PARALLELIZING THE JIKES RESEARCH VIRTUAL MACHINE

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

2006

Christos-Efthymios K. Kotselidis
School of Computer Science

Table of Contents

Abstract **6**

Declaration **7**

Copyright **8**

Acknowledgements **9**

1. Introduction **10**

1.1 Scope and Objectives 10
1.2 Organization of the thesis 11

2. The Jikes Research Virtual Machine **12**

2.1 Structure Overview 12
2.1.1 Runtime System 14
2.1.2 Thread and Synchronization System 14
2.1.3 Memory Management System 15
2.1.4 Compiler System 16
2.2 The BootImageWriter 16
2.3 The Optimizing Compiler 17
2.3.1 The Intermediate Representation 17
2.3.2 Method Compilation 18
2.4 The Adaptive Optimization System (AOS) 21
2.4.1 The Controller Subsystem 22
2.4.2 The Runtime Measurement Subsystem 23
2.4.3 The Recompilation Subsystem 24
2.4.4 The AOS Database 24

3. Parallelizing the BootImageWriter **25**

3.1 Previous Work 25
3.2 The Clone Problem 26
3.3 Implementation 28
3.3.1 Jikes RVM design deficiencies 30
3.4 Results and Performance Issues 31
3.4.1 Hardware and Software 31
3.4.2 Results 31
3.4.3 Performance Issues 33

4. Parallelizing the Runtime Compiler **36**

4.1 The AOS Architecture 36
4.1.1 Selective Optimization 38
4.2 Implementation 41
4.2.1 AOS Thread Model 41
4.2.2 Techniques of creating multiple compilation threads 42

4.3 Results and Performance Issues 44

5. Conclusions 49

Acronyms 52

References 53

List of Tables

Table 3.1: Results of compiling the BootImage with single or multiple threads..31

Table 3.2: Results of compiling the BootImage with single or multiple threads(IBM'JVM)…… 32

Table 4.1: Results after running SpecJVM98 Benchmarks using one compilation thread…….. 45

Table 4.2: Results after running SpecJVM98 Benchmarks using two compilation threads…..……. 46

Table 4.3: Results after running SpecJBB05 Benchmark using one and two compilation threads……49

List of Figures

Figure 2.1: IR flow from Bytecode to Final Assembly. 18
Figure 2.2: Optimized compilation of a single method. 20
Figure 2.3: Overview of the AOS. 21
Figure 2.4: Overview of the Runtime Measurement System. 23
Figure 3.1: Former implementation of the newExecution method. 26
Figure 3.2: Problem caused by clone's shallow copies. 27
Figure 3.3: Current implementation of the newExecution method. 28
Figure 3.4: Netbeans profiler snapshot during BootImage Compilation. 31
Figure 4.1: Overview of the AOS. 34

Abstract

Compilers have been an area of research since the early stage of computers' development. The introduction of multicore and multiprocessor architectures creates a new field of research. Modern Compilers and Virtual Machines can benefit from these architectures. The introduction of parallelism during compilation can improve significantly the performance of Virtual Machines. This thesis examines the advantages of parallelizing the Jikes Research Virtual Machine. The parallelization will be applied in two different parts. The first part is the BootImage creation and the second part is the Runtime compiler.

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

Acknowledgements

First of all I would like to thank Ian Rogers for his invaluable help. Without him this project would never exist. Being the brain of the project, he spent hours helping me and guiding me throughout the difficult task of understanding Jikes RVM.

Furthermore, I would like to thank my supervisor Dr. Chris Kirkham. He motivated me to work hard and supported me morally throughout the project.

Many thanks go to all my friends (Tasos, Pavlos, Kostas and Kostas) here in Manchester for their support during this year, my friends back in Greece and all the guys in the Jamaica Group that helped me always when I had problems during the project.

I thank my beloved parents, Konstantinos and Anastasia, for their help throughout my life. I thank them for their emotional and financial support. Without them I would not be able to study abroad and therefore to complete this thesis. Furthermore I thank my brother Panayiotis for his company, my grandmothers Xarikleia and Vasilia and of course my grandfather Christos Mavridis.

Special thanks go to Dimitris Stamos (cool) ,Giorgos Kritharas and Tim Vouras for being my friends and letting me win during our Pro tournaments.

Last but of course not least, I thank my special Austrian (EU) friend Fiona Schwab for the laughter we had this year. I thank her for motivating me, boosting my morale and correcting my assignments (FS) throughout this year.

1. Introduction

This thesis introduces the functionality of the Jikes Research Virtual Machine (Jikes RVM) [1, 20] as well as the subsystems that can be parallelized in order to achieve better performance. The work has been carried out in the Jamaica Research Group at the University of Manchester.

1.1 Scope and Objectives

The recent development of multicore and multiprocessor computers has defined new requirements for the software to be implemented. Programmers now must take advantage of these high performance computers by embedding parallelism in their programs. An attempt was made to detect the segments of code that could be executed in parallel in order to improve Jikes RVM's performance. The two main parts which were rewritten to be parallel are the **BootImage creation** and the **Runtime Compiler**. An effort was made in order to enhance not only the performance during the bootstrap of the Jikes RVM but also the performance of the Runtime System which results in faster compilation of source code due to parallelism. The objectives of the work were to examine the behavior of the Jikes RVM in multithreaded environments while using various techniques. The factors that influence our results had to be analyzed and solutions to any potential drawbacks had to be implemented.

1.2 Organization of the thesis

Chapter 2 discusses the structure of the Jikes RVM focusing on the optimizing compiler, the BootImage Writer and the Adaptive Optimization System.

Chapter 3 discusses in detail the parallelization of the BootImageWriter as well as the parallelization of the Optimizing compiler.

Chapter 4 discusses in detail the parallelization of the Runtime Compiler including the results.

Chapter 5 states the conclusions and the considerations about the work. It also suggests potential future work that maybe done, derived from the results.

2. The Jikes Research Virtual Machine

The Jikes Research Virtual Machine is derived from IBM's internal project called Jalapeno [2]. In 2001 IBM donated the software to the Open Source Community. Currently it comprises an Open Source testbed virtual machine for Java written in Java. It encompasses a variety of sophisticated features such as an optimizing compiler, adaptive optimization system (AOS) [4], Garbage Collection framework and Thread Scheduling scheme.
Currently Jikes RVM comprises an important system where new techniques concerning virtual machines can be applied. A significant number of Universities and Institutions use it in order to employ innovative techniques on dynamic compilation, garbage collection and thread scheduling.

2.1 Structure Overview

The Jikes RVM consists of four core subsystems [19, 3]:

The **Run-time subsystem** provides functionality concerning the run-time aspects of Jikes RVM. It is mostly written in Java providing services such as I/O, exception handling, dynamic class loading, reflection, etc. All the services are provided through Jikes RVM *Magic* class.

The **Thread and Synchronization subsystem** provides functionality concerning the threads implementation and scheduling in Jikes RVM. The Jikes RVM creates `pthreads` and assigns a Virtual Processor object on each one of them. All java threads are then multiplexed on the virtual processors. Shared queues are held in order to support the interconnection among the threads.

The **Memory Management subsystem** provides the functionality necessary for the allocation and collection of the system's objects. All the memory management tools are integrated in a single unified library called MMTK [11, 17] which includes a variety of GC mechanisms.

The **Compiler subsystem** includes the various compilers of the Jikes RVM. Two types of compiler exist. Firstly, the bytecode to native code compiler which is fast but performs no optimization (Baseline/Quick compiler) and secondly the optimizing compiler which compiles segments of code using optimization techniques with the use of Intermediate Representation (IR) [10, 13]. The optimizing compiler is part of the adaptive optimization system (AOS) which performs dynamic optimization while executing source code.

The fact that Jikes RVM [14] is written in Java results in its incapability to bootstrap itself. To provide a solution to this problem Jikes RVM constructs the BootImage which is a set of predefined core classes necessary to initialize the rest of the system. An ordinary Java program called BootImageWriter constructs a snapshot of the entire system by compiling and writing the core services' classes in a file called BootImage. An external pre-installed Virtual Machine is used in order to construct the BootImage.

2.1.1 Runtime System

2.1.1.1 Exception Handling

Depending on the nature of the exception, Jikes RVM responds accordingly. If the exception is a normal Java Exception, Jikes RVM can handle it internally [7]. If the exception is generated from the hardware (division by zero, null pointer exception, array index out of bounds) then a hardware interrupt is created and a C handler delivers it to Jikes RVM. The corresponding exception is built and then it is delivered to the `deliverException` method. The tasks that this method performs are to save the object's state and data in order to retrieve them later and to deliver the exception to the appropriate catch block.

