ADDING AN INTERPRETER TO THE JIKES RVM

A thesis submitted to the University of Manchester
for the degree of Master of Science
in the faculty of Science and Engineering

2006

Anastasios D. Katsigiannis
School of Computer Science

Table of Contents

List of Figures 3

Abstract 4

Declaration 5

Copyright 6

Acknowledgments 7

Chapter 1 Introduction 8

1.1 Background 8

1.2 Goals and Objectives....9

1.3 Thesis layout....11

Chapter 2 Bytecode Interpreters 12

2.1 VMs and Bytecode interpreters....12

2.2 Pure bytecode interpreters....13

2.3 Direct-threading bytecode interpreters....15

2.4 Inline-threading bytecode interpreters....17

Chapter 3 Just-In-Time (JIT) Compilers 20

3.1 JIT compilers vs traditional compilers....20

3.2 JIT compiler evolution....22

3.3 Dynamic compilation system....23

3.4 JIT/JVM Interaction....25

Chapter 4 The Jikes Research Virtual Machine (RVM) 26

4.1 The Jikes RVM structure....26

4.2 The Java Stack....30

4.3 The AOS system....31

4.4 Building Jikes....33

Chapter 5 Implementation Details 35

5.1 The new approach for Jikes RVM....35

5.2 The VM_Interpreter..38
5.3 Inserting the VM_Interpreter into Jikes..41
Chapter 6 Conclusions **44**
Bibliography **47**

List of Figures

Figure 1: Opcodes and opcode operands..........14

Figure 2: Switch based interpreter..........14

Figure 3: Direct-threading interpreter..........16

Figure 4: Inline threading interpreter..........17

Figure 5: Java threaded interpreter..........18

Figure 6: Structure of a JIT compiler..........21

Figure 7: Jikes' Basic Architecture..........27

Figure 8: JTOC organization..........29

Figure 9: Stack & Stack Frames [Inside the Java 2 Virtual Machine, Bill Venners] 30

Figure 10: Adding two variables..........31

Figure 11: The AOS system..........32

Figure 12: The AOS with adaptive inlining..........33

Figure 13: Compilation vs Interpretation..........36

Figure 14: The new AOS system..........37

Figure 15 : Example of one of the interpreter's case branches..........40

Abstract

The increase of speed and the reduction of the memory footprint produced by the Java Virtual Machines is one the of the most challenging things that todays software engineers are dealing with. In the case of Jikes Research Virtual Machine one approach towards these goals is the addition of an interpreter as an extra module to the existing system. This thesis presents the theoretical background, the implementation details and the results of such an attempt.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Copyright in text of this thesis rests with the Author. Copies (by any process) either in full, or of extracts, may be made only in accordance with instructions given by the Author and lodged in the John Rylands University Library of Manchester. Details may be obtained from the Librarian. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the Author.

The ownership of any intellectual property rights which may be described in this thesis is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

Further information on the conditions under which disclosures and exploitation may take place is available from the head of School of Computer Science.

Acknowledgments

At first I would like to thank my parents Dimitrios Katsigiannis and Nikolitsa Katsigianni for all of their support and help throughout my one year studying at Manchester. Moreover I would like to thank Chris Kirkham for his advices and for the moral support for the completion of this thesis. But without any doubt the person that was the real catalyst for this thesis was Ian Rogers. He was always there willfully and tirelessly answering all of my questions and providing his knowledge and experience for the implementation and the debugging for some of interpreter's “darkest” spots. My thanks also go to my Greek friends in Manchester, Christos Kotselidis, Paulos Kaimalakis, Kostas Malakasis and Kostas Chrisostomidis for the great time we had all this year and for having someone to share the same mentality. My gratitude also goes to Diomidis Spinellis, my first programming professor, for his teaching and for being the most important reason for my involving with the programming world. Finally I would like to thank Xenia Pakou for her patience all this year and for been the most sweet person to look forward to see at every semester break.

Chapter 1

Introduction

In this thesis, the implementation of an interpreter as an extra module of the Jikes RVM is demonstrated. The addition to the existing Jikes RVM virtual machine was made as part of the work that is been done by the Jamaica research group of the University of Manchester.

1.1 Background

It is widely claimed among the computer world, that one of the main reasons that Java had such a big success was without doubt its capability to run the same piece of software on a plethora of computer systems. The operating system as well as the hardware architecture of each one of these systems can vary but at the same time they can run exactly the same Java code and produce exactly the same output [1]. Java achieves that because, in contrast to other popular languages (C, C++, Pascal, Visual Basic,..), is a bytecode language. This means that instead of producing executable code after the compilation or the interpretation of the source code, it produces an intermediate code that is called bytecode. For the bytecode to be able to run on any computer system and platform and produce the expected results, another piece of software is needed. This software is a Virtual Machine (VM) written specifically for the right combination of hardware architecture and operating system. So we can have a Java VM (JVM) written for the Intel 32 architecture that runs on Win XP, a JVM for

1 One of the few exceptions is the Sun's Pluggable Look And Feel (PLAF) Java libraries that for legal reasons restrict the users of a specific platform to use the GUI representation that belongs to another platform.

the Intel 64 architecture that runs on Linux, or even a JVM for a mobile phone architecture that runs on Symbian OS. Based on the above we can say that JVMs are an inextricable part of Java and are as old as the language itself[2]. As a result of that the development of JVMs has passed through many stages with many different designs and implementations. In later chapters we will see in more detail how the JVMs have evolved through time mainly by focusing on the different internal architecture. But it is not only the differentiation in their internal structure that distinguishes the different JVMs. It is also the legal license and what comes with them and the possibility that the source code of such software is freely available. Here the main two categories are JMVs that are proprietary software, and as such we can't have any view at the source code, and JVMs that their source code is freely distributed (such softwares can have a GPL license, or a FreeBSD license). This differentiation is very important because only in the latter case do we have the opportunity to see the code and change it in any way without being part of a software company that develops a specific JVM. In the academic environment where this thesis was implemented that latter was the only possibility if we wanted to do research on JVMs. The JVM that the Jamaica group uses for research purposes is the Jikes Research Virtual Machine (Jikes RVM) an open source JVM written in Java that is used by many research groups. In the next section we will explain what were the specific goal and objectives behind this thesis and what we wanted to achieve by adding an interpreter to the Jikes RVM.

1.2 Goals and Objectives

The original idea for the addition of an interpreter to the Jikes RVM was brought up by the Jamaica research group. The research group among other things is concentrated with experimenting on the Jikes RVM and tries to find ways that would increase its

2 The are nowadays some companies (eg. Imsys, aJile) that produce processors that can natively run bytecode without the use of a virtual machine.

performance. The Jikes RVM was originally implemented in such a way that it would use a compiler to make the translation between the bytecode and the native code. Much of the effort, before this thesis, for the increase of the performance of the JVM, was put on finding new ways to improve the speed of the compiler that runs inside Jikes. One of the things that was discovered was that much of the time that is needed by a computer system to execute a Java bytecode is spent during its compilation phase. So in order to increase the performance the virtual machine had to be modified in such a way that the cost from that phase would be as low as possible. The solution that was found wasn't an new one. Instead it was an old one that is also a little bit “forgotten” by the implementors of Jikes RVM. It was the interpreter.

The first generation of JVMs had an interpreter while the second had only a compiler. The third generation was a hybrid implementation that used an interpreter and a compiler. Today we can find commercial JVMs (IBM DK, Sun Hot Spot) that have an interpreter and a compiler coexisting in the same environment. But as far as we know there isn't any open source VM that takes advantage of that schema. Now how the interpreter could alter the speed of the VM is something to be analyzed in more detail later in this thesis.

After finding what should be done, several other things remained in order to have the expected results.

- Insert the interpreter inside Jikes in such a way that it wouldn't alter its architecture, but instead it should work without any problem with the rest of the modules.
- Take results from the new Jikes counting its speed for various bytecodes.
- Compare the results with the ones taken before the insertion of the interpreter and if they weren't they expected ones correct its implementation.

1.3 Thesis layout

Below we show how the structure of the current thesis and we give a short abstract for every chapter.

Chapter 2 describes the theoretical background of the bytecode interpreters. At the beginning there is a description on how the bytecode interpreters fit into todays VMs. Then the various interpreter implementations are presented along with the advantages and disadvantages of every approach.

Chapter 3 is about Just In Time (JIT) Compilers. Firstly the JIT compilers are compared to the traditional compilers. The history of the JIT compliler evolution follows next and the chapter closes with a description of the dynamic compilation system, which is the part of the JVM that determines the overall compilation strategy of the system.

