators, DeCaf processes all the Java bytecodes and each bytecode is executed in a single cycle. Few native bytecodes are added, to provide access to internal registers, cache control, and non-cached memory accesses, and to support additional memory addressing modes. DeCaf is designed to operate either as a stand-alone processor or as a coprocessor. It has a peak rate of four instructions per cycle and it handles approximately 90 percent of the total Java bytecodes. DeCaf processor is connected to a host processor through a memory-mapped or a coprocessor interface. There are two control lines, one to trigger DeCaf and one to trigger the host because the two processors cannot run at the same time. This technique avoids a multiprocessor environment and associated software hassles, and minimizes also many debug troubles.

3.3.6 Zucotto Xpresso

Zucotto Xpresso [6] can also run as a coprocessor or as a stand-alone processor. It supports only 80 percent of the most commonly used bytecodes directly in hardware, while the other 20 percent, that are not supported by Xpresso, are translated and

executed by a JVM. In this case a switch to host processor is performed. Moreover, Xpresso implements some customized bytecodes in order to allow direct memory access and to improve the virtual machine performance. It includes also an efficient hardware support for garbage collection and puts a high priority on power consumption.

3.4 Software Optimizations Techniques

Many software acceleration techniques were proposed in order to accelerate the Java virtual machine. Some of them consist in designing and implementing more efficient virtual machine components, others consist in using extensive static analysis to optimize programs before execution. A third category of acceleration techniques consists in adding to or replacing the interpreter of the Java virtual machine by a dynamic compiler, that translates all or part of the Java bytecodes to platform-dependent executable machine code. This technique used to translate the platform-independent bytecodes into platform-dependent binaries is very efficient in terms of performance. In fact, results show that dynamic compilation is the best solution for Java virtual machine performance and can dramatically increase the execution speed of Java programs. However, some of these techniques are inappropriate in the context of embedded systems owing to their large code size. In the sequel, we present in detail the three categories of the Java virtual machine acceleration techniques [2, 5, 8, 14, 15, 35, 27].

3.4.1 General Optimization

Java portability is one of the main features of the Java language. This portability is made possible thanks to the interpretation mechanism. Java execution is based on a pure bytecode interpretation technique, which lies at the heart of the Java virtual machine and emulates the bytecode execution by executing the corresponding machine code on a particular platform. Although this technique provides high portability, it is the major cause of the Java virtual machine poor performance. This issue constitutes the motivation underlying many research initiatives aiming at introducing general optimization techniques, that accelerate the Java virtual machine and at the same time maintain the interpretation as the only Java execution model.

General optimization techniques consist in designing and implementing more efficient virtual machine components such as better interpreter, garbage collector, threading system, lookup mechanism, etc.. It is worth to mention that these techniques do not need important additional overhead in terms of memory and energy, hence, they may be feasible and efficient in the context of embedded systems. In the sequel, we describe some general optimizations techniques built in the Java virtual machine.

Romizing

Sun introduces a Romizing technique for the Kilo virtual machine in order to speed up the class loading mechanism. The Romizer links directly the Java classes in the virtual machine and reduces considerably the virtual machine startup. In other words, instead of launching the virtual machine alone and then loading the *.class* files one by one during runtime, the Romizer translates Java classes into C structures that contain all the information needed at runtime. These structures are saved in a C file and stored in the ROM. Consequently, the Java virtual machine executes the binary code of the pre-compiled classes from the ROM. As a result, the interpretation overhead is reduced, loading time for romized *.class* files is saved, RAM memory space is saved, however the executable file size is increased.

Fast-bytecodes implementation

The Java virtual machine implements quick versions of some bytecodes that need to call some virtual machine runtime services during interpretation. Reference resolution and method lookup are instances of such services. The aim of this optimization is to accelerate the interpretation process. The technique used saves the time needed to call the virtual machine services another time if the same bytecode with same arguments is encountered. When one of these bytecodes is called for the first time, the runtime services are called normally and the bytecode is replaced by a corresponding fast version. After that a cache entry containing the computed values is created and saved. Once the bytecode is called another time, its fast version is considered and the already saved values are used. This technique avoids to call the virtual machine services many times by extracting the already computed values from the cache.

Direct Threading Interpretation

This technique enhances the interpretation mechanism by eliminating the central dispatch [12]. Each bytecode of a method being interpreted is replaced by an address of the corresponding implementation which ends with the required dispatch to the next bytecode. Table 3.1 shows an instance code of this technique. For instance, the bytecode "ICONST_1" is replaced by its corresponding implementation address saved in the stack (ICONST_1 : *SP++) followed by a jump instruction to the implementation address of the next bytecode (next()), which is in this case "ICONST_2".

```
void *ByteCode[]={&&ICONST_1, &&ICONST_2, &&IADD, ...}
int JavaStack[STACKSIZE];
#define next() goto **(IP++) // Dispatch instruction
void **IP=ByteCode;

ICONST_1 : *SP++;
           next();
ICONST_2 : *SP++;
           next();
IADD     : - -SP;
           SP[-1] += *SP;
           next();
```

Table 3.1: Direct Threading Interpretation Code

Inline Threading Interpretation

The inline threading interpretation technique is based on the direct threading technique and improves it by eliminating the dispatch overhead within basic blocks [26, 13, 14]. It proceeds as follows: The bytecode sequences that form the basic blocks are first identified. Then, an implementation is dynamically created for such sequences through copying and concatenating each bytecode implementation in a new buffer. Finally, the dispatch code is copied at the end. Table 3.2 shows this interpretation technique for the sequence of ICONST_1 and INEG bytecodes.

Dynamic Method Lookup Acceleration

When an invocation occurs, a Java virtual machine mechanism called "lookup" determines the method to be executed. A search for the method is performed starting from one class and continuing recursively in the super-classes. Once the corresponding method is found, it will be executed directly, otherwise an error is signaled. This operation is very expensive in terms of execution and it occurs very often. Many techniques have been proposed to enhance this mechanism by saving previous lookup results in the cache for future uses. These techniques avoid calling the lookup many times to determine a particular method. Global cache, small inline cache and polymorphic inline cache are three known techniques that are based on this strategy of acceleration[9]:

- Global cache technique stores the previous lookup results in a global cache table consisting of three columns: The receiver class, the method name and the method address. Once the current class and the method name matches those that are saved in the cache, the code at the method address is executed and a call to the

```
ICONST_1_BEGIN : *SP++ = 1;
ICONST_1_END : **(IP++);

INEG_BEGIN : SP[-1] = -SP[-1];
INEG_END : **(IP++);

DISPATCH_BEGIN : goto **(IP++);
DISPATCH_END;
```

a. Bytecode Implementation.

```
int ICONST_1_LENGTH = (&&ICONST_1_END - &&ICONST_1_BEGIN);
int INEG_LENGTH = (&&INEG_END - &&INEG_BEGIN);
int DISPATCH_LENGTH = ( &&DISPATCH_END - &&DISPATCH_BEGIN);

void *Buffer = malloc (ICONST1_SIZE + INEG_SIZE + DISPATCH_SIZE);
void *Current = Buffer;

memcpy (Current, &&ICONST_1_BEGIN, ICONST_1_LENGTH);
Current += ICONST_1_LENGTH;
memcpy (Current, &&INEG_BEGIN, INEG_LENGTH);
Current += INEG_LENGTH;
memcpy (Current, &&DISTPATCH_BEGIN, DISPATCH_LENGTH);
```

b. Inlining Bytecode Implementation.

```
ICONST_1 body : *SP++ = 1;
INEG body : SP[-1] = -SP[-1];
DISPATCH body : goto **(pc++);
```

Table 3.2: Inline Threading Interpretation Code

lookup subroutine is avoided.

- The inline cache technique saves the previous lookup results in the code itself at each call site. The call instruction is changed to direct invocation of the method found by the default method lookup.
- Polymorphic inline cache extends the inline cache by using a stub routine that initially calls the method lookup. Each time the lookup is called, this stub function is extended with information that helps to determine the method to be executed.

Thread Synchronization Optimization

The thread synchronization mechanism is provided by Java through monitors that give exclusive access to shared data [3]. An object is an instance of such shared data. Objects are locked and unlocked by threads during execution using a lock structure. This structure, which is stored in the object header, contains some information about object lock states. So, added per-object states and two or three words of object header spaces have to be added in order to support the synchronization mechanism. Indeed, performing this mechanism affects particularly the embedded virtual machine because of the limited storage spaces . Some techniques are proposed to compact the size of the object header from three words to one word . Compacting the memory used by objects results in faster and more efficient threads synchronization mechanism.

Acceleration of the Verification

The verification process of the classical Java virtual machine is not relevant for embedded devices due to their limited resources. The verification process requires a lot of memory in order to perform its data flow analysis. To deal with this problem, Sun has divided the verification process into two steps: Pre-verification and verification. The pre-verification step, which is done off-line (before execution), performs all the data flow analysis and stores all the resulting information in the pre-verified *.class* file. The second step is the verification, during which a verification of the information stored in the pre-verified *.class* file is performed. This technique is very efficient and saves the time needed to perform the data flow analysis at runtime.

3.4.2 Static Compilation

All the Java virtual machine acceleration techniques that are performed before execution are part of the static compilation category. Ahead of Time (AOT) and Way Ahead of Time (WAT) are two known static compilation techniques[11].

An AOT compiler compiles all the application code on the target device at loading time before its execution. The resulting machine code is saved in the ROM in order to be used for future executions. This can improve considerably the Java virtual machine performance. However, the main drawback of this technique is the startup penalty it takes to translate the bytecodes of the loaded application into native machine code. This startup penalty can be significant, even though it is performed only the first time the application is compiled.

A WAT compiler can compile all the applications on the target device or on the host development platform before execution. The resulting machine code is stored on the target device for future execution. There is no penalty startup by using this technique because all the compilation operations are performed before even the loading process. However, the Java virtual machine using this technique can loose its portability if the applications are compiled on a host development platform and executed on a different one.

Moreover, static compilation techniques are inefficient to load dynamically applications from other locations. Executing these applications need an interpreter in order to translate the applications bytecodes into dependent-platform machine code. So, a Java virtual machine should support compilation and interpretation mode at the same time for complete execution. Although this strategy can be efficient in the context of classical Java virtual machine, it cannot be relevant for embedded Java virtual machine. Handling interpretation and compilation need additional memory space, which is not available on embedded devices with resources-constrained features.

3.4.3 Dynamic Compilation

Dynamic compilation techniques are very efficient in terms of performance, while they maintain most of the Java features. In dynamic compilation, Java bytecodes are translated into machine code at runtime. The resulting machine-dependent binaries are then executed on the processor of the target device. Most of the dynamic compilation techniques use the method as a unit of compilation. So, when a method is invoked, it is compiled instead of being interpreted, and the resulting machine code is then executed. Although compiling and executing a method may take more time than interpreting it, the fact, that this process is performed once during a Java program execution, makes dynamic compilation worthy. Eventual invocation of an already compiled method will be performed by a simple call to its corresponding machine code saved in the cache memory. As a result, several calls for the machine code of a compiled method will produce, for sure, a considerable speedup during a Java program execution.

A classical dynamic compiler compiles a method at its first invocation. This means

that all the invoked methods, even those that are not frequently called, will be compiled, and the machine code will be saved in the cache. In fact, this strategy of dynamic compilation has some disadvantages. The first one is the time spent to compile infrequently called methods, while interpreting them can take less time. So, compiling such methods introduces a time overhead. Indeed, this drawback exists only in the first released strategy of dynamic compilation. Later on, many approaches have been proposed in order to deal with this problem. Some of them use different compilers to compile frequently and infrequently called methods. Others use a mixed-mode approach, in which a method is interpreted until a certain condition is satisfied, then it will be compiled. An example of such condition can be as follows: the number of method invocations is greater than a certain threshold. Results show that efficient implementation of such approaches can solve very important part of this problem and can avoid the unnecessary compilation time.

Another drawback of dynamic compilation can be the implementation of the compiler itself. The compilation is performed at runtime. So, any delay caused by the compiler can affect directly the performance of the Java virtual machine, and hence, the performance of the whole Java program execution. For this reason, the compilation time should be minimized as much as possible. However, this does not mean that we should only focuss on the efficiency of the compiler. In fact, a compiler, that compiles methods very quickly and produces a poor quality code, could affect the execution performance. A dynamic compiler needs to balance the compilation time and the quality of the generated code in order to improve considerably the execution.

Potential disadvantages can exist if the dynamic compilation is applied in embedded context. Excepting the drawbacks mentioned above, dynamic compilation seems to be a very efficient technique to enhance Java performance in desktop and servers. However, embedded systems do not have the same hardware capabilities as desktop and server systems. The latter have fast busses, hundred of megabytes of RAM and microprocessors operating at over 2 GHz with on-chip 512 kilobytes caches, while embedded systems can have an available memory less than 512 kilobytes to handle the entire Java runtime environment, 300 kilobytes of RAM, 1 megabytes of flash and ROM, microprocessors operating at over 32 MHz and low battery capacity. These limited hardware capabilities of embedded systems bind some restrictions on dynamic compilation techniques and prevent dynamic compilers from accomplishing the same performance results as on desktop and servers.