2.1.1.2 The MAGIC Class

In order for the Jikes RVM to communicate with the underlying Operating System (OS) a special MAGIC class is used. The methods in the MAGIC class have empty bodies and are identifiable by the compilers. While compiling, the correspondent machine code of every MAGIC method is inserted in order to bypass the type system. Users can not invoke MAGIC methods which consequently guarantee Jikes RVM's integrity. Some services that MAGIC methods can implement are: Object Allocation, Garbage Collection, Dynamic Linking, Exception Handling, Reflection and I/O.

2.1.2 Thread and Synchronization System

The Jikes RVM multiplexes Virtual Processor objects (`VM_VirtualProcessor`) on operating system's `pthreads`. A Virtual Processor object is assigned for each `pthread`. Java threads are then multiplexed on the Virtual Processor objects. Various queues are held on every Virtual Processor. The total number of the VPs can be defined either by passing an argument while running the Jikes RVM or by editing the source code. A number of queues containing the threads that are not running are held on every Virtual Processor.

Such queues are:

- **idlequeue** contains an idle thread which runs when the VP does not have any other thread to execute.
- **readyqueue** contains threads ready to be executed.
- **transferqueue**, which can be accessed from other VPs too, contains the threads that are transferred to a particular VP from other VPs. It is used for load balancing.

Considering thread scheduling, Jikes RVM does not use any particular time slicing algorithm. Each running thread stops either after a voluntarily call to a yield method or if it is blocked by a lock. Jikes RVM supports three kinds of locks:

- **Processor locks** are Java objects (`VM_ProcessorLock`) which contain a single field that states the VP that owns the lock. It is mostly used for load balancing.
- **Thin locks** are bits stored in the header of an object. They are used as a locking mechanism in case multiple threads contest for a particular object.
- **Thick locks** are Java objects. They are "heavyweight" implemented containing numerous queues and are used when multiple requests from multiple users exist for a specific object.

2.1.3 Memory Management System

As already mentioned, Jikes RVM integrates all the GC mechanisms and memory managers in a unified platform called MMTK. Numerous Garbage Collectors such as incremental, conservative, real-time and concurrent are contained in the MMTK. In general 55 collector-neutral mechanisms, 5 GC sub-components and 8 GC algorithms are included in the MMTK.

2.1.4 Compiler System

Two types of compilers exist in the Jikes RVM [1, 19].

- The **Baseline** compiler compiles the source code into native code without performing any optimizations. The compiled code is generated through one pass. The `VM_BaselineCompiler` class contains switch statements that generate the bytecodes through the architecture specific `VM_Compiler` methods.
- The **Optimizing** compiler provides high level optimizations. It constructs an Intermediate Representation (IR) on which various optimization phases are performed until the optimized IR will be translated into machine code. The parallelization of the Optimizing compiler is one of the key issues this thesis will examine and therefore it is presented in detail in section 2.3.

Another type of compiler, which is implemented only for PPC architecture, of the Jikes RVM is referenced in its documentation:

- The **Quick** compiler is a middle solution between the Baseline and the Optimizing Compilers. It generates code through one pass, like the Baseline compiler, performing some primitive optimizations.

2.2 The BootImageWriter

As already mentioned, the BootImageWriter is an ordinary Java program that constructs the BootImage. The BootImage is a file which contains compiled core classes of the Jikes RVM necessary for its bootstrap. During the build of the Jikes RVM the BootImageWriter compiles and writes into a file all these core classes. The way the core classes (**primordials**) are compiled is defined by the type of build selected while running the **jconfigure** command. Currently there are four main configurations for the Jikes RVM:

- The **prototype** configuration excludes the optimizing compiler and the adaptive optimization system. Therefore is simple and fast but provides poor performance.

- The **prototype-opt** is similar to the prototype configuration including the optimizing compiler and the adaptive optimization system.
- The **development** configuration includes the optimizing compiler and the adaptive optimization system resulting in a fully functional RVM. Building the system using this configuration will significantly prolong the build time. The compilation of the BootImage is performed by the optimizing compiler. This fact differentiate the development configuration from the prototype-opt configuration where the BootImage is build by the baseline compiler.
- The **production** configuration is similar to the development one excluding all the assertions. This configuration is the one with the highest performance and takes approximately the same time to build as the development configuration.

2.3 The Optimizing Compiler

The optimizing compiler [4, 12] is one of the core elements that compose the Jikes RVM. It provides high level optimizations with the use of the Intermediate Representation (IR).

2.3.1 The Intermediate Representation

The fundamental unit of compilation and optimization in the Jikes RVM is a single method. Each method's bytecodes are translated into the IR and all the optimization phases are applied to it until the method is finally translated to machine code. Three categories of IR exist in the Jikes RVM:

- High-Level IR (HIR) is similar to bytecodes. The operations applied to HIR are similar to those applied on bytecodes. It is a register-based representation providing more flexibility than tree or stack based representations.
- Low-Level IR (LIR) introduces the Jikes RVM details into the IR. Such details are the Jikes RVM object model, the write barriers, etc.
- Machine-Level IR (MIR) is similar to the target assembly language. It is the last step before the code is translated to machine code.

Optimizations are performed in each of these IRs.

The sequence of the steps that are applied into a single method is depicted in figure 2.1.

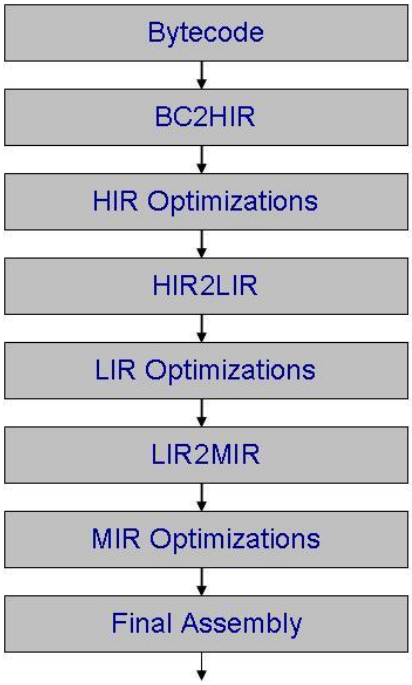

Figure 2.1: IR flow from Bytecode to Final Assembly

2.3.2 Method Compilation

All the optimization phases performed on a method during its compilation are called Compiler Phases. An instance of the `OPT_CompilationPlan` class contains all the necessary information to compile a method. The necessary elements for a compilation to be performed are:

- The optimization plan which contains all the optimization plan elements. Each individual optimization plan element is an instance of the `OPT_CompilerPhase` class. Each element represents one optimization phase that might be performed on the IR of a method. The optimization plans then are wrapped in the

`OPT_OptimizationPlanAtomicElement` or `OPT_OptimizationPlanCompositeElement` class (both classes are subclasses of the `OPT_OptimizationPlanElement` abstract class).

- The IR of a method. The IR of a method is an instance of the `OPT_IR` class. All the optimizations are performed on this object which is altered until its final transformation which is the machine code.
- The method compiled which is an instance of the `VM_NormalMethod` class.
- The Options object that is used for the compilation which is an instance of the `OPT_Options` class.

The steps that Jikes RVM does in order to compile a method are the following:

1. It retrieves a new Optimization Plan. The Optimization Plan is an array which contains `OPT_OptimizationPlanElement` objects (`OPT_OptimizationPlanElement[]`).The retrieval of the Optimization plan is performed by the `OPT_OptimizationPlanner` class. This class contains the createOptimizationPlan() method which returns a new Optimization Plan. It encloses all the `OPT_OptimizationPlanElement` objects in an array under the guidance of the `OPT_Options` object that is passed as an argument to the method. For each element its shouldPerform() method is invoked in order for the optimization planner to decide if that compiler phase must be included in the optimization plan or not. The shouldPerform() method checks in the `OPT_Options` object if a specific compiler phase has to be encountered or not.
2. It constructs the `OPT_CompilationPlan` object which contains the method to be compiled, the optimization plan and the compilation options.

3. It invokes the optimizing compiler to compile the compilation plan and to return the compiled method (`OPT_Compiler.compile(cp))`.

The above procedure is performed for all the methods that have to be compiled. Figure 2.2 depicts the compilation procedure.