Chapter 4 describes that internals of the Jikes RVM. For all the subsystems of Jikes the focus in on the Adaptive Optimization System (AOS) and the way that Jikes organizes its objects. Finally there are some details on the way that Jikes is build.

Chapter 5 gives all the implementation details of the new system. There is presentation of the new architecture of Jikes and the reason behind all the decisions taken on the implementation phase. At the end there is a description on how the new module was inserted into Jikes.

Chapter 6 gives some general conclusions for this thesis, describes the results of the overall attempt to run Jikes with the interpreter enabled and some possible future work that can be done to improve this work.

Chapter 2

Bytecode Interpreters

“An interpreter is a program that executes other programs[3]”. Starting from this general definition of an interpreter it can easily be said what a bytecode interpreter is. So a definition for the bytecode interpreter can be as “a program that executes the bytecode representation of the program”. The bytecode representation is nothing more than the intermediate code that is produced in bytecoded languages such as Java, Smalltalk and Caml. These languages have major engineering advantages over the typical conventional languages (C, Assembly, Lisp, ...) like their capability to run on a variety of systems, their higher level of abstraction, increased debugging capabilities, and runtime type checking. On the other hand their greatest disadvantage is the typically poor performance of the bytecode interpreter compared to the compiled code. In the next sections we will examine in more detail the bytecode interpreter and understand how we can increase its performance.

2.1 VMs and Bytecode interpreters

It is already mentioned that the bytecode is in an indermediate step between the source code and the native code which is executed by the processor and that in order to run any bytecode we need a Virtual Machine. The bytecode itself is normally created by the compilation of the source code, but the time it takes to produce the bytecode almost never affects the execution speed of our program (some optimizations on the bytecode may be able to alter in a small percentage the execution speed). What is

3 Free On-line Dictionary of Computing (http://foldoc.org/)

really important for the performance of every bytecode language is the how fast bytecode is executed as native code. As a result of that we need VMs that are fast in this particular mapping. Unfortunately the first VMs weren't very fast and the reason was that they used only a simple bytecode interpreter as their only means that would achieve the effect of executing bytecode. Nowadays VMs have evolved and they use more sophisticated interpreters accompanied by bytecode compilers. Now we will present the various types of bytecode interpreters as they have evolved through time and justify our decision to implement a particular type inside the Jikes RVM.

2.2 Pure bytecode interpreters

Before looking at the each different category of bytecode interpreters we will examine the basic structure of a bytecode file. The basic structural elements for every bytecode representation are the opcodes (Operation Codes) and the operands. The standard opcodes for a given language is nothing more than an instruction set architecture (ISA) that is interpreted by the VM, while the operands are the data that some of the opcodes must act upon. In Figure 1 we see how the opcodes and the operands look inside a Java bytecode file (*.class* file).

```
  67: ldc    #14; //String
  69: invokevirtual  #9; //Method
java/lang/StringBuffer.append:(Ljava/lang/String;)Ljava/lang/StringBuffer;
  72: iload  6
  74: invokevirtual  #13; //Method java/lang/StringBuffer.append:(I)Ljava/lang/StringBuffer;
  77: invokevirtual  #11; //Method java/lang/StringBuffer.toString:()Ljava/lang/String;
  80: invokevirtual  #12; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
  83: iload  5
  85: i2l
  86: bipush  32
  88: lshl
  89: iload  6
```


Figure 1: Opcodes and opcode operands

The opcodes and the operands that are stored inside a bytecode file are what the interpreter will take as input in order to produce the expected native code. The mechanism that is used for such a translation to take place varies nowadays but in the first VMs the interpreters had basically the following formation (Figure 2).

```
for(;;)  {

   instructionPointer++;
   opcode  = *instructionPointer;

      switch (opcode) {
            case opcode1: {
            /*....*/
            break;
         }
            case opcode2: {
            /*....*/
            break;
         }
            case opcode3: {
            /*....*/
            break;
         }

            //*....*/
      }

 }
```

Figure 2: Switch based interpreter

We can see that the main structure of the interpreter is nothing more than a big infinite *for* loop that encloses a *switch* statement for all the possible language opcodes. Inside each *case* we implement each opcode and at every end of this statement we put a *break* condition so that the control will be passed outside the *switch* and return to the beginning of the *for* loop. The *instructionPointer* is simply a pointer that points to the next opcode to be executed. Every time that we jump from the end to the beginning of the *for* loop the following steps are [Piumatra 98]:

- increment the instructionPointer;
- fetch the next opcode from the memory;
- a redundant range check on the argument of switch;
- fetch the address of the destination case label from a table;
- jump to that address;

and at the end of each opcode

- execute the requested opcode
- jump back to the start of the *for* body to fetch the next opcode;

From the above we can conclude that the pure bytecode interpreters have the big advantage of been easily writable and understandable while at the same time can be highly portable. But their major downside is their slow speed (compared to the other types of bytecode interpreters).

2.3 Direct-threading bytecode interpreters

The second category of interpreters that is used by bytecode languages are the threaded code interpreters. In these interpreters we don't have a big switch statement that will look up and fetch the address of the destination *case* from a table, but instead we will jump directly to the address of the next opcode. An example of such an interpreter can be seen below [Piumatra 98].

```
void * code[] ={...,
            &&opcode_push3,
            &&opcode_push4,
            &&opcode_add,...};

opcode implementions:

/* dispath next instruction */
#define NEXT()  goto **++instructionPointer

void **instructionPointer = code -1;
/* start execution : dispatch first opcode */
NEXT();
/* opcode implementation ...*/
opcode_push3:
*++stackPointer = 3;
NEXT();
opcode_push4:
*++stackPointer = 4;
NEXT();
opcode_push4:
--stackPointer ;
*stackPointer +=stackPointer[1];
NEXT();
/*.....*/
```

Figure 3: Direct-threading interpreter

Here the steps we have to make in order to execute the code are:

- increment the instuctionPointer
- fetch the next opcode address from the memory;
- jump to that address
- execute the opcode

From the above it is shown that there is a reduction to the half of steps that are needed during the execution of a certain bytecode if a threaded interpreter is used, which means that the speed is increased significantly. There has been a reduction in running expensive instructions like the one jump instruction and the one memory reference. Unfortunately this gain comes with a cost. The cost has to do with the portability of the code and its simpleness. Here we must notice that what this

implementation really achieves compared to the pure bytecode interpreter, is a way of elimination the extra steps that the switch statement imposes. Moreover the recent GCC 4.1 compiler can use the *-fthread-jumps* argument in order to achieve this optimization and transform any switch statement to an optimized “threaded” interpreter.

2.4 Inline-threading bytecode interpreters

An improved version of the direct-threading is inline-threading. The basic idea here is that before we run any bytecode file we can find and replace basic blocks of opcodes with their corresponding implementation. This will increase the file size but all the fetches and jumps to the memory will disappear and the VM will simply execute the bytecode. The disadvantage of this type of implementation is the increased size of the bytecode as well as the extra time we have to spend in order to transform the bytecode file. The next figure shows a part of an inline threaded compiler implemented in C [Gagnon, 2001].

(a) Instruction Implementations

```
ICONST_1_START: *sp++ = 1;
ICONST_1_END: goto **(pc++);
INEG_START: sp[-1] = -sp[-1];
INEG_END: goto **(pc++);
DISPATCH_START: goto **(pc++);
DISPATCH_END: ;
```

(c) Inlined Instruction Sequence

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

(b) Sequence Computation

```
/* Implement the sequence ICONST_1 INEG */
size_t iconst_size = (&&ICONST_1_END - &&ICONST_1_START);
size_t ineg_size = (&&INEG_END - &&INEG_START);
size_t dispatch_size = (&&DISPATCH_END - &&DISPATCH_START);
void *buf = malloc(iconst_size + ineg_size + dispatch_size);
void *current = buf;
memcpy(current, &&ICONST_START, iconst_size); current += iconst_size;
memcpy(current, &&INEG_START, ineg_size); current += ineg_size;
memcpy(current, &&DISPATCH_START, dispatch_size);
...
/* Now, it is possible to execute the sequence using: */
goto **buf;
```

Figure 4: Inline threading interpreter

Writing an inline-threading interpreter in Java (like we wanted in this thesis since Jikes RVM is written in Java) in a way similar to the above example is difficult, are source of problems is there is no way to handle pointers like in C. However writing an interpreter in Java using the threading technique can be achieved but the implementation as well as the logic differs. In order to achieve inlining in Java we could write the following code [Ian Rogers]:

```
class Interpreter {
    static Interpreter interpreters[NUM_INSTRUCTIONS];
    static {
        interpreters[NOP] = new Interpreter() {
          Interpreter execute() {
                IP +=4;
                return interpreters [memory [IP]];
                }
          };
        interpreters[ADD] = new Interpreter() {
          Interpreter execute() {
                IP +=4;
                // do add...
                return interpreters [memory [IP]];
                }
          };
          //..........
      }
      //.............
    static void execute(Address IP) {
        Interpreter currentInterpreter = interpreters[memory[IP]];
        while(true) {
          currentInterpreter = currentInterpreter.execute();
        }
      }
}
```

Figure 5: Java threaded interpreter

The basic logic we use for the Java implementation is that certain sequences of bytecode instructions are more common that other (e.g. a compare is usually before a branch). The above threaded interpreter is better than the pure that has a *switch* statement since the branch predictor can predict where the next bytecode will be. The compilation system can “predict” in a right way the most of times so we can have a

speed up in the execution time. The acceleration in the speed that we will have after using this kind of implementation inside a JVM is about 10% compared to the switch based.