Moreover, when thinking about endowing a dynamic compiler into an embedded Java virtual machine, many implementation issues emerge because of the lack of memory space. Including code optimization mechanisms that need additional memory space is practically impossible, from which the difficulty to obtain good quality and smaller generated code. Furthermore, additional cache management is needed in order to orga-

nize the storage of generated machine code. The cache memory available in embedded systems is not enough to store all the code generated by a dynamic compiler. Java bytecodes are much smaller than native processor instructions, the produced native code can be 8 times the size of bytecodes. Thus, a cache management mechanism should be efficiently implemented in order to free unused code and find space for newly generated machine code. As a result, the implementation of a dynamic compiler should be very frugal with memory and beneficial considering the costs in memory and power.

Many dynamic compilation approaches have been proposed in order to deal with all the disadvantages mentioned above and cover all the target platform's features. In the sequel, we distinguish the three main approaches: Classical or Just-In-Time (JIT) dynamic compilation, adaptive dynamic compilation and selective dynamic compilation. The adaptive and selective are based on the Just-In-Time dynamic compilation and improve it in some issues. The method is the unit of compilation in the three approaches. This strategy allows the compiler to use the Java virtual machine services in order to, gather the required references, arguments and local variables of a particular method, and determine all the context structures related to it. Using such virtual machines services avoids complex implementations that require additional memory space and time to be executed.

Classical or JIT Dynamic Compilation

Just-In-Time (JIT) is the first and original approach proposed for dynamic compilation. JIT consists in compiling a method when it is invoked for the first time and saving the generated machine code for future calls. This technique is based on one mode of bytecode translation, which is the compilation mode, while the interpreter is not needed any more. This means that all the invoked methods, even those that are not frequently called, are compiled. This strategy of dynamic compilation has some disadvantages. The first one is the time spent to compile infrequently called methods, while interpreting them can take less time. The second drawback is that the quality of generated code is the same for all the methods, whatever the frequencies of their invocations are. Indeed, this issue can affect those methods that are invoked very often (sometimes million of times). Low quality generated code for such kind of methods can lead to poor performance of the Java virtual machine. Furthermore, this approach is not relevant for embedded Java virtual machine because it requires large memory space, which is not available in embedded systems.

Adaptive Dynamic Compilation

Adaptive Dynamic Compilation (DAC) is an alternative acceleration for JIT compilation. This approach is proposed in order to improve the classical dynamic compilation and produce different qualities of generated code, depending on the invocation frequency of each method. It consists in using different compilers that produce different levels of code quality. A fast compiler, that produces low quality code, is used for infrequently called methods, while a slow compiler, that produces a high quality optimized code, is used to translate frequently called methods. All the methods are first compiled with the fast compiler and then, according to the frequency of a given method, this method can be chosen to be recompiled by the slow compiler or not. Since frequently called method can be called thousands and millions of times, spending more time to generate high quality code can enhance considerably the execution performance. Once a method machine code is generated by the slow compiler, the recent machine code generated by the fast compiler is totally discarded and freed from the cache memory. This approach is not relevant for embedded Java virtual machine because the lack of memory space makes the implementation of two compilers at the same system almost impossible.

Selective Dynamic Compilation

The Selective Dynamic Compilation (SDC) is proposed in order to improve the Just-In-Time dynamic compilation and use a mixed mode approach for machine code generation. The mixed mode approach consists in using both the interpreter and the compiler. The interpreter is used for infrequently called methods, while the compiler is used for frequently called methods. A given method will be interpreted until a certain condition is satisfied, then it will be compiled. This condition can be specified in the compiler implementation. For instance, a method will be interpreted until reaching the limit allowed for a method to be interpreted. Reaching this interpretation limit leads to declare the given method as frequently called method or "hotspot", which means that the method should be compiled. The generated code is saved in the cache for future calls. The strategy followed in this approach makes it relevant for embedded systems. Not all the invoked methods are compiled, which means that the memory space needed for compiled code can be managed in order to fit with the memory space available in embedded systems. A very lightweight implementation of such selective dynamic compiler is required to reduce the generated code size as much as possible. A detailed explanation for this approach is presented in the next section.

3.5 Selective Dynamic Compilation

The selective dynamic compilation is the approach we based on in the implementation of our Armed E-Bunny selective dynamic compiler targeting ARM processors. It is the most efficient approach that can be endowed into the embedded Java virtual machine and can fit with the limited-resource specifications of the embedded systems. Selective dynamic compilation optimizes programs at runtime, based on information available only at runtime, thus offering the potential for greater performance and less memory use. Selective dynamic compilers usually base their optimizations on run-time-computed values of particular variables and data structures called run-time constants. It deviates from JIT compilation by selecting and compiling only those Java code fragments that are frequently executed. An example of Java code fragment can be a method. This technique can significantly accelerate the virtual machine and at the same time reduces the memory overhead because only a part of a program is compiled.

A dynamic compilation performs its operations at runtime, so the compilation time in addition to the generated code execution time are accounted in the overall execution time of a Java program into the Java virtual machine. Moreover, the compiler and the generated code are stored in the memory of the target system, which is very limited for embedded systems. The following are the main constraints that should exist in a dynamic compiler in order to integrate it efficiently into an embedded Java virtual machine:

- The compilation time is encountered in the overall execution time. Hence, the compilation should be as fast as possible.
- The compiler code should be maintained in the target machine memory during execution. Hence, the size of the compiler code should be as small as possible.
- The performance of a high quality code is better then the performance of low quality code. Hence, the quality of the generated code should be as high as possible.
- The generated code size should be as small as possible, particularly for embedded systems, where the memory space is too limited.
- The acceleration accomplished by endowing such dynamic compiler into the Java virtual machine should be significant.

The architecture of a selective dynamic compiler is based on five main components, including the interpreter of the virtual machine. These components are: the method, the profiler, the VM interpreter, the compiler and the cache manager. Each one of

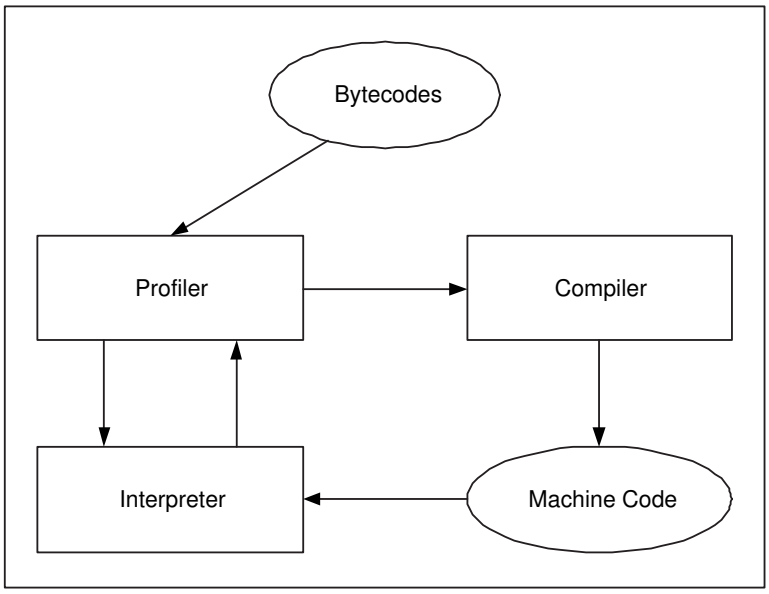

Figure 3.3: Dynamic Compiler into JVM

these components has a specific role in translating the Java bytecodes into machine code. Figure 3.3 shows the architecture of a selective dynamic compiler integrated into a Java virtual machine. The following subsections explain the components of a dynamic compiler and shows how dynamic compilation proceeds.

3.5.1 Profiler

The profiler has many related roles and communicates with all the other components of a dynamic compiler. A profiler has the task to detect the frequently executed piece of code. The piece of code used in most of the dynamic compilation techniques is the method. A profiler selects the set of methods where the program spends most of its time. A detected frequently called method is declared as hotspot method. The profiler performs its task through three related and consecutive steps. First, it monitors and traces events that occur during runtime. Second, it sets the cost of these events. Third, it attributes the cost of these events to specific parts of the program. By doing that, a profiler is able to forecast future through monitoring past events. In fact, the future events can be to switch from the interpretation mode to the compilation mode of the Java bytecodes. Time-based profiling, counter-based profiling and sample-based profiling are three distinguishable approaches that can be implemented into a dynamic compilation profiler [18].

Time-based Profiler

A time-based profiler is based on calculating the time spent in each method, so it is able to predict if a method is frequently invoked or not. Some variables or instructions are introduced in the implementation of some bytecodes such as calls, returns, throw and catch in order to record time. Each time a method is called, the current time is checked and added to the current called method structure. A method is declared as frequent method if the calculated time reaches a time threshold. The time threshold is normally specified in the compiler implementation. This approach is complete in the sense that all called methods are checked by the profiler. On the other hand, its main disadvantages are the overhead time and code size resulting from executing and storing the added variables and instructions. Indeed, these disadvantages reduce the use of this approach in many dynamic compiler profilers, particularly those addressed to embedded systems.

Counter-based Profiler

A counter based profiler is based on calculating the number of times a method is invoked. A counter is added to the structure of each method and is initially set to zero. This counter is incremented by one each time a given method is executed. It is incremented also at each back-edge branch if the method includes loop iterations. The role of the profiler is to, check the counter of the invoked method before translating its bytecodes, and compare it to the threshold specified in the implementation. If the method's counter reaches the threshold, the profiler declares the method as frequently called or hotspot method. Considering a method as hotspot means that the method should be compiled. Like time-based profiling, this approach is also complete. Even though the overhead caused by introducing and updating the method counters is a disadvantage, the small size of added code makes counter-based profiling efficient and particularly relevant for embedded systems.

Sample-based Profiler

A sample-based profiler collects its data in a cyclical manner. It samples the running application periodically when the application reaches a predefined points. Method entries and loop edges are instances of such points. One strategy of sampling is to accord to the method containing the loop the sample taken on a loop back-edge, and to accord to both calling and called methods the sample taken at a method entry. Another strategy of sampling is to trace the Java stack periodically in order to detect the methods currently executing and accord samples to them. Whenever the number of samples of a method reaches the threshold specified in the implementation, the method

is declared as frequently called or hotspot method, and then it is compiled. The main advantage of this approach is the reduced overhead. However, it is not complete and may not cover all the executed methods. Sampling is performed periodically and not continuously, which means that some methods can be executed without being sampled. In fact, this incompleteness makes this approach less used in most dynamic compilers, even though it is useful for limited resources systems.

3.5.2 Compiler

The compiler is the principal component responsible of translating Java bytecodes into platform-dependent machine code. As mentioned above, many constraints should be respected and balanced among them according to the characteristics and capabilities of the target platforms. The most important issue at the end is to integrate into the machine virtual a compilation mechanism more efficient than interpretation. A compiler can perform one or more passes over bytecodes and generates the corresponding machine code. The design of a compiler is based on the following issues: The compilation unit, the compilation time, the quality of the machine code and the nature of the machine code (stack-based or register based).

The compilation unit is the fragment of code chosen to be compiled. Specifying the compilation unit is very important and influences directly the design of the dynamic compiler [1]. The method is the compilation unit used in most dynamic compilers due to the following reasons:

- A method is represented by an internal Java virtual machine structure and many JVM services are dedicated to perform many operations on it. So, it is easy to identify a method, get its bytecodes, get its arguments and local variables and add some fields to its structure useful for compilation and machine code execution.
- The switching mechanism between interpretation, compilation and machine code execution are based on some interpreter control points applied on methods. Due to the complexity of such mechanisms, re-implementing them is worthless.
- The representation of a method by JVM structure simplifies the cache management mechanism. Whenever the cache is full and some machine code are freed, the new state will be easily updated in the corresponding method structure.

On the other hand, using the method as compilation unit has the following two main disadvantages:

- The cache will be quickly full due to the code size of a method. With respect to the size of code generated each time the compiler is called, only few methods can be compiled and their corresponding machine code can be stored in the cache.

- Experience shows that only 20% of a method declared as hotspot is really frequently called, while the remaining of the code is infrequently invoked. This means that the cache is 80% occupied by a code that is not really hotspot.

The compilation is performed at runtime, so the compilation time is part of the overall execution time and should be as fast as possible. A quick compiler should respect some constraints in order to attain efficient results . First, the number of passes over bytecodes should be reduced to a maximum of one or two passes, while the production of the machine code should be so quick. Second, some dynamic compilers construct first intermediate representation of bytecodes. This step can be avoided. Moreover, very few optimizations are permitted during compilation due to the lack of memory available on embedded systems.

Regarding the quality of the code, high quality code results in high execution performance. The compiler should generate the highest quality possible of machine code, even if this is very difficult in case of quick compilation strategy. At the same time, a compiler should take into account the size of the compiler and the size of the generated code, specifically when integrated into embedded virtual machines.

The nature of the machine code generated is also a very important issue when designing and implementing a dynamic compiler. It has a big and direct influence on the size and execution speedup of the generated machine code, on the complexity of switching between interpretation and compilation mode, and on the compilation overhead. The stack-based and register-based are the two possible natures of generated machine code. In stack-based code, all data operations are performed through a stack, while in register-based code, all data operations are performed using registers. Results show that generated stack-based machine code for a given function performs 30 percent more operations than generated register-based machine code for the same function. Hence, register-based code seems to be more efficient than stack-based code. However, since Java bytecodes are stack-based, it is easier and faster to generate stack-based machine code corresponding to a given Java program [31].