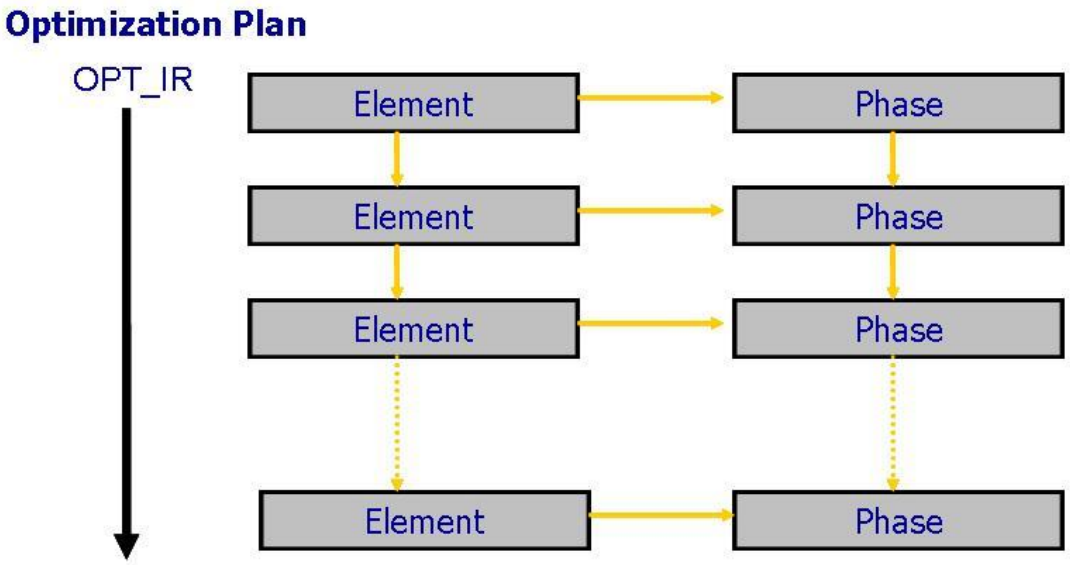

Figure 2.2: Optimized compilation of a single method

Having that specific architecture in the optimizing compiler, it is obvious that multiple optimization plans for different methods will share the same optimization elements. To avoid interference among the states of different compiler phases, when we want to create a new optimization plan for a new method, the newExecution() method of each compiler phase is invoked. As will be demonstrated in section 3.2, the original implementation of the newExecution() method was causing problems during the parallelization of the optimizing compiler.

2.4 The Adaptive Optimization System (AOS)

The optimizing compiler discussed in section 2.3 is the key component of the heart of Jikes RVM which is the Adaptive Optimization System [1, 4, 5]. The AOS performs on line monitoring and measuring of the methods that are being compiled. The general idea behind it is to track "hot" methods (methods that are being used frequently) and to recompile them using progressive optimization levels. The general architecture of the AOS is illustrated in figure 2.3.

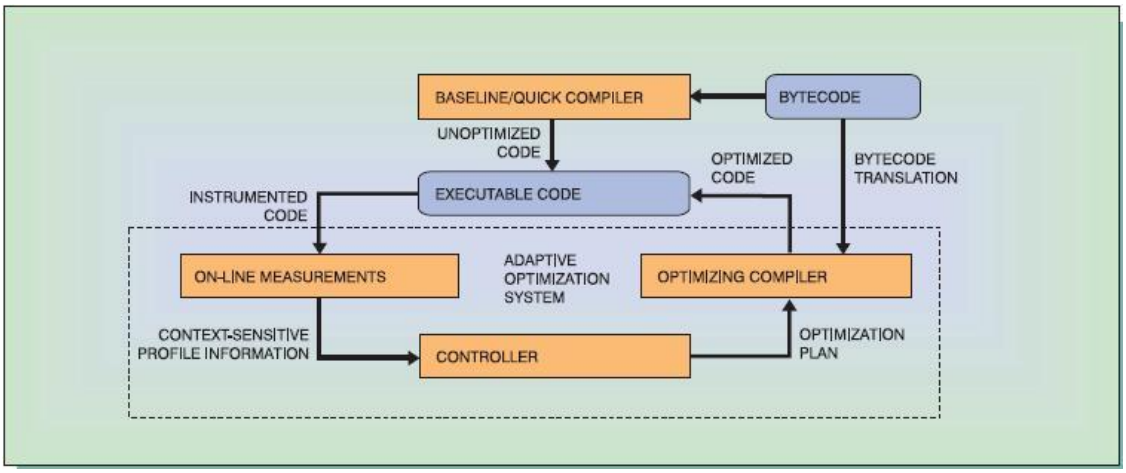

Figure 2.3: Overview of the AOS [1]

- The On-Line Measurement System monitors the methods' execution using sampling and profiling techniques. Information about methods is written in the AOS Database. The profiling information is forwarded to the controller subsystem which in turn decides the actions that must be performed for a particular method (Optimizing recompilation or Modification of the profiling techniques).
- The Controller System then constructs the recompilation optimization plan and passes it to the optimizing compiler for recompilation. The Controller System determines which methods should be recompiled and at what optimization level.

The AOS of the Jikes RVM is composed of five Java threads [6]:

- Three organizer threads that perform the monitoring and are part of the on-line measurement system.
- The controller thread which is the main thread for the controller system and performs all the coordination between the on-line measurement system and the recompilation system.
- The compilation thread which performs the recompilation of "hot" methods.

The communication among the various threads is performed via shared blocking priority queues.

The following sections will discuss the components of the AOS in more detail.

2.4.1 The Controller Subsystem

The Controller system [5] is the coordinator of the AOS. It organizes the Runtime Measurement system along with the Recompilation system. It initializes all the profilers used by the measurement system defining the form of profiling, the duration and the rules. It also constructs recompilation plans according to the information gathered from the AOS Database and the Profiling subsystem. These recompilation plans are forwarded to the recompilation system which performs the compilation of "hot" methods.

Depending on the information gathered by the AOS Database and the Runtime Measurement Subsystem the controller may decide either to continue profiling using a different strategy or to recompile the method.

The main part of the Controller Subsystem is the Controller Thread. This is a normal Java thread whose primary task is to run an infinitive loop and dequeue events the On-line Measurement subsystem has placed. If no events are placed in the queue then the controller thread is blocked until a new event is placed. All the events implement an interface which provides the process method which the controller calls.

2.4.2 The Runtime Measurement Subsystem

The role of the Runtime Measurement subsystem is to monitor the execution of the methods and to gather information regarding each individual method. After that, it forwards the information either to the AOS database for storage or to the controller subsystem for the decision making. A more detailed view of the Runtime Measurement System is illustrated in figure 2.4.

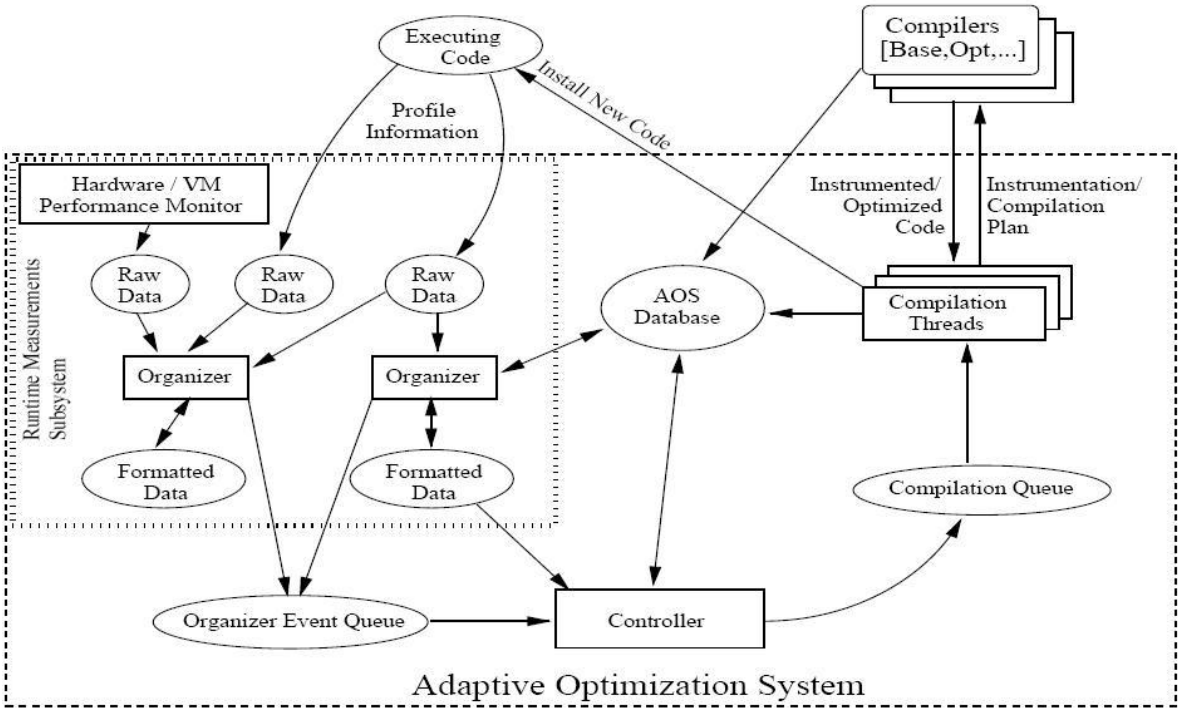

Figure 2.4: Overview of the Runtime Measurement System [1]

As shown in figure 2.4 the Runtime Measurement System encloses several subsystems that perform data profiling. These systems perform instrumentation of the executing code, VM and hardware performance monitoring. The outcome of these subsystems is raw profiling data which in turn are analyzed by separate threads, called **organizers**. The organizer threads (`VM_Organizer`) are created by the Controller.

