JIKES RVM ADAPTIVE OPTIMIZATION SYSTEM WITH INTELLIGENT ALGORITHMS

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

September 2004

By
Jisheng Zhao
Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

Future high-performance virtual machines will improve performance through sophisticated online feedback-directed optimization. This thesis presents the research work of applying intelligent optimization algorithms on the Jikes RVM adaptive optimization system (AOS); then the new AOS can search for optimal DNA (combination of optimizing compiler's parameters) and recompile the hot methods with new DNAs to gain better performance. The new system is written completely in Java (as is the original version of adaptive optimization system), applying the intelligent optimization techniques not only to application code, but also to the Jikes RVM virtual machine itself.

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

Chapter 1

Introduction

1.1 Motivation

Currently, virtual machines face significant performance challenges beyond those confronted by traditional static optimizers. First, portable program representations and dynamic language features, such as dynamic class loading, force the deferral of most optimizations until runtime, including a runtime optimization overhead. Second, modular program representations preclude many forms of whole-program interprocedural optimization. Third, virtual machines incur additional costs for runtime services such as security guarantees and automatic memory management.

To address the challenges listed above, vendors have invested considerable resources into adaptive optimization systems in production virtual machines. The motivation of this thesis is improving the runtime performance for Jikes RVM Java Virtual Machine by applying optimization algorithms on Jikes RVM's adaptive optimization subsystem that can perform adaptive recompilation at runtime. The new adaptive optimization system will perform searching in the optimization space at runtime to find out the DNA that can result in better performance. The DNA is a combination of the optimization compiler's parameters for the method that will be recompiled by the adaptive optimization system at runtime.

This problem can also be regarded as investigating an approach to improve compiler heuristics with machine learning approaches at runtime. Many compiler

problems are NP-Complete [1]. Thus, implementations can't be optimal. One optimization approach (a compiler optimization function) is beneficial to one kind of application or system configuration, but it may not gain performance improvement in another kind of application or may make its performance even worse than before. We can't record the optimization strategies for every application and system configuration, so the proposal is to use intelligent algorithms to automatically search the priority function space (in this thesis, we name it as the optimization space) at runtime to find a suitable DNA that can gain performance improvement.

1.2 Adaptive system with intelligent optimization

An adaptive system is a system that can change its behavior dynamically. This means that a system that can perform adaptive optimization aims to improve performance by monitoring the system's behavior and then making optimization decisions.

In this thesis, the adaptive optimization has an extra function – searching in the program optimization space to find optimization strategy dynamically. The traditional optimizing compiler is primarily based on static analysis. The optimization strategy will be swiftly made redundant, due to architectural evolution (i.e. change to a new generation of processor), application requirement (i.e. some application focus on floating point computation, some need large memory) or the changing of the hardware configuration (i.e. increase/reduce the number of processors, memory). It is extremely difficult to accurately model the interaction between program and machine. The adaptive optimization uses runtime information to make decisions about how to optimize the program code, so it can exploit

[1] NP ("non-deterministic polynomial-time") is the set of decision problem solvable in polynomial time on a non-deterministic Turing machine.
A decision problem C is NP-complete if :

1. it is in NP
2. every other problem in NP is reducible to it.

"reducible" here means that for every problem L, there is a polynomial-time many-one reduction, an deterministic algorithm which transforms instances $l \in L$ into instances $c \in C$, such that the answer to c is YES if and only if the answer to l is YES.

the exponential growth in hardware performance. There are two intelligent optimization algorithms used for adaptive optimization. i.e. Genetic Algorithm and Hill Climbing.

Jikes RVM could be viewed as a adaptive system, because it has an adaptive optimization subsystem that performs dynamic recompilation for hot methods. But its current recompilation strategy is based on static heuristics (the adaptive optimization system make judgment by statistic data got by static analysis). The research work in this thesis applies the intelligent algorithms (Genetic Algorithm, Random-Mutation Hill Climbing) on the adaptive optimization system to perform intelligent searching and improve heuristics dynamically.

Another benefit that can be got from adaptive optimization is that it reduces the complexity of the optimization compiler [SAMO02].

1.3 Contribution

The main contributions of this thesis are summarized as below:

- Applies the intelligent optimization algorithms (Genetic Algorithm and Random-Mutation Hill Climbing) to the Jikes RVM Java Virtual Machine's adaptive optimization system, so the Jikes RVM's adaptive optimization system can search for optimal DNA and use the optimizing compiler to recompile methods with these DNA to improve the runtime performance. This is named as online optimization.

- Implements the offline optimization. The optimum combination of compiler's parameters got at runtime with the intelligent optimization algorithms can be stored and reloaded the next time the VM is booted, so the current iteration can benefit from the previous iteration.

- Studies the impact of the intelligent optimization algorithm. Because adaptive optimization is a trade-off, performing the searching and recompilation will not only improve the recompiled methods' runtime performance, but also consume the computing resource. In addition, the searching will also introduce some risk in runtime (described in Chapter 3).

1.4 Outline

This chapter gives a brief introduction to the background and contribution of the research. The detailed background knowledge and concepts will be described in chapter 2, including the architecture of Jikes RVM and its adaptive optimization system, the adaptive system concept, genetic algorithm and RMHC algorithm.

Chapter 3 presents the design and implementation of the intelligent optimization algorithms on Jikes RVM adaptive optimization subsystem.

Chapter 4 gives the Spec JVM98 test results of the new version of Adaptive Optimization System compared with the old version.

Chapter 5 gives the analysis of the test results and presents a summary of this thesis and a critique of the work, before outlining future work.

Chapter 2

Background

This thesis presents an improvement to the Jikes RVM's adaptive optimization system. The improved 'intelligent' adaptive system relies on either a Genetic Algorithm(GA) or Random-Mutation Hill Climbing (RMHC). This chapter will present an overview of the Jikes RVM in Section 2.1. Section 2.2 describes the particular adaptive optimization system in the Jikes RVM. Section 2.3 describes adaptive compilation systems in general. Finally, Section 2.4 gives an overview of genetic algorithms and the RMHC algorithm.

2.1 Jikes RVM

The Jikes RVM is a Java Virtual Machine written in Java and used for research work and as a test bed for new ideas [AAea00]. It features a modular design with flexible subsystems for memory management and garbage collection, dynamic compilation and adaptive compilation. Subsection 2.1.1 describes the overall structure of the Jikes RVM. Following this object model, the runtime system, dynamic compilation, memory management and threading are described in subsections 2.1.2, 2.1.3, 2.1.4, 2.1.5 and 2.1.6 respectively.

2.1.1 The implementation of the VM

The Jikes RVM is written in Java [ABC+00]. It compiles itself on a host JVM, writes itself out to disk and then uses a small C stub program to load itself back into memory. To enable it to have access to the OS, a callback mechanism is provided into C stubs that call the OS. The underlying OS signal handling

features are also used to provide information to the Jikes RVM such as null pointer exceptions. The Jikes RVM also supports calling Java Native Interface (JNI) compatible C libraries that may be loaded at runtime. By having the majority of the Jikes RVM written in Java, it is relatively straight_forward to port it to new operating systems and hardware.

Unlike other JVMs, the Jikes RVM doesn't have a bytecode interpreter but must always compile bytecodes. Compilation of bytecodes is triggered the first time a method is executed. The structure of the JVM is configured using a tool which creates a customized set of Java source files that are then built. Java bytecode can either be dynamically compiled or built into the initial RVM image. For the build configuration known as production a lot of the Jikes RVM's bytecode is compiled into the RVM image (unlike a prototype build). The build configuration also controls what classes are used for basic functions such as garbage collection.

The Jikes RVM augments the Java language to allow:

- Invoking operating system services.
- Using architecture-specific machine instructions
- Accessing machine registers and memory
- Coercing object references to raw addresses and vice versa
- Transfering execution to an arbitrary address

These capabilities must be available to the VM, however they must also be prevented from becoming available to user applications. The Jikes RVM achieves this using Java's package security mechanism. For the VM, the Jikes RVM has a special VM_MAGIC class. The bodies of these methods are empty to allow the Java source compiler (javac) to compile them. The Jikes RVM's compilers ignore the resulting bytecodes, as they recognize the name of VM_MAGIC class, and insert the necessary machine code inline. Thanks to the package mechanism and the Java class loader, the user code does not evade Java's security restrictions. The Jikes RVM's class loaders will verify, when they encounter a call to a VM_MAGIC method, that the method being compiled is authorized to access that part of the JVM.

2.1.2 Object Model

The object model is an important aspect of an object oriented language. Java's object model allows references to objects but the underlying pointers to locations in memory are hidden (unlike with languages such as C++). Java uses a garbage collector to reclaim unused areas of memory. An efficient object model within the JVM can improve performance (for example by reducing cache misses or by being fast to execute) and the amount of memory used.

The Java language's object model deals with issues such as: field and array access, virtual method dispatch, dynamic type checking, and null pointer checks. For different VMs, there are different object model implementation approaches. The general approach of the Jikes RVM's object model is given below:

- Object Header: in the Jikes RVM there is a two-word object header associated with each object. It supports virtual method dispatch, dynamic type checking, memory management, synchronization, and hashing. One word of the header is a status word including three fields: the first is used for locking; the second holds the default hash value of hashed objects and the third is used by the memory management system. The other word of an object header is a reference to the TIB (described later) that will be helpful for dynamic type checking and virtual method dispatch.

- Type Information Block (TIB) serves as the Jikes RVM's virtual method table. It is an array of Java object references. The first component is a reference to an object that describes the class of this object (including its superclass, the interfaces it implements, offsets of any object reference fields). The remaining components are references to compiled method bodies (executable code, and lazy compilation stubs) for the virtual methods of the class (in the Jikes RVM compiled code is placed in a special object called VM_CodeArray).

- Jikes RVM Table of Contents(JTOC, shown in Figure 2.1) is a global data structure used internally by the Jikes RVM. It is a single int array with some of the values being the addresses of objects. All of Jikes RVM's global data structures are accessible through the JTOC, including static fields and references to all static method bodies, literals, numeric constants and references to string constants. The JTOC also holds references to all of

a class's TIBs. The Jikes RVM uses a machine register to always hold a reference to the JTOC, allowing it to be easily accessed.

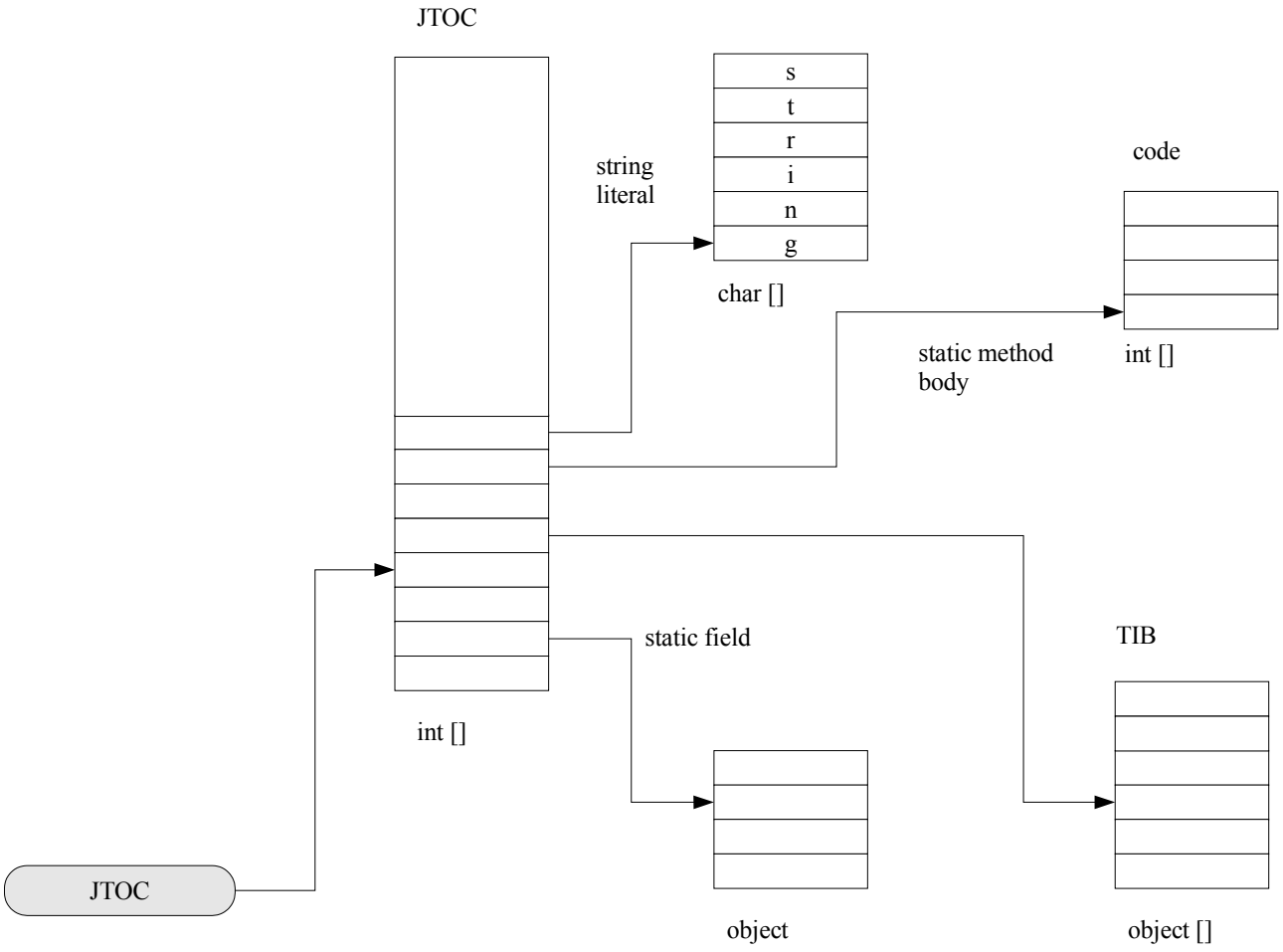

Figure 2.1: JTOC

- Virtual Methods Dispatch: The compiled method bodies are arrays of machine instructions. To perform virtual method dispatch, the Jikes RVM loads the TIB pointer from the object header, computes the offset of the method's compiled code within the TIB, then loads this value and branches to it.