The following are three points that should be taken into account when talking about stack-based versus register-based machine code:

- Register-based code is more compact than stack-based code. Hence, the quantity of register-based code that can be generated during compilation will be obviously smaller than the quantity of stack-based code.

- Register-based code can be run faster than stack-based code because access to register is faster than access to memory (stack). Hence, register-based code is more relevant and efficient for Java virtual machine performance.

- Stack-based code is more relevant than register-based code for the switch between interpretation and compilation mode, which is frequently performed during execution. The interpretation is stack-based, so switching to stack-based compilation will be easier and faster. A switch to register-based compilation mode requires an entire transfer, for all the method's contexts and variables needed for execution, from the stack to registers, and vice versa for an inverse switch.

3.5.3 Cache Manager

The role of a dynamic compiler is to compile the bytecodes of a particular method and to generate its corresponding machine code. Once this machine code is generated, it needs to be saved in the cache for future execution. Since the size of generated code for a given method is 4 to 8 times the size of its bytecodes, the cache memory, and in particular the cache memory of embedded systems, is not sufficient to store all the generated machine code. Hence, a cache management should be performed in order to free space for new storages. The main issue in cache management is how to choose the elements to be freed. Choosing these elements depends on many parameters such as method size, method compilation time, method invocation frequency, time of the last method invocation, etc.. In this context, many algorithms are proposed for implementing a cache manager. In the sequel, we present some known cache management algorithms used in many dynamic compilers:

LRU (Least Recently Used)

This algorithm is based on a chronological order to access to an element [22]. It chooses the element that has not been accessed for the longest period of time in order to be freed from the cache. To perform this task, the time of the last access is associated to each element and is updated at each eventual access. The drawback of this algorithm is that it causes cache fragmentation. Since the new stored code may not fit totally in the freed space, hence some small spaces in the cache will remain free and unusable for new storage.

LFA (Least Frequently Accessed)

This algorithm is very similar to LRU. It is also based on a chronological order of the use of an element. A counter is associated with each element and is incremented by one each time this element is used. This algorithm chooses the element that has the smallest counter value to be freed from the cache. The main drawback of this algorithm is the overhead of adding and updating the element counter, in addition to cache fragmentation.

LRC (Least Recently Created)

This algorithm is also called FIFO (First In First Out). It is based on a chronological order of an element loading. The entry to be freed is the oldest among those currently in the cache. To perform this task, the cache manager checks the time of loading the element in the cache, which is associated with each element. The main advantage of this algorithm over LRU and LFA is that it avoids cache fragmentation. The free cache spaces that remain after inserting elements can remain useful for eventual storage.

Second Chance

This algorithm enhances LRC. The aim of this enhancement is to keep the recently used elements in the cache. A bit is associated with each element saved in the cache. This bit is initially set to zero. When an old element is used, this element is set to one. Once the cache management is performed, the bit of the oldest element is checked. If its value is zero, the element is freed, otherwise, the bit is reset to zero and the element is put at the bottom of the list.

LE (Largest Element)

This algorithm chooses the largest element stored in the cache in order to be freed. This strategy can minimize the number of freed elements during cache management, but creates more cache fragmentation. The overhead caused by adding information about the size of each element in the cache is also another drawback of this algorithm.

BFE (Best Fit Element)

This algorithm is based on the size of the new entry to be inserted. It scans the cache in order to search for the best element size that fits with or greater than the size of the new element to be stored. If all the element sizes are smaller than the new element size,

cached elements are grouped in pairs or two and new searches may be applied. The overhead caused by scanning the cache, sometimes many times, is an essential drawback of this algorithm.

Full Cache Flush

This algorithm is very simple. Whenever the cache is full and there is no space that fits with the size of the new entry, all the elements are freed from the cache. Although this strategy is very lightweight, the main drawback is that the hot entries can be removed. This issue may lead at the end to additional compilation overhead.

3.6 JVMs Endowed with Dynamic Compilers

Dynamic compilation is a popular approach for acceleration. Almost all the standard Java virtual machines are endowed with dynamic compilers. In the sequel, we present four of these virtual machines and we describe briefly the dynamic compilations techniques used inside.

3.6.1 VM Hotspot

The Java HotSpot VM [35], which is the core component of Java 2 Standard Edition, is equipped with a selective dynamic compiler. Performance critical methods are detected by means of a counter-based profiler with additional heuristics. These heuristics investigate the caller methods in order to compile them with the triggering method. On-Stack-Replacement technique is also used to trigger compilation while a method is still running. The Java Hotspot VM dynamic compiler applies several classical optimizations and is considered as one of the most efficient in the market. Another important reason of this efficiency is the aggressive method inlining it applies. Indeed, methods that are detected frequent are not only compiled but also inlined. The benefit of this optimization is that it produces larger blocks of code for the compiler to perform optimizations. Operating on large blocks of code increases the effectiveness of classical optimizations. However, using On-Stack-Replacement technique or applying expensive optimizations such as aggressive method inlining is not adequate for embedded systems because they require important resources, particularly memory space.

3.6.2 Intel ORP

Intel ORP is a complete Java virtual machine implementation triggered by Intel. ORP allows the different Java runtime components to be developed in complete isolation. The dynamic compiler built inside is based on the Just-In-Time (Classical) approach, which means that there is no interpreter used inside. At the same time, the compilation is performed by two different compilers, as in the adaptive approach. The first is called fast compiler, and the second is called optimizing compiler. All methods are first compiled by the the fast compiler, and then only the frequently called method are recompiled by the optimizing compiler. The new generated code for a given method replaces the old one already generated by the fast compiler. The difference of performance between the two compilers is approximately 30%.

3.6.3 IBM Jalapeño: Jikes RVM

IBM Jalapeño [2] is a Java virtual machine equipped with a JIT compiler with different levels of optimizations. That is, according to the frequency of a method, this is compiled with the appropriate level of optimization. The profiler used by Jalapeño is very complex and is part of a bigger system called Adaptive Optimization System (AOS). As Java Hotspot VM, the features of IBM Jalapeño cannot be applied in an embedded context due to the lack of resources. For instance, a JIT approach is not suitable because it requires that all methods are compiled, whereas embedded systems lack the necessary memory to store all the methods' generated code.

3.7 Embedded JVMs Endowed with Dynamic Compilers

Due to its efficiency in terms of performance, dynamic compilation became a popular approach for acceleration. Almost all standard Java virtual machine are endowed with dynamic compilers. However, the features of these virtual machines cannot be applied in an embedded context due to the lack of available resources. An example of these restricted resources is the memory size, which is a precious resource on embedded systems.

The deployment of such dynamic compilation techniques into embedded Java virtual machines faces two major difficulties. First, the size of the dynamic compiler should be small because it should be maintained in the memory during the program execution. Second, heavyweight optimizations are not affordable because of their overhead. Hence,

embedded dynamic compilers should be extremely frugal with memory resources. Despite these difficulties, dynamic compilation is also used in CLDC-based embedded virtual machines [39, 29, 31]. However, except one paper about KJIT [31], no detailed information about these systems is available in the literature due to commercial reasons. In the sequel, we present the most known embedded Java virtual machines that are endowed with dynamic compilers.

3.7.1 CLDC Hotspot

CLDC Hotspot VM [39] is an embedded virtual machine introduced by Sun Microsystems. As its name indicates, it is strongly inspired by the standard Java Hotsopt VM [35]. All the features of Java Hotspot VM (desktop version) that can be adapted to embedded environment are applied. A selective dynamic compiler is built inside. Performance critical methods are detected by a single statistical profiler. The compilation is done in one pass. Constant folding, constant propagation and loop peeling are the three basic optimizations applied. The space required by CLDC Hotspot is almost the double of the space required by the Kilo virtual machine (KVM). It reaches 1 megabyte. No more details are provided about the CLDC Hotspot dynamic compiler.

3.7.2 KJIT

KJIT is a lightweight dynamic compiler that uses as its foundation the KVM [31]. KJIT does not use any form of profiling for the simple reason that all methods are compiled. This strategy seems to be very heavyweight and only feasible in server or desktop systems. The key idea to make this strategy adequate for embedded Java virtual machines is to compile only a subset of method bytecodes. This technique is a good solution to make KJIT adaptive for embedded systems. The remaining bytecodes continue to be handled by the interpreter. To handle that, efficient execution switches between the compiler and the interpreter are applied. Whenever one of the interpreted bytecodes is encountered, a switch back from the compiled mode to the interpreted mode is performed. This requires an efficient switching mechanism since this operation is highly frequent. KJIT performs such switch mechanism by pre-processing the bytecodes before their compilation. However, there is a significant overhead in terms of time and space required for performing the pre-processing. For instance, the pre-processed code is 30% larger than the original one.

3.7.3 Jbed Micro Edition CLDC

Jbed is a Java virtual machine for small mobile devices [29]. Jbed is deployed by Esmertec AG and intended to IntelXscale and Strong ARM architectures. The dynamic compiler of Jbed is called FastBCC. It compiles all bytecodes and dynamically loaded classes at load time instead of waiting their first executions. It links also the loaded classes into the application. Hence, the execution delay is eliminated. FastBCC is a small, one-pass compiler embedded in the virtual machine. Results show that Jbed virtual machine is four time faster then the KVM.

3.7.4 EVM

EVM is an embedded Java virtual machine based on dynamic compilation. It is deployed by Insignia for its Jeode platform[23]. All the acceleration techniques inside it are designed to meet the Java specifications for resource-constrained devices. Moreover, an adaptive dynamic compiler is embedded in it. It uses also a precise concurrent garbage collection and a predictable system behavior. Only the frequently executed code is compiled and the resulting generated code is stored into a memory buffer. Additional optimizations are also performed on the stored generated code. Results show that EVM is six times faster than an interpretive virtual machine.

3.7.5 IBM J9

IBM J9 is a fast Java virtual machine deployed by IBM and based on JDK1.2.2 technology [33]. J9 is addressed for embedded devices. It contains an adaptive dynamic compiler supported on many PowerPC platforms. Many features can be selected at runtime in order to meet the target platform characteristics, so memory utilization can be minimized differently depending on each device. Moreover, J9 supports a precise garbage collection that ensures no memory leaks and manages automatically memory allocation and de-allocation.

3.7.6 Wonka

Wonka is a Java virtual machine deployed by Acunia and addressed for resource-constrained embedded systems. It is an extremely portable virtual machine for a variety of markets and does not require a host operating system. It contains a concurrent garbage collection that manipulates very effectively the memory and keeps the minimum fragmentation. A J-Spot native compiler is intended to be employed in the

next version of Wonka.

3.8 Conclusion

Some disadvantages can exist if the dynamic compilation is applied into embedded context. Dynamic compilation seems to be a very efficient technique to accelerate Java performance in desktop and servers. However, embedded systems do not have the same hardware capabilities as desktop and server systems. These limited hardware capabilities of embedded systems bind some restrictions on dynamic compilation techniques and prevent dynamic compilers from accomplishing the same performance results as on desktop and servers. Moreover, when thinking about endowing a dynamic compiler into an embedded Java virtual machine, many implementation difficulties are imposed due to the lack of memory space and battery life available. In this chapter, we presented first the optimization techniques for the Java virtual machine, and particularly we focussed on the software techniques. Then, we discussed the dynamic compilation and we showed its performance efficiency over other techniques. Finally, we detailed dynamic compilation into embedded context and we presented some virtual machines endowed with dynamic compilers.

Chapter 4

Armed E-Bunny

4.1 Introduction

In this chapter, we describe the elaboration of a dynamic compiler, called Armed E-Bunny, that targets the ARM platform. The proposed system uses, as starting virtual machine, the last version of Sun's Kilo virtual machine (KVM). The architecture of Armed E-Bunny is inspired by [10]. This technology is based on a selective dynamic compiler built inside the J2ME/CLDC (Java 2 Micro Edition for Connected Limited Device Configuration). Our results show that our work comes up with an efficient, lightweight and low-footprint accelerated Java virtual machine ready to be executed on ARM embedded machines. This chapter presents first the architecture of ARM platform, then it details the architecture, the design as well as the implementation and debugging issues of Armed E-Bunny. Our experimental results prove that a speedup of 360% over the last version of Sun's KVM is accomplished by Armed E-Bunny with a footprint overhead that does not exceed 119 kilobytes.

4.2 ARM Platform Architecture

As long as the use of wireless systems such as PDAs, cell phones, pagers, etc. is becoming a need in our everyday life, the ARM architecture is becoming the industry leading 16/32 bit embedded system processor solution due to its performance and RISC (Reduced Instruction Set Computer) features. ARM powered microprocessor are being routinely designed into a wider range of products than any other 32-bit processor. This wide applicability is made possible by the ARM architecture, resulting in optimal system solutions at the crossroads of high performance, small memory size and low power consumption. In this section, we present a brief description of ARM architecture, and particularly the issue needed for the implementation of an ARM assembler. ARM

Processor	Mode	Description
User	user	Normal Program Execution mode
FIQ	fiq	Fast Interrupt for high speed data transfer
IRQ	irq	Used for general-purpose interrupt handling
Superviser	svc	A protected mode for the operating system
Abort	abt	Implements virtual memory and/or memory protection
Udefined	und	Supports software emulation of hardware processors
System	sys	Runs privileged operating system tasks

Table 4.1: ARM Processor Mode

processor modes, ARM registers, ARM instruction set, ARM instruction decoding and subroutine calls are the main points discussed in the following subsections [19, 20].