2.4.3 The Recompilation Subsystem

The Recompilation subsystem consists of a single recompilation thread (`VM_CompilationThread`). As already mentioned the controller system places compilation plans of methods to be executed in the compilation queue. The compilation thread extracts these compilation plans from the queue and executes them. The main components of a compilation plan are:

- The **optimization plan**, which defines the nature of the optimizations that must be applied on a method during its recompilation.
- The **profiling data**, derived from the Runtime Measurement system, which directs the feedback-directed optimizations created by the optimizing compiler.
- The **instrumentation plans**, which indicate any potential intrusive instrumentation that should be inserted to the generated code by the optimizing compiler.

The controller thread runs an infinitive loop checking the compilation queue. If any available plan exists then the controller thread retrieves it and executes it. If not then the thread is blocked until the controller places a plan in the queue.

2.4.4 The AOS Database

The AOS Database acts as the repository of the adaptive optimization system. The various subsystems that comprise the AOS store in the AOS Database decisions that have been taken concerning certain methods, events or analysis results.

3. Parallelizing the BootImageWriter

The parallelization of the BootImageWriter is the first step that has to be taken in order to parallelize the Runtime Compiler. While building the Jikes RVM using the development mode the optimizing compiler is compiled and used for the compilation of the **primordials**. An attempt to parallelize the BootImageWriter would benefit us in the following two aspects:

- Firstly, we would achieve a speedup during the compilation of the Jikes RVM using the time consuming development mode. It takes approximately 13 minutes when compiled single-threaded.
- Secondly, the parallelization of the BootImageWriter would provide sufficient debugging mechanisms while attempting to parallelize the optimizing compiler. During the BootImage compilation, the threads created are pure Java threads and an external virtual machine is used. Therefore the debugging mechanisms provided by the VM can be used in order for errors to be traced and fixed.

3.1 Previous Work

Initially, the BootImageWriter was made parallel for the baseline compiler. Unfortunately, this support became broken and the system was never made to run for the optimizing compiler. The main reason behind the incapability of the optimizing compiler to run in parallel was the original implementation of the newExecution() method of the `OPT_CompilerPhase` class as explained in section 3.2.

3.2 The Clone Problem

As explained in section 2.3.2, during the compilation of a method by the optimizing compiler, an optimization plan is created and applied on each individual method. Apparently, different methods will need to share the same optimization phases (`OPT_CompilerPhase`). In order to achieve that, the original implementation of Jikes RVM was calling the newExecution() method of each compiler phase that has to be shared among different optimization plans. The newExecution() method returned a clone instance of the compiler phase as shown in figure 3.1.

```
public OPT_CompilerPhase newExecution (OPT_IR ir)
    try {
      return (OPT_CompilerPhase)this.clone();
    } catch (CloneNotSupportedException e) {
       return null;
    }
 }
```

Figure 3.1: Former implementation of the newExecution method

The use of the clone() method [15, 16] in order to return a new compiler phase was the reason why the optimizing compiler could not run in parallel. The clone() method as implemented in the Java programming language returns shallow copies of fields instead of deep copies.

The problem caused by the shallow copy is illustrated in figure 3.2.

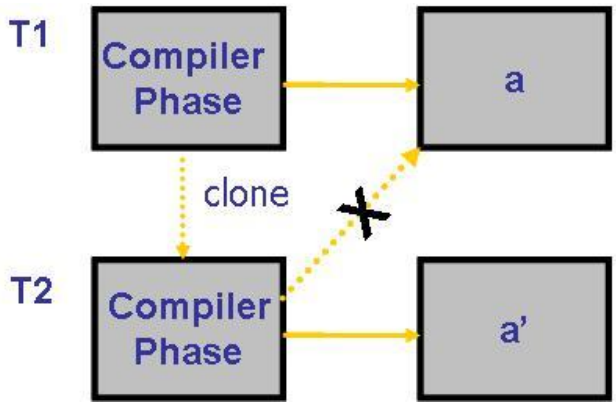

Figure 3.2: Problem caused by clone's shallow copies

In a potential case, we have two threads: T1 and T2. A compiler phase is created on T1 which references an object **a**. With the original implementation of the newInstance() method, if thread T2 wants to create a new compiler phase similar to the compiler phase of T1 then a new compiler phase will be created by the clone method **but** the reference to object **a** will remain the same. This results in the two threads accessing and modifying the same object **a**.

The compiler phases reference other objects which are shared among cloned versions. This causes severe problems during compilation with the optimizing compiler as shared objects are modified among different compiler phases. For example instances of the `OPT_Options` class are referenced by compiler phases. Using clone, multiple threads change the fields of these objects which consequently results in methods compiled with different options than they were supposed to.

3.3 Implementation

In order to enable proper separation of the compiler phases, so they can be compiled in parallel, we had to replace the clone method's shallow copies with deep copies.
The solution implemented to deal with this problem is the use of the newInstance() method which returns deep copies.
An abstract method getClassContructor() has been added to the OPT_CompilerPhase class. Each compiler phase overrides that method returning the Constructor object of the specific compiler phase. The Class("…").forName() method returns the runtime class descriptor of a class. Then we call the getContructors() method in order to retrieve the constructor of the particular compiler phase. The implementation of the newExecution() method in the OPT_CompilerPhase is shown in figure 3.3.

```
public OPT_CompilerPhase newExecution(OPT_IR ir) {
    try{
       Constructor cons = getClassConstructor();
       if(cons != null) {
         return (OPT_CompilerPhase)cons.newInstance(initargs);
       } else {
    throw new Error("Error, no constructor found in phase " +
                   this.getClass() +
                   " make sure a public constructor is declared");
              }
    }catch(Exception e){
      throw new Error("Failed to create phase " + this.getClass(), e);
     }
 }
```

Figure 3.3: Current implementation of the newExecution method

The newExecution() method now returns a new instance of the compiler phase. The deep copy is achieved by assigning all the fields of the returned classes to separate objects. Every compiler phase that does not require a deep copy because it has no fields, overrides the newExecution method returning *this* (i.e. compiler phases with no fields are shared among threads as they pose no hazard).

Other implementation details in order to succeed in the parallelization of the optimizing compiler are:

- Extra segments of synchronization had to be added on shared variables and data structures. The most important synchronization occurs in the BURS algorithm [18].
- In order to enable the separation of the methods being compiled in order to compile them in parallel, discrete optimization plans for every method must be formulated. This is implemented by creating two vectors. The first vector stores the optimization plan of every method while the second stores the compilation options for every method. The access to these two vectors is synchronized. When a method needs to be compiled by the optimizing compiler, a new optimization plan is created by the optimization planner which is stored in the first vector. A cloned version of the master options is also stored in the second vector at the same index as the optimization plan was stored in the first vector. If a free optimization plan exists then it is locked and used by the method being compiled. Otherwise the pre mentioned procedure takes place.
- Redeclaration of the static variables to non-static in order not to be shared by different compiler phases.
- Extra minor fixes had to be added too such as the report of additional statistics (3.3.1) method and redeclaration of inner classes to outer.

3.3.1 Jikes RVM design deficiencies

Objects which are instances of the `OPT_CompilerPhase` class implement the reportAdditionalStats() method. The role of this method is to report additional statistics concerning the optimized compilation of a method. The original implementation of this method is:

- As already stated, each compiler phase (`OPT_CompilerPhase`) is wrapped in an Atomic Element object (`OPT_OptimizationPlanAtomicElement`). Each Atomic Element has a reference to the compiler phase it contains through an instance variable. The Atomic Element in turn contains two double instance variables which are used to measure some additional statistics (counter1, counter2). When the reportAdditionalStats() method is implemented in the compiler phases, access to these double variables is made via the Atomic Element.

  The first time the method is called, it works correctly. When the newExecution() method is called then the link between the Compiler Phase and the Atomic Element is lost due to re-initialization of all the fields in the compiler phase. Therefore, during the compilation of the BootImage, NullPointer Exceptions were indicating the lost of the connection between the Compiler Phases and their Atomic Elements.

  The solution implemented to solve that problem is the creation of a HashMap which will contain the values of these two variables for all the compiler phases. The HashMap will be initialized once during the creation of the optimization plan and afterwards all the compiler phases will store in this HashMap pair values (key: Name of the Compiler Phase, data: The values of the two variables).

The above changes has been made in all the Compiler Phases that implement the reportAdditionalStats() method.

3.4 Results and Performance Issues

This section includes all the results and the performance issues obtained during the study of the multithreaded BootImage compilation.