- Static fields and methods are stored in the JTOC, so they can be accessed easily and quickly.

- Null pointer checking is performed by hardware. In the Jikes RVM, object references are machine addresses. Any attempts to access a null pointer

are trapped by the hardware as memory violations. The Jikes RVM can get information for the null pointer exception from the interrupt handler, encapsulate the information into a null pointer exception object and transfer it to the method that caused the exception.

- Method invocation stacks (shown in Figure 2.2) A stack frame contains space to save nonvolatile registers, a local data area (the use of which is compiler-dependent), and an area for parameters that are to be passed to called methods and that will not fit in the Jikes RVM's volatile registers. It also contains a compiled-method identifier (identifying information about the method for the stack frame, it is used to perform runtime profiling), a next-instruction pointer (the return address for any called method) and a previous-frame pointer (used to locate the previous call stack frame - it is used to walk through all the stack frames to find which method should handle an exception).

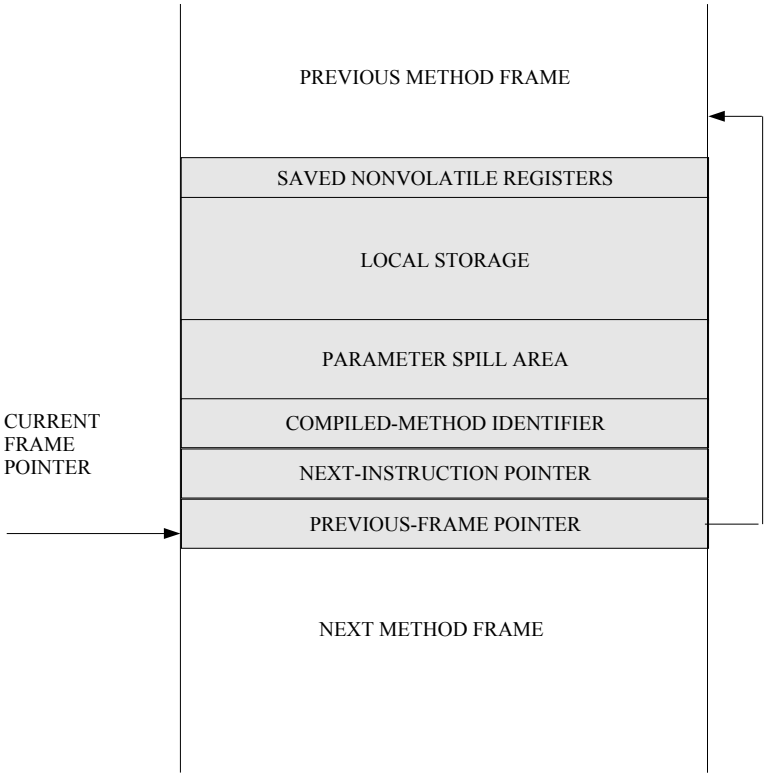

Figure 2.2: Stack Frame

2.1.3 Runtime Systems

The Jikes RVM's runtime system provides such services as: exception handling, dynamic type checking, dynamic class loading, interface invocation, I/O and reflection. The majority of all the runtime service code is implemented in Java, a few functions need OS support provided by C code.

- Exception handling: A signal will be generated if a null pointer is dereferenced, an array index is out of bounds, an integer is divided by zero, or a thread's method-invocation stack overflows. The signals are caught by a small C signal handler that will invoke a Java method using JNI. The Java method builds the appropriate exception object, copies it to the appropriate receiver and transfers control to the receiver's catch block [1].

- Dynamic class loading: the class loader loads the class the first time it is referenced during execution of an application. The compiler is used to emit machine code for methods within the class the first time a method is executed (the first method invocation goes to a trampoline that compiles the method and then updates references to it). In general, the class loader uses the baseline compiler at first; the compiled method will have the chance to be recompiled by the optimizing compiler in future via the adaptive compilation framework.

- I/O: basic I/O requires operating system support. The I/O operations are implemented by a system call interface that interacts with a support library written in C code.

- Reflection: is supported by the Jikes RVM's implementation of core objects and the class loader.

2.1.4 Dynamic Compilation in the Jikes RVM

Java bytecode interpreters are used in JVMs to allow bytecode to be machine independent. However, performance of interpreters is many times worse than code written explicitly for a particular machine. Dynamic compilation is used by JVMs to bridge this performance divide . It gives both native performance

[1] To implement dispatching exceptions, the Jikes RVM searches the current thread's stack frame to find the method that handles the exception; if it can't find the method, the thread will be killed.

and platform independence. This section will give a brief description of the Jikes RVM's dynamic compilers [BCF+99], their major functions and working scenarios.

The Jikes RVM employs a compile-only strategy; it compiles all methods to native code before they execute. Currently, the RVM system includes two fully operational compilers:

1. Baseline compiler: translates Java bytecodes directly into native code by simulating Java's operand stack. But the performance of the compilation result of this compiler is only slightly better than Java bytecode interpretation. This is because the compiler does not perform register allocation, it just performs the stack operations in the Java Spec.

2. Optimizing compiler: translates Java bytecodes into intermediate representation, upon which it performs a variety of optimizations. The compiler performs register allocation with some efficient algorithms. The compiler's optimizations can be grouped into three levels:

   - Level 0 consists mainly of a set of optimizations performed on-the-fly during the translation from Java bytecodes to the intermediate representation. As these optimizations reduce the size of the generated IR, performing them tends to reduce overall compilation time.
   - Level 1 augments level 0 with additional local optimization such as common subexpression elimination, array bounds check elimination and redundant load elimination. It also adds inlining based on static-size heuristics, global flow-insensitive copy and constant propagation, global flow-insensitive dead assignment elimination.
   - Level 2 augments level 1 with SSA-based flow-sensitive optimizations.

The intermediate representation optimization can be divide into three stage: (shown in Figure 2.3)

1. High-level intermediate representation (HIR) optimization.

2. Low-level intermediate representation (LIR) optimization.

3. Machine-specific intermediate representation (MIR) optimization.

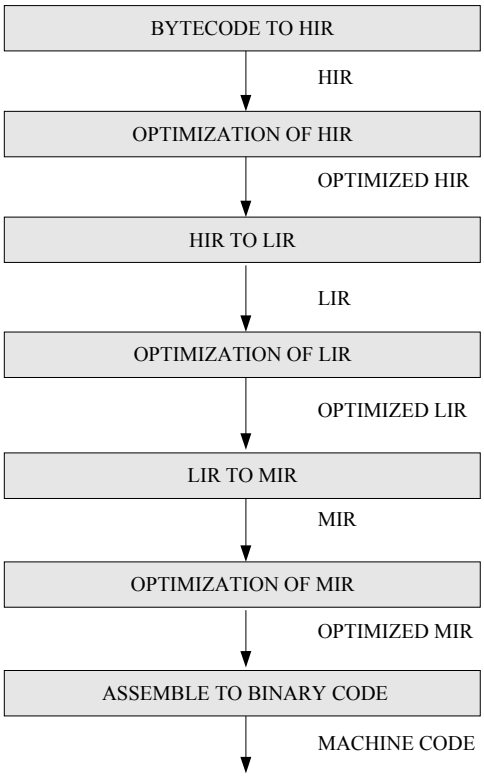

Figure 2.3: Optimization Stages

The compilers in Jikes RVM can be invoked in three scenarios listed below: (shown in Figure 2.4)

1. When the executing code reaches an unresolved reference (the class type does not exist in JTOC), causing a new class to be loaded (the class's TIB will be installed in JTOC), the class loader invokes a compiler (the compiler could be baseline compiler or optimizing compiler) to compile the class initializer. The class loader also initializes the compiled code for all methods to a lazy compilation stub.

2. When the executing code attempts to invoke a method that has been loaded and not yet been compiled, the lazy compilation stub is executed, which leads to the compilation of the method (the compiler could be baseline compiler or optimizing compiler).

3. The adaptive optimization system can invoke a compiler when profiling data suggests that recompiling a method with additional optimization may be beneficial. The adaptive optimization controller (describe later) evaluates

which optimization level can be applied to the method and builds the compilation plan that indicates to the optimizing compiler how to optimize the method. (This scenario is also the focus of this thesis).

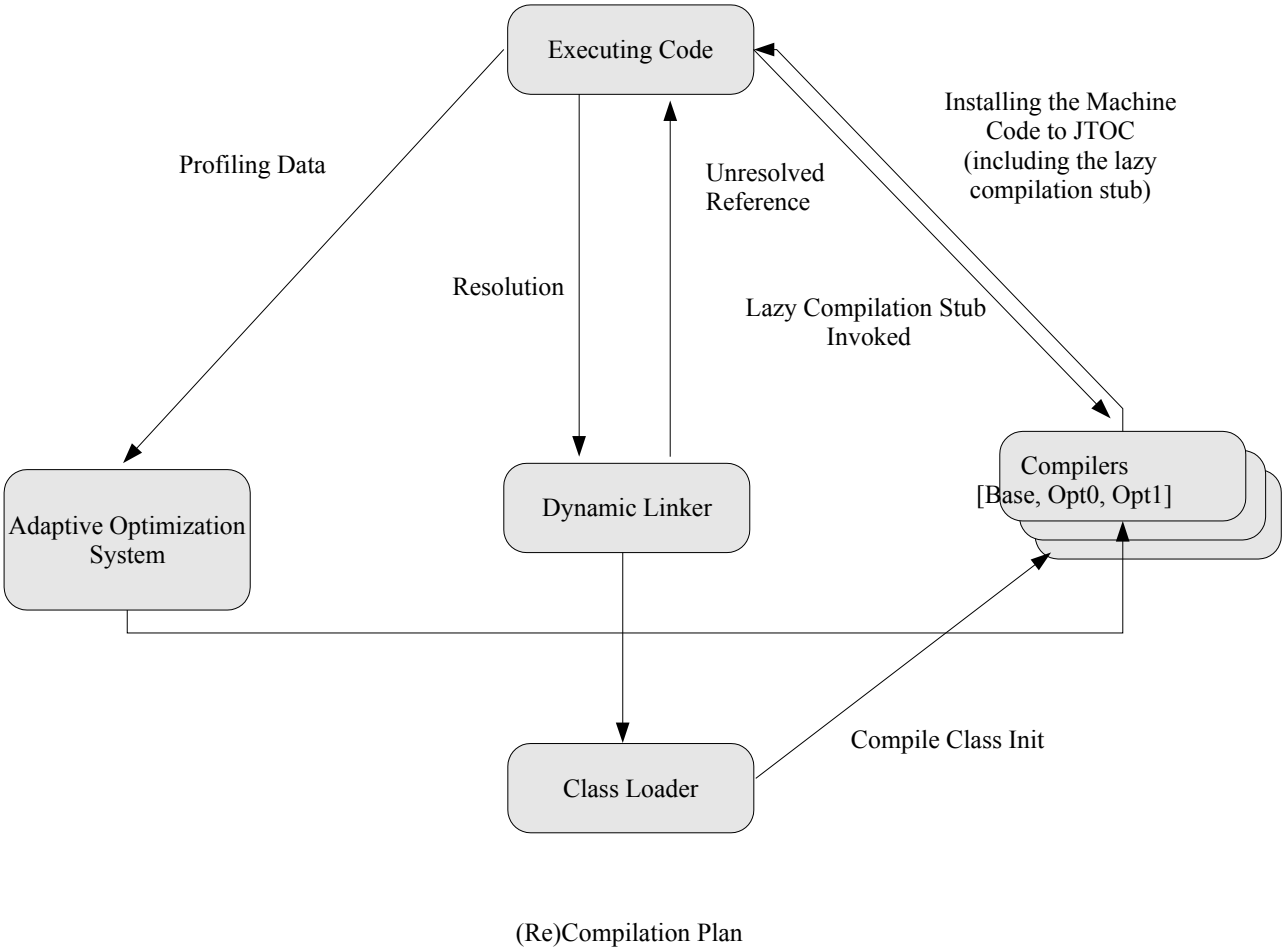

Figure 2.4: Compilation Scenarios

Another thing the compilers need to do is building the boot image. Both baseline compiler and optimizing compiler can perform this task. They act as a static compiler that works offline. (showed in Figure 2.5). The machine code generated by the baseline compiler or optimizing compiler is stored in the boot image file by the boot image writer [2].

[2]Boot image writer is a tool in Jikes RVM. It uses the compiler (baseline compiler or optimizing compiler) to build the boot image with the VM's class files.

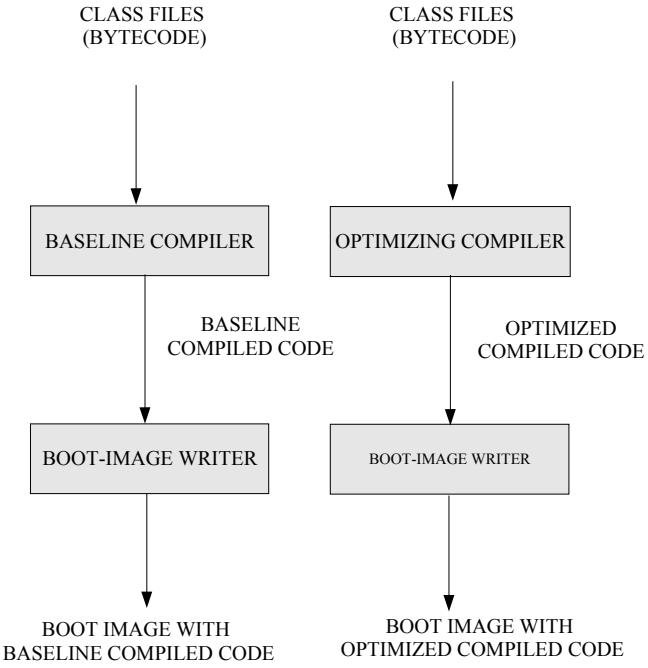

Figure 2.5: Bootwriter Scenarios

2.1.5 Memory Management

Automatic garbage collection is a very useful feature of Java and the most challenging to implement efficiently. There are many approaches to automatic memory management. There is not a memory management strategy that will be optimal for every scenarios. Jikes RVM has a flexible architecture in memory management: MMTK [BS04], it provides a standard interface for memory management, so it can support a family of memory managers for object allocation and garbage collection. There are four major types of managers supported: copying, noncopying, generational copying and generational noncopying. Each manager consists of a concurrent object allocator and a stop-the-world, parallel, type-accurate garbage collector.