4.2.1 ARM Processor Modes

ARM is an architecture that supports seven processor modes. Table 4.1 illustrates these modes. Switches between modes can be performed under software control, by external interrupts or by exception processing. The user mode is the mode mostly used in most application program execution. User mode does not allow the access to some protected system resources or to switch to other modes. All modes, except the user mode, are known as privilege modes. They have full access to system resources and can switch among them freely. In our case, the user mode is used all the time.

4.2.2 ARM Registers

ARM has a total of 37 registers divided as follows: 30 general purpose registers, 6 status registers and one register for the program counter. Only 15 of the 30 general purpose registers are available at a time and vary from one mode to another. Table 4.2 shows all these registers. We use only the user mode registers in the implementation of our compiler. The following is a brief description for the role of each user mode register (*R0-R15*):

R0-R3 They hold the first four words of incoming arguments and intermediate values within a routine. *R0* and *R1* can hold the return values of a subroutine.

R4-R10 These are truly general purpose registers with no special task assigned to them by the processor architecture.

R11 It is reserved to hold the value of the frame pointer in the stack. It is also known as *FP*.

User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Table 4.2: ARM Registers

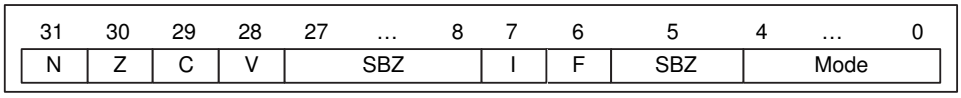

Figure 4.1: Structure of the Processor Status Registers

R12 It is used to hold temporary and intermediate values needed for subroutine calls. It can hold also other values as general purpose registers.

R13 It is reserved to hold the value of the stack pointer. It is also know as *SP*.

R14 It is reserved to hold the value of the return address for a subroutine. When a subroutine call is performed, *R14* is set to the value of the next instruction. It is also known as *LP*.

R15 It is reserved to hold the program counter. It is used to identify which instruction to be performed next. It is often referred as an instruction pointer. It is also known as *PC*.

There is also another register known as CPSR (Current Processor Status Register). It holds the current status of the processor. This includes various condition code flags, interrupt status, processor mode and other status and control information. Figure 4.1 shows the structure of this register. As shown in the figure, the status register contains 4 flag bits:

N Negative flag.

Z Zero flag.

C Carry flag.

V Overflow flag.

Many instructions, including comparison, arithmetic, logic and move, can modify these flags. Since all the instructions are executed under a particular condition, all of them check the values of these flags in order to verify if the condition is satisfied or not. This point will be explained in the instruction set section.

4.2.3 ARM Instruction Set

The ARM architecture [19, 20] is based on 32 bit Reduced Instruction Set Computer (RISC) principles, and the instruction set and related decoding mechanism are much simple than those of microprogrammed Complex Instruction Set Computer. Table 4.3 shows the ARM instruction set summary with the role of each one of them and figure 4.2 shows the format of these instructions. This format is used in the implementation of the assembler of our dynamic compiler. The instruction machine code size is fixed to 32 bits. Although this simplicity results in a high instruction throughput and impressive speedup, it requires sometimes the generation of many instructions to perform a simple operation or to replace some un-existing instructions. For instance, to load an immediate value greater than 255 into a register, a list of instructions including bits manipulation operations are performed. Another example is the code generation of the division arithmetic operation.

Condition Field

As mentioned in the register section, the CPSR flags are needed to verify the condition satisfaction. All ARM instructions are conditionally executed with respect to the CPSR condition code and the instruction's condition field. The instruction's condition field specifies the condition under which the instruction can be executed. During execution, a verification is performed in order to verify if the *C, N, Z* and *V* flag values fulfil the condition. If the condition is satisfied, the instruction is executed, otherwise it is ignored. Table 4.4 shows the fifteen possible conditions that can be represented in the 4 bits condition field of an instruction.

Mnemonic	Instruction	Action
ADC	Add with carry	R_d := R_n + Op2 + Carry
ADD	Add	R_d := R_n + Op2
AND	AND	R_d := R_n AND Op2
B	Branch	R15 := address
BIC	Bit Clear	R_d := R_n AND NOT Op2
BL	Branch with Link	R14 := R15, R15 := address
BX	Branch and Exchange	R15 := R_n, T bit := R_n[0]
CDP	Coprocessor Data Processing	Coprocessor-specific
CMN	Compare Negative	CPSR flags := R_n + Op2
CMP	Compare	CPSR flags := R_n - Op2
EOR	Exclusive OR	R_d := (R_n AND NOT Op2) OR (Op2 AND NOT R_n)
LDC	Load Coprocessor from Memory	Coprocessor Load
LDM	Load Multiple Registers	Stack Manipulation (POP)
LDR	Load Register from Memory	R_d := (address)
MCR	Move CPU Register to Coprocessor Register	cR_n := rR_n {<op>cR_m}
MLA	Multiply Accumulate	R_d := (R_m * R_s) + R_n
MOV	Move Register or Constant	R_d := Op2
MRC	Move from Coprocessor Register to CPU Register	R_n := cR_n {<op>cR_m}
MRS	Move PSR status/flags to Register	R_n := PSR
MSR	Move Register to PSR status/flags	PSR := R_m
MUL	Multiply	R_d := R_m * R_s
MVN	Move Negative Register	R_d := 0xFFFFFFFF EOR Op2
ORR	OR	R_d := R_n OR Op2
RSB	Reverse Subtract	R_d := Op2 - R_n
RSC	Reverse Subtract with Carry	R_d := Op2 - R_n - 1 + Carry
SBC	Subtract with Carry	R_d := R_n - Op2 - 1 + Carry
STC	Store Coprocessor Register to Memory	address := CR_n
STM	Store Multiple	Stack Manipulation (PUSH)
STR	Store Register to Memory	(address) := R_d
SUB	Subtract	R_d := R_n - Op2
SWI	Software Interrupt	OS call
SWP	Swap Register with Memory	R_d := [R_n], [R_n] := R_m
TEQ	Test Bitwise Equality	CPSR flags := R_n EOR Op2
TST	Test Bits	CPSR flags := R_n AND Op2

Table 4.3: ARM Instruction Set

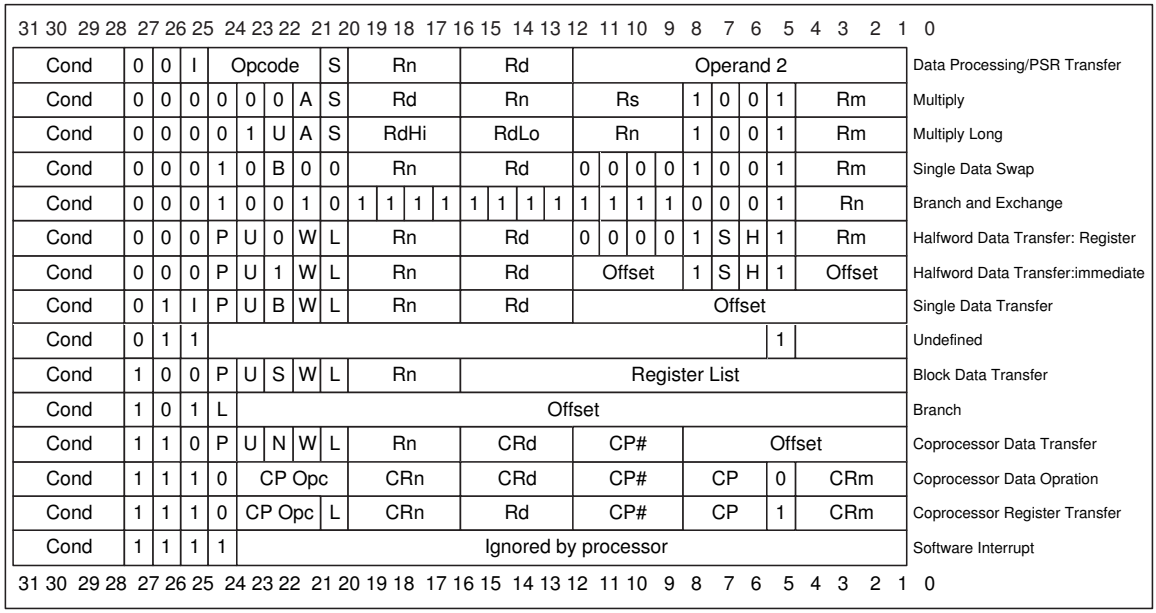

Figure 4.2: ARM Instruction Set Formats

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear and Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Table 4.4: Condition Code Summary

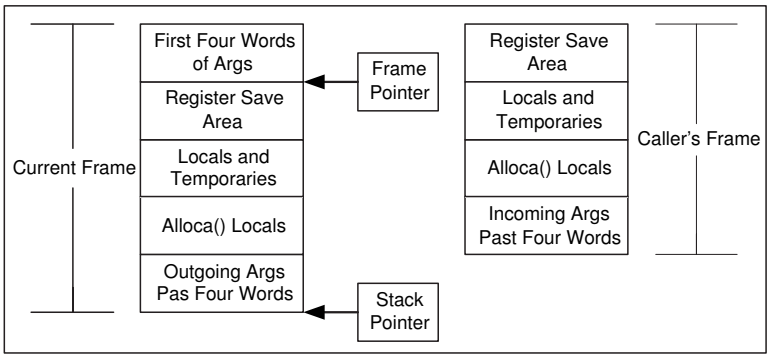

Figure 4.3: ARM Stack during Subroutine Call

4.2.4 Subroutine Call

A list of instructions are executed to save and restore contexts before and after a function call. They are called respectively Prologue and Epilogue [7]. During Prologue, a sequence of instructions pushes the incoming arguments in *R0, R1, R2* and *R3* to the argument locations in the stack, saves the values of the registers *R11, R13, R14* and *R15* and updates the stack parameters. During Epilogue, the saved registers are restored and the returned values are loaded in *R0* and/or *R1* if they exist. Figure 4.3 shows the stack format during a subroutine call.

4.3 Armed E-Bunny Architecture

Armed E-Bunny contains six major components: the Method Initializer, the Profiler, the KVM's Interpreter, the Machine Code Execution engine, the ARM compiler and the cache manager. Figure 4.4 depicts the architecture of Armed E-Bunny and shows the relationship between its components. The Virtual Machine Execution engine and the Interpreter exist already in KVM. The fragment of code on which interpretation or compilation is applied is the method.

Many features like reduced memory footprint and efficient use of different stacks make our Armed E-Bunny an appropriate Java acceleration technology for embedded systems. Armed E-Bunny is a selective dynamic compiler. Only the frequently called methods are compiled and saved in the cache structure. This strategy led us to have reduced memory footprint results that does not exceed 119KB. Furthermore, the use of two stacks (one for interpretation and one for compiled method execution) is a real advantage to preserve the portability to a high extend of the virtual machine. However, the drawback of this way is its complexity due to the following reasons. First, method's related data (e.g arguments) should be transferred between the Java and native stacks

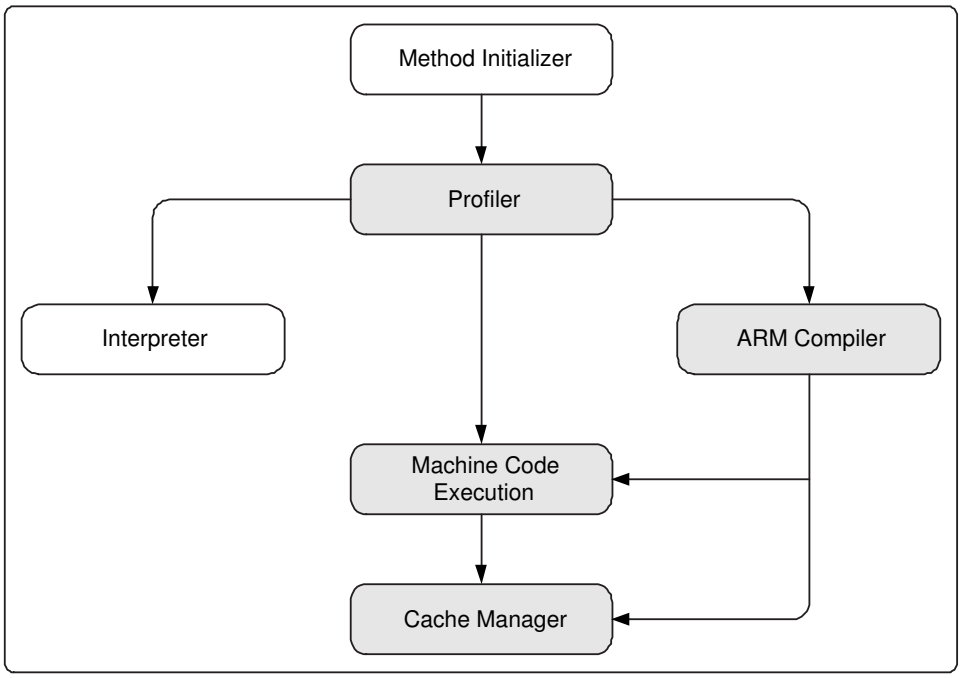

Figure 4.4: Armed E-Bunny Architecture

when necessary. Second, since ARM architecture specifies a technique for a subroutine call that relies on the registers more than the stack, we were obliged, each time a method is called, to transfer data needed (e.g subroutine arguments) from the stack to registers and vice versa. Here we describe the mechanism of method interpretation and compilation.