Chapter 3

Just-In-Time (JIT) Compilers

Using an interpreter was the first and most simple way the first JVMs used in order to execute bytecode. We have seen how interpreters evolved and the basic characteristics that each of them has. Independently of the interpreter that is implemented inside the JVM we are restricted on how fast the bytecode is executed. This is because the interpreters are based on the idea that we execute each bytecode ignoring the fact that the same bytecode could have been run previously or that it may be executed in the future. So any acceleration is achieved if only if we "predict" the bytecodes that will follow or if we improve our implementation[Piumarta and Riccardi, 1998]. A different approach for executing bytecode is the use of JIT compilers. In the next sections we will have a deeper look at this JVM component and examine how the JVM/JIT interaction can be achieved.

3.1 JIT compilers vs traditional compilers

Being part of a JVM a JIT compiler has many characteristics that are similar to a traditional one but also is unique in some things. Like the compilers that were developed along with the first computer languages a JIT compiler eventually translates some code to machine code. The difference this time is the source code, which is not a common human written code but instead a machine produced bytecode. This is very important since a JIT compiler will have a much simpler front-end and back-end. There is no need for source language error reporting (a task done by the compiler that produced the bytecode) and also there is no need for the generation of object files or

relocatable code. The JIT compilers don't compile the entire bytecode file, but only what is needed by the runtime environment methods. As a result a JIT compiler may never have a full view of the entire program. The final thing that distinguishes this kind of compiler from tradition ones, is their significance to the execution time of a bytecode program. The compilation time and space consumption are very important since the JIT compiler will execute the needed methods on the fly and a good performance is based mainly in this JVM component. A visualization of the structure of a JIT compiler is shown in the next figure.

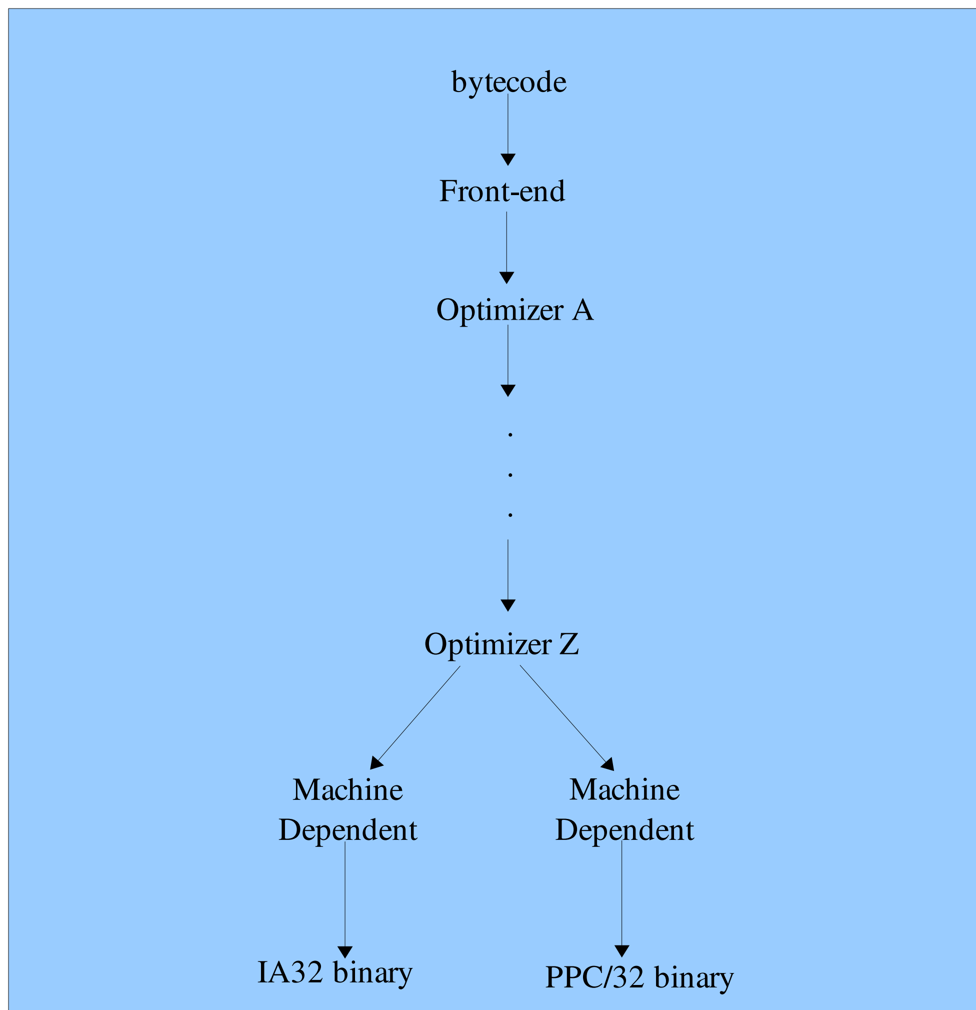

Figure 6: Structure of a JIT compiler

3.2 JIT compiler evolution

The first JIT compilers that were implemented inside a JVM had a very simple strategy on how to operate. They used a single level execution model and compiled every method with a standard set of optimizations. This means they didn't use any kind of dynamic execution of methods and they compiled the whole bytecode file regardless of whether some of it was never used. As one can easily assume the outcome of this approach wasn't very satisfactory since the level of optimization was trivial and the compiled code size was quite big. More details on the research that was done during the first years of JIT compilers can be found at [Adl-Tabatabai et al.1998], and Cacao [Krall 1998].

The next step was to use dynamic compilation, instead of static, and more sophisticated optimizations. Now the compiler wouldn't execute every method but only the ones that really were needed during its runtime and at the same time would impose more optimizations. The side effect that was noticed was the compilation overhead which was increased noticeably. This has to do with delays in the application startup and in the creation of code that had excessive size.

The workaround for the problems that were imposed by the previous approaches was a dynamic compilation system that would define a better dynamic compilation framework and also adapt a strategy that could use multiple levels of optimizations. A two-level execution model consisting of one interpreter and one compiler or two compilers with different optimization levels of compilation would replace the previous one. The results that were achived with this type of JVMs were better than before since the compilation overhead dropped.

Even though there was some progress, the results were not as good as expected. The problem this time would be how to manage the equilibrium between the optimization effectiveness and the compilation overhead caused by the increasing gap of the trade-off level between the two execution modes. The solution to that was the creation of a framework for online profile collection accompanied by a feedback

system. Both of them are inserted inside the dynamic compilation system and communicate with the rest of the modules. This generation of JVMs also uses multiple execution modes (combination of interpreter or baseline compiler and optimizing compiler), but moreover can track the mostly frequent methods (hot spots)[4]. The information gathering is done by the profiler framework and depending on how often a method is called or how important a method is, different optimization levels are used. As a result the hot spot methods can be recompiled by the optimizing compiler while the less significant ones can be run by the interpreter or the baseline compiler. The most advanced JVMs in our days use this approach as the basic architecture of their dynamic compilation system even though some differences can still exist. In the next section we take a deeper look inside this system, since it is the heart of any JIT compiler and the one that plays the most important role in the execution speed of the bytecode.

3.3 Dynamic compilation system

The dynamic compilation system is the architecture that most JIT compilers use nowadays. The basic targets that a good system should aim at are:

- to be as fast as possible
- to have a not very complicated framework so that various optimizations can be added without resulting in a compilation overhead increase

The details of such system may vary depending on the JVM, but in general there are three important characteristics that define the classification of the dynamic compilation system.

4 Statistic analysis of programs has shown that most of the programs follow the 90/10 rule, which means that for 90% of the execution time of a program 10% of the code is used [Hennesey & Patterson, Quantitative Computer Architecture].