3.4.1 Hardware and Software

All the experiments have been carried out on the Antigua computer in the labs of the Jamaica Research Group at the University of Manchester. Antigua is a dual CPU Xeon hyper-threaded computer and therefore we could take advantage of its four processor contexts.

The external JVMs used are: Sun's VM version 1.5 and IBM's JVM.

3.4.2 Results

The results gathered after compiling the BootImage single and multithreaded using Sun's VM are shown in table 3.1.

Table 3.1: Results of compiling the BootImage with single or multiple threads (Sun JVM)

	1 thread	**2 threads**	**3 threads**
Real time	13m21.241s	10m53.217s	10m24.801s
User time	17m27.601s	24m7.286s	28m58.989s
Sys time	0m17.701s	0m18.061s	0m17.805s

The results gathered after compiling the BootImage single and multithreaded using IBM's VM are shown in table 3.2.

Table 3.2: Results of compiling the BootImage with single or multiple threads (IBM'JVM)

	1 thread	**2 threads**	**3 threads**
Real time	13m11.274s	10m12.808s	9m10.064s
User time	16m29.602s	21m57.422s	26m22.739s
Sys time	0m21.189s	0m23.793s	0m25.458s

These results are gathered by the application of Linux **time** command during the compilation. The Real time is the actual clock time that passed from the start of the compilation until its completion. The User time is the time the CPUs worked during the compilation. Since the BootImage is compiled on a multicore environment, the User time is the sum of all the times on each core.

Sun JVM

An approximate 20% decrease of the real time is achieved while compiling the BootImage multithreaded. However, a 43% increase of the User time while compiling with two threads and a 65% increase while compiling using three threads is noticed.

IBM JVM

An approximate 20% decrease of the real time is achieved while compiling the BootImage with two threads and an additional 10% decrease while compiling with three threads. However, a 32% increase of the User time while compiling with two threads and a 60% increase while compiling using three threads is noticed.

The behavior of the multi threaded model is better while using the IBM's JVM. This may be due to a better thread scheduling and synchronization scheme that IBM's JVM may use.

3.4.3 Performance Issues

A first explanation of why we have this increase in the system's user time was the synchronization of the Java threads. In order to test our assumption we profiled the compilation using the **Netbeans** profiler [23]. The output of the profiling is illustrated in figure 3.4.

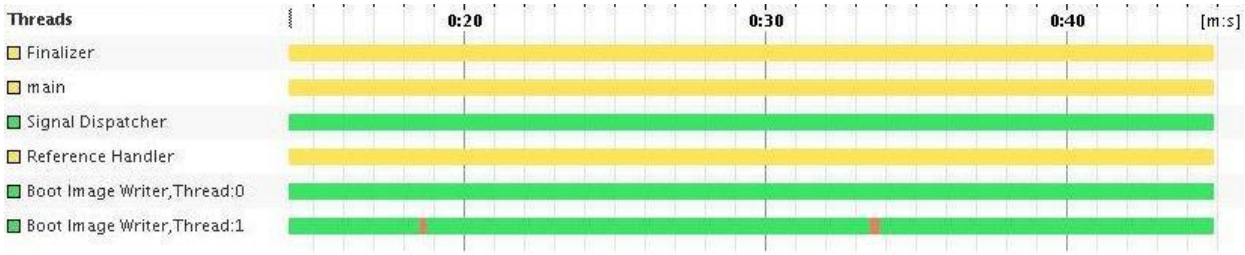

Figure 3.4: Netbeans profiler snapshot during BootImage Compilation

The two threads used for the BootImage compilation (BootImageWriter, Thread 0/1) are indeed run in parallel. The green color represents the threads in their run state while the red color represents the threads in their monitor state. The overall time the threads are in monitor state is approximately 1% of the overall execution time which means that the synchronization segment of code added does not slow down the compilation. Besides the **Netbeans** profiler, other commercial profilers such as **JProfiler** [22] were used in order to confirm our results. All the profilers produced the same output.

The profilers that profile Java programs reflect the situation and the synchronization in the application level. They do not reveal what is happening on the actual contexts of the computer. The output depicted in figure 3.4 is the same on multicore and on single core computers. That fact indicates that although the threads in the application level run in parallel, they do not necessarily run in parallel on the hardware level. We tried to find profiling tools to research the compilation behavior on the actual hardware but unfortunately no tools were available.

Another noticeable element is that the Java runtime system understands when executed in a multicore environment and automatically takes advantage of all the available CPUs. This behavior was tested with the use of Linux **mpstat** command. This command outputs the utilization on each core individually providing a clearer image about the contexts' utilization. While compiling using one thread, all the contexts showed a utilization of 30% on average. These results confirmed that Java uses all the available contexts while compiling in a multicore environment. When the BootImage was compiled using more than one thread the utilization of the contexts increased significantly. The corresponding contexts on which the separate compilation threads were assigned produced a 99.9% utilization while the remaining contexts had an average utilization of 70%. The Linux kernel was scheduling the threads, cycling them on the contexts.

Another idea that could justify this increase in the user time might be a race condition that was taking place among the running threads. The initial assumption was the following: A thread (T1) is compiling one method which a second thread (T2) tries to access at the same time. T2 can not realize that the method that it wants to compile is being compiled by T1 and instead of waiting until the compilation finishes and access the compiled method, it recompiles again the same method. In order to check the validity of the assumption the total number of the methods being compiled during the compilation of the BootImage was counted. Both for single and multithreaded compilation the number of methods were equal (17999 methods). This fact shows that no race condition takes place. If a race condition was taking place the number of methods compiled would be greater in the multithreaded compilation.

The next step was to find the distribution of the extra time on the compiling methods. The strategy followed was moving from general compilation times to more detailed compilation times. The result of this approach was the observation of an increase of the compilation time of each compiler phase of each method while compiling with multiple threads. Having in mind the amount of compiler phases

applied on each method of every class being compiled, we can conclude that the overall increase of the user time is the sum of the extra time spent on each compiler phase.

Two other explanations of the increase of the user time during the multi-threaded compilation may be:

1. The fact that Java takes advantage of all the available contexts while compiling on multicore environment adds an overhead due to the thread synchronization among the contexts.
2. The machine on which the measurements were taken has a shared cache memory between its contexts. The fact that the compilation threads loads data from the memory to the cache memory may cause the problem of cache thrashing.

The BootImage writer is an ordinary Java program which means that the threads created during the parallel compilation are ordinary Java threads. The VM used for the BootImage compilation is the external VM that has to be preinstalled in order to initialize the Jikes RVM. This results in the fact that all the scheduling and synchronization mechanisms are not transparent compared to the Jikes RVM. Java acts as a black box providing us with limited opportunities of investigating threads' behavior.

The availability of a proper multiprocessor computer with shared cache memories may have solved our questions concerning the multithreaded model.

4. Parallelizing the Runtime Compiler

The parallelization of the optimizing compiler in the context of the BootImage compilation was the first step for the parallelization of the Runtime Compiler. The Runtime compiler in its current implementation uses one compilation thread to recompile at a pre-specified optimization level the "hot" methods. The aim here is to create multiple compilation threads in order to take advantage of multicore or multiprocessor systems. The architecture of the AOS along with the changes made in order to create multiple compilation threads will be introduced in the later sections.

4.1 The AOS Architecture

As already introduced in section 2.4, the AOS is the framework that Jikes RVM uses for adaptive compilation. The AOS encompasses four subsystems as shown in figure 4.1

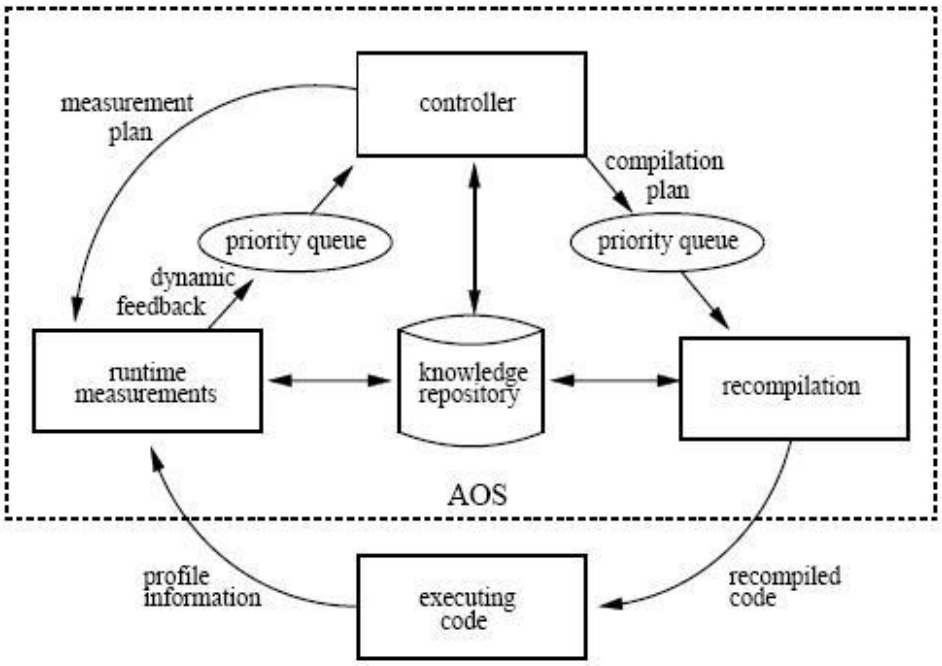

Figure 4.1: Overview of the AOS [1]