2.1.6 Thread and Synchronization Model

Rather than mapping Java threads to operating system threads directly, Jikes RVM multiplexes Java threads on virtual processors that are implemented as

AIX / Linux pthreads [3]. This design is helpful for implementing such concerns as:

- Effecting a rapid transition between mutation (by normal threads) and garbage collection.
- Implementing locking without using OS kernel services.
- Supporting rapid thread switching.

Jikes RVM's threads are neither run-until-blocked nor fully preemptive; a thread can be preempted, but only at predefined yield points [4].

Jikes RVM has three kinds of locks for supporting system synchronization (i.e. thread scheduling and load balancing that require atomic access to global data structures) and user synchronization (the user threads access to their global data synchronously).

- Processor Lock is a low-level primitive used for thread scheduling (and load balancing) and to implement other locking mechanisms.
- Thin Lock is a object-level lock used for synchronizing the thread's access to an object. This lock mechanism, based on bits in an object header, is used for locking in the absence of contention.
- Thick Lock is an object lock implemented in a higher level compared with processor lock and thin lock. It has a enteringQueue used as a queue of threads that are contending for the lock and a waitingQueue used as a queue of theads awaiting notification on the object that the thick lock currently governs. A thin lock can be converted to a thick lock.

2.2 The adaptive optimization system in Jikes RVM

The research work in this thesis is based on Jikes RVM's adaptive optimization system. So here is an introduction to the original version of adaptive optimization

[3]The pthread is a standard specification for thread operations in POSIX. Most operating systems support this standard i.e. AIX, Linux, Solaris and Windows NT, although there may be differences in physical implementation.

[4]The yield point is a stub code that performs thread switch operations, it can be inserted into the compiled Java code by compilers in Jikes. The compilers provide location information for object references on a thread's stack at yield points.

system in Jikes RVM. Section 2.2.1 introduces the system architecture of the Jikes RVM's adaptive optimization system and the following one, 2.2.2, gives an introduction to the adaptive optimization system's working mechanism.

2.2.1 The adaptive optimization system architecture

The Jikes RVM adaptive optimization system (AOS) consists of three components: runtime measurements subsystem, adaptive controller subsystem and recompilation subsystem. These components encompass one or more separate threads of control, so they work asynchronously. The internal structure of the adaptive optimization system is depicted by Figure 2.6. The different components communicate with each other via the event queues. The event queues provide the ability for Bilateral communication. There's also a common data structure: the AOS database shared by all AOS components.

Runtime Measurements Subsystem

The runtime measurements subsystem gathers information about the executing methods, summarizes the information and then either passes the summary along to the controller via the organizer event queue or records the information in the AOS database.

Several systems, including instrumentation in the executing code, hardware performance monitors, and VM instrumentation, produce raw profiling data as the program runs. Usually, these systems perform only extremely limited processing of the raw data as it is produced. Instead, separate threads called organizers periodically process and analyze the raw data. Such a design separates the generation of raw profiling data from the data analysis and gets two benefits:

1. It allows multiple organizers to process the same raw data in different ways. i.e. optimizing the method and adaptive inlining use the same raw profiling data (methods counting).

2. This separation allows low level profiling code to execute under strict resource constraints. Recall that we monitor not just application code, but also system services of the VM. So the profiling operation should complete its task in a short time period. i.e. The profiling operations will be performed in thread switching or memory allocator, if the analysis work were also performed here, there would be a very effect on system performance.

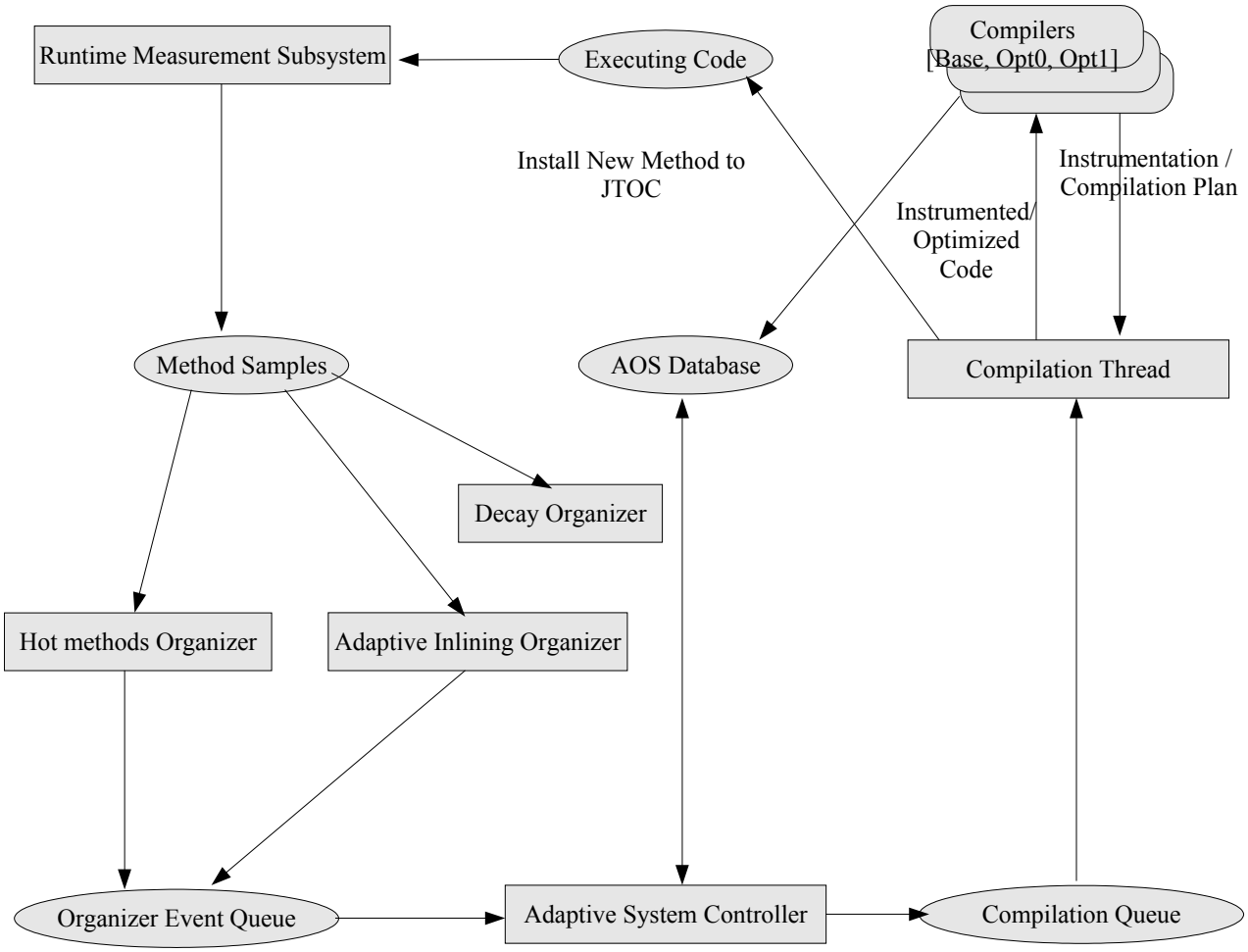

Figure 2.6: Adaptive Optimization System

There are several types of organizer in the adaptive systems: hot method organizer, adaptive inlining organizer and decay organizer. To perform more adaptive profiling functions, we can add more organizers in the system. In this thesis, we will focus on the methods counting data.

Adaptive Controller Subsystem

The major task of the adaptive controller subsystem is making compilation decisions, including whether or not to perform method recompilation, on-stack replacement, and adaptive inlining. The controller subsystem orchestrates and conducts the other components of the adaptive optimization system. It coordinates the activities of the runtime measurements subsystem and the recompilation subsystem. It initiates all runtime measurement subsystem profiling activities by

determining what profiling should occur, under what conditions, and for how long.

The adaptive controller subsystem receives information from the runtime measurement subsystem and AOS database, and uses the information to make compilation decisions. Such actions include:

- instruct the runtime measurements subsystem to continue or change its profiling strategy, which could include using the recompilation subsystem to insert intrusive profiling.

- recompile one or more methods using profiling data to improve their performance.

The compilation decisions will be passed to the recompilation subsystem that will direct the actions of the various compilers.

The controller communicates with the other two components using priority queues; it extracts measurement events from a queue that is filled by the runtime measurements subsystem and inserts recompilation decisions into a queue that compilation threads process. When these queues are empty, the dequeuing threads sleep. The various system components also communicate indirectly by reading and writing information in the AOS database (The AOS database stores the counting data for method invocations, yield points, instructions and debugging data).

Recompilation Subsystem

The recompilation subsystem consists of compilation threads that select compilers, perform compilation operations and install the recompilation results into JTOC (the new methods and inlined methods). The operations that need the compilation thread to perform are encapsulated in the compilation plans that are inserted into the compilation queue by the controller (it could be regarded as a series of command objects). The compilation plan consists of three components:

1. Optimization Plan specifies which optimizations the complier should apply during compilation.

2. Profiling Data directs the optimizing compiler's feedback-directed optimizations.

3. Instrumentation Plan dictates which, if any, intrusive instrumentation the compiler should insert into the generated code.

As the recompilation operations are performed by the compilation threads (separate from the application's threads), they can occur in parallel. This differs from the initial compilation of a method, which occurs the first time a method is invoked; and the lazy compilation, compilation occurs in the application thread that attempted to invoke the method that hadn't been compiled.

AOS Database

The AOS database provides a repository where the adaptive optimization system records decisions, events, and static analysis results and can be queried by various adaptive system components.

2.2.2 The working mechanism for Adaptive Optimization System

The adaptive optimization system's working mechanism is based on a Multi-level recompilation strategy and feedback-directed optimization (FDO). The paragraphs below describe the FDO and sampling mechanism in the Jikes RVM, the Multi-level recompilation strategy that is the kernel algorithm for Multi-level recompilation and, briefly, Feedback-directed inlining.

Feedback-directed optimization (FDO) and sampling mechanism

The optimization implemented in Jikes RVM adaptive optimization system is an online feedback-directed optimization (using the online profiling information as the feedback to perform runtime optimization)[AHR02]. There are three factors as compelling motivation for FDO:

1. FDO can overcome the limitations of static optimizer technology by exploiting dynamic information that cannot be inferred statically.

2. FDO enables the system to change and revert decisions when and if conditions change.

3. Runtime binding allows more flexible and easy-to-change software systems.

To implement fully automatic online FDO effectively, a VM must address the following challenges:

- Compensate for the overhead in collecting and processing profile information and performing associated runtime transformations
- Account for only partial profile availability and changing conditions that affect profile data stability.

To reach the aim of effective online FDO, the virtual machine needs to collect accurate profile data with low overhead. Jikes RVM performs Call-stack sampling to get accurate profile data for method call with low overhead; this sampling implementation takes advantage of existing mechanisms in the Jikes RVM. As mentioned in subsection 2.1.2, there is a field that contains a compiled-method identifier in Jikes's method stack frame, so we can get the information about which method is being called (the identifier of current method) from the current thread's stack frame before it switches and a counter associated with the current method is incremented. The system attributes a sample taken on a call edge to the current method. A sample taken in a method prologue is credited to both the calling and current method, capturing the fact that control is in transition between both methods. This sample technique provides a basic mechanism to estimate the time spent in execution of each method. In the adaptive optimization system implementation, the organizer threads that need the method counting data install sampling objects (method listeners) to record raw data regarding the execution profile. During a thread switch, the VM invokes the update method of this listener, which records the currently active method in a raw data buffer. This activity costs only a few additional cycles during each thread switch, and will not result in obvious performance impact. After collecting the number of samples specified by its current sample size, the method listener wakes the organizer threads. A woken organizer thread analyzes the counting data, generates the event object and sends it to the adaptive controller system. i.e. the hot method organizer discovers a hot method, generates a hot method event and sends it to the controller.

Multi-level recompilation strategy

Multi-level recompilation is the major mechanism for the adaptive optimization system to make the judgment whether or not to recompile a method and how to

recompile it.

Given a hot method from the hot method organizer by the event queue, the adaptive controller must decide whether it is profitable to recompile the method with additional optimizations. The controller uses a cost-benefit analysis to make this calculation. At first, the optimization levels available to the controller are numbered from 0 to N[5]. For a method m currently compiled at level i, the controller estimates the following quantities:

- T_i, the expected time the program will spend executing method m, if m is not recompiled.
- C_j, the cost of recompiling method m at optimization level j, for $i \leq j \leq N$
- T_j, the expected time the program will spend executing method m in the future, if m is recompiled at level j.

The system will choose the j that minimizes the quantity $C_j + T_j$, if $C_j + T_j < T_i$ the controller decides to recompile m at level j; otherwise it decides to not recompile. But the factors in this model are unknowable in practice. The process of estimating future cost and benefits is an ongoing open research problem. The current controller implementation is based on the fairly simple estimates described below:

- To estimate T_i :

  At first, the controller assumes the program will execute for twice its current duration. So, if the application has run for$n$ seconds, the controller assumes it will run for n more seconds. Then define T_f to be the future expected running time of the program. By using the sampling techniques described previously, the VM system keeps track of where the application spends time as it runs. The system uses a weighted average of these samples to estimate the percentage of future time P_m in each method. From the percentage estimate and the future time estimate, the controller predicts the future time spent in each method as: $T_i = T_f * P_m$. The weight of each sample starts at one and decays periodically. Thus, the execution behavior of the recent past exerts the most influence on the estimates of future program behavior.

[5]The compilers in the current RVM (baseline, opt0, opt1, opt2) are mapped as level 0, 1, 2, 3.

- To estimate T_j :

  When the controller recompiles methods, it adjusts the future estimates to account for the new optimization level, and expected speedup due to recompilation. The adaptive system estimates the effectiveness of each optimization level as a constant based on offline measurements. Let S_k be the speedup estimate for code at level k compared to level 0; if method m is at level i, the future expected running time at level j is given by: $T_j = T_i * S_i/S_j$

- To estimate C_j :

  Currently, the Jikes RVM uses a linear model (a function of method size, so C_j is determined by the method size) of the compilation speed for each optimization level. This model is also calibrated offline.