The first characteristic has to do with the lack or not of an interpreter along with one or more compilers. The interpreter is used in the first moments of the execution of a bytecode where we don't apply any optimization to any method. The same role can be played by a baseline compiler that is simpler than the optimizing one and doesn't have a big compilation overheard. Another important thing is that the use of a baseline compiler instead of an interpreter would mean a significantly bigger memory footprint. On the other hand the integration of an interpreter and a compiler would increase the complexity of the system and significantly more effort has to be taken during implementation of such system in order to get the wanted results.

The second characteristic has to do with how the compilation system monitors the behavior of a bytecode program and promotes a method from a lower optimization level to a higher. Depending on the JVM we could have a variety of strategies. In one of them we could have a mechanism system which uses a counter associated with each method. The system can insert code to test the value of the counters. If the counter reaches a specific value then the code is "upgraded" to a higher optimization level and is recompiled by the optimizing compiler. An approach like this one is used by the Intel Open Runtime Platform (ORP) [Cierniak et al. 2002]. A different strategy could be the one that is used by the IBM DK for Java [Suganuma et al. 2005]. In this case a sampling-based profiler is used which collects information about the programs' thread execution. The profiler keeps track which program threads use the CPU most time and then which methods inside these threads are currently executing. Based on the results the profiler creates a list of all the "hot" methods sorted by the hotness counter (a counter associated with every method indicating how "hot" each method is). After that the profiler, which runs all the time that the program is been executed, sends the group of the "hot" methods to the recompilation controller for recompilation at a higher optimization level.

The third characteristic has to do with what kind of profile information is collected by the dynamic compilation system in order to be passed to the higher optimization levels. The instrumenting profiler used by the IBM DK for Java

[Suganuma et al. 2005], is used to collect data from the "hot" methods based on an instrumentation plan from the recompilation controller. Later this data will be used by the dynamic compiler to increase the optimization of the method. The way the profiler takes out the desired data from a method is by inserting dynamic code inside the compiled code and rewriting the entry code of the target. After collecting the desired data the generated instrumentation code automatically uninstalls itself from the target code.

3.4 JIT/JVM Interaction

The interaction of JIT compilers and JVMs is very important since the way they communicate defines the implementation details of both programs. The runtime services often require JIT support (memory management, exception delivery and symbolic debugging) while at the same time the JITed code requires extensive runtime support (type checking, memory allocation, use of hardware traps and signal handlers) [Hind, ACACES 2006]. The collaboration of the two pieces of software will enable optimization opportunities like effective inline code sequences and customizable dynamic type checking code sequence. On the other hand a more general implementation of a JIT compiler that will fit into several JVMs will make the system more insensitive to changes that can be applied to any part of the JVM or the JIT. The second approach of a more independent JIT is much more difficult to implement but not impossible if very careful design is made.

Chapter 4

The Jikes Research Virtual Machine (RVM)

Undoubtedly the JIT compiler is among the most important modules that a modern JVM uses for the execution of bytecode. Other modules play also a significant role, like the class loader and the memory manager. But in order to have a better understanding on the internal structure of a JVM the best way is to examine a real one. This is also the scope of this chapter that takes an inside look at the internals of the Jikes RVM, the open source JVM that was used in this thesis. Such a task is also an inevitable process for somebody who wants to alter or add an extra feature to the JVM. In our case the extra element was the interpreter.

4.1 The Jikes RVM structure

The Jikes RVM is an open source Java Virtual Machine written in the Java programming language. The project was originally named Jalapeño and was initially developed at the IBM® T.J. Watson Research Center [http://jikesrvm.sourceforge.net/]. In 2001 the project went open source and this was the right opportunity for the community to explore its internals and contribute to its evolution. The basic architecture of the JVM is represented in the following frame.

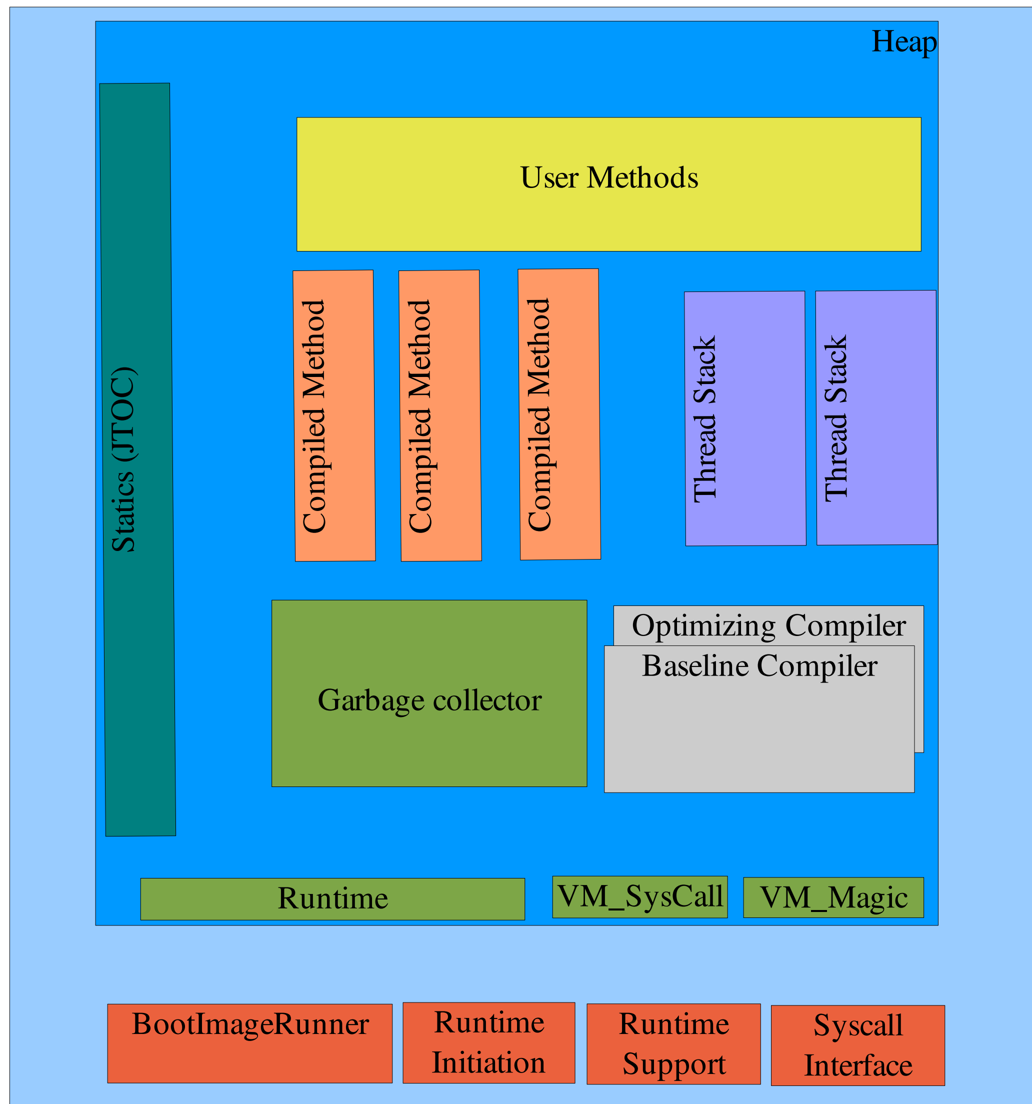

Figure 7: Jikes' Basic Architecture

Jikes is constructed of four basic components:

- **Core runtime**, which includes, among other things, the thread scheduler, the class loader, the library support and the verifier. The core runtime is generally responsible for managing all the underlying data structures required to execute

applications and interface with libraries.

- **Compilers**, that include the baseline compiler, the optimizing compiler and the JNI (Java Native Interface) which is one of the few parts of Jikes not written in Java[5]. This component is used for generating executable code from bytecodes.
- **Memory managers**, that are responsible for the allocation and collection of objects during the execution of an application.
- **AOS (Adaptive Optimization System)**, that has as its task the monitoring of the execution of the bytecodes and dynamically uses the different kinds of compiler in order to achieve the best execution performance.

The first module that runs when Jikes is loaded into memory is the Boot Image Writer. This is a Java program that runs under an existing JVM. Then a list of "core" Jikes classes is loaded and the *init()* method is invoked for these classes. The copies of the loaded classes and the objects created by the JVM environment (reflection is used), are transformed into a byte array and copied to the boot image. From this point on the Boot Image Runner is called (it allocates memory for the virtual machine image, reads the image from disk into memory, and branches to the image startup code) and the boot image eventually runs. One important thing that should be mentioned here is that everything in Jikes is an object (this also comes from the fact that Jikes is written in Java). But not all objects are "genuine" Java objects instead some are Jikes objects. In this category we can include the execution stacks and the Java Table Of Contents (JTOC). The execution stacks are declared as *int[]* and it is the compiler's job to identify locations within frames that hold references for later use by the garbage collectors. The JTOC (which for our case is called Jikes Table Of Contents) contains references to all the Jikes Objects as well as static fields of reference. The next figure shows how JTOC is organized [Alpern et al. 2000].