Initially a method is supposed to be interpreted. Once the method is loaded by the virtual machine, the profiler checks its frequency by verifying if its counter has reached the threshold defined in our implementation. The result of this verification identifies if the method is hotspot [1]or not. If it is recognized as hotspot, a quick switch to the Machine Code Execution or to the ARM Compiler is performed by the profiler. Otherwise, the Interpreter takes its advantage over the other components and continues its work normally.

During the switch mechanism, two cases are possible. If the method is already compiled, the reference that points to its machine code in the cache is called. Otherwise, the ARM compiler translates the given method into ARM machine code and stores it in the cache. Once the translation is completed, the corresponding machine code is called and a reference to this code is saved in the structure of the method for future calls. Detailed explanation about all these mechanisms is described in section 4.4. Here we highlight the roles of the six main components of our proposed embedded ARM dynamic

[1]Frequently called method.

compiler's architecture and we describe the interaction between its components.

Method Initializer

This component already exists in the virtual machine. Its is responsible for loading all the method's references and parameters needed for both interpretation and compilation.

Profiler

The profiler has many related roles and communicates with all the other components. By checking the method's counter, this component is able to identify if a method is a hotspot or not and specify the mode in which the decoding should be performed. Once a switch to the compilation mode is done, additional role to the profiler allows it to choose either to execute the method's corresponding machine code found in the cache directly or to call the compiler and then run the relevant machine code. The ARM compiler is called only if the method is not already compiled.

Interpreter

The interpreter of KVM decodes the bytecodes of a given method into executable machine code. In Armed E-Bunny, the interpreter communicates with the profiler before starting its decoding process. If the method is identified as a hotspot, the interpreter stops its work and the profiler switches to other components.

Machine Code Executer

This component is responsible of invoking the machine code of a method. Once the stacks and the registers are filled with the data needed for method execution, the reference of the machine code found in the cache is called. A possible switch back to the interpreter is triggered from this component.

ARM Compiler

The ARM compiler is called by the profiler. It is a one pass compiler. Its role is to go through the bytecodes of a given method and to generate the corresponding ARM executable machine code. Once the generation is done, a management of the cache is applied, if needed, and the code is saved inside. A reference to the generated code

is saved in the method structure for future invocations. In some special cases, the compiler switches to the interpreter in order to gather some information or to invoke some complicated native methods and then returns back to continue its translation. Detailed explanation about its functionality is provided in section 4.4.

Cache Manager

Generated native code is saved in a particular structure in the permanent space in the heap called the cache. Since this structure has a fixed size, a management should be applied in order to find enough space for the generated code. This management process is invoked only if the cache is full. Actually, this process passes through all the methods generated in the cache, selects the ones that have not been called for the largest period of time and removes them. We use the LRU algorithm (Least Recently used) [22]. A queue is used to keep the chronological order of invoked methods. This queue is updated each time a compiled method is invoked. The only disadvantage in LRU algorithm is that some methods may be recompiled several times. However, our experiments show its efficiency.

4.4 Armed E-Bunny Design and Implementation

Our system covers, besides the compilation of all kind of bytecodes, the different issues of the integration of a dynamic compiler into a virtual machine such as garbage collection, exception handling, etc. In this section, we discuss in detail the design of Armed E-Bunny. Profiling and mode switching, one pass method compilation, garbage collection, exception handling and threads are the following main points discussed.

4.4.1 Profiling and Mode Switching

The profiler of Armed E-Bunny performs simple check over the frequency of a method in order to identify it as hotspot or not hotspot. If a method is recognized as hotspot, a switch from interpretation to compilation mode is applied. Otherwise, the interpreter continues its execution. In order to perform this check, a counter is added to the structure of the method and is updated each time the method is called.

Once the virtual machine finishes loading the parameter of a method, the profiler compares the value of its counter to the threshold specified in the implementation. Depending on the result, the profiler either

```
// Profiler
if (methodCounter >= threshold)
begin //Hotspot Method

  if (currentMethodNotCompiled)
     compileCurrentMethod;

  if (currentMethodCompiled)
  begin //Machine code execution

//The 3 following instructions prepare the native
//stack before execution

     PushNativeStack methodArguments;
     LeaveSpace; //for local variables
     LeaveSpace; //for returned values
     call currentMethodMachineCode;

     popNativeStack returnedValue;
     pushJavaStack returnedValue;

end //Machine Code execution
end //Hotspot Method
```

Table 4.5: Profiler Algorithm

- compiles the method and then executes its corresponding generated code if its counter reaches the threshold and it is not already compiled. Or,
- executes the method's generated codes if its counter reaches the threshold and it is already compiled. Or,
- continues the interpretation of the corresponding method.

When one of the first two cases is chosen, all the method parameters and information needed are transferred from the Java stack to the native stack before execution. Then, the results are pushed back in the Java stack after finishing the execution of the code generated. Table 4.5 summarizes briefly the implementation of the profiler.

4.4.2 One Pass Method Compilation

Our compilation spans over a lightweight one pass compilation technique that generates a code of reasonably good quality. The generated code is stack-based as Java bytecode, but uses many information that are computed at the compilation level. The main

role of the compiler is to pass through the method bytecodes and translate them into executable machine code (binaries) for ARM machines. The generated code is saved in the permanent memory and a reference to it is saved in the structure of the method for future calls.

In addition to bytecodes translation, and like all compilers, a list of ARM instructions are generated at the beginning of each method in order to save the values of some registers and variables. Also, an opposite list is generated at the end of the same method to restore the saved values and switch back to the interpreter. These two processes are called respectively Machine Code Prologue and Epilogue and are used to save and restore contexts. This section explains in details all the steps that our compiler passes through.

Machine Code Prologue and Epilogue

Re-establishing the calling method context after the execution of the called method, handling native garbage collection and manipulating threads are the main reasons for generating the prologue and epilogue. During prologue, the values of the registers *R10-R15* are pushed into the native stack, the value of frame pointer is saved in the thread data structure, the reference of the generated code is saved in the method data structure and the current method counter is incremented by 1. During epilogue, the values of the registers *R10-R15* are restored and the value of the frame pointer saved in the thread data structure is updated. Indeed, the prologue instructions figure on top of the method generated code and the epilogue instructions figure in the generated code of the return bytecodes (*return, ireturn, areturn and lreturn*). Table 4.6 shows the set of ARM assembly instructions used for the prologue and epilogue.

Bytecodes Translation

Unlike common compilers, Armed E-Bunny is based on a bytecodes translation technique which avoids complex computations. The translator passes through the method's bytecodes using a while loop, identifies the bytecode and then generates its corresponding ARM machine code. The machine code generation is implemented by following a top down strategy. First, each bytecode is translated to a list of C functions called 'Gen' functions (e.g *GenMOV*). Each one of them is able to generate a list of one or more ARM assembly language instructions. Second, inside the 'Gen' functions, each instruction is transformed to its equivalent in ARM machine code hexadecimal form by our own ARM assembler implemented inside the virtual machine. Finally, each time an instruction is generated, there is a function responsible of saving it into a frame in the *MachineCode* table to be eventually executed. Table 4.7 shows an example of such

```
//begin Prologue
mov R12, R13;
push R12;
push R10;
mov R10, R13;
mov CurrentThread->LastFramePointer, R10;
push R14;
push R11;

//begin Epilogue
pop R11;
pop R14;
pop R10;
pop R12;
mov R13, R12;
mov CurrentThread->LastFramePointer, R10;
```

Table 4.6: Prologue and Epilogue

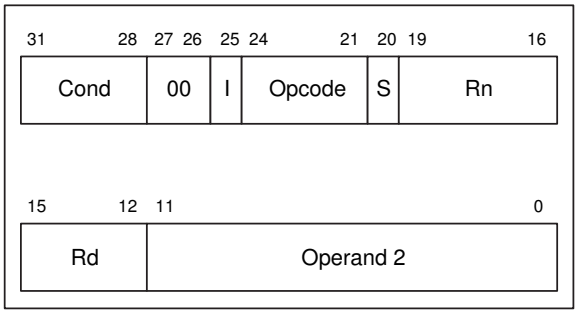

Figure 4.5: Data Processing Instruction

translation technique.

Actually, our ARM assembler that is implemented inside the virtual machine does not perform the work of a real assembler. However, its role is to transform directly part of the assembler instructions into ARM machine code ready to be executed. We refer in our implementation to the documentation of ARM assembly language and we follow the same architecture used in transforming assembly instruction to executable machine code. Figure 4.5 shows the binary representation of the data processing instructions which includes all the arithmetic (e.g. *add, sub*), logic (e.g. *and, or*), shifts and moves instructions.

The size of all machine code instructions is 32 bits. The first 12 bits are used for representing the second operand. The register R_n is the first operand register. The register R_d is the destination register, S (bit 20) sets the condition code and permits to update the values of the *CPSR* flags. The bits 21 to 24 specify the opcode

IADD bytecode Implementation

```
//begin
  GenPopToRegister(R0);
  GenAddRegisterToRegisterContent(R13,R0);
//end
```

GenPopToRegister(R0) Implementation

```
//begin
//This function generates the hexadecimal values
//of the assembler instruction

  SetMachineCode(0x........); //ldr R0, [R13]!
//end
```

GenAddRegToRegContent(R13,R0) Implementation

```
//begin
//This function generates the hexadecimal values
//of the assembler instruction

  SetMachineCode(0x........); //ldr R1, [R13]
  SetMachineCode(0x........); //add R1, R0
  SetMachineCode(0x........); //str [R13], R1
//end
```

SetMachineCode(0x........) Implementation

```
//begin
  *(MachineCodeTable) = 0x........;
//end
```

Table 4.7: Machine Code Generation

of an instruction. The bit *I* specifies if the second operand is an immediate value and the last four bits sets the condition under which the instruction will be executed. Since all the instructions size is fixed, in some cases we were obliged to represent an assembly instruction by a set of many machine code instructions. A machine code instruction is the 32 bits binary number that is understood by the ARM microprocessor (e.g *0xe3a01002* is the machine code representation of *mov R1, R2*). For instance, loading an immediate value in a register requires the generation of a sequence of many instructions in addition to many bit-manipulation operations. Table 4.8 shows the implementation of this mechanism.

The above strategy of translation is applied on all the bytecodes, even though we differentiate them with respect to their implementation complexity and functionalities. Some bytecodes such as loads (e.g. *iload*), stores (e.g. *astore*), stack manipulation (e.g. *push, pop*), arithmetic except division, logic and shift (e.g. *iadd, iand*, ishr), and branching (e.g. *ifne, ificmpeq*)are directly translated into machine code, which reproduces the interpreter behavior on the native stack. Other bytecodes such as field access, object creation, array manipulation, method invocation, return, monitor, casting and exception require some virtual machine services (e.g. method lookup, field reference resolution) at runtime in order to be translated. Generating the corresponding native code instruction by instruction, including virtual machine services, yields a complex and very bulky code. For this reason, we adopted in Armed E-Bunny a different approach, which allows to call these services from the native code. In addition to calling virtual machine services, we implemented some C functions to generate some complicated operations of some bytecodes. These C functions use the same stack we use for native code, so we do not need to transfer the method parameters to the Java stack each time we need to switch to C mode. Hence, the resulting generated machine code is compact and less complex. In the sequel, we present the generated ARM assembly instructions of a representative bytecode for the mostly used category of bytecodes, distinguished with respect to their functionalities. Notice that the following ARM assembly instructions are represented in hexadecimal before saving them into the *MachineCode* table.