Feedback-directed inlining

The adaptive system also supports the online feedback-directed inlining[6]. The system takes a statistical sample of the method calls in the running application and maintains an approximation to the dynamic call graph based on this data. Using this approximate dynamic call graph, the system identifies hot edges to inline, and passes the information to the optimizing compiler.

Jikes RVM installs an edge listener (the same mechanism as described earlier) to capture the calling edge (including the caller, call site and callee). The calling edges should be inserted to a buffer. When the buffer becomes full, the edge listener notifies the Dynamic Call Graph (DCG) organizer to wake up and process the edges.

The DCG organizer uses the edges in the buffer to update the weights in a dynamic call graph it maintains. Periodically, the DCG organizer invokes the adaptive inlining organizer to recompute adaptive inlining decisions. The adaptive inline organizer will generate the compilation plan about the inlining operation and send it to recompilation threads.

[6]This function has not been involved in the work in this thesis. So, the description is brief.

2.3 Adaptive System

A system is adaptive if it changes its behavior by itself (based on its model of the user, or based on the task context).

In nature, various species have the ability to self-replicate and the ability to adapt to the changing of living environments by changing their genes.

In computer science, the goal is imbuing computer programs with intelligence, with the life-like ability to self-replicate, and with the adaptive capability to learn and to control their environments (computing resource, i.e. processors, memory, etc).

When considering how to improve the runtime performance for Jikes RVM, we should know that this problem is located in the domain of adaptive systems. Jikes RVM should change its behavior at runtime. One viable solution for this problem is recompiling some hot methods at runtime or we can say that Jikes rebuilds part of itself to change its behavior at runtime (the recompilation work is performed by Jikes RVM adaptive optimization system that is described in section 2.2 and chapter 3).

Subsection 2.3.1 gives more detail about adaptive compilation. Subsection 2.3.2 discusses the features of online and offline adaptive compilation and the difference between them.

2.3.1 Adaptive Compilation

The current version of adaptive optimization system in Jikes is based on the Multi-level optimization strategy (described in subsection 2.2.2). It is a static heuristic (the adaptive optimization system uses a static optimization strategy to recompile the methods). As the compiler's parameters often have 2 or more configuration values (boolean, enum, int, double, etc), so this may result in various different combination of compiler's parameters. There may be a combination of parameters that will lead to better performance for some method.

The best combination of parameters (optimization strategy) will be different in different runtime environment, so a static optimization strategy can not fit this requirement. In addition, the best optimization strategies for the methods that need to be recompiled may also be different.

To reach the maximal optimization, the adaptive system should perform

searching on the hypothesis [7] space where the hypotheses in this space should be a combination of the optimization compiler's parameters. So we need to adopt a machine learning approach to perform the searching. In this thesis, two unsupervised learning algorithms have been evaluated, namely Genetic Algorithm and Random-Mutation Hill Climbing (described in section 2.4).

As mentioned in Chapter 1, adaptive compilation is a trend in compiler developments [AFG+03]. It reduces the complexity of the structure of a modern compiler and improves the efficiency [AFG+00]. The investigation in this thesis (applying a machine learning approach in the adaptive optimization system to improve runtime performance) should be helpful for the future work for dynamic compilation research in Jikes RVM.

2.3.2 Online / Offline implementation

As much research work had been performed on the adaptive compilation domain; we can divide the work into two categories: online optimization and offline optimization.

- Online optimization: this performs the optimization operations at system runtime. Some virtual machine based language have such a mechanism for runtime optimization. One of the typical system is Jikes RVM, the adaptive optimization system can recompile hot methods at runtime.

  As mentioned before, the optimization strategies in the current version of the adaptive optimization system don't search in hypothesis space for the optimum combination of the compiler's parameters. We should apply the intelligent algorithm in it to perform dynamic searching in the hypothesis space. An important problem which needs to be considered is the negative influence on runtime performance when the VM performs the searching process by an intelligent algorithm. It is a trade-off that the improvement got from the optimization should exceed the cost of the optimization operations.

- Offline optimization: this performs the optimization operations during the time that the system doesn't run, but the optimization are based on the profiling information got at system runtime. Many researchers have used this searching approach to find the optimization compilation strategies [A.P98],

[7] A hypothesis is a conception in machine learning; it can be understand as a candidate solution to a problem.

especially in compiling parallel programs. Compared with online optimization, the advantage of this approach is that it needn't consume any runtime resource for optimization, so we needn't concentrate on the trade-off. But the result of the optimization must be got at the next iteration. For a system with a long runtime, this approach is not suitable. In this thesis, online and offline optimization strategies have both been implemented in Jikes RVM's adaptive optimization system (described in chapter 3) and evaluated (described in chapter 4).

2.4 Genetic Algorithm / Random-Mutation Hill Climbing

The aim of the adaptive optimization compilation is finding out the most suitable combination of compiler's parameters for method recompilation. The machine learning approach is a good choice for solving this problem. To perform adaptive optimization with the machine learning approach, we need an intelligent algorithm to perform the searching. As there is not any information that can indicate whether or not the searching direction is correct before we start the searching, the learning process should be unsupervised. Subsection 2.4.1 describes the Genetic Algorithm and the following one, 2.4.2 describes the Random-Mutation Hill Climbing.

2.4.1 Genetic Algorithm (GA)

Genetic algorithms provide a learning method motivated by an analogy to biological evolution. Rather than search from general-to-specific hypotheses, or from simple-to-complex, GAs generate successor hypotheses by repeatedly mutating and recombining parts of the best currently known hypotheses [Mit96]. At each step, a collection of hypotheses called the current population is updated by replacing some fraction of the population by offspring of the most fit current hypotheses. This process forms a generate-and-test beam-search of hypotheses, in which variants of the best current hypotheses are most likely to be considered next. GAs are widely used in problem solving including evolving computer programs, data analysis and prediction, evolving neural networks, etc. The popularity of GAs is motivated by a number of factors including:

- Evolution is known to be a successful, robust method for adaptation within biological systems.
- GAs can search spaces of hypotheses containing complex interacting parts, where the impact of each part on overall hypothesis fitness may be difficult to model.
- Genetic algorithms are easily parallelized and can take advantage of the decreasing costs of powerful computer hardware.

Here is a brief description of this algorithm:

1. Start with a randomly generated population of chromosomes (hypotheses).
2. Calculation the fitness $f(x)$ of each chromosome x in the population.
3. Repeat the following steps until n offspring have been created:
   (a) Select a pair of parent chromosomes from the current population, the probability of selection being an increasing function of fitness. Selection is done with replacement, meaning that the same chromosome can be selected more than once to become a parent.
   (b) With probability p_c(the crossover probability or crossover rate), crossover the pair at a randomly chosen position in the chromosome (chosen with uniform probability) to form two offspring. If no crossover takes place, form two offspring that are exact copies of their respective parents.
   (c) Mutate the two offspring at each locus with probability p_m(the mutation probability or mutation rate), and place the resulting chromosomes in the new population.
4. Replace the current population with the new population.
5. Go to step 2.

As mentioned above, the search performed by GAs is based on an analogy to biological evolution. A diverse population of competing hypotheses is maintained. At each iteration, the most fit members of the population are selected to produce new offspring that replace the least fit members of the population. Hypotheses are often encoded by strings that are combined by crossover operations, and subjected to random mutations.

GAs conduct a randomized, parallel, hill-climbing search for hypotheses that optimize a predefined fitness function, and illustrate how learning can be viewed as a special case of optimization. In particular, the learning task is to find the optimal hypothesis, according to the predefined fitness function.

2.4.2 Random-Mutation Hill Climbing (RMHC)

Hill Climbing is a simple and widely used approach [SP95] in the artificial intelligence domain. As it is easily implemented and understood, it is chosen as another intelligence algorithm used in adaptive system to perform heuristic searching.

Here is a brief description of this algorithm:

1. Choose a chromosome at random. Call this string $best - evaluated$.
2. Choose a locus at random to flip. If the flip leads to an equal or higher fitness, then set $best - evaluated$ to the resulting chromosome.
3. Go to step 2 until an optimum chromosome has been found or until a maximum number of evaluations have been performed.
4. Return the current value of $best - evaluated$.

Although this algorithm does not have the advantages of GAs, RHMC can not perform efficient searching on a complex space of hypothesis (this algorithm is more easily influenced by local minima) and can not be easily parallelized, but RMHC can perform faster searching in the space. We will evaluate the results of tests in Chapters 4 and 5.

Chapter 3

Design and Implementation

In general, the optimizing compiler implementation relies on heuristics (some special optimization policies). They perform well in practice, but require a lot of tweaking to find a good combination of the optimization parameters (the combination of several different optimization policies to gain the best performance). We hope that we can perform searching at runtime, so we need only construct a general optimizing compiler and the adaptive system will perform the searching for a good combination of the optimization policies.

This chapter will concentrate on two parts. The first part is an algorithm design (discussed in section 3.1), including the mechanism of the searching engine and the two searching algorithms (Random-Mutation Hill Climbing and Genetic Algorithm). Section 3.2 describes how to apply the searching algorithms within the current adaptive optimization system in Jikes RVM.

3.1 Algorithm Design

Two optimization algorithms (Genetic Algorithm, Random-Mutation Hill Climbing) based on a basic tree-based searching engine have been used to perform the searching process. Online feedback profiling information was used to perform adaptive optimization. The searching algorithm should be unsupervised (GA, RMHC), because it is not known which combination of compiler's parameters is better or worse before the start of the runtime search.

Subsection 3.1.1 describes the design and working mechanism of the Tree-based searching engine. Subsections 3.1.2 and 3.1.3 describe how to map Genetic Algorithm and RMHC algorithm onto the Tree-based searching engine and some

special problems for each algorithm.

3.1.1 Tree-based searching engine

The tree-based searching engine is the basic infrastructure in the new adaptive optimization system.

Here are some basic issues for the tree-based searchiing engine:

- Major Aim

  The major aim is finding a sequence of optimization compiler parameters that will lead to better performance than before. The searching will be performed in a search space composed of various combinations of compiler parameters. For example, there are 60 boolean parameters (i.e. switches with states on/off or true/false), the number of combinations of these parameters should be 2^{60}. Obviously, we can't test all the combinations and choose one that has the best performance. Because we will perform searching operations on the runtime system, we should find an acceptable combination as soon as possible to gain better performance.

- Searching Risk

  The searching should be more like a random-searching. We just generate a new combination of parameters / DNA based on the previous combination of parameters / DNA, but we can't estimate the runtime performance for the method that was compiled with this new DNA. We can say that there is chance in the searching; if we can't find a good DNA, the runtime performance could be worse than before. How to reduce the searching risk and increase the performance is a trade-off.

- Why use tree-based searching

  Two unsupervised algorithms, RMHC and Genetic Algorithm, were used to perform searching. The two algorithms share a common feature in the basic framework for algorithm implementation, a tree-based framework is suitable for hill climbing like algorithms. So the tree-based framework was chosen as the basic infrastructure to gain the benefits of code reuse and easy extensibility. Another benefit we can get from this framework is that the tree-based searching can be driven by a searching bias towards the direction that can lead to better runtime performance.

type	name	value type	min value	step	max value
V	IC_MAX_INLINE_DEPTH	int	2	1	6

Table 3.1: Example of an encapsulating policy

A tree-based searching approach is used as a basic framework for the searching engine. The sought object is the combination of the optimization compiler's parameters, so we should consider how to express these combinations (encapsulating policy) at first and then the searching process (Per-method optimization and All-method optimization) can be considered.

The two subsections below describe the encapsulating policy for the combinations of compiler's parameters and the searching process, including two searching policies (per-method optimization and all-method optimization).

Encapsulating policy

To use a tree-based strategy, the first problem is how to encapsulate the optimization compiler's parameters as a tree node or how to encode the optimization compiler's parameters as DNA. Each node in the tree is a combination of optimization compiler parameters that is called as DNA, so the tree should be called a DNA tree.

The parameters that need to be encapsulated include such types as boolean, int, float, double, enumerate. The boolean and enumerate types have small number of states, but the numbers of states for int, float and double types using large. I assign the fixed number of states for int, float and double types in my implementation, the number of states is configurable in .dat files. e.g. we can define the parameter is the following way:

the value of IC_MAX_INLINE_DEPTH can be any integer value between 2 and 6 in step of 1 (2, 3, 4, 5, 6) or in step of 2 (2, 4, 6). Why should we use a fixed number of states for each parameter? Because we will perform mutation operation on each parameter, the value state is more easy to understand (a description of the mutation operation occurs in subsection 3.1.2).

Finally, all the optimization compiler's parameters can be encapsulated in an option class OPT_DNAOptions (this class extends from class OPT_Options) that encapsulate the parameter's value and related operation methods. The DNA node objects in the DNA tree should contain OPT_DNAOptions.

Searching process

The search should start from the root node in the tree. The root node encapsulates the basic configuration of the optimization compiler's parameters. When we find that the performance of the method that was compiled by the parameters in the root node can't reach the fitness threshold (to be described in chapter 4), we will search a new node in the tree with breadth-first searching and apply the configuration in the new node to the optimization compiler. If there's no new node, we should generate one with some strategies that will be mentioned below.

There are two different optimization strategies in the searching process: Per-method optimization and All -method optimization. A detailed description of the two searching process will be explained now.