---

5 The other parts of Jikes also not written in Java are the Boot Image Runner and two signal handlers (one that captures hardware traps and trap instructions and one that passes timer interrupts to the running Jikes system).

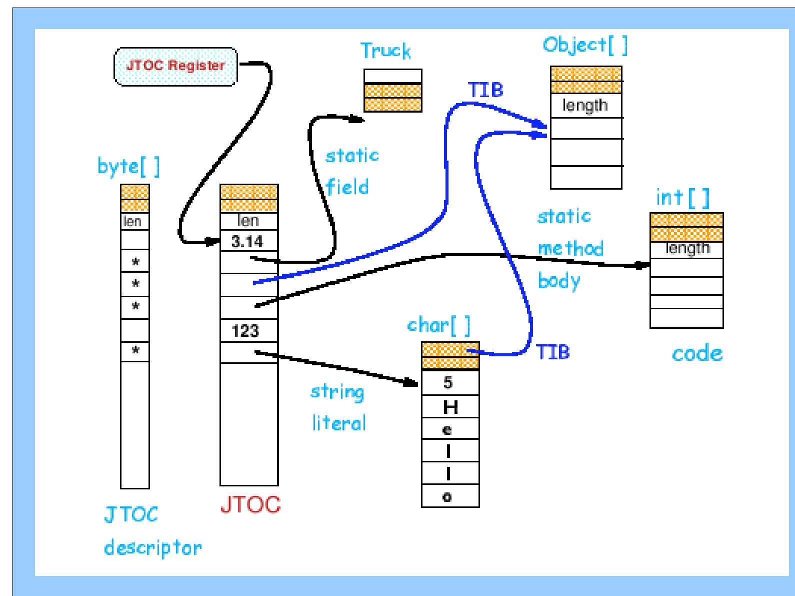

Figure 8: JTOC organization

The TIB (Type Information Block) shown above is an array of Java object references. Its first component describes the object's class (including its superclass, the interfaces it implements, offsets of any object reference fields, etc.). The other components are compiled method bodies (executable code) for the virtual methods (which is another building block of the Jikes object) of the class. As a result the TIB serves as Jikes' virtual method table. The TIB along with a word called status word comprise the object's header. The last pieces that form a Jikes object are the method invocation stacks which are created for each method invocation of that object (more details about the Jike's object are described in Alpern et al. 2000). All the above information is necessary to understand the internals of Jikes since such knowledge is used for the construction of the interpreter.

4.2 The Java Stack

Another very important aspect of every JVM is the way that the Java stack is organized. This has to do with the way each Java bytecode is executed but in the case of Jikes its importance is bigger. Because Jikes itself is written in Java the understanding of the Java stack is be very important for the better understanding of the processes that take place inside the JVM.

Every time a Java method is called, the JVM creates a new Java stack frame for that thread. The Java stack stores a thread's state in discrete frames (Figure 9).

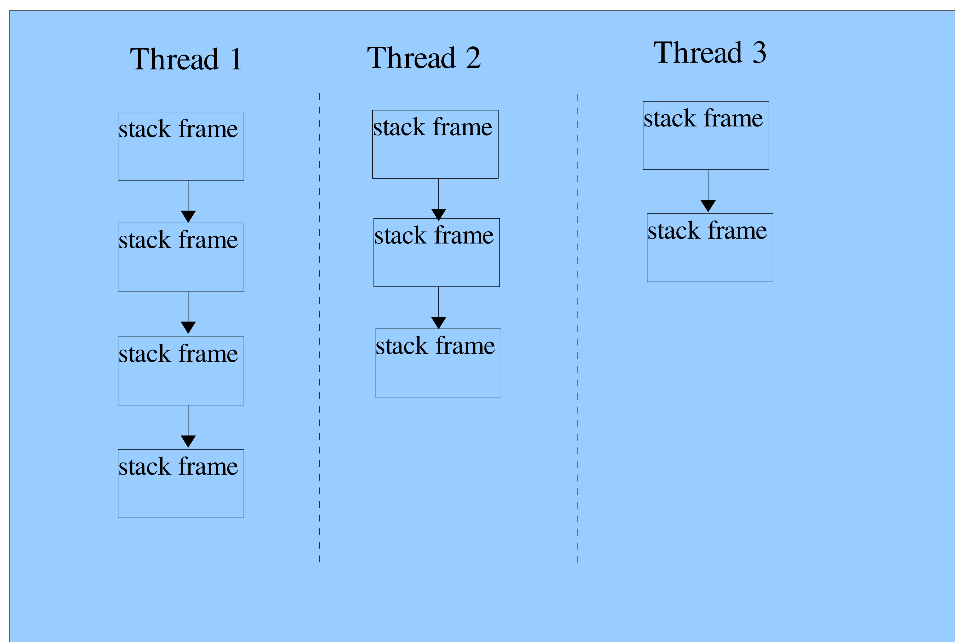

Figure 9: Stack & Stack Frames [Inside the Java 2 Virtual Machine, Bill Venners]

The method that is currently running is called the thread's current method and the stack frame for the current method is the called the current stack frame. Every stack frame has three parts: the local variables, the operand stack and the frame data. The local variable part is an array that stores all the method's local variables (i*nt, float, char, byte, short, double, long, references, returnAddress*). The local variable's array is

accessed via array indices. Similar to that is the operand stack which is also an array but this time the access to the array's values is done only by pushing and popping. The operand stack is the basic data structure that is used for the execution of the opcodes (the instruction set of the method's bytecode stream). What is meant by the last one is that all the Java bytecodes are composed by a well defined set of instruction. These instructions are executed by the JIT compilers or the interpreter of every JVM. Some of Java's opcodes as they are defined in the language specification are: *istore*, *dsub, areturn, ...*. The opcodes and the data that accompanies them, are pushed and popped from the operand stack. Below is an example of the addition of two local variables.

```
iload_0      //push the int inlocal variable 0
iload_1      //push the int inlocal variable 1
iadd         //pop two ints, add them, push result
istore_2     // pop int, store into local variable 2
```

Figure 10: Adding two variables

The last part of the stack frame is the frame data which is used for support of the constant pool resolution (a place where all the constants are stored), the normal method return, and the exception dispatch.

4.3 The AOS system

Undoubtedly the part of the Jikes RVM that would be the most important for this thesis would be the AOS subsystem. This is because is responsible for the compilation strategy which is the the major speed factor of the JVM. The insertion of the interpreter would mean an alteration to the AOS system. The “pre-interpreter” Jikes AOS system is shown below [Jikes Tutorial 2001, Hind & Attanasio].