Loads, Stores and Stack Manipulation Bytecodes This category includes the following bytecodes: *aconst_null, iconst_x, lconst_x, bipush, sipush, ldc, ldc_w, ldc2_w, iload, iload_x, lload, lload_x, aload, aload_x, iallaload, aaload, baload, caload, saload, istore, istore_x, lstore_x, astore_x, iastore, lastore, aastore, bastore, castore, sastore, pop, pop2, dup, dup_x1, dup_x2, dup2, dup2_x1, dup2_x2, swap*. Bytecodes of this category perform operations on the top of the stack, load local variables and constant pool entries onto the operand stack and registers, and store the register and stack values onto local variables. Table 4.9 shows the translation of *istore* bytecode.

```
// Move an immediate value imm into a register Reg
// The function InstSetMachineCode fills the table of MachineCode
// with the ARM machine code

if(imm <= 255 && imm >= 0)
  InstSetMachineCode(0xe3a00000 | (((int) Reg) << 12) | ((BYTE) imm));
  // mov Reg, (BYTE) imm
else if(imm > 255 && imm <= 65535)
{
  InstSetMachineCode(0xe3a00000 | (((int) Reg) << 12) | ((BYTE) imm));
  // mov Reg, (BYTE) imm;
  InstSetMachineCode(0xe3a00000 | ((BYTE) (imm>>8));
  // mov R0, (BYTE) (imm>>8);
  InstSetMachineCode(0xe1900400 | (((int) Reg) << 16) | (((int) Reg) << 12));
  // orr Reg, R0<<8;
}
else if(imm > 65535 && imm <= 16777215)
{
  InstSetMachineCode(0xe3a00000 | (((int) Reg) << 12) | ((BYTE) imm);
  // mov Reg, (BYTE) imm;
  InstSetMachineCode(0xe3a00000 | ((BYTE) (imm>>8));
  // mov R0, (BYTE) (imm>>8);
  InstSetMachineCode(0xe3a01000 | ((BYTE) (imm>>16));
  // mov R1, (BYTE) (imm>>16);
  InstSetMachineCode(0xe1900400 | (((int) Reg) << 16) | (((int) Reg) << 12));
  // orr Reg, R0<<8;
  InstSetMachineCode(0xe1900801 | (((int) Reg) << 16) | (((int) Reg) << 12));
  // orr Reg, R1<<16;
}
else
{
  InstSetMachineCode(0xe3a00000 | (((int) Reg) << 12) | ((BYTE) imm);
  // mov Reg, (BYTE) imm;
  InstSetMachineCode(0xe3a00000 | ((BYTE) (imm>>8));
  // mov R0, (BYTE) (imm>>8);
  InstSetMachineCode(0xe3a01000 | ((BYTE) (imm>>16));
  // mov R1, (BYTE) (imm>>16);
  InstSetMachineCode(0xe3a02000 | ((BYTE) (imm>>24));
  // mov R2, (BYTE) (imm>>24);
  InstSetMachineCode(0xe1900400 | (((int) Reg) << 16) | (((int) Reg) << 12));
  // orr Reg, R0<<8;
  InstSetMachineCode(0xe1900801 | (((int) Reg) << 16) | (((int) Reg) << 12));
  // orr Reg, R1<<16;
  InstSetMachineCode(0xe1900c04 | (((int) Reg) << 16) | (((int) Reg) << 12));
  // orr Reg, R2<<24;
}
```

Table 4.8: Loading Immediate Value into a Register

```
// istore : Store integer into local variable
{
  //[ ]: means the content of
  //fsize is the number of local variables of the current method
  //The displacement is multiplied by 4 because the addresses
  //in the stack grow by 4

  int index=ip[1];
  //ip is the address of the bytecode and ip[1] is the content of ip+1

  ldm sp!, {Rn};
  //pop the content of SP (top of the stack) to register Rn

  str Rn, [R10, 4*fsize-index+3];
  //Store Rn at the address [R10]+(4*fsize-index+3)
  //R10 contains the frame pointer
  //(4*fsize-index+3) is an operation that calculateds the location
  //of the local variable in the stack
}
```

Table 4.9: *istore_x* Translation

Arithmetic and Logical Bytecodes This category includes the following bytecodes: *iadd, ladd, isub, lsub, imul, lmul, idiv, ldiv, irem, lrem, ineg, lneg, iand, land, ior, lor, ixor, lxor, iinc.* Bytecodes of this category perform arithmetic and logical operations. Table 4.10 shows the translation of *isub* bytecode.

Shift and Type Conversion Bytecodes This category of bytecodes includes the following bytecodes: *ishl, lshl, ishr, lshr, iushr, lushr, i2l, l2i, i2b, i2c, i2s.* Bytecodes of this category perform shift and type conversion operations. They are mostly executed through the ARM *mov* instruction. Table 4.11 shows the translation of *ishr* bytecode.

Branching Bytecodes This category includes the following bytecodes: *ifeq, ifne, iflt, ifge, ifgt, ifle, if_icompeq, if_icompne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, if_acmpeq, if_acmpne, goto, tableswitch, lookupswitch, goto_w.* Bytecodes of this category perform unconditional and conditional branching, which can serve for the if/else, switch and loop operations. Branching can be performed in two ways: forward and backward. When applying backward branching, a simple jump to the address of the instruction already generated is applied. Such operation is not complicated since the addresses of all the generated instructions are saved at the compilation time in a defined structure. On the other hand, when applying forward branching, the address of the instruction or bytecode to which the jump

```
// isub : Subtract integer
{
  ldm sp!, {Rd};
  //pop the content of SP (top of the stack) to register Rd

  ldm sp!, {Rn};
  //pop the content of SP (top of the stack) to register Rn

  sub Rn, Rn, Rd;
  //Rn=Rn-Rd

  stm sp!, {Rn};
  //push the register Rn to the content of SP (top of the stack)
}
```

Table 4.10: *isub* Translation

```
// ishr : Arithmetic shift right
{
  ldm sp!, {Rd};
  //pop the content of SP (top of the stack) to register Rd

  ldm sp!, {Rn};
  //pop the content of SP (top of the stack) to register Rn

  mov Rn, Rn, asr Rd;
  //shift right arithmetically Rd times the content of Rn

  stm sp!, {Rn};
  //push the register Rn to the content of SP (top of the stack)
}
```

Table 4.11: *ishl* Translation

```
// goto : Branch always
{
  if(backward_branch)
    Generate jump instruction;
  else //forward branch
    Leave free space;

  Continue bytecode generation Until reaching the target bytecode;

  if(target_bytecode_reached)
  {
    Load the address of the corresponding free space leaved;
    Generate jump instruction at the loaded address;
  }
}
```

Table 4.12: *goto* Translation Algorithm

should be performed does not exist yet in the structure containing the addresses. For this reason, the complete translation of the forward branching is postponed until the target bytecode is reached and compiled, while the address of the uncomplete generated instruction is saved in a particular structure defined for this purpose. Whenever the compiler reaches the target instruction or bytecode, the address of the uncomplete generated instruction is loaded and its content is filled with the corresponding ARM machine code. Table 4.12 shows the translation algorithm of *goto* bytecode.

Allocation, Array Manipulation and Field Access Bytecodes This category includes the following bytecodes: *new, newarray, anewarray, arraylength, multianewarray , putfield, getfield, putstatic, getstatic*. Bytecodes of this category create and manipulate objects and arrays, and access to object fields using symbolic references in order to get or set their values. Indeed, such bytecodes need to call some KVM services in order to resolve field and class references, initialize classes and allocate memory space in the heap. To deal with that, we implement some functions in ARM assembly and C languages in order to call these KVM subroutines and return the resolved references and results into registers. Table 4.13 shows the translation of *getfield* bytecode.

Invoke Bytecodes This category includes the following bytecodes: *invokeVirtual, invokeSpecial, invokeInterface and invokeStatic*. Bytecodes of this category permit a method to call another method. In fact, such bytecodes need to call some KVM services such as method reference resolution and method lookup in order to

```
// getfield : get field value in object
{
  mov R0, ip;
  // ip is the address of the bytecode
  ldm sp!, {R1};
  //pop the content of SP (top of the stack) to register R1

  mov LP, PC;
  mov PC, &getfield_function;
  //These two instructions are used to call the function
  //getfield_function
  //The arguments of getfield_function ip and object_reference
  //are sent into R0 and R1. The function getfield_function
  //calls some KVM services to get the field value of the
  //addressed object and return the result into the register R0

  stm sp!, {R0};
  //push the register R0 to the content of SP (top of the stack)
}
```

Table 4.13: *getfield* Translation

load the method references, parameters and local variables and verify them. To deal with that, we apply the same strategy as with the allocation, array manipulation and field access bytecodes, i.e implementing some functions to call these KVM subroutines. Moreover, according to our strategy of compilation, compiled method can only call a compiled or native method, which means that a switch to interpretation mode is not allowed at this level of execution. Hence, there are three different cases to be treated. First, if the invoked method is native, a special techniques is used in order to call the corresponding native function. This technique is detailed in the section "Java Native Methods". Second, if the invoked method is already compiled, a simple call to its executable machine code is performed. The third case is when the method is not yet compiled. In such case, the method is first compiled and then its generated machine code is executed. Table 4.14 shows the translation of *invokeVirtual* bytecode.

Return Bytecodes This category includes the following bytecodes: *return, ireturn, areturn and lreturn*. Bytecodes of this category are always placed at the end of a method in order to return to the calling method. In addition to the return operation, such bytecodes restore the calling method context (epilog) by pushing the returned values in the stack, restoring the old values of the status registers and deleting the method frame from the stack. Table 4.15 shows the translation

```
// invokeVirtual : invoke virtual method
{
  mov R0, ip;
  // ip is the address of the bytecode
  sub SP, SP, #8;
  //This instruction is used to leave 2 empty spaces
  //in the native stack

  mov LP, PC;
  mov PC, &invokevirtual_function;
  //These two instructions are used to call the function
  //invokevirtual_function. The arguments of invokevirtual_function ip
  //is sent into R0. The function invokevirtual_function
  //returns its results into R0, R1 and the top of the native stack

  //Switch// R0
  1: add SP, SP, R1
      //If the value of R0 is 1, this means that the called method
      //is Java native and was treated in the function
      //invokevirtual_function
      //This instruction is used to update the native stack

  0: ldr R0, [SP]
     //If the value of R0 is 0, this means that the address of the
     //method to be executed is sent in the register R1 and the
     //number of local variables is pushed at the top of the native stack

     sub SP, SP, R0
     //This instruction is used to leave empty space for local variables

     mov LP, PC;
     mov PC, R1
     //These two instructions are used to call the machine code
     //of the method

     add SP, SP, R0
     //This instruction is used to update the native stack
     //after the method call

     UPDATE_Interpreter_Global_Varibles;
}
```

Table 4.14: *invokeVirutal* Translation

```
// return : return from method
{
  ldm R13!, {R11, R14, R10, R12};
  //This instruction is used to pop the top four values
  //in the stack into R11, R14, R10 and R12 respectively
  mov R13, R12;
  mov CurrentThread->LastFramePointer, R10;

  mov R0, (fsize*4)+8;
  //fsize is the number of local variables of the current method
  //(fsize*4)+8) is the total frame size reserved in the stack
  //for a given method
  //fsize is multiplied by 4 because the addresses
  //in the stack grow by 4

  mov PC, LP;
  //This instruction is used to return
}
```

Table 4.15: *return* Translation

of *return* bytecode.

Fast ByteCodes Translation

In KVM, some method's bytecodes such as *getfield, putstatic, invokevirtual*, etc. are replaced by fast bytecodes the first time they are executed in this method [21]. Such bytecodes need virtual machine services such as *resolveMethodReference*, which calls a set of functions in order to load the method references and parameters. To avoid that each time a method is called, KVM uses a mechanism that replaces some bytecodes by their corresponding fast bytecodes (e.g. *invokevirtual* by *fastinvokevirtual*) and saves all the values needed in the cache. The next time the same method is executed, all the references and parameters are loaded from the cache instead of calling the virtual machine functions. A reverse process is also applied in case the cache is full and the fast bytecodes are replaced back by their original versions. The KVM documentation mentioned that this mechanism accelerated the execution of the virtual machine by a factor of 5%.

Armed E-Bunny compiles also all the fast bytecodes. It uses the same KVM's mechanism to replace the bytecodes by their corresponding fast bytecodes and load the method references and parameters from the cache. Indeed, at this level of compilation of this kind of bytecodes, the system do not need to switch the execution mode in order

```
// ldiv :
{
  Pop_Argument1_ToRegister(R1, R0);
  Pop_Argument2_ToRegister(R3, R2);

  Call_Function_longdivision;

  Push_ReturnedValue_FromRegister(R0, R1);
}

// Function longdivision(long64 arg1, long64 arg2)
{
  return ll_rem(arg2,arg1);
  // ll_rem is a function of the C compiler that manipulates the long division
}
```

Table 4.16: *ldiv* Translation Algorithm

to call the virtual machine services that are needed. A simple call from a C function to another one is performed. The reverse mechanism is called automatically by the interpreter whenever is needed.

C functions

In the implementation of Armed E-Bunny, we followed the strategy of calling C functions to generate the complicated operations of some bytecodes instead of generating the corresponding native code instruction by instruction. Some of these functions are provided by the virtual machine, others are implemented in our compiler. Invoking these methods from native mode is very simple while calling them from machine code requires a set of pre-execution operations to be performed. Indeed, unlike other architectures, ARM's function receives its arguments inside the registers *R0-R3* and returns its result in the register *R0*. For this reason, each time we needed to invoke a C function, we had to transfer all the arguments from the stack to registers and then the result back from the registers to the stack. Although this procedure seems to be a little bit elaborated, experiments have shown its efficiency over other strategies. Table 4.16 shows the translation algorithm of *ldiv*, which is an instance of bytecodes that need to call a C function in order to complete their translations.

```
//Algorith of the code introduced in the implementation of the
//invoke bytecodes
if (Native_Method)
{
  Pop_Arguments_Form_Native_Stack(thisMethod);
  Push_Arguments_Into_Java_Stack(thisMethod);

 //begin calling the KVM functions responsible of executing native methods

  Call_invokeNativeFunction(thisMethod);
  Call_TRACE_METHOD_EXIT(thisMethod);

  //end calling the KVM functions responsible of executing native methods

  Pop_Returned_Values_Form_Java_Stack(thisMethod);
  Push_Returned_Values_Into_Native_Stack(thisMethod);
}
```

Table 4.17: Java Native Methods Algorithm

Java Native Methods

The Java virtual machine provides a list of Java native methods that are neither interpreted nor compiled. These methods are implemented directly in the C language. They are based on the Java stack. In Armed E-bunny, the profiler deals with this kind of methods and calls their corresponding native functions before switching to the compilation mode. However, these subroutines may be called also during compilation by the invoke bytecodes (i.e. *invokevirtual, invokeinterface and invokestatic*). For this reason, a process of three steps is performed inside the implementation of these bytecodes once a native method is detected. Table 4.17 illustrates the algorithm of this process.