1. Per-method optimization searching:

   There are three method A, B and C, and they were compiled with the optimization compiler's parameters encapsulated in DNA node 1. So the three methods have reference to DNA node 1. (shown in Figure 3.1)

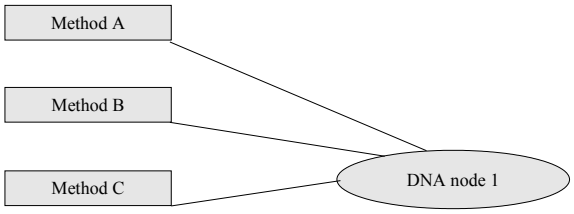

Figure 3.1: per-method optimization: step 1

When method A is considered to be recompiled, we should search for a new node in the current DNA tree. As there's only the root node in the DNA tree, so we should generate a new DNA node 2 from DNA node 1 and method A will be recompiled with options in DNA node 2. (shown in Figure 3.2)

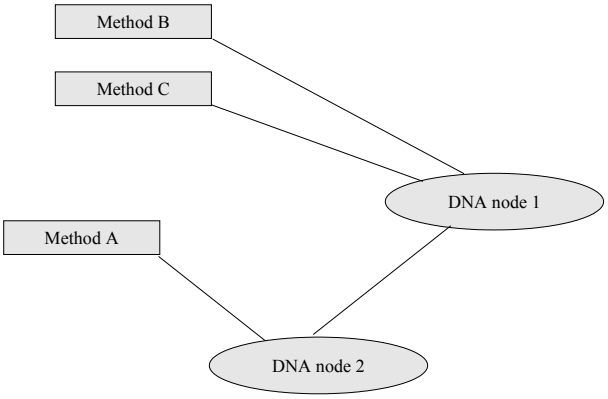

Figure 3.2: per-method optimization: step 2

When method A is considered to be recompiled again and we find that method A has better performance on node 2 than node 1, this means that we should go on with this searching path, so we generate a new node 3 based on node 2. When methods B and C are considered to be recompiled, they perform a breadth-first search on the DNA tree, then node 2 should be the first choice. (shown in Figure 3.3)

When method B is considered to be recompiled again and we find that method B has worse performance on node 2 than node 1, this means that the current searching direction is not suitable and we should go back to the previous node to find a new direction, so we back track to node 1 and generate a new node 4 based on node 1. (shown in Figure 3.4)

When method B is considered to be recompiled again and we find that method B has better performance on node 4 than node 1, then we generate node 5. (shown in Figure 3.5)

The searching progress will stop when the methods' performance reaches the fitness threshold. This means that the searching algorithm converges to an optimum point.

2. All-method optimization searching

   In the description above, each method has its own reference to a special DNA; we can regard it as a Per-method optimization strategy. In contrast,

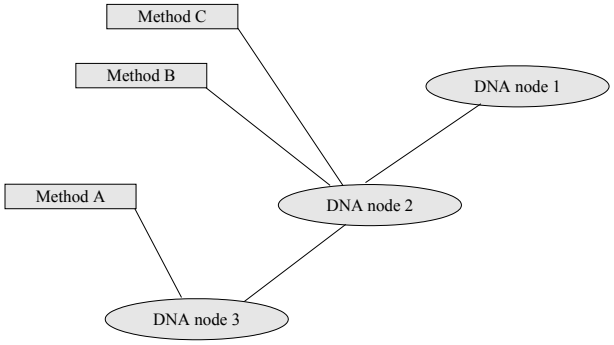

Figure 3.3: per-method optimization: step 3

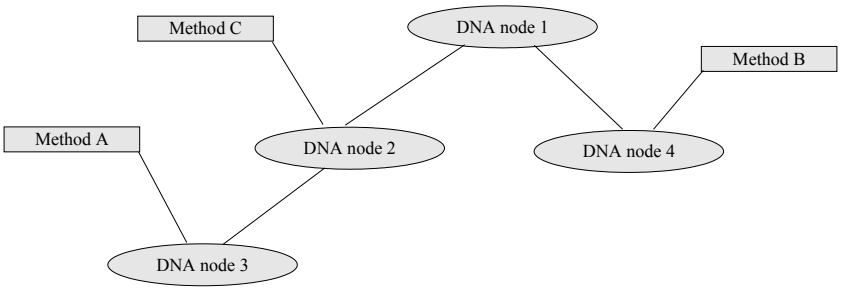

Figure 3.4: per-method optimization: step 4

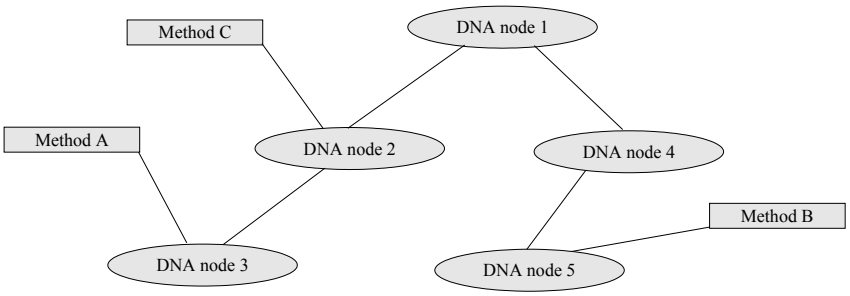

Figure 3.5: per-method optimization: step 5

here all methods have the reference to the same DNA, so it can be defined as All-method optimization.

The example is the same as used to illustrate Per-method optimization searching: There are three method A, B and C, and they were compiled with the optimization compiler's parameters encapsulated in DNA node 1. So the three methods have reference to DNA node 1. DNA node 1 is set as the Current Node. (shown in Figure 3.6)

When method A is considered to be recompiled, we should search for a new node in the current DNA tree. As there's only the root node in the DNA tree, we should generate a new DNA node 2 from DNA node 1 and method A will be recompiled with options in DNA node 2. DNA node 2 is set as the current node. (shown in Figure 3.7)

When method A is considered to be recompiled again and we find that method A has better performance on node 2 than node 1, then we generate a new node 3 based on node 2. DNA node 3 is set as the current node. When method B and C are considered to be recompiled, they get reference to the current node: DNA node 3. (shown in Figure 3.8)

When method B is considered to be recompiled again and we find that

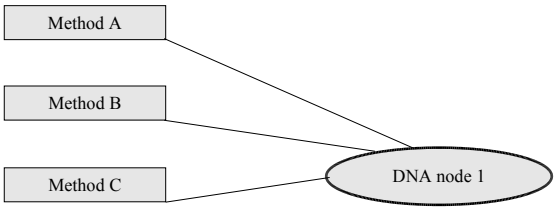

Figure 3.6: all-method optimization: step 1

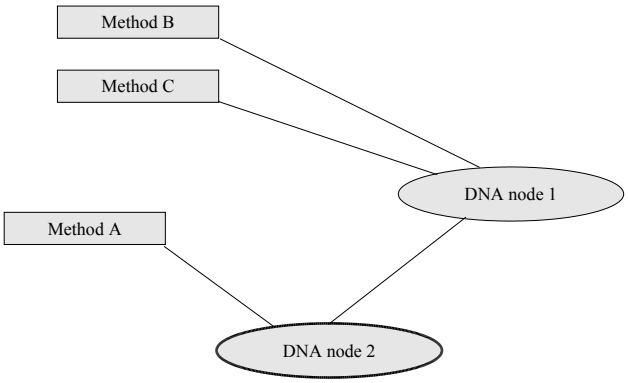

Figure 3.7: all-method optimization: step 2

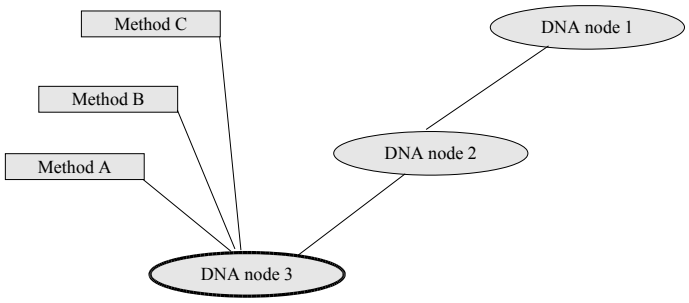

Figure 3.8: all-method optimization: step 3

method B has worse performance on node 3 than node 2, we should back track to node 2 and generate a new node 4 based on node 2. DNA node 4 is set as the current node. (shown in Figure 3.9)

When method B is considered to be recompiled again and we find that method B has better performance on node 4 than node 2, we generate node 5. DNA node 5 is set as current node. (shown in Figure 3.10)

The process will stop when the methods' performance reaches the threshold, then the search algorithm converges to the optimum point.

It is not exactly that all the recompiled methods must be recompiled by the same compiler's parameters / DNA. As we can observe in Figure 3.10, some methods (A and C) that had been recompiled by the previous DNAs reached the performance threshold and couldn't get the opportunity to be recompiled again. So All-methods optimization is an approximate description, as the searching process is going on at runtime, only the hot methods that will be called more frequently will get more opportunity to be recompiled. We may understand the All-method optimization as All-hot-method optimization. It is not worth recompiling methods A and C with the current

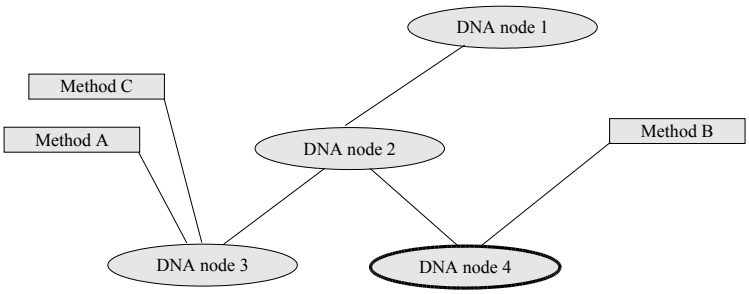

Figure 3.9: all-method optimization: step 4

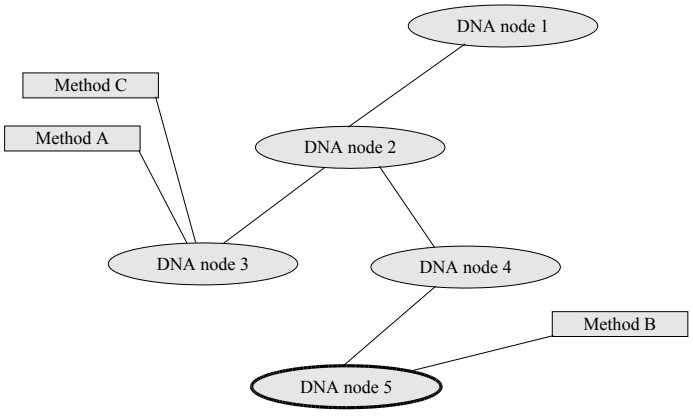

Figure 3.10: all-method optimization: step 5

node (DNA node 5), because we can't estimate the future running performance of the recompiled methods and we should also consider the cost of recompilation.

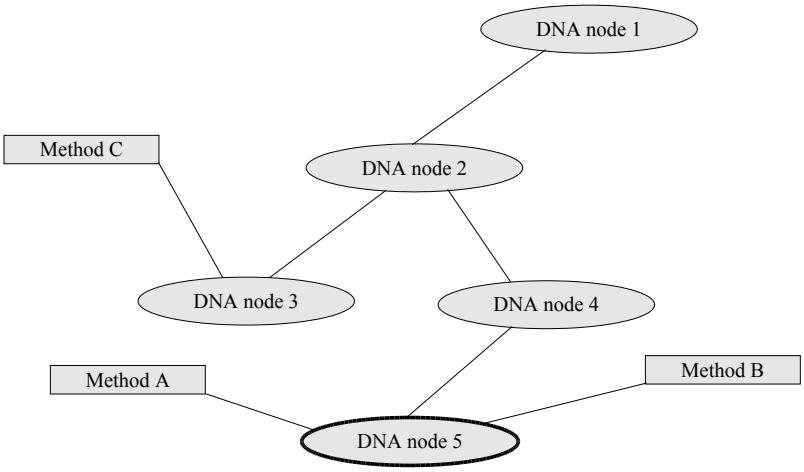

Figure 3.11: all-method optimization: step 6

An important problem is the searching risk in All-method optimization. When method A needs to be recompiled again (shown in Figure 3.11), the current node is node 5; there are 3 different changes between node 3 and node 5, so more luck is needed or the performance may be worse than before. In Per-method optimization, there is less difference between two recompilation operations, because the searching direction is driven by the feedback runtime performance and we just need to generate a new node based on node 3 or back track to node 2 and try node 4. So we can say that All-method optimization has more searching risk than Per-method optimization. Another problem that we should consider for All-method optimization is that if there's not a DNA that is suitable for all methods that need to be recompiled, the searching algorithm will not converge. The adaptive system will perform infinite searching. So we shouldn't require all methods to be recompiled with same DNA node. If one method's performance has reached

the threshold, just stop recompiling it.

The tree-based searching engine is a framework for different searching algorithms in our implementation, so it can be used as a common infrastructure when we implement different searching algorithms. In this thesis, two searching algorithms (RMHC and GA) were applied in this framework.

3.1.2 RMHC implementation

This section describes how to implement the RMHC searching algorithm, including how to map the RMHC algorithm to the tree-based searching engine and the mechanism for generating new DNA node.

A brief description of the algorithm

- *step*1: Choose an initial DNA as the current best-evaluated DNA and evaluate its fitness.
- *step*2: Choose a locus at random to mutate on this DNA.
- *step*3: Evaluate the fitness of the new DNA. If it leads to better fitness, set it as the best-evaluated DNA, if not, give it up.
- *step*4: If the fitness reaches the threshold return the best-evaluated DNA, if not return step 2.

Mapping the algorithm to the tree-based searching engine

As you can observe, RMHC can be easily mapped to the tree-based searching engine:

- For *step*1: Our searching process starts from the root node in the DNA tree, this is the same as choosing the initial best-evaluated DNA.
- For *step*2: When generating a new DNA node in the tree, we perform the mutation operation on the parent DNA node.
- For *step*3: We measure the runtime performance for a recompiled method (the approach used for measurement will be explained in subsection 3.2.2), if it is better than before, we generate a new DNA node based on the node

that is referenced by the method currently; this is the same as evaluating the fitness of the new DNA and setting it as the best-evaluated DNA. If the runtime performance is worse than before, we should back track to the parent node and generate a new DNA node based on the parent node; this is giving up the current DNA.

- For *step*4: if the runtime performance of the methods reach a threshold, we stop searching, if not, we should try a new DNA.

The tree-based searching process is very suitable for *hillclimbing* like searching algorithm.

How to generate the new DNA node (The mutation operation)

When we generate new DNA node, we just need to perform a mutation operation on the parent node to generate the sub nodes.

The mutation operation works like this:

- For a *boolean* parameter: just invert the value. e.g. $true- > false$ or $false- > true$

- For an *enumerate* parameter: select one of the enumerate value at random e.g. there are three enumerate value E_VAL1, E_VAL2, E_VAL3, the parameter's current value is E_VAL1, we just need to select a value from E_VAL2 and E_VAL3 at random.