- The **Runtime Measurement Subsystem** profiles the execution of the application producing data for the decision making regarding the recompilation of frequent compiled (hot) methods.
- The **Controller Subsystem** is the backbone of the AOS. All the decisions, regarding the nature of the profiling data gathered from the Runtime Measurement subsystem along with the actions that have to be performed concerning "hot" segments of code, are included in this subsystem.
- The **Recompilation Subsystem** performs the recompilation of "hot" methods. This research focuses on this particular subsystem. The Recompilation Subsystem includes the compilation threads which we want to study. Currently a single compilation thread is initialized and run. The aim is to create more that one and to find an efficient way of deciding their number.
- The **AOS Database** which acts the role of the knowledge repository of the AOS. Profiling data concerning execution code and recompilation plans are stored in the database for future retrieval and study.

The communication between the sub-components of the AOS is made via **blocking priority queues**. The subsystems communicate asynchronously via a producer-consumer model.

The Adaptive Optimization System performs two major tasks: selective optimization [6] (4.1.1) and feedback-directed inlining [6].

4.1.1 Selective Optimization

The general idea behind the selective optimization is the discovery of segments of code on which significant execution time is spent. The fundamental unit of compilation and optimization in the Jikes RVM is a single method. Hence, all the segments of code that will be optimized concern methods.

If these methods were compiled at an optimization level by the optimizing compiler, then a significant speed-up of the system's performance would be achieved. Therefore, the AOS tries to find these "hot" methods and recompiles them with the optimizing compiler. The new compiled methods are then installed in the system and any potential future calls to these methods will invoke the optimized code.

In order for the Jikes RVM to identify "hot" methods, it installs a listener which checks the executed code and records the methods that are being executed. The listener checks the thread's call stack at every yield point and records into a buffer the method's id that is being executed. When the buffer is full then the organizers start working with the raw data the listener has gathered.

When the buffer becomes full, the listener suspends and the Hot Method Organizer (`VM_HotMethodOrganizer`) starts processing the data. The Hot Method Organizer holds a structure in which it stores the processed data. The data structure (`VM_MethodSampleData`) records for every method (id) the number of times it has been executed. When this task is completed for all the methods included in the buffer, the Hot Method Organizer constructs an event for each method and places them into the controller's input queue. The events contain the total number of times a method has been executed since the beginning of the execution. The Hot Method Organizer then re-registers the listener and become suspended until the next buffer is ready to be processed.

The Controller, which periodically checks its input queue, dequeues the events and processes them. The decisions that should be taken for each method are defined by a cost-benefit model (`VM_AnalyticalModel`). The Controller may decide either to recompile the particular method at an optimization level or to continue profiling the method using a different strategy.

Two classes compose the controller. The `VM_Controller` class contains the options of the controller subsystem. The `VM_ControllerThread` class which is an ordinary Java thread, runs an infinitive loop dequeueing and processing events placed in the controller input queue.

All the controller events implement an interface which provides the `proccess` method. This fact provides a significant level of abstraction and extensibility to the AOS. Any new events need only implement that interface in order to be added to the AOS.

The cost-benefit model that governs the AOS is represented by an abstract class (`VM_AnalyticalModel`) with several subclasses. Furthermore the recompilation strategies are all implemented by extending the `VM_RecompilationStrategy` or the `VM_AnalyticalModel` class.

If the controller decides to recompile a method, it creates an event and places it into the recompilation queue. The event includes the method to be recompiled and the optimization plan which will be applied.

The recompilation thread, which is the focus of this thesis, dequeues the event and recompiles the method using the optimization plan included in the event. After the recompilation of the method is finished, it installs the new compiled method for future calls.

Cost Benefit Model

The decision of whether or not a method should be recompiled at an optimized level is taken according to a Cost Benefit Model [8]. The model is contained in the `VM_AnalyticalModel` class. The following three measures are used for the decision making:

For a method compiled at level *i*:

- T_i is the time that will be spent executing the method at that particular level i.e. without performing any or more optimizations.
- C_j is the time spent recompiling the method at optimization level *j*.
- T_j is the time that the program will spend in executing a method after it has been recompiled at optimization level *j*.

Then the comparison made is the following:

$$\mathbf{C_j + T_j < T_i}$$

If the sum of the time spent while recompiling the method at optimization level j and the future execution time of the method (after recompiled at level *j*) is less than the execution time spent for the method without any recompilation, then the controller recompiles the method. In other case the controller chooses not to recompile the method.

The quantities that have to be calculated are unknowable in practice. The Controller calculates these quantities using a simple model (`VM_CompilerDNA`). The efficient estimation of Cost-Benefit values is still a subject of research.

4.2 Implementation

The former implementation of the Jikes RVM uses one compilation thread to perform the optimized recompilation of "hot" methods. The way that Jikes RVM may take advantage of a multicore or multiprocessor machine is by creating multiple recompilation threads. An effort was made to create multiple recompilation threads using different methodologies. In order to decide the strategy that should be followed concerning the creation of the multiple threads, the thread model of the AOS had to be studied.

4.2.1 AOS Thread Model

When the AOS starts, five major threads are created. Apart from the various other threads such as the application thread and the GC threads, three monitor threads, one controller thread and one recompilation thread are created. The monitor threads are mostly suspended on a wait queue. They are woken thanks to execution of adaptive metric code inserted into the compiled code by the AOS. In general the monitor loading is very light due to its role which is the incrementation of counters and the insertion of data into tables.

The recompilation thread is most of the time suspended. It is woken when a lazily compiled method is invoked or the AOS decides to recompile a "hot" method. The invocation of a lazily compiled method is rare and may occur when the system is bootstrapping. Therefore the loading is light. When the AOS decides to recompile a method at an optimization level, the caller will be suspended until the end of the compilation.

Regarding the other daemons that exist in the system, a finalizer thread, a debugger thread and GC threads (one per Virtual Processor) are created. All of them are mostly suspended in wait queues. The GC threads run only when everything else is stopped. Therefore, they can be ignored when measuring the occupied VPs. The finalizer threads run only after GC and only if a class requires their functionality.

The debugger thread currently does not work, so it also can be ignored while measuring the available VPs.

In conclusion, the threads that occupy the system most of the time are the application threads and idle threads executing.

4.2.2 Techniques of creating multiple compilation threads

In order to create multiple compilation threads two techniques have been implemented. The first technique is to measure the VPs that currently run idle threads. Compilation threads will be created and assigned on these available idle VPs. The second technique is the creation of custom threads. Considering the first method an efficient way of measuring the available free VPs had to be found.

4.2.3.1 Measuring the available idle VPs

The scheduler of the AOS includes a mechanism that allows the identification of the non-deamon threads (application threads) in the system. It has a counter which shows the number of the active application threads in the system. If the counter hits zero then the VM is terminated. Unfortunately, this mechanism does not reveal if the active application threads are in running or in suspended state and therefore it can not be used for measuring the free VPs.

Java [16] provides a method which returns the available processors of a system. However, this method (`Runtime.getAvailableProcessors()`) returns the total number of the processors that exist in the system.

Each VP contains a field that stores the active thread which currently runs on it. If the active thread field is an instance of the `VM_IdleThread,` it means that this particular VP currently runs no threads and therefore a new thread can be assigned on it. This approach however has some drawbacks too. If a VP runs a daemon thread such as a monitor thread and does not have any other threads in its wait queue, the idle count will give a value which is an underestimate. The monitor

thread after a short time will be suspended and the VP will be idle. Consequently a better solution concerning the calculation of the idle VPs would be the measurement of the daemons too, besides the idle threads. However two problems associated with this solution does not allow us to use this technique. Firstly, the compilation threads have to be excluded because of the amount of time they run and also because these are the threads that have to be counted as running threads. Secondly, a running daemon may be masking an application thread in the VP's runnable queue. In order to be able to count the daemons running on each VP and the application threads in the runnable queue, a locking mechanism has to be used while checking the queues. A `VM_VirtualProcessorLock` must be assigned to each VP while examining its state. Due to performance issues, because of the load added by using the locking mechanism, this solution is not recommended.