First, the compiler use the Java stack to push the method's arguments. Second, the *invokenativefunction* of KVM is called in order to invoke the method. Finally, when the execution is accomplished, the results and the arguments are popped from the Java stack and only the results are pushed back into the native stack. Indeed, this mechanism allows us to switch successfully between the two stacks without the need to return back to the interpreter.

4.4.3 Garbage Collection

Our system allows native method calls and memory allocation during compilation mode. This means that the garbage collection of KVM may be called and some references saved

in the heap may be lost because the current algorithm of KVM garbage collection doesn't take into account the objet allocated in the native stack.

The current KVM garbage collection goes through three steps: mark, sweep and compact. Based on the result of marking, the garbage sweeps the free chunks to constitute consistent blocks and compact the heap, leading to a move of the object inside it. Objects addresses are then changed and therefore, references to them are updated. Hence, the main issue here is to enhance the marking algorithm, which scans only the Java stack, in order to scan the native stack and mark its live referenced objects.

This is done in our compiler by adding some features to the KVM garbage collection. Using C and ARM assembly language code, information about a thread native stack are gathered and passed to the marking process, which passes over the given native stack, scans it and marks all its live objects in the heap. Then, a switch to the sweep and compact functions of the KVM is performed. At the end of this mechanism, if necessary, the native stack references are updated through a code added to the KVM functions responsible of references updating. Table 4.18 shows the algorithm of the code used to mark and update the references of the native stack objects. This code is executed on the native stack of each live thread at the moment when the garbage collection is triggered. By doing that, the garbage collection will treat, whenever is called, all the native marked object as if they are Java stack object references.

4.4.4 Exception Handling

Exception handling is an important feature of the Java language which has specific semantics to be respected [21]. Our dynamic compiler handles it by generating efficient code for the bytecode *athrow* which is responsible of raising an exception. Indeed, additional ARM assembly code is also added to the functions that are called by *athrow* and new issues relevant to exception propagation are introduced. During compilation mode, the method that throws the exception is always compiled. However, the method that catches the exception can be either interpreted or compiled. In the two situations, a call to virtual machine functions is applied to throw the exception. Table 4.19 shows the algorithm of *athrow* implementation.

If the method catching the exception is compiled, the additional code added to the virtual machine *throwexception* function is used to locate the native instruction corresponding to the bytecode handling the exception. Once the handled native code is located, the compilation mode continues and a jump to the native instruction is executed. Otherwise, a switch to the interpreter is applied to continue its normal exception handling process.

```
if (Compiled_Method)
{
   //begin marking the native stack

   Load_Native_Stack_Parameters(thisThread);
   Load_Method_Frame(thisMethod);
   Select_Live_Objects(thisMethod_Frame);
   Mark_Native_Stack_References(Live_Objects);

   //end marking the native stack

   Call_KVM_Sweep/Compact;

   // begin updating the native stack references

   Load_Native_Stack_Parameters(thisThread);
   Load_Method_Frame(thisMethod);
   Select_Live_Objects(thisMethod_Frame);
   update_Native_Stack_References(Live_Objects);

   //end updating the native stack references
}
else

   Execute_KVM_Garbage_Collection;
```

Table 4.18: Garbage Collection Algorithm

```
// Athrow Implementaion
CallThrowException;

if (exceptinHandled)

begin //begin exception handling

  if (isCompiled(catchingMethod)

      jumpTo nativeCatchingMethod;
      //Continue in compilation mode

  else

      jumpTo interpreterCatchingMethod;
      //Switch to interpreter mode

end //end exception handling

else

InterpreterUnhandledException;
```

Table 4.19: Exception Handling Algorithm

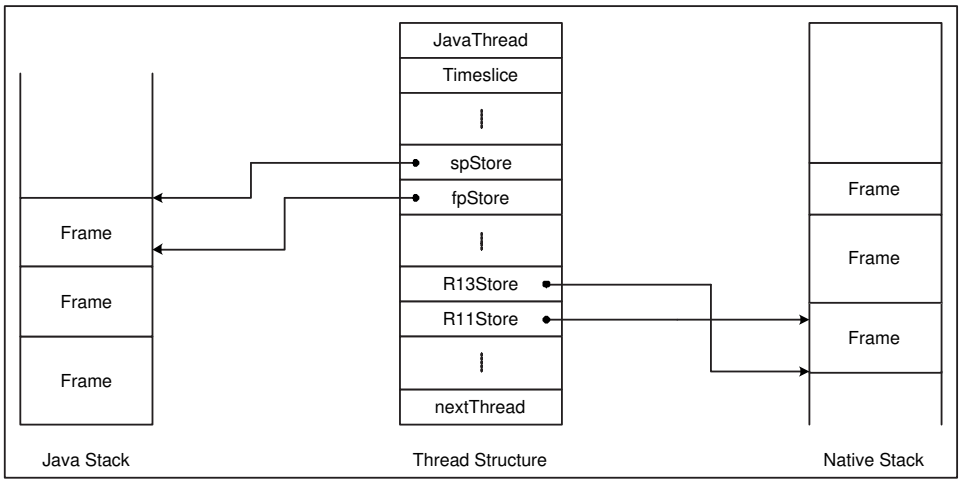

Figure 4.6: Thread Structure

4.4.5 Threads

The technique that has been used in handling threads is inspired by [10] and is still under enhancement. During interpretation, the KVM runs its original threads switch services, while during compilation, additional code is generated for bytecodes causing transfer control and contexts saving. During interpretation mode, the methods are executed on the Java stack , while during compilation mode, the generated code is executed on the native stack. In this context, the data structure representing the thread in the virtual machine must hold information about both Java and native stack in order to handle switching issues. These structures are updated each time a switch between threads occurred. Figure 4.6 shows the thread structure during compilation. Whenever the register holding the time-slice value reaches the zero value, a thread switch is triggered. Actually, the application of this switch mechanism prevents a compiled thread from going into an infinite loop and the virtual machine from hanging indefinitely.

4.5 Debugging

Armed E-Bunny is implemented in the C programming language and the ARM assembly language. Our dynamic compiler is cross-compiled on an Intel workstation using the GNU arm-linux-gcc and then is ported on an Embedded-Linux Handheld for execution. For the debugging and visualization issues, we used the GNU arm-linux-gdb together with ddd built on an Intel workstation, while the server of this debugger is installed on the Handheld. A connection between the debugger and its server permitted us to trace the execution of the virtual machine. Figure 4.7 shows a snapshot of a debugging session.

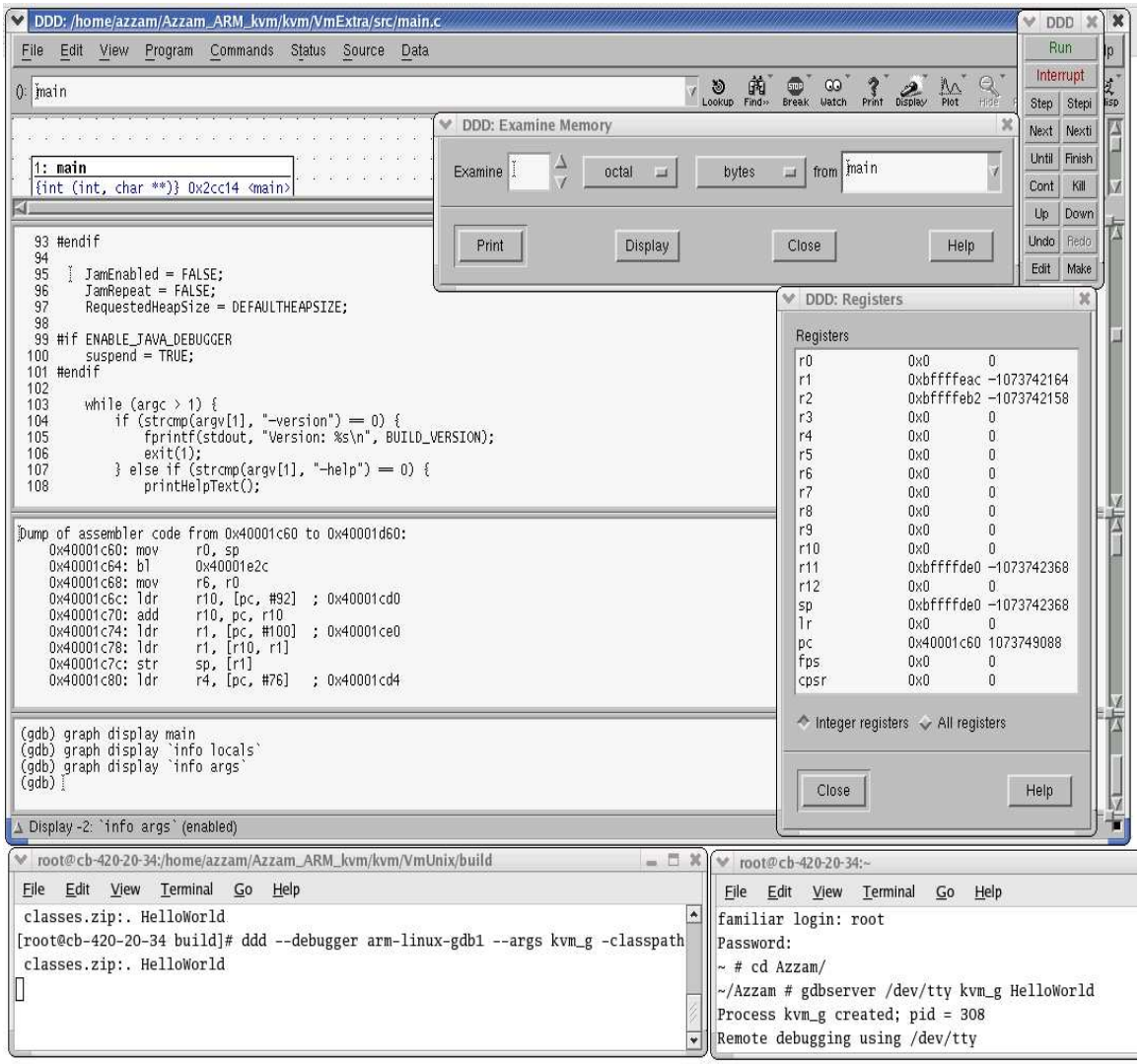

Figure 4.7: Debugging Session

	KVM 1.0.4	Armed E-Bunny	Speedup
Sieve Score	40	83	2.075
Loop Score	35	79	2.26
Logic Score	39	108	2.77
String Score	129	651	5.05
Method Score	35	81	2.32
Overall Score	55	200	3.64

Table 4.20: Comparison of KVM and Armed E-Bunny Performance

Debugging using these tools was difficult due to their restricted options and the delay exhibited during program tracing. Moreover, each time we needed to debug a new version of the modified virtual machine, we were obliged to transfer the executable file from the workstation to the Handheld, which also takes time. However, the execution speedup we reached made all this worthwhile.

4.6 Experimental Results

To test the results of Armed E-Bunny in the virtual machine, we ported its ARM executable to a Handheld and we executed it. Our results shows that Armed E-Bunny requires additional memory space that does not exceed 119KB, including the executable footprint overhead and the translated code storage.

The performance of Armed E-Bunny selective dynamic compiler is evaluated by running the CaffeineMark benchmark on the original version of KVM 1.0.4 with and without Armed E-Bunny. The results, which is illustrated in table 4.20, demonstrated that Armed E-Bunny produces an overall speedup of 360 % over the original KVM 1.0.4. Figure 4.8 shows a snapshot and a comparison chart of our tests on an Ipaq H3600 under Embedded-Linux.