- For value type parameter ($int, float$ and *double*) : select one of the values between the max value and min value with the predefined step. e.g. the parameter's max value is 6, min value is 2 and step is 1, current value is 4, so we need to select one of the number in {2, 3, 5, 6}; if the step is 2, we need to select one of the number in {2, 6}.

The approach above is the mutation operation on one of the parameters in the DNA. In GA implementation (mentioned later), we use the same approach for mutation operations.

Another thing that needs to be considered is the Mutation rate: this is the number of parameters to be chosen to be mutated in each mutation operation. In my experience, the performance would be best with a mutation rate less than 2% (there are less than 60 parameters in the DNA, so we mutate only one parameter at each mutation operation)

3.1.3 GA implementation

This section describes how to implement the Genetic Algorithm for searching, including how to map the GA algorithm to the tree-based searching engine, the mechanisms for generating the initial population and new DNA node by GA operations and how to estimate the DNA's probability for being selected.

A brief description of the algorithm

- *step*1: Start with randomly generated DNA population[1]
- *step*2: Calculate the fitness $f(x)$ of DNA x in the population.
- *step*3: Perform the evolution operation to generate new DNA by GA operator and replace the old DNA with new one in the population
- *step*4: Go to step 2, until reach threshold.

GA operators:

- *Selection* fitness-proportionate selection
- *Crossover*
- *Mutation* same approach described in subsection 3.1.2

Mapping the algorithm to tree-based searching engine

GA is more complex than RMHC, because the searching in GA is based on the population evolution, but it can still be mapped to tree-based searching engine easily:

- For *step*1 : we should generate a group of candidate DNA as the population for future use, and then start the search from the root node; the root node should be one of the DNA in the population.
- For *step*2 : to calculate the DNA's fitness, we should measure the runtime performance for the methods that are recompiled with the DNA.

[1]The population should be a group of DNAs (like the DNA nodes in RMHC), the number of DNAs is a configurable parameter of the VM. The DNAs in the population will be generated when the VM is booted.

- For *step*3 : if the performance is better than before we should increase the probability value[2] of the current DNA node that is referenced by the method in the population and generate the new DNA node based on the population; if not, we should decrease the probability value of the DNA corresponding to the current DNA node, back track to the current DNA node's parent node and generate new node based on the population. The generation of new DNA node is based on GA operations (the details are given in the next section). All the DNA nodes share the same population (there is a field that refers to the population in the DNA node object, the data structure in memory is shown in Figure 3.12). The difference between each DNA node is their corresponding DNA that will be used for recompiling the method that has a reference to this node.

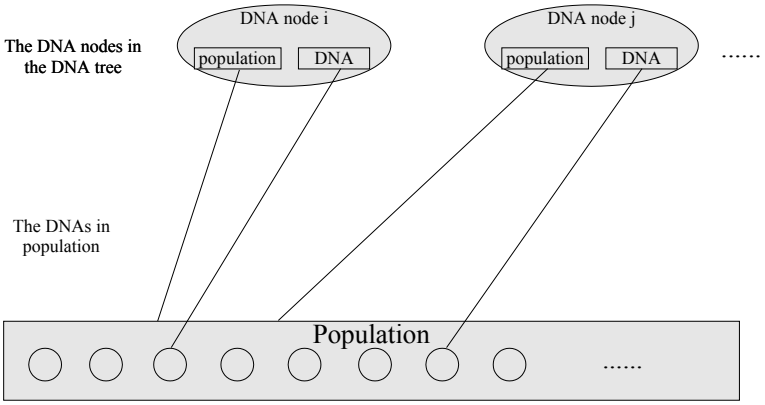

Figure 3.12: Population Reference

- For *step*4 : if the runtime performance for the methods reach the threshold, we should stop the searching process; if not, continue the searching process (back to step 2).

[2]The probability value is an attribute in each DNA in GA. It is used to calculate the probability that the DNA will be selected. The detail about the fitness-proportionate selection will be given below

DNA	DNA_1	DNA_2	DNA_3	...	DNA_m
Probability	p_1	p_2	p_3	...	p_m

$$p_i = 1/m(i = 1..m)$$

Table 3.2: DNAs' probability value

How to generate the initial population

To generate the initial population at first, perform mutation operations on the DNA encapsulated by root DNA node (the initial configuration of the optimization compiler's parameters). The DNA in the initial population all have the same *probability* for being selected. (shown in Figure 3.13 and Table 3.2)

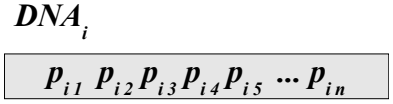

Figure 3.13: DNA structure

How to generate a new DNA node by GA operations

In each time we generate the new DNA node, we select two DNA from the population as the parent DNA by probability and perform *crossover* operation (shown in Figure 3.14) with a predefined *mask string*[3] on the parent DNA to get new DNA. We may perform *mutation* on the new DNA (shown in Figure 3.15) and replace one of its parents with this new DNA in the population. Random crossover was not be implemented, because it would increase the cost of computing resource obviously (it needs to add more loop operations to find the random selected positions in DNA).

[3]The symmetric mask string is used in this implementation, it means that the adaptive optimization system perform symmetry crossover on the two parent DNAs (shown in Figure 3.14). The motivation of such an implementation is to reduce the cost of evolution operation at runtime.

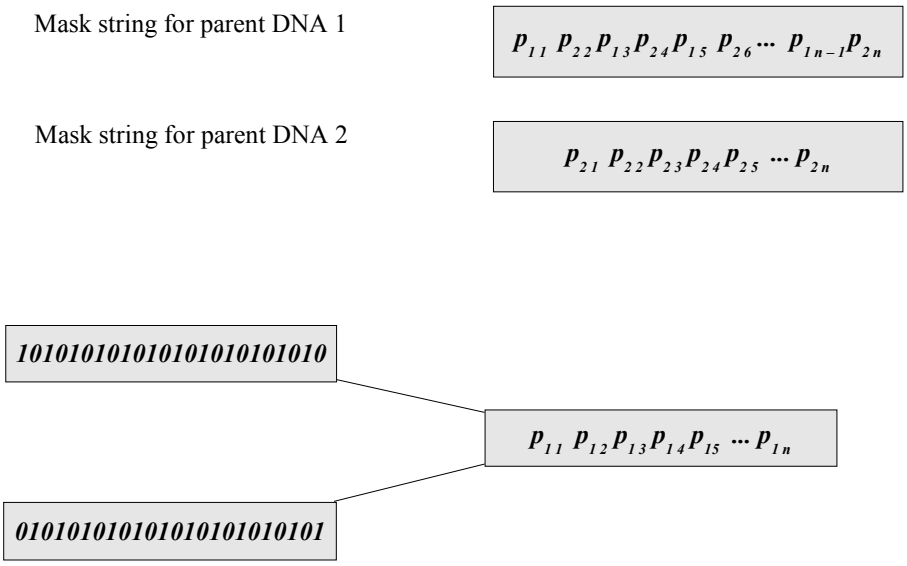

Figure 3.14: Crossover Operation

How to estimate the DNA's probability for being selected

A traditional approach is used to implement $fitness - proportionate\ selection$.

Each DNA_i in the population has a integer attribute v_i and when a DNA is selected from the population, a random number between 0 and $\sum_{j=1}^{m} v_j$, is generated. If the value of this random number is between $\sum_{j=1}^{i-1} v_j$ and $\sum_{j=1}^{i} v_j$, the i^{th} DNA in the population will be selected. The probability that DNA_i will be selected is $v_i / \sum_{j=1}^{m} v_j$.

If a DNA_i in the population can gain better performance, the value of v_i should be increased, this increases its probability for being selected (it means that this DNA should be a good DNA and our searching process prefers the DNA with good fitness, so we should increase the good DNA's probability for being selected), and if not, we should decrease v_i, i.e. decrease the probability.

Hitchhiking problem in GA

GA apply probability on the operators, so the good DNA will gain better opportunity to be selected and the searching process will more easily converge more

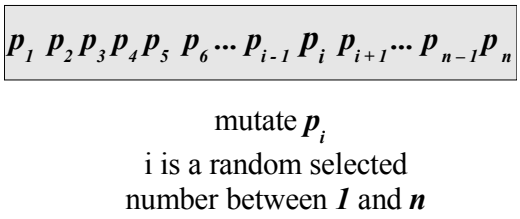

Figure 3.15: Mutation Operation

easy to converge to a good fitness point. But this will result in a Hitchhiking Problem (the details will be explained in chapter 4 combined with the data analysis). In short, once a high-fitness DNA is discovered, its high fitness allows this DNA to spread quickly in the population. This will slow the discovery of the useful change of parameters in other positions. So, hitchhiking seriously limits the search of the GA by restricting the DNA sampled at certain loci.

3.2 System design and implementation

The major ideas for searching algorithm have been discussed above, The detailed issues in practically implementing the ideas in the Jikes RVM adaptive system will be given in this section. Subsection 3.2.1 describes the searching mechanism performed in the Jikes RVM adaptive optimization system. Subsection 3.2.2 describes the profiling mechanism in Jikes RVM. The last subsection, 3.2.3, introduces about the testing and evaluating issues (online and offline iterations).

3.2.1 Searching mechanism

When and how to start the searching

Our searching mechanism is based on the Multi-Level optimization adaptive system1, so we can preserve the benefit from it. Here is a brief introduction to the Multi-level recompilation strategy:

Multi-level recompilation strategy:

- T_i, the expected time the program will spend executing method m, if m is

not recompiled.

- C_j, the cost of recompiling method m at optimization level j.
- T_j, the expected time the program will spend executing method m in the future at optimization level j.
- the *controller* decides to recompile method m at optimization level j, if $C_j + T_j < T_i$; the controller prefers to find the highest optimization level for each recompiled method. e.g. for method A, its current optimization level is opt0, if $C_1 + T_1 < T_0$ and $C_2 + T_2 < T_0$, we will choose opt2 as the new optimization level for recompiling.

In the mechanism mentioned before, we search a higher optimization level for the method that needs to be recompiled, when the controller's criterior prevents use of any higher optimization levels, we can start the searching on the current optimization level (this was the method compiled in last time) to find if there are any optimized combination of the optimization compiler's parameters that can result in higher performance, then the new search should be performed using our tree-based searching engine on the current optimization level.

e.g. there is a method needing to be recompiled, its current optimization level is opt 1 (the compiler that the method was compiled in last time is opt1 compiler), but we can not find a higher optimization level to recompile it (determined by the equation $C_j + T_j < T_i$), so we should start the searching on the DNA tree corresponding to optimization level 1.

In general, there are three optimization levels in Jikes RVM, opt0, opt1, opt2 (like the $O0, O1, O2$ compilation parameters in some C/C++ compiler), so there are three DNA trees in our adaptive system (shown in Figure 3.16).

The different searching policies: RMHC and Genetic Algorithm

As mentioned in subsections 3.1.2 and 3.1.3, these two algorithms can be easy to map in the tree-based searching. In the adaptive system, there is an interface VM_RecompilationStrategy that is used for perform Multi-level recompilation strategy. The implementation of the interface was modified to encapsulate the tree-based searching mechanism described in the previous section and extend two branches (shown in Figure 3.17):

VM_DNARecompileStrategy (correspond to RMHC)

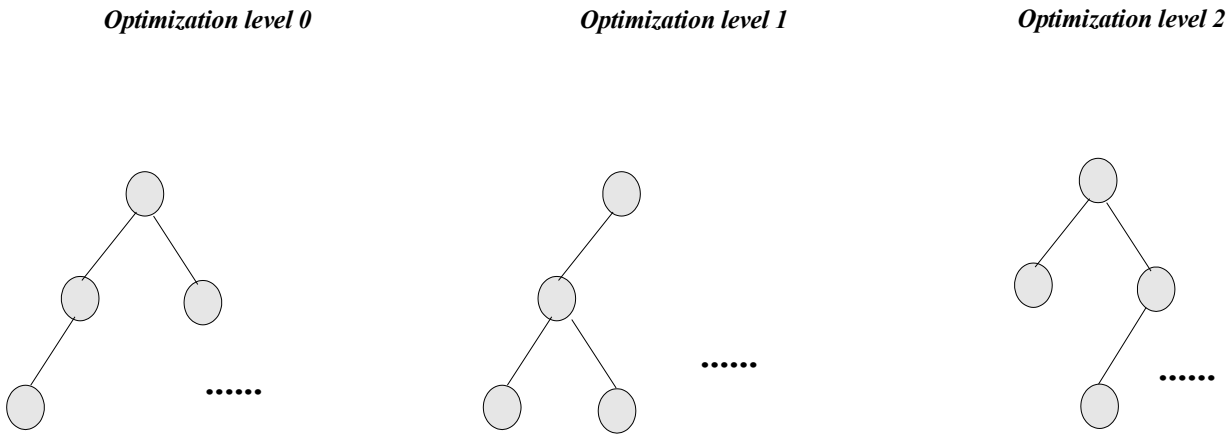

Figure 3.16: DNA trees

VM_GARecompileStrategy (correspond to Genetic Algorithm)

The two new interface implementation classes encapsulate some internal data structure used to implement the different algorithms. A switch was added in the RVM, so the searching policy can be selected when the VM is booted.

Per-method optimization and All-method optimization

Per-method optimization is easy to understand, each recompiled method has its own referenced DNA. For All-method optimization, there are three current DNA nodes corresponding to the three DNA trees, so the methods will reference to same DNA node in its corresponding optimization level. i.e. for method A, its current compiler is baseline compiler (it means that we didn't perform optimization compilation on it), if method A need to be recompiled and the

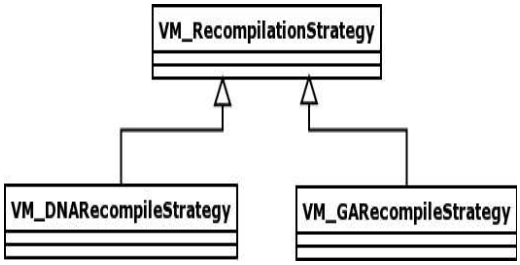

Figure 3.17: Interface Hierarchy

suitable optimization level is opt1, we should recompile method A with the current DNA node in the DNA tree corresponding to opt1. If the suitable optimization level is opt2, the we should use the DNA tree corresponding to opt2.