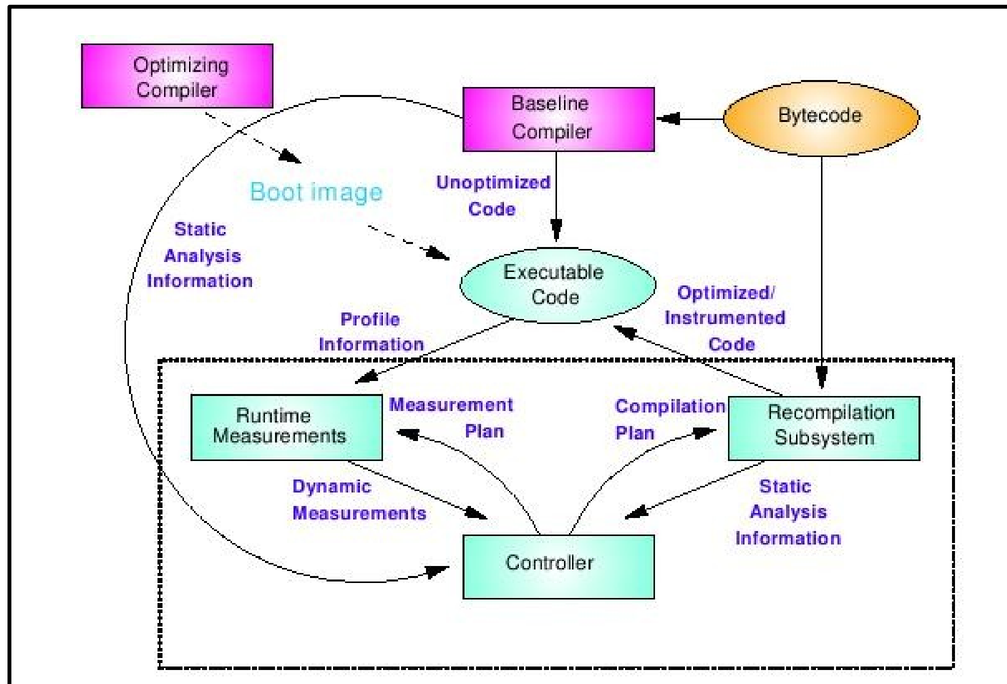

Figure 11: The AOS system

In this system the bytecode is initially compiled by the baseline compiler. Jikes doesn't have any information about hot methods and as a result all methods are treated the same way. As time passes and the JVM executes code, Jikes' timer-based sampling mechanism gathers information about the more often used methods and methods that have greater significance for the execution speed. The sampling profiler collects this information at yield points (method prologues and loop back edges) throughout the entire program execution. This has as a result those methods which are declared hot are passed to the optimizing compiler for further optimizations. Jikes has three different levels that indicate how hot a method is. The profiling system that Jikes has is very flexible and can adaptively adjust to changes in an application's dynamic behavior. An extra optimization of the AOS system has to do with method inlining. Here also a profiler, that periodically takes statistical samples , is used to decide which methods should be inlined. The details of the AOS with the adaptive inlining is shown delow [Jikes Tutorial 2001, Hind & Attanasio].

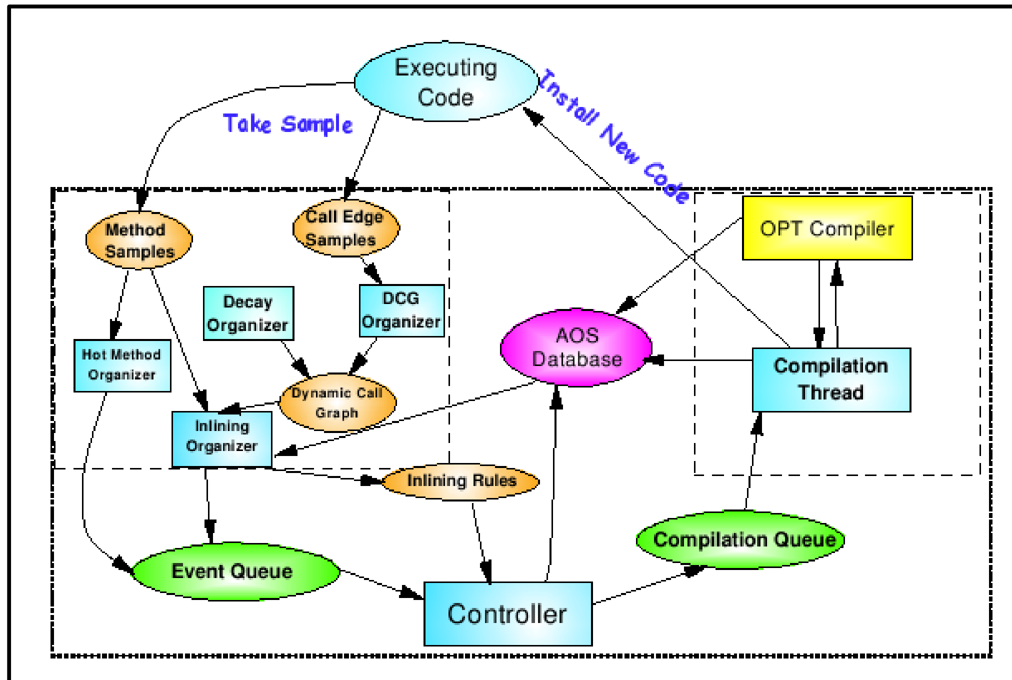

Figure 12: The AOS with adaptive inlining

4.4 Building Jikes

For someone to be able to run Jikes what he needs to do is to execute a collection of Unix shell scripts that contain configuration parameters. One of the most important parameters that the user must define is the level of optimization that Jikes will use. This is defined through the *jconfigure* script. There can be three different buildings of Jikes. The first one is the *prototype* build that constructs the JVM with the baseline compiler as the only compiler to be used for the execution of the bytecodes. The second build is the *production* that has the baseline and the optimizing compiler and the debugging information turned off. The last one is the *development* that is exactly the same as the production but has the debugging information turned on. The way that Jikes is built has an obvious speed impact with the *production* and *development* being

significantly faster than the *prototype*. The major advantage of the *prototype* is the build time that is much faster than the other two (almost 1 minute spent for the *prototype* build on a 2.0Ghz Intel Centrino processor compared to 20 minutes for the *production* and *development*).

Chapter 5

Implementation Details

After the presentation of the theoretical background and the Jikes RVM environment, more details are given on how the addition to the interpreter was achieved, what strategy was followed, what were the problems and what was finally achieved.

5.1 The new approach for Jikes RVM

As already mentioned Jikes follows the compile-only approach. The baseline and the optimizing compilers are used to run methods with varying significance. Both of them cooperate with the rest of the Adaptive Optimization system to produce the best results. This compiler-only use has a significant disadvantage that has to do with the compilation overhead. The time that we need to read a method and compile it, is a slow process compared to interpretation. The interpreter is faster than any complier at the first moments of any bytecobe execution since the way it is implemented allows the bytecode to be executed almost immediately. But as time passes and the compiler has finished the bytecode reading and has applied the various optimizations, the execution becomes faster using the compiler module. While the interpreter for every invocation of the same method will do the same procedure as the first time the method was called, the compiler will use the already existing code with all the optimizations applied. So it becomes clear that for the first execution moments the interpreter is faster, while the compiler becomes the best choice only after it has finished the reading and the analysis of the needed bytecode. In order to achieve the best possible performance, an

implementation that combines these two modules in the best possible way should be written. As a result of the above, it was decided that an interpreter should be written as an Jikes RVM extra module that would run before any compiler only for a certain amount of time. The next figure shows exactly that.

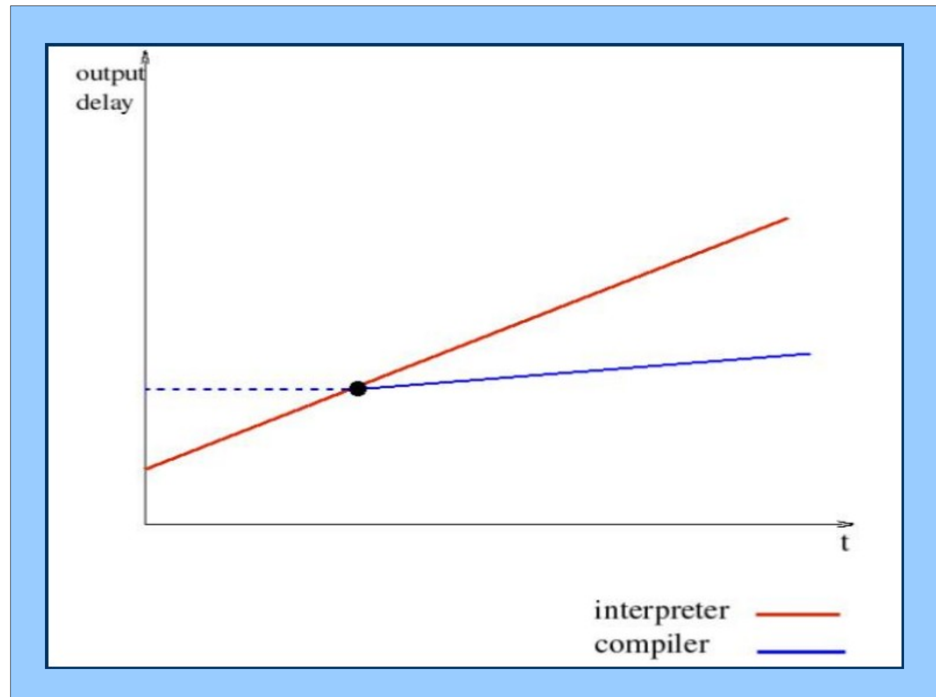

Figure 13: Compilation vs Interpretation

The important thing in the above graph is to find the crossing point of the two lines. This point indicates for how long the interpreter will be used and at the same time at which moment the baseline compiler will take over. The dashed line shows the period that the compiler doesn't produce any output and fulfills all the needed tasks in order to produce some output. Another important thing here is that this optimization scheme will only work in programs that aren't too small and have methods or structures within methods (e.g. *for* and *while* loops) called more than one or two times. In the opposite case the compiler shouldn't be used at all. But whatever the case the interpreter module should be planned carefully inside the rest of the AOS system and cooperate

harmonically with the rest of the JVM. Structurally the new system with the interpreter added will look like the next figure:

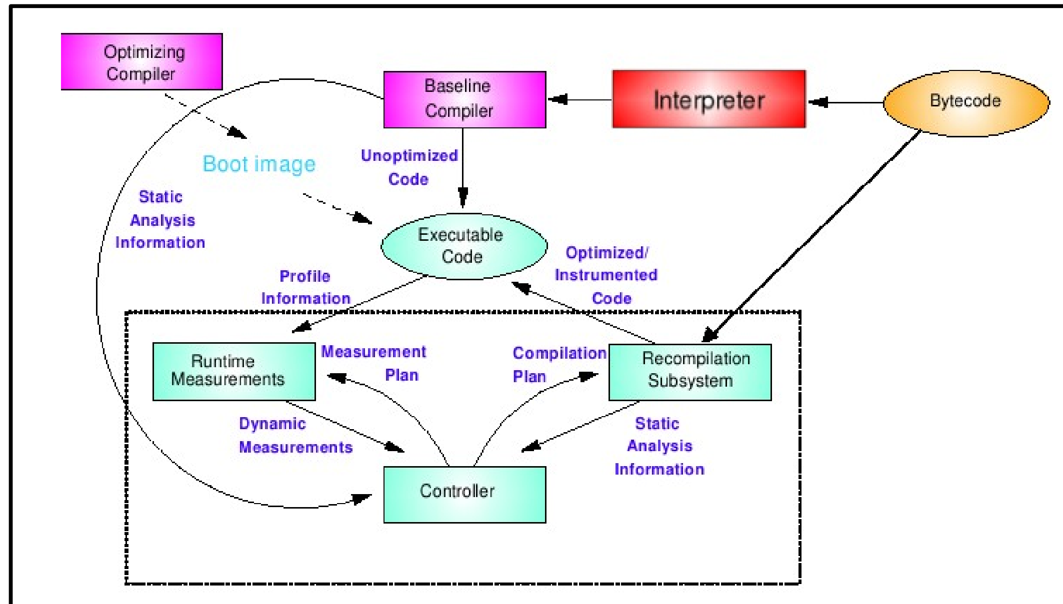

Figure 14: The new AOS system

It is seen from the above that the interpreter is put before the baseline compiler and it is the first module that starts to execute bytecode. Now the criterion on which the decision to move to the baseline compiler is based is something that can only be decided after testing and experimenting. This means that the choice to jump to the next module can be made after one, two, three or even more identical method invocations. There isn't an obvious way to say beforehand what is the best number of times that the interpreter has to be called before the baseline compiler comes in. Since many suggestions can be made on that, what was decided to be done in this implementation was to extend the arguments that Jikes RVM takes by adding a new one (`-X :interpret=[num]`) in which the person that runs a *.class* file can explicitly give the number of times that a method needs to be invoked before the execution control is passed to the baseline compiler. In order to achieve such a modification

`VM_CommandLineArgs.java` file needed to be changed accordingly. IBM DK, that also uses an interpreter and a compiler inside its dynamic compilation system, uses a slightly different approach [Suganuma et al 01, 05]. In this JVM also all methods are initially executed by an interpreter (Mixed Mode Interpreter) but the difference is that it uses an internal counter for accumulating both method invocation frequencies and loop iterations. The counter is decremented whenever the method is invoked or loops within the method are iterated and when the counter reaches zero the first JIT compilation is triggered for that method. But the important thing here is no information is given on how the initial value of the counter is given. Moreover IBM BK passes the runtime information gathered during the interpreter run to the JIT compilers so they can use it for their optimizations. Comparing this approach with the one that was followed for Jikes RVM what can be said is that in the latter case a simpler architecture was followed (no loop iteration counter, no runtime information passed to JIT compilers) that probably wouldn't give the best possible results but on the other hand it could give the user the possibility to explicitly define the counter number that defines the number of times the interpreter runs for each method before control is passed to the JIT compilers.

5.2 The VM_Interpreter

After defining the basic architecture of the “interpreting” Jikes, more details are given on the actual implementation of the interpreter. Since the interpreter was a completely new module for Jikes, a new Java file having the main body of the interpreter had to be created. This file is called `VM_Interpreter` and is part of the `com.ibm.JikesRVM` package. The main body of the `VM_Interpreter` is basically a big *switch* statement for all the possible Java opcodes [Inside the Java Virtual Mashine, Bill Venners]. The Java opcodes are the fundamental blocks of every Java *.class* file and with the static variables and also some other data form the source

bytecode file that will be created. Java has 228 possible opcodes and for each opcode a different *case* branch was implemented. At this point what must be explained is the reason why the pure bytecode implementation (see section 2.2) was chosen instead of a faster direct-threading(see section 2.3) or inline-threading interpreter (see section 2.4). The main reason for this choice was the implementation language. As mentioned there is no way to implement a direct-threading bytecode interpreter using a safe language such as Java. An inline- threaded interpreter can be written though but it needs a “prediction” scheme for the next opcode that will be interpreted. Since such an approach is significantly more complex than the “switched” interpreter, the decision that was taken was in favor of the simpler and faster to be implemented pure bytecode interpreter.

After deciding the interpreter's basic architecture, the thing that was needed to be determined was how the operand stack would be implemented and the how the local variables would be handled. This decision was very important since the operand stack and the structure that would handle the local variables are the major data structures of the interpreter. Since all the opcodes that would be executed would use them, the way they are implemented plays a significant role on how the rest of the interpreter would be written and an effective implementation would mean increased speed. For this implementation the decision that was finally taken was based on the following logic. Because Java can have two different kinds of variables (primitive types and objects), probably the best approach would be to have two types of operands stacks and two types of local variable handlers. As a result the *int operandStack[]* and *int localVariables[]* were created for the primitive types and the *Object operandStackObjects[]* and *Object localVariablesObjects[]* for the objects respectively. It can be observed that the *operandStack[]* and *localVariables[]* are used for all the different kind of primitive types (*int, float, char, byte, short, double, long*). The problem here is that the bit size for different variable types varies (e.g *ints* are 32 bits long and *doubles* are 64 bits long). While the conversion of smaller primitive types to *int* is a very easy process in Java (a simple cast is needed) the conversion of

doubles and *longs* is not so trivial. The workaround for this problem was to divide the 64 bit long variables to two 32 bit chunks and store them in two consecutive places in the *int* array. Similarly when there was a need to use a 64 bit variable the two 32 bit parts were concatenated to give the original number. A slightly different approach to this scheme would be to have as many operand stacks as the number of primitive types. In this case there would be operand stacks like a *double operandStack[]* and a *float operandStack[]*. In this case some time for the conversion of the various types would be saved but more memory would be used. Which of the two approaches is the better is something that has to tested in order to be determined. A future implementation of the interpreter with the multiple operand stacks approach would determine the better solution. A very important thing that should be mentioned here is the way that we reference the objects or the primitive types that are stored inside the two stacks. The way that we achieve that is by the use of an integer called *stackPtr*. This stack pointer can be incremented or decremented by code inside the various *cases* of the *which* statement but the key thing here is that *stackPtr* is a shared variable, used in common by both operand stacks. In the next figure we give an example of how the above data structures and variables are used in the interpreter for a random opcode.

```
/*...*/
case JBC_iadd: {
    stackPtr--;
    int value2 = operandStack[stackPtr];
    stackPtr--;
    int value1 = operandStack[stackPtr];

    int result = value1 + value2;

    operandStack[stackPtr] = result;
    stackPtr++;
    break;
      }
/*...*/
```

Figure 15 : Example of one of the interpreter's case branches

It was mentioned before that Java uses a method based organization for the execution of its code. As a result of that the interpreter also used a “per method” approach and a new instance of the interpreter is created for a every method invocation. Extra consideration has to be taken for static methods because they are methods that don't have any local variables and have to be treated in a different way from the rest. For non static methods a few other Java methods were written that were used in various opcode *cases* and had to do with the reading of a method's arguments (`private static int readArgs(Object args[], ...)`) , the insertion of the result of a method back on the stack (`private static int putResultOnStack(int operandStack[],...)` and the conversion betwe,en the different Java primitive types (`private static long makeLong(int i1, int i2)` and `private static double makeDouble(int i1, int i2)`).