Considering the above issues, the most efficient solution in order to decide the available VPs is the measurement of the VPs that currently run threads which are instances of the `VM_IdleThread`.

A method has been added to the `VM_Scheduler` class which returns the number of the available idle processors. The method added is the getCurrentAvailableProcessors() which checks the processors[] vector and increments a counter when a processor's current thread is the idle thread. If no processors are available one processor is returned in order for the compilation thread to run.

4.2.3.2 Using custom compilation threads

A new class has been added to the Jikes RVM. The class `VM_CompilationController` is used in order to be able to define the way the compilation threads are created. Concerning the creation of custom compilation threads, the user simply sets the number of the desired compilation threads. A loop creates the compilation threads. All the compilation threads retrieve methods for recompilation from a shared compilation queue where the controller places the "hot" methods. The access to this queue is synchronized to avoid multiple compilations of same methods.

4.3 Results and Performance Issues

Two benchmarks have been used (SpecJBB05 and SpecJVM98) to measure the performance of the multithreaded model. The benchmarks have been run using a variety of combinations between VPs and compilation threads.

All the compilations took place on Antigua computer. The measurements are the average of three measurements for each result. The executions were carefully tested in order to avoid interferences with processes running on the machine. Therefore all the Benchmarks were run when the machine was not executing any other users' processes.

Tables 4.1 and 4.2 contain the results gathered for SpecJVM98 using one and two compilation threads.

Table 4.1: Results after running SpecJVM98 Benchmarks using one compilation thread

Benchmarks	Number of Virtual Processors used			
(100 iterations)	**1 VP**	**2 VPs**	**3 VPs**	**4VPs**
	real 9m43.735s	real 9m45.623s	real 10m36.913s	real 24m25.190s
compress	user 9m4.934s	user 9m 4.238s	user 10m25.507s	user 54m21.176s
	sys 0m16.165s	sys 0m16.345s	sys 0m16.677s	sys 0m16.169s
	real 3m32.184s	real 3m44.947s	real 4m7.219s	real 97m13.490s
jess	user 3m14.396s	user 3m29.805s	user 3m37.938s	user 283m1.922s
	sys 0m16.113s	sys 0m16.397s	sys 0m16.625s	sys 0m19.833s
	real 25m8.733s	real 25m23.741s	real 25m35.012s	
db	user 24m4.638s	user 24m39.544s	user 25m10.990s	-
	sys 0m26.142s	sys 0m26.134s	sys 0m25.694s	
	real 7m24.500s	real 8m58.802s	real 7m44.866s	real 11m52.713s
mpegaudio	user 7m7.475s	user 8m12.955s	user 7m8.915s	user 19m9.336s
	sys 0m2.792s	sys 0m2.704s	sys 0m2.744s	sys 0m3.188s
	real 9m50.184s	real 10m8.025s	real 9m50.625s	
jack	user 6m7.179s	user 6m35.621s	user 6m32.321s	-
	sys 3m27.089s	sys 3m36.706s	sys 3m38.074s	

Some benchmarks could not run using 4 VPs. A first observation from the results is that when more than one VP is used, the real time increases. The time increases by a small extent moving from one to two or three VPs. However, when trying to use four VPs the programs slow down significantly in most cases while in other cases they even could not complete. This is probably caused by the scheduling scheme Jikes RVM uses.

The Jikes RVM sits on top of the underlying OS which means that two scheduling mechanisms work simultaneously. Firstly, the OS schedules the running threads on the different contexts and secondly, the Jikes RVM's schedules the threads multiplexed on the VPs.

The behavior of the running threads was observed with the **mpstat** command. The threads were cycling on the available contexts (4) of the Antigua computer.

The benchmarks were tested again under the same conditions using two compilation threads. Table 4.2 illustrates the results.

Table 4.2: Results after running SpecJVM98 Benchmarks using two compilation threads

Benchmarks	Number of Virtual Processors used			
(100 iterations)	**1 VP**	**2 VPs**	**3 VPs**	**4VPs**
	real 8m59.256s	real 9m30.931s	real 8m54.462s	real 20m56.295s
compress	user 8m39.124s	user 9m29.160s	user 9m11.118s	user 41m2.898s
	sys 0m16.201s	sys 0m15.977s	sys 0m15.989s	sys 0m16.677s
	real 3m31.057s	real 4m58.099s	real 3m40.168s	
jess	user 3m14.684s	user 3m28.421s	user 3m31.644s	-
	sys 0m15.701s	sys 0m15.833s	sys 0m15.737s	
	real 24m39.942s	real 26m11.862s	real 25m41.302s	
db	user 24m9.687s	user 25m4.890s	user 25m41.420s	-
	sys 0m26.086s	sys 0m25.262s	sys 0m25.598s	
	real 7m58.566s	real 7m33.637s	real 7m32.44s	real 11m27.664s
mpegaudio	user 7m51.281s	user 7m26.592s	user 7m21.104s	user 17m12.525s
	sys 0m2.932s	sys 0m3.524s	sys 0m2.916s	sys 0m3.176s
	real 10m9.253s	real 10m14.448s	real 10m36.853s	
jack	user 6m9.591s	user 6m32.609s	user 6m42.509s	-
	sys 3m28.493s	sys 3m36.174s	sys 3m38.870s	

A first noticeable element is the behavior of the Benchmarks when run with two VPs. Both the real and the user time increases. While compiling with one, three or four VPs the time varies but in small percentages. In general a speedup is achieved in these cases.

The percentage of the improvement of the performance is determined by the number of "hot" methods found in every run along with their size. According to IBM's papers, the optimized recompilation on a single run of SpecJVM98 takes approximately the 6.5% of the execution time. Therefore the portion of time that we try to speed-up using parallel compilation threads is relatively small compared to the total time of the execution. The parallelization of the recompilation occurs only when the compilation queue is loaded enough so both the compilation threads can recompile a method at the same time. If the recompilation of methods occurs rarely during execution or if it is spread among the execution time then it is more possible that only one compilation thread will be used.

The size of the methods being recompiled plays an important role too. If the methods are small then either one recompilation thread will recompile them, because the second would never manage to get one, or even if they were recompiled in parallel the actual time that the threads would run in parallel would be very small.

Furthermore, the fact that the machine used is a dual-core hyperthreaded Xeon (four contexts) with shared cache memory between the processor contexts may influence the results. Cache thrashing problems may be a factor negatively affecting the results.

The presence of a benchmark in which plenty of heavy methods had to be compiled so the parallel recompilation would occur in a significant extent would be a better indicator of the performance. On the other hand, testing the Jikes RVM in this situation may not be a good indicator concerning real applications.

The load of the schedulers (OS and Jikes RVM) has to be considered when studying the performance of the Jikes RVM. As shown in table 4.1 the real time increases when more than one VP is used. The reason is the load added by the scheduler. In some cases the Benchmarks had problems even for completing properly. Hence, in order to find the most efficient way of creating multiple compilation threads Jikes RVM has to be tested in real situations.

As explained in section 4.1.1, an equation is used in order to determine whether a method should be recompiled or not. In the equation used ($\mathbf{C_j + T_j < T_i}$), C_j is the time spent for recompiling the method at optimization level *j*. The calculated time is based on tests made while using one compilation thread. Adjustments have to be made to the values if compiling with multiple compilation threads.

The SpecJBB05 benchmark behaved differently to the SpecJVM98. The results are shown in table 4.3.

Table 4.3: Results after running SpecJBB05 Benchmark using one and two compilation threads

SpecJBB05	Number of Virtual Processors used			
	1 VP	**2 VPs**	**3 VPs**	**4VPs**
1 compilation thread	real 32m27.358s user 20m3.967s sys 2m11.262s	real 31m9.448s user 37m0.027s sys 18m49.111s	real 25m25.948s user 43m50.804s sys 28m22.234s	-
2 compilation threads	real 32m18.336s user 18m4.688s sys 14m9.081s	real 28m48.697s user 33m43.302s sys 19m31.217s	real 25m21.519s user 43m31.447s sys 28m42.232s	-

The real time decreases while using more VPs. A speedup is achieved while compiling with two compilation threads. The speedup is minor while using one and three VPs. When two VPs is used the real time decreased by three minutes. Besides the decrease of the real time, a decrease of the user time is achieved too. However, the system time increases dramatically. The results can vary considerably with the Benchmarks.

5. Conclusions

Modern Compilers and Virtual Machines can benefit by the introduction of multicore and multiprocessor architectures. Parallel compilation on multiple cores or processors should improve significantly the execution time of applications. This thesis examined the potential of parallelizing the Jikes Research Virtual Machine.

Parallelism was embedded in two parts of the Jikes RVM. The first part was the BootImage creation while the second was the Runtime Compiler. The main component of the two pre-mentioned parts is the optimizing compiler. An effort was made in order to parallelize the optimizing compiler.