3.2.2 Profiling mechanism

The adaptive optimization system is driven by the runtime feedback profiling information. The basic framework and mechanism of the runtime profiling (Call-stack sampling) in Jikes RVM have been given in chapter 2. Here is a more detailed description about the profiling mechanism in the new adaptive optimization system, including how to measure the runtime performance, how to generate and store the runtime sampling information, how to make the recompiling decision and how to recompile the method.

Measure the runtime performance

As mentioned in Chapter 2, Jikes counts the method when the thread will be switched. We will preserve this basic measurement mechanism (I can't figure out better mechanism). In general, we collect the method calls in 20 thread switch operations (it is a configurable parameter in Jikes) and this is defined as a time segment (this is an approximate value, because in general the thread will be switched in a fixed time interval, 10 ns, but the threads could be activated / deactivated explicitly in the VM itself and most multi-thread applications, so the time segment will not be regular value). If we find 5 calls made by method A, it means that method A cost 5 / 20 = 25% of whole runtime in one time segment (although it is not very precise). For each recompiled method, we record its

performance in the current time segment and the time segment that the method was recompiled in last time, then the data can be used for comparing in future.

How to generate and store the runtime sampling information

In Jikes RVM adaptive system, a hot method event would be generated for each method that had been captured by the sampling mechanism at thread switching in one time segment (in general, 20 thread switches).

There are two peices of data that need to be recorded: performance in the current time segment and in the previous time segment

- For the performance in current time segment: The class VM_HotMethodEvent is extended to append a new attribute for storing the number of call times, so the controller can get the current performance when it receive the hot method event.

- For the performance in time segment that the method was recompiled in last time: Before the method is recompiled, a *ControllerPlan* that will be used by the optimization compiler for recompiling should be generated and stored in an internal hash table so the status of the recompiled method can be monitored in future. The class VM_ControllerPlan was extended to append a new attribute for storing the number of call times got from the hot method event, so the performance in the time segment that the method was recompiled in last time could be recorded.

How to make the judgment whether or not to recompile the method with higher optimization level or new combination of compiler's parameter (DNA)

To recompile the method with a higher optimization level, we need use the Multi-level optimization mechanism, it make the judgment by the equation $C_j + T_j < T_i$ mentioned before. The cost of recompiling and running the new compiled method should be less than the cost of running the old compiled method. Every method that was captured in one time segment will be evaluated whether or not to be recompiled. To perform searching in the same optimization level (try to find an optimized configuration of compiler parameters / DNA), we should set a threshold (it is a configurable parameter in Jikes) to make a judgment. When one method has more calls than the threshold in one time segment, it should be recompiled,

because it means that the method's performance is not acceptable and it should be optimized. We can't estimate the cost of the performance of the method that was recompiled by the new DNA. We can only evaluate whether or not the new compiled method's performance is better than before by comparing the recorded performance information.

As we have mentioned in this subsection, the performance in the current time segment and the previous time segment have been recorded. Such information can help to decide whether or not to go on with the current searching path (the searching path is the path from the root node to one of the leaf nodes in the tree-based searching engine), if the performance is better than before, we should go on (generate the new DNA node), if not, we should back track to find new direction (back track to the parent DNA node, and generate a new leaf node).

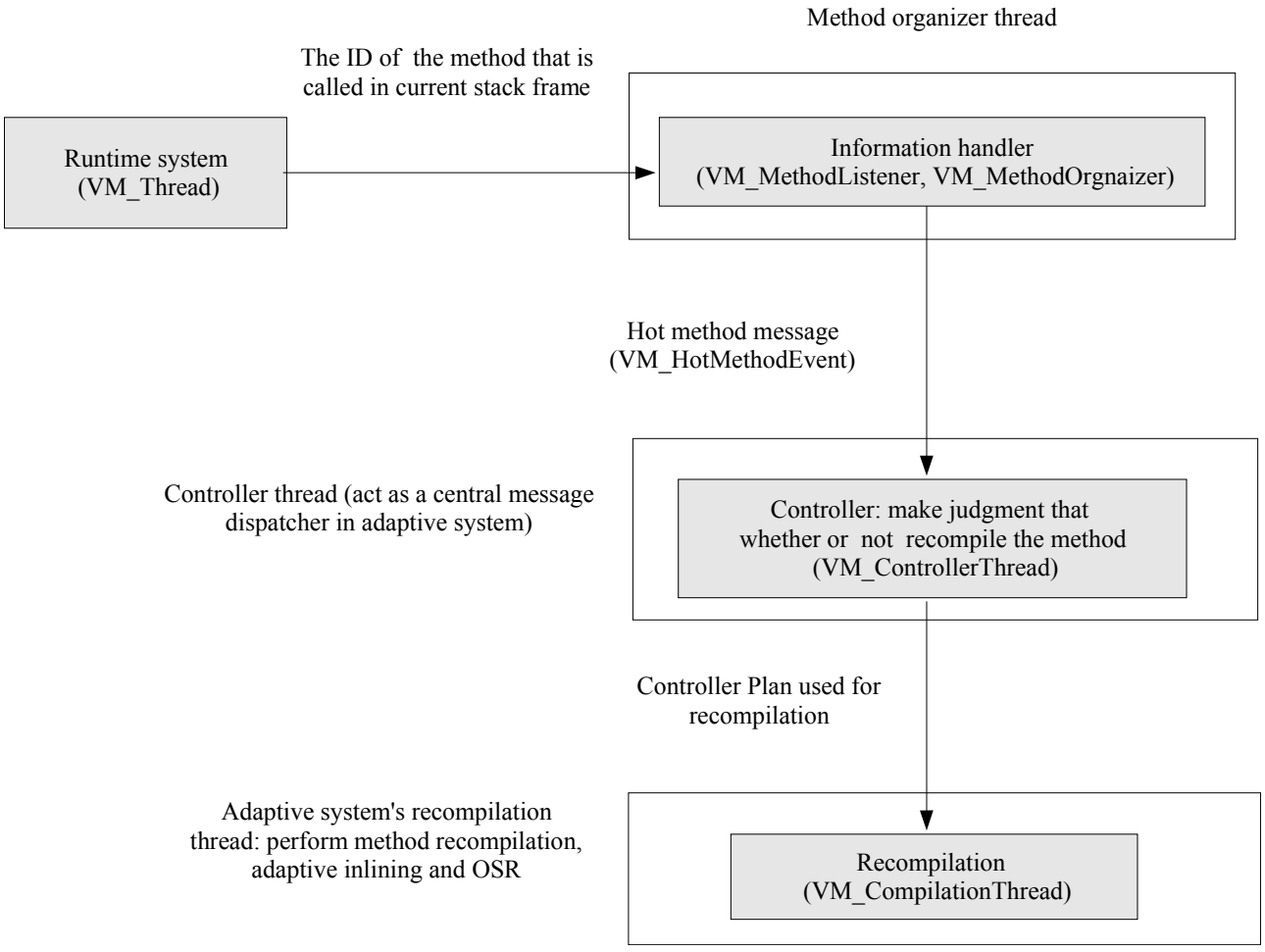

Figure 3.18: Recompilation Process

How to recompile the method

The process from collecting method calls until the end of recompiling the hot method is an asynchronous process, the work load is distributed in three threads that communicate with each other via the shared message queues (Figure 3.18). As there are three process threads, the recompilation process could be divided into three stages.

- At first, the runtime system captures the method call on the thread stack frame when the thread was switched, and records the method ID into a array until it is full (20 times in general, or we can configure it as other value, eg 30, 50 ...). The array that stores the method ID will be sent to the Organizer thread. This thread will count the number of method calls for each method and generate the hot method event, and then send the event message to the Controller Thread.

- The Controller Thread will make a judgment about whether or not the method encapsulated in the hot method event message should be recompiled; if so, it will generate the Controller Plan that will be used by the optimization compiler for recompilation. The searching mechanism (tree-based searching engine with searching algorithm RMHC / GA and the Multi-Level optimization strategy) just happens in this stage. We will use the functions that were encapsulated in VM_DNARecompileStrategy or VM_GARecompileStrategy to make a judgment and generate the Controller Plan. If the method should be recompiled and we generate the Controller Plan, it should be sent to the Recompilation Thread.

- The Recompilation Thread's job is very simple: recompile the method with the Controller Plan, install it into the JTOC, so the new recompiled method will be called next time.

All of the message queues (From Organizer thread to Controller thread and from Controller thread to Recompilation thread) were encapsulated in VM_Controller object that is a shared common data structure used for adaptive system.

3.2.3 Online / Offline iteration

This section explains the term - iteration that will be used in evaluating the results of tests, including the difference between online/offline iteration and how

to perform searching in these two types of iteration.

What is an iteration

We define the iteration as a complete run for a Java application.

The difference between online iterations and offline iterations

- For Online Iterations: we will run a test application several times, but the VM will not reboot each time the test application is run.
- For Offline Iterations: we still run the test application several times, but the VM will reboot each time the test application is run.

Performing the search in these two scenarios

- For Online Iterations: we just define a number of iterations in the Spec applications (Spec jvm98 harness). The VM will not be rebooted between each iteration, so we can go on searching from the first iteration to the last one and get the performance data in each iteration. The DNA tree could be built consistently. When we apply Per-method optimization searching on this scenario, we can find that the .application's performance will be increasingly better and reaches an optimum point when the searching converges after several iterations. If we apply All-method optimization searching, it will not be so easy to converge (the test results and analysis will be given in the next two chapters).
- For Offline Iterations: the VM will be rebooted for each iteration. To continue the search, we should record the DNA each time the VM will be shut down and reload the DNA at the time the VM will be booted again. As restoring the DNA tree and the relationship between the DNA nodes and the methods is a complex piece of engineering, so the All-method optimization is more realistic in this scenario. The *currentnode* defined in the All-method optimization should be stored, but it is not certain that the current node must be a *good* node (the node could result in bad performance, but the searching had stopped in this iteration), so the current node's parent could be a good choice for the DNA that will be stored, although the searching engine may perform more searching in the next iteration. This storage

policy has proved that to be feasible in practical benchmark test. When the VM is booted again, the stored DNA node will be reloaded and assigned as the root node in the DNA tree.

Chapter 4

Experimental Results

This chapter gives the experimental results of the new adaptive optimization system for Jikes RVM (using the intelligent searching algorithm).

Section 4.1 describes the basic configurations of the test bed. Section 4.2 describes the online iteration test, section 4.3 describes the offline iteration test, and section 4.4 gives a summary of this chapter.

4.1 System configurations

The experimental results in this chapter were obtained on an Intel Architecture machine with one 2 Giga Hz Intel P4 processor running SUSE Linux. The system has 512 M memory.

The Jikes RVM boot image was compiled using the optimizing compiler.

- building configuration: FastAdaptiveGenMS (the production build of Jikes RVM on linux platform)
- Adaptive Optimization System configurations:
  - Max optimization level : opt2
  - Initial sample size : 20 (the method sample organizer thread will be activated approximately every 200ms to report the hot methods to the controller)
  - Initial population size : 50 (for Genetic Algorithm searching, i.e. there are 50 DNAs in the population)

SPEC jvm 98		
Name	ID	Description
compress	201	An implementation of the Lempel_Ziv compression algorithm
jess	202	Java expert shell system
javac	213	JDK 1.0.2 Java compiler
mpegaudio	222	Decompression of audio files
raytrace	205	raytracing algorithm
mtrt	227	Variant of a two-thread raytracing algorithm

Table 4.1: Benchmark Items Description

- Iteration times for online iteration : 30

We evaluate the system using the SPEC jvm98 benchmarks [www98]. Table 4.1 provides a description of each benchmark.

4.2 Online iteration test

In the online iteration test, Jikes RVM will not be rebooted until it has finished all the iterations. The searching process will keep going during the Jikes RVM run, then each iteration can get the searching result from its previous iteration except the first one (or we can regard that each iteration can get benefit from its previous one). As we have set the fitness threshold (mentioned in chapter 3), the searching process will converge after several iterations and the remaining iterations' performance should be steady. When the searching process converges, it should reach an optimization point (all the methods recompiled with new DNA have reached the fitness threshold), the least iteration's performance should be better than non-searching iteration (the online iteration performed by the original version of Jikes RVM adaptive optimization system).

Here is an ideal result of test (shown in Figure 4.1). The result of non-AOS (Jikes RVM doesn't perform any adaptive operations) should be a steady line from the first iteration to the last one. The result of non-searching AOS (the original version of Jikes RVM adaptive optimization system) is a curve that drops down after the first few iterations. The searching AOS (the new version of adaptive optimization system that perform runtime searching) is also a curve, but it drops more gradually than non-searching AOS, because of the cost for searching. The searching AOS's performance should be better than non-searching AOS when it

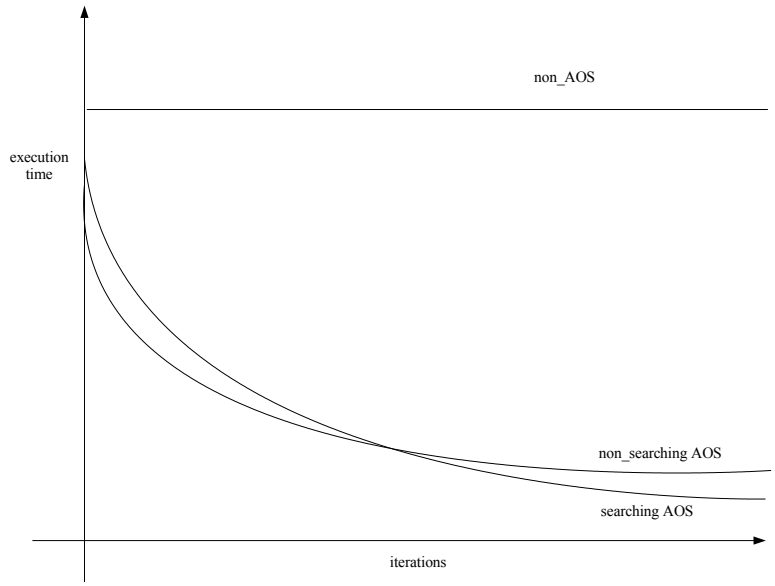

Figure 4.1: Ideal Result

converges (reaches the fitness threshold).