5.3 Inserting the VM_Interpreter into Jikes

Writing the VM_Interpreter was only half of the job that needed to be handled so that Jikes would have a new working module. The other half is how this module is inserted inside an already working JVM and which files of the original Jikes have to be altered. The most obvious alteration would have to be inside the VM_Runtime module and more specifically inside the VM_DynamicLinker file. The `static void lazyMethodInvoker()` method was changed in a way that after a method invocation the VM_Interpreter would be called. But the way that this method was changed wasn't a very simple one since some VM_Magic code had to be written. Basically the implementation would be a special baseline IA32 compiler “hack” that would allow a “magical” return value from a method even though the stack thinks this method returns void and takes no parameters. The next figure shows exactly which objects are called with every method invocation.

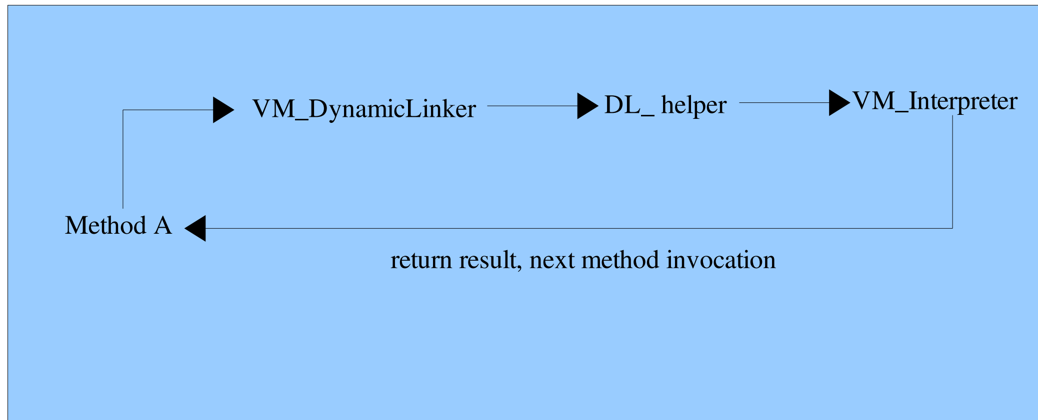

Figure 13: VM_Interpreter interaction with other Jikes' objects

Apart from the VM_DynamicLinker a few other files were changed so that that various opcodes would be executed properly. Below there is a list with all the files that were modified.

File	**Comment**
jconfigure	Include the interpreter in Jikes' built
VM_NormalMethod.java	Add some additional properties to VM_NormalMethod
OPT_GenerateMagic.java	Add some more “magic” for the optimizing compiler
VM_DynamicLinker	Link the interpreter with the rest system
VM_Entrypoints.java	Add some code so that the Jikes RVM could use the interpreter module
VM_Magic.java	Add more “magic” methods
VM_MagicNames.java	Add some magic variables used in the *return* on the methods

File	Comment
VM_Reflection.java	Modify the reflection system of Jikes
VM_CommandLineArgs.java	Add the X:-interpret argument to Jikes

Chapter 6

Conclusions

After giving the implementation details it is time for the conclusions for the addition of the bytecode interpreter to the Jikes RVM presented. Obviously the heart of the implementation is the VM_Interpreter.java file. Inside it is the implementation of all the possible opcodes. In order to get any results and to check the effect of the addition to the overall performance, we had to thoroughly examine the execution of different kind of methods that would use all the possible Java opcodes. For this reason various test files that checked all the possible aspects of the Java language were used. This task was extremely difficult since the testing of the execution for a numerous of *.class* files was very time consuming and the detection and correction of the various bugs proved to be a really hard task. The outcome of this process was the following:

All the possible opcodes were tested and for the majority of the *.class* files the interpreter produced the expected results.

Unfortunately some bugs still remain that prevent the evaluation of the overall performance of new version of Jikes. From a group of available JVM benchmarks (*mtrt, jess, comp,db, mpeg, jack, javac, jcc, G.M,....*) only one, the JavaLinPack could be executed using the interpreter. Moreover this benchmark could only be run using the *prototype* built of Jikes (the slowest build since it uses only the baseline compiler). The

result for that was that the “interpreted” Jikes run the benchmark almost 15 times slower than the non interpreter Jikes. Better performance is expected if the interpreter ware build using a *production* configuration. Sadly this wasn't possible.

The main reason that prevented the other benchmark bytecodes to run had to do with the complexity of debugging the interpreter inside the Jikes' RVM environment. In many cases significant code had to be altered in other files so that the interpreter could cooperate harmonically with the rest of Jikes' modules. The debugging process has reached a point at which other Jikes' modules have to be examined for bugs since there are some indications that some implementation errors, that prevent the interpreter running correctly for all the bytecodes, may be located in other files (there problem is probably located on the part of Jikes that deals with reflection). The bad thing is that probably an extra one man month or more for code debugging is required. The good thing is that even at this stage the interpreter can have big GC advantages, as well as being a useful research tool, to other Jikes RVM users.

FUTURE WORK

Since the implementation of the interpreter is not completely finished a list of things can be done in the future to enhance the behavior of the interpreter and as a result the behavior of Jikes RVM.

- The most important thing is to fix the remaining bugs so that proper measurements can be taken.
- By adding an extra operand stack for the 64 bit long variables (*long, doubles*) there could be an increase of the interpreter's speed. This could save processing time since no conversion between 32 and 64 bit long variables would be needed. On the other hand the memory footprint would be increased. The best solution is to be could be determined through benchmark.
- Writing a different implementation of the the interpreter by using a threaded approach instead of the *switch – case* could lead to a probable 10% increase in the execution speed.

EPILOGUE

This thesis is probably the most interesting and challenging programming task that the author has ever taken. The road to the completion of the implementation of such a demanding project was a long and tough one but it was truly unforgettable for the experience that was gained and the lessons that were learned. Even though the project isn't 100% finished the author would feel great satisfaction if he has helped in his way in the research that is done in the area of JVMs.

Bibliography

[1] TOSHIO SUGANUMA, TOSHIAKI YASUE, MOTOHIRO KAWAHITO, HIDEAKI KOMATSU, and TOSHIO NAKATANI IBM Tokyo Research Laboratory: Design and Evaluation of Dynamic Optimizations for a Java Just-In-Time Compiler. In *ACM Transactions on Programming Languages and Systems, Vol. 27, No. 4, July 2005, Pages 732–785.*

[2] MICHAEL HIND: Dynamic Compilation and Adaptive Optimization in Virtual Machines. In *ACACES 2006.*

[3] MICHAEL G. BURKE, JONG-DEOK CHOI, STEPHEN FINK, DAVID GROVE, MICHAEL HIND, VIVEK SARKAR, MAURICIO J. SERRANO V.C. SREEDHAR HARINI SRINIVASAN, JOHN WHALEY IBM Thomas J. Watson Research Center: The Jalapeno Dynamic Optimizing Compiler for Java. *In 1999 ACM Java Grande Conference, San Francisco, California, June 12-14, 1999.*

[4] KAZUAKI ISHIZAKI, MOTOHIRO KAWAHITO, TOSHIAKI YASUE, MIKIO TAKEUCHI, TAKESHI OGASAWARA, TOSHIO SUGANUMA, TAMIYA ONODERA, HIDEAKI KOMATSU, TOSHIO NAKATANI IBM Tokyo Research Laboratory: Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler. *In JAVA'99 San Francisco California USA.*

[5] YUNHE SHI, DAVID GREGG, ANDREW BEATTY, M. ANTON ERTL: Virtual Machine Showdown: Stacks Versus Registers. *In VEE'05 June 11-12, 2005, Chicago Illinois, USA.*

[6] IAN PIUMATRA, FABIO RICCARDI: Optimizing direct threaded code by selective inlining: *ACM 1998.*

[7] B. ALPERN, S. AUGART, S.M. BLACKBURN, M. BUTRICO, A. COCCHI, P. CHENG, J.DOLBY, S. FINK, D. GROVE, M. HIND, K. S. McKINLEY, M.

MERGEN, J.E.B. MOSS, T. NGO, V. SARKAR, M. TRAPP: The Jikes Research Virtual Machine project: Building an open source community. In IBM SYSTEMS JOURNAL, VOL 44, NO 2, 2005.

[8] ETIENNE GAGNON AND LAURIE HENDREN Sable Research Group: Effective Inline-Threaded Interpretation of Java Bytecode Using Preparation Sequences.

[9] IBM. The Jikes Research Virtual Machine User's guide, 2004. Manual accompanying the JikesRVM source distribution.

[10] IAN ROGERS: Threaded Interpreters: Message sent to comp.compilers newsgroup Nov. 2005

[11] SPINELLIS, D. Code reading: The open-source perspective. Effective Software Development. Addison Wesley, 2003.

[12] TIM LINDHOLM FRANK YELLIN: The Java Virtual Machine Specification Second Edition. Online edition.

[13] BILL VENNERS. Inside the Java Virtual Machine. McGraw-Hill Companies.