The main reason behind the incapability of the optimizing compiler to run in parallel was the original implementation of compiler phases. Specifically, the use of the clone() method while creating new compiler phases caused severe problems resulting in the optimizing compiler crashing. The newExecution() method was rewritten replacing the clone() method's shallow copies with deep copies. Reflection was used in order to create new instances (deep copies) of the compiler phases. Extra fixes had to be added too in order to allow parallel execution of the optimizing compiler.

After successfully parallelizing the optimizing compiler, the BootImage creation of the Jikes RVM was tested. A 20% decrease of the real time achieved while compiling the BootImage using two threads. However, a significant increase of the user time is observed. The increase of the user time may be caused by the scheduling of the threads on the contexts and the synchronization between them. The threads used for the BootImage compilation are pure Java threads. Java acts as a black box providing limited opportunities while investigating threads' behavior. Some of the assumptions made about the threads' behavior were tested and various scenarios have been excluded.

The next step was the parallelization of the Runtime Compiler. The optimizing compiler as part of the Adaptive Optimization System (AOS) is used when "hot" methods have to be recompiled at an optimization level. Formerly, the Jikes RVM was using one compilation thread to perform the recompilation. An effort was made to create multiple compilation threads. Two techniques of creating multiple compilation threads have been implemented. The first technique concerns the creation of custom compilation threads while the second concerns the creation of compilation threads according to the availability of idle processors. The available Virtual Processors are sampled and the number of the recompilation threads created is equal to the idle VPs (Virtual Processors whose current thread is instance of the `VM_IdleThread`).

In order to study the behavior and the performance of the Jikes RVM two benchmarks were used. Both SpecJVM98 and SpecJBB05 produced similar results. The performance was slightly improved in most cases. However, there were cases where the execution time increased probably due to scheduling issues.

Future Work

The research carried out as part of this thesis can act as a starting point for future work concerning the parallelization of the Jikes RVM. Several design deficiencies of the optimizing compiler have been discovered along with the scheduling problems.

A further study of Jikes RVM’s scheduler would reveal any problems concerning the thread scheduling on the Virtual Processors. The execution time increased significantly while using more than one Virtual Processor. That may be due to problems caused by Jikes RVM’s inefficient scheduling. Furthermore, while testing the Benchmarks warning messages concerning the Virtual Processors and the Garbage Collection indicate that Jikes RVM can not function efficiently with multiple Virtual Processors.

Concerning the hardware used while measuring Jikes RVM's performance, a proper multiprocessor machine without shared cache memory between the processors could exclude some of the possible factors affecting the performance. In order to determine if problems caused by shared cache memory such as cache thrashing affect the performance Jikes RVM has to be tested on a multiprocessor machine.

The development of a proper Benchmark may provide a clearer image of the benefits of parallel compilation. In section 4.3 the disadvantages of the Benchmarks used for the measurements are explained.

Acronyms

Definitions for most acronyms were taken from the WikiPedia [9] project.

GC Garbage Collection. A system of automatic memory management which seeks to reclaim memory used by objects which will never be referenced in the future.

Jikes RVM Jikes Research Virtual Machine. A JVM implemented by IBM as a research program and released under an open source licence.

JVM Java Virtual Machine. A VM that executes Java bytecodes.

OS Operating System. A basic set of programs that communicate with the computer harware and share resources in order to enable user programs to run.

VM Virtual Machine. An environment between the computer platform and the end user which allows the execution of programs not designed for the current architecture.

IR Intermediate Represenation. A form of representing the bytecodes on which optimization techniques can be applied.

MMTk Memory Management Toolkit. A framework that includes all the Garbage Collection mechanisms in the Jikes RVM.

VP Virtual Processor. A Java object that represents a physical processor of the system. Java Threads in the Jikes RVM are multiplexed on Virtual Processors. A Virtual Processor object is assigned for each `pthread`.

AOS Adaptive Optimization System. A framework which performs adaptive optimizing recompilation during a program's execution.

References

[1] ALPERN, B.,ATTANASIO, C.R.,BARTON, J.J.,CHENG, P.,CHOI, J.-D., COCCHI, A.,FINK, S.J.,GROVE, D.,HIND, M.,HUMMEL, S.F.,LIEBER, D.,LITVINOV, V.,MERGEN, M.F.,NGO, T.,RUSSELL, J.R.,SARKAR, V.,SERRANO, M.J.,SHEPHERD, J.C.,SMITH, S.E.,SREEDHAR, V.C.,SRINIVASAN, H., AND WHALEY, J. (2000) The Jalapeno virtual machine, IBM Systems Journal.

[2] ALPERN, B.,AUGART, S.,BLACKBURN, S.M.,BUTRICO, M.,COCCHI, A.,CHENG, P.,DOLBY, J.,FINK, S.,GROVE, D.,HIND, M.,McKINLEY, K.S.,MERGEN, M.,MOSS, J.E.B.,NGO, T.,SARKAR, V., AND TRAPP, M. (2005) The Jikes Research Virtual Machine project:Building an open-source research community, IBM Systems Journal.

[3] ALPERN, B.,BARTON, J.J.,HUMMEL, S.F., NGO, T.,SHEPHERD, J.C.,ATTANASIO, C.R.,COCCHI, A.,LIEBER, D.,MERGEN, M., AND SMITH, S. Implementing Jalapeno in Java, 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99), Denver, Colorado, November 1, 1999.

[4] BURKE, M.G.,CHOI, J-D.,FINK, S.,GROVE, D.,HIND, M.,SARKAR, V.,SERRANO, M.J.,SREEDHAR, V.C.,SRINIVASAN, H., AND WHALEY, J. The Jalapeno Dynamic Optimizing Compiler for Java, 1999 ACM Java Grande Conference, San Francisco, June 12-14, 1999.

[5] ARNOLD, M.,FINK, S.,GROVE, D.,HIND, M., AND SWEENEY, P.F. Adaptive Optimization in the Jalapeno JVM: The Controller's Analytical Model, 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3), December 10, 2000, Monterey, California.

[6] ARNOLD, M.,FINK, S.,GROVE, D.,HIND, M., AND SWEENEY, P.F. Architecture and Policy for Adaptive Optimization in Virtual Machines.

[7] GOUSIOS, G. JikesNode: A Java Operating System, MSc thesis, University of Manchester,2005.

[8] ARNOLD, M.,FINK, S.,GROVE, D.,HIND, M., AND SWEENEY, P.F. Adaptive Optimization in the Jalapeno JVM, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000), Minneapolis, Minnesota, October 15-19, 2000.

[9] ALPERN, B.,COCCHI, A., LIEBER, D.,MERGEN, M.,AND SARKAR, V. Jalapeno-A Compiler-Supported Java Virtual Machine for Servers, Workshop on Compiler Support for Software System (WCSSS 99), Atlanta, GA, May 1999.

[10] MUCHNICK, S.S. Advanced Compiler Design and Implementation, Morgan Kaufmann ed.,1997.

[11] BLACKBURN, S. AND CHENG, P. MMTk:The Memory Management Toolkit, Jikes RVM(Jalapeno) Presentations [3],2004.

[12] FINK, S.,GROVE, D., AND HIND, M. Dynamic Compilation and Adaptive Optimization in Virtual Machines, Jikes RVM(Jalapeno) Presentations [3],2004.

[13] GROVE, D.,AND HIND, M., The Design and Implementation of the Jikes RVM Optimizing Compiler,Jikes RVM(Jalapeno) Presentations [3],2002.

[14] ATTANASIO, D. AND HIND, M. The Design and Implementation of the Jalapeno Research VM for Java, Jikes RVM(Jalapeno) Presentations [3] ,2001.

[15] ECKEL, B. Thinking in JAVA(4th Edition),Prentice Hall PTR,2006.

[16] SUN MICROSYSTEMS, The Java Tutorial(3rd Edition), http://java.sun.com

[17] BLACKBURN, S. M., CHENG, P., AND McKINLEY, K. S. Oil and water? High performance garbage collection in Java with MMTk. In Proceedings of ICSE 2004, 26th International Conference on Software Engineering (Edinburgh, Scotland, May 2004).

[18] PELEGRI-LLOPART, E. AND GRAHAM, S. L. 1988. Optimal code generation for expression trees: an application BURS theory. In Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (San Diego, California, United States, January 10 - 13, 1988). POPL '88. ACM Press, New York, NY, 294-308.

[19] IBM. The Jikes Research Virtual Machine User's guide, 2004. Jikes RVM manual.

[20] The Jikes Research Virtual Machine (RVM), 2004. http://jikesrvm.sourceforge.net

[21] Wikipedia, The Free Encyclopedia. http://en.wikipedia.org

[22] Ej-Technologies, JProfiler, http://www.ej-technologies.com/products/jprofiler/overview.html

[23] NetBeans, Profiler, http://profiler.netbeans.org