4.7 Conclusion

This chapter described a new acceleration technology for Java embedded virtual machines target ARM 16/32-bit embedded system processors. This technology is based on a selective dynamic compiler built inside the J2ME/CLDC (Java 2 Micro Edition for Connected Limited Device Configuration). Our results show that our work comes up with an efficient, lightweight and low-footprint accelerated Java virtual machine ready to be executed on ARM embedded machines. In this chapter, we presented first the architecture of ARM platform, then we detailed the architecture, the design as well as

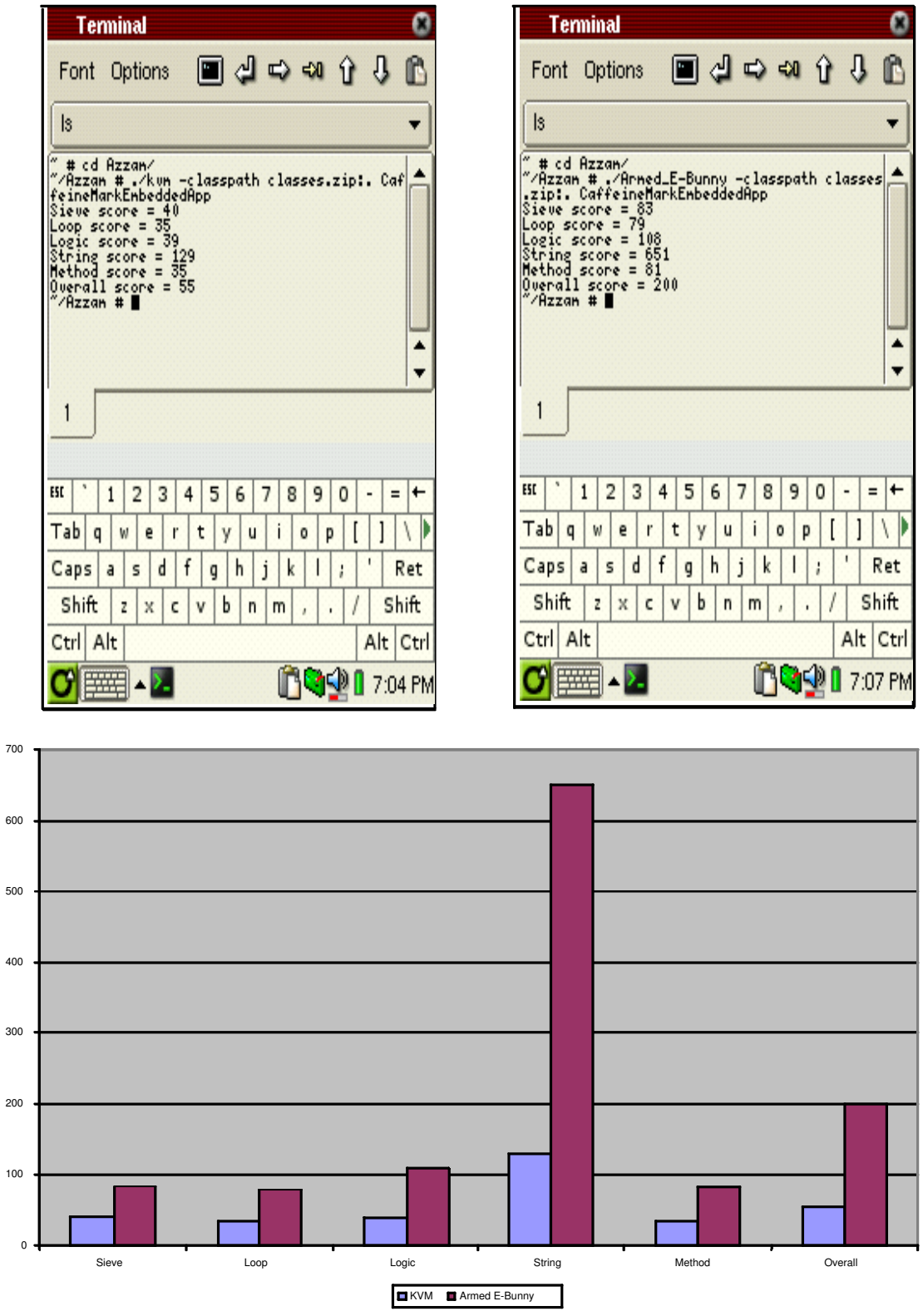

Figure 4.8: Caffeine Scores of the KVM and Armed E-Bunny

the implementation and debugging issues of Armed E-Bunny. Our experimental results prove that a speedup of 360% over the last version of Sun's KVM is accomplished by Armed E-Bunny with a footprint overhead that does not exceed 119 kilobytes.

Chapter 5

Conclusion

The poor execution performance is still the major problem of Java. Enhancing the performance of the Java virtual machine is becoming an interesting domain of research for many developers. Consequently, many optimization techniques have been proposed. In the beginning of this document, we presented the main components of the Java virtual machine and we tried to highlight all the features related to a Java program execution. Furthermore, we described in detail the Kilo virtual machine, on which we applied our acceleration technique. Although the main advantage of the Java virtual machine is its portability thanks to its interpreter, the poor performance of the interpretation mechanism is a severe drawback that affects the whole Java program execution. All the Java virtual machine acceleration techniques are discussed in chapter 3. In addition, we described some of the existent Java virtual machines endowed with dynamic compilers. The Java acceleration techniques are divided into two main approaches: hardware and software acceleration. Hardware acceleration can achieve a significant speedup in term of virtual machine performance, but the high power consumption and the cost of the acceleration technologies leaded developers to deviate to software acceleration. Among the software acceleration techniques, the dynamic compilation proved that it is the most efficient, even if it makes the Java virtual machine looses part of its portability.

In this work, we were concerned with the acceleration of the Java virtual machine embedded into resource-constrained devices. Embedded devices have limited hardware capabilities illustrated particularly into very small memory and low battery life. Such resource limitations stand in the way of the acceleration techniques requiring huge data structures and energy consumption, and make many of them not applicable in the context of embedded systems and then not relevant to accelerate the embedded Kilo virtual machine (KVM). In fact, these limitations bind some restrictions on dynamic compilation techniques and prevent dynamic compilers from accomplishing the same performance results as on desktop and servers. Endowing a dynamic compiler into an embedded virtual machine poses many implementation difficulties. The overhead

memory and power needed by the dynamic compilers should fit with the system memory and battery capabilities.

The architecture of ARM platform, as well as the architecture, design and implementation of Armed E-Bunny, our selective dynamic compiler targeting ARM processors, are highlighted in chapter 4. The ARM architecture is becoming the industry leading 16/32 bit embedded system processor solution. ARM powered microprocessor are being routinely designed into a wider range of embedded systems than any other 32-bit processor. The wide applicability of ARM architecture, that results in optimal system solutions at the crossroads of high performance, small memory size and low power consumption, leaded us to choose ARM processors as the target of Armed E-Bunny. Armed E-Bunny is a dynamic compiler based on the selective dynamic compilation approach. It is built inside the last version of Sun Kilo virtual machine (KVM 1.0.4). Our dynamic compiler uses a very lightweight profiler that allows only frequently executed method to be compiled. All the expensive profiling techniques and the heavyweight optimizations are avoided. A one pass compiler passes through the bytecodes of the frequently called method and translates them into stack-based machine code which is stored in the cache. A cache manager is also implemented in order to free memory space for newly generated code whenever the cache is full. Our results demonstrated that our system, Armed E-Bunny, accomplished a significant performance speedup (3.6 times better than KVM) with a footprint overhead that does not exceed 119 kilobytes. We are very satisfied of our work and we are still motivated to accomplish more.

5.1 Future Work

Regarding our future work, we are still working on enhancing the quality of our ARM dynamic compiler in order to accomplish better speedup. At the same time, we are trying to optimize our cache management, garbage collection and threading mechanisms. Moreover, we are planning to port MIDP/CLDC together (including Armed E-Bunny) onto an Embedded-Linux Handhelds. By doing that, we will have an entire accelerated J2ME/CLDC ready to execute any midlet (including graphical ones) on embedded devices.

Bibliography

[1] A. Aho and J. Ulman. *Principles of Compiler Design*. Addison Wesley, 1977.

[2] B. Alpern, C. Attanasio, J. Barton, M. Burke, P. Cheng, J. Choi, A. Cocchi, S. Fink, D. Grove, S. H. M. Hind, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. Russell, V. Sarkar, M. Serrano, J. Shepherd, S. Smith, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno Virtual Machine. *IBM Systems Journal*, 39(1):211–238, Feb. 2000.

[3] D. F. Bacon, S. J. Fink, and D. Grove. Space- and Time-Efficient Implementation of the Java Object Model. In *Proceedings of ECOOP 2002, Object-Oriented Programming, 16th European Conference*, pages 111–132, Malaga, Spain, June 2002.

[4] Bytecodes. Method Call Overhead. http://www.bytecodes.com/techHACCEL2B.html, 2003.

[5] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java under Dynamic Optimizations. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, pages 13–26, Vancouver, Canada, June 2000.

[6] G. Comeau. Java Companion Processors versus Accelerators. http://www.zucotto.com, 2003.

[7] M. Corporation. ARM Guide. Technical report, Microsoft Corporation, USA, August 2003.

[8] T. Cramer, R. Friedman, T. Miller, D. Seherger, R. Wilson, and M. Wolczko. Compiling Java just in time. *IEEE Micro*, 17(3):36–43, 1997.

[9] M. Debbabi, M. Erhioui, L. Ketari, N. Tawbi, H. Yahyaoui, and S. Zhioua. Method Call Acceleration in Embedded Java Virtual Machines. volume 2659 of *Lecture Notes in Computer Science*, pages 750–759. Springer-Verlag, 2003.

[10] M. Debbabi, A. Gherbi, L. Ketari, C. Talhi, N. Tawbi, H. Yahyaoui, and S. Zhioua. E-Bunny: A Dynamic Compiler for Embedded Java Virtual Machines. In *Proceedings of the Third International Conference on the Principles and Practice of Programming in Java*, pages 100–107, Las Vegas, USA, 2004. ACM Press.

[11] L. Dickman. A comparison of interpreted java, wat, aot, jit, and dac. Technical report, esmertec, 2002.

[12] M. A. Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, 1993.

[13] V. K. Geetha Manjunath. A small hybrid jit for embedded systems. *SIGPLAN Notices*, 35(4):44–50, April 2000.

[14] G. Hedin, editor. *Effective Inline-Threaded Interpretation of Java Bytecode Using Preparation Sequences*, volume 2622 of *Lecture Notes in Computer Science*. Springer, 2003.

[15] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. Java bytecode to native code translation: the Caffeine prototype and preliminary results. In IEEE, editor, *Proceedings of the 29th annual IEEE/ACM International Symposium on Microarchitecture, December 2-4, 1996, Paris, France*. IEEE Computer Society Press, 1996.

[16] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[17] J. Kamdar. Embedded java in information appliances. Technical report, NAZOMI Communications, 2000.

[18] S. Liang and D. Viswanathan. Comprehensive profiling support in the Java Virtual Machine. In USENIX, editor, *Proceedings of the fifth USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99): May 3-7, 1999, San Diego, California, USA*, 1999.

[19] A. Limited. ARM7TDMI Data Sheet. Technical report, ARM Limited, 2001.

[20] A. Limited. ARM Developer Suite Assembler Guide. Technical report, ARM Limited, 2001.

[21] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[22] S. M. Majercik and M. L. Littman. Using Caching to Solve Larger Probabilistic Planning Problems. In *Proceedings of the 15th National Conference on Artificial*

Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98), pages 954–960, Menlo Park, July 26–30 1998. AAAI Press.

[23] P. Manze. Jeode Platform Java for resource-constrained devices. *Handheld and Wireless solutions*, 2:72–76, 2002.

[24] F. Maruyama. OpenJIT 2: The Design and Implementation of Application Framework for JIT Compilers. In USENIX, editor, *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM'01): April 23–24, 2001, Monterey, California, USA. Berkeley, CA*, Berkeley, CA, USA, 2001. USENIX.

[25] Nazomi. Bootsing the performance of Java Software on Smart Handheld Devices and Internet Appliance. http://www.nazomi.com, 2003.

[26] I. Piumarta and F. Riccardi. Optimizing Direct-threaded Code by Selective Inlining. In *In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[27] R. Radhakrishnan, N. Vijaykrishnan, L. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java runtime systems: characterization and architectural implications. *IEEE Transactions on Computers 50*, 2001.

[28] T. Sayeed, A. Taivalsaari, and F. Yellin. Inside The K Virtual Machine. http://java.sun.com/javaone/javaone2001/pdfs/1113.pdf, June 2001.

[29] K. Schmid. Esmertec's Jbed Micro Edition CLDC and Jbed Profile for MID. Technical report, Esmertec AG, Dubendorf, Switzerland, Spring 2002.

[30] B. Shannon. Java 2 platform enterprise edition, v 1.3. Technical report, July 2001.

[31] N. Shaylor. A Just-in-Time Compiler for Memory-Constrained Low-Power Devices. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 119–126, San Francisco, CA, USA, Aug. 2002.

[32] A. l. steve steele, Java Program Manager. Accelerating to meet the challenge of embedded java. Technical report, ARM, Cambridge, UK, November 2001.

[33] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[34] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-in-Time Compiler. *ACM SIGPLAN Notices*, 36(11):180–195, Nov. 2001.

[35] Sun. The Java HotSpot Performance Engine Architecture. Technical report, California, USA, Apr. 1999.

[36] Sun. Java 2 Platform, Micro Edition, Version 1.0 Connected, Limited Device Configuration. Specification. Technical report, California, USA, May 2000.

[37] Sun. KVM Porting Guide. Technical report, Sun MicroSystems, California, USA, September 2001.

[38] Sun. MIDP APIs for Wireless Applications: A Brief Tour for Software Developers. Technical report, Sun MicroSystems, California, USA, February 2001.

[39] Sun. The CLDC HotSpot Implementation Virtual Machine. Technical report, J2ME, California, 2002.

[40] Sun. Java 2 Platform, Standard Edition, v 1.4.2 API Specification. Technical report, April 2003.

[41] I. Transvirtual Technologies. Kaffe virtual machine documentation. http://www.kaffe.org/, July 2002.

[42] P. Wilson. Uniprocessor garbage collector techniques. *International Workshop Memory Managment*, 637, 1992.

[43] B. Yang, S. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. Chung, S. Kim, K. Ebcioğlu, and E. Altman. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 128–138, Newport Beach, California, Oct. 12–16, 1999. IEEE Computer Society Press.

[44] Decaf the wireless java core specification. Technical report, Aurora VLSI, Inc., Santa Clara, USA, December 2001.