Subsection 4.2.1 will give the practical tests on Spec jvm98 benchmark. Subsection 4.2.2 analyzes the results and compares them with the ideal result.

4.2.1 Benchmark test

4.2.2 Data analysis

For per-method optimization, the results of raytrace (shown in Figure 4.2 and Figure 4.8), mtrt (shown in Figure 4.3 and Figure 4.9), compress (shown in Figure 4.4 and Figure 4.10) and mpegaudio (shown in Figure 4.7 and Figure 4.12) have obvious improvement in performance. But the results of jess (shown in Figure 4.5), have little improvement and javac (shown in Figure 4.6 and Figure 4.11 result is even worse than before.

For all-method optimization, the result of raytrace (shown in Figure 4.13 and Figure 4.16), mtrt (shown in Figure 4.14 and Figure 4.17), compress (shown in 4.15) have obvious improvement in performance. For mpegaudio test (shown in Figure 4.18), the searching process doesn't converge (didn't give the figures). The results of testing for new adaptive optimization system on each benchmark are not always better than the original version. Sometimes they are worse than before (including raytrace, mtrt and compress). To explain this phenomenon, there are

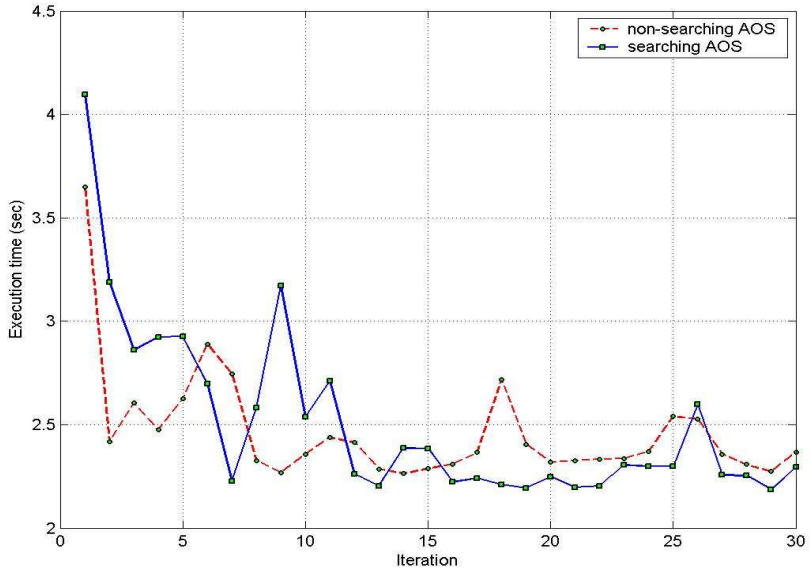

Figure 4.2: Per-method optimization benchmark type: raytrace, fitness threshold = 0.25, searching algorithm: RMHC

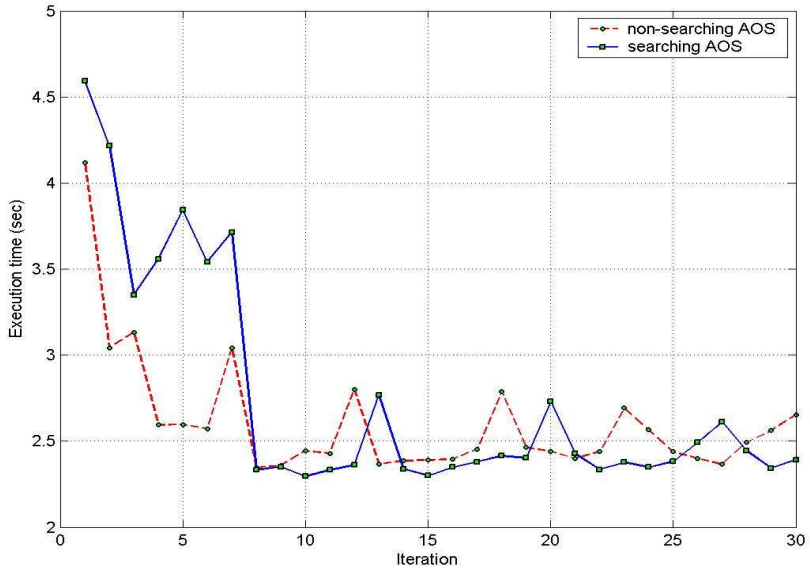

Figure 4.3: Per-method optimization benchmark type: mtrt, fitness threshold = 0.25, searching algorithm: RMHC

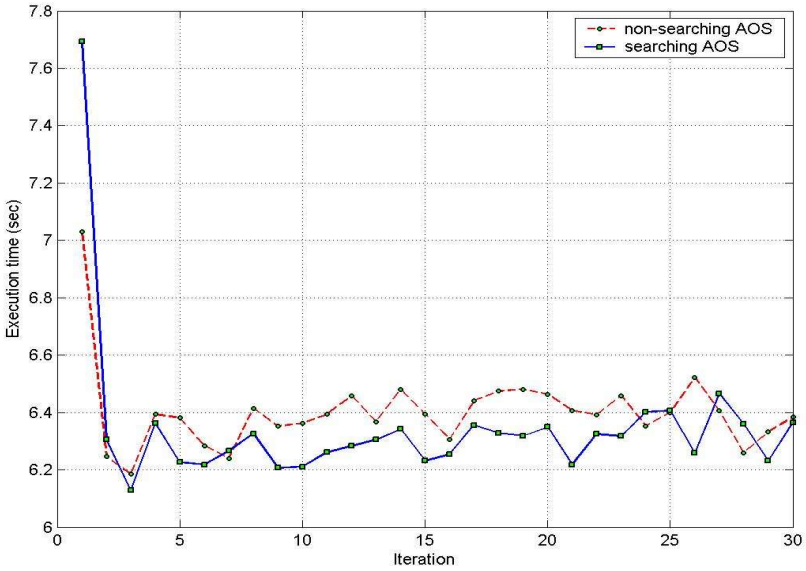

Figure 4.4: Per-method optimization benchmark type: compress, fitness threshold = 0.25, searching algorithm: RMHC

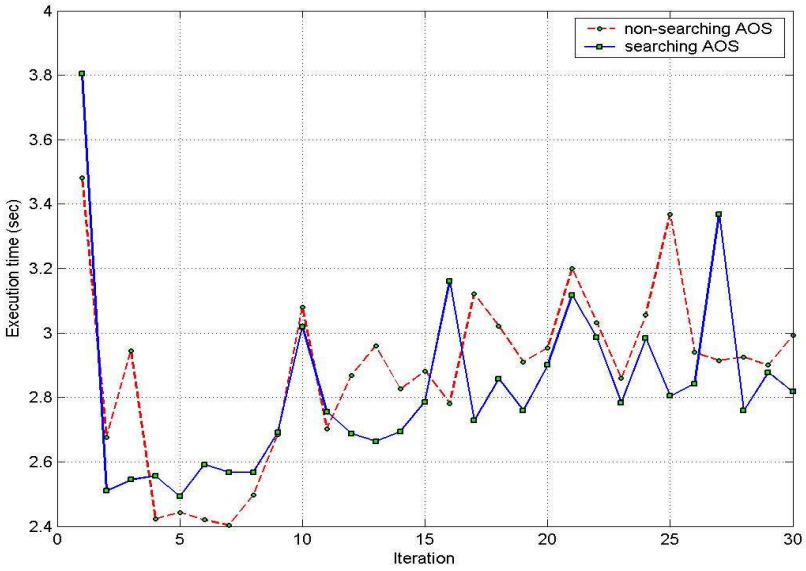

Figure 4.5: Per-method optimization benchmark type: jess, fitness threshold = 0.15, searching algorithm: RMHC

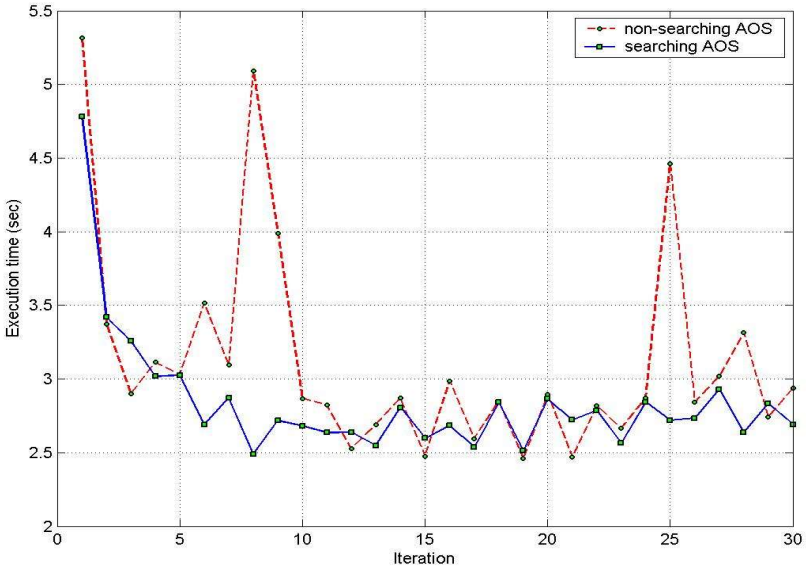

Figure 4.6: Per-method optimization benchmark type: javac, fitness threshold = 0.25, searching algorithm: RMHC

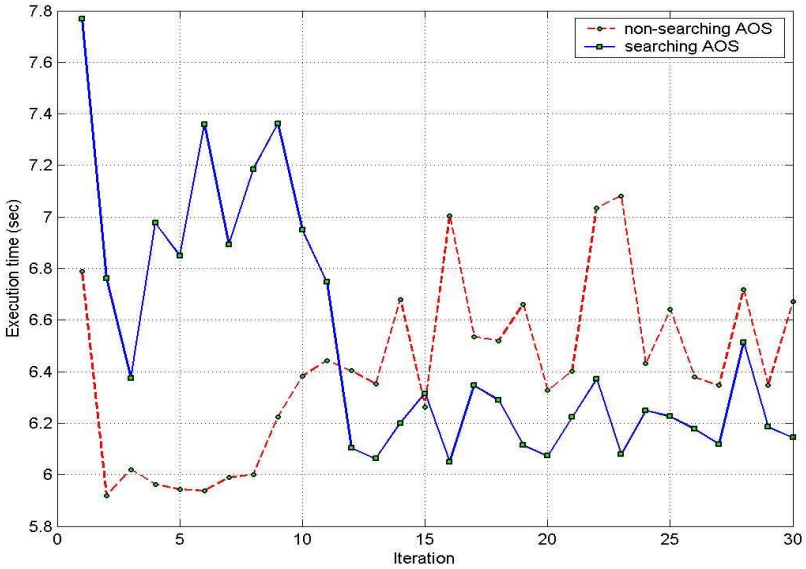

Figure 4.7: Per-method optimization benchmark type: mpegaudio, fitness threshold = 0.3, searching algorithm: RMHC

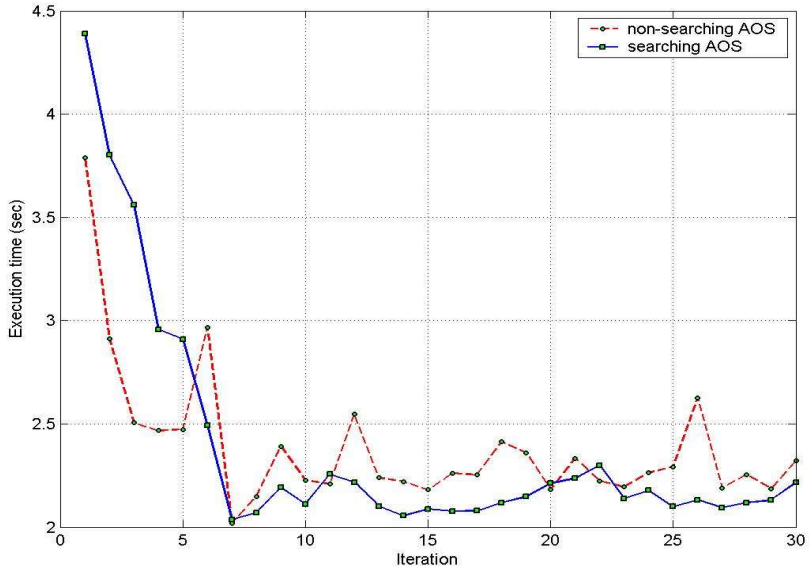

Figure 4.8: Per-method optimization benchmark type: raytrace, fitness threshold = 0.25, searching algorithm: GA

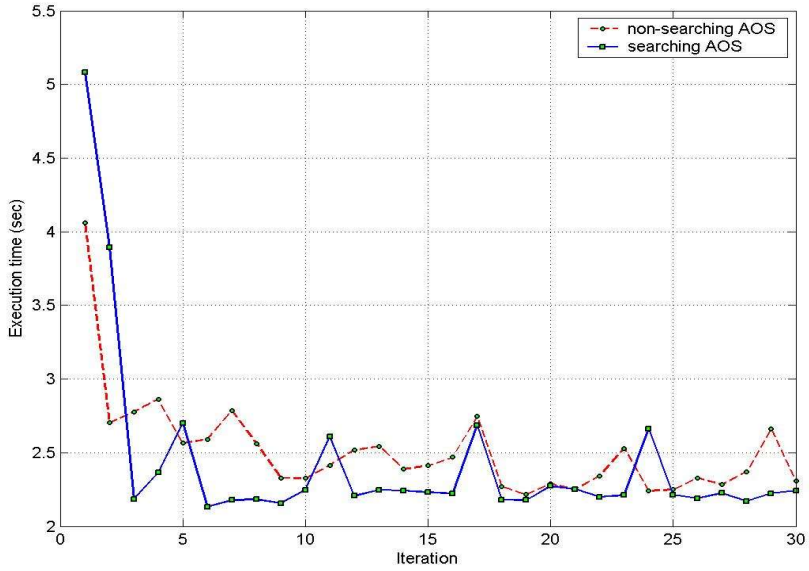

Figure 4.9: Per-method optimization benchmark type: mtrt, fitness threshold = 0.25, searching algorithm: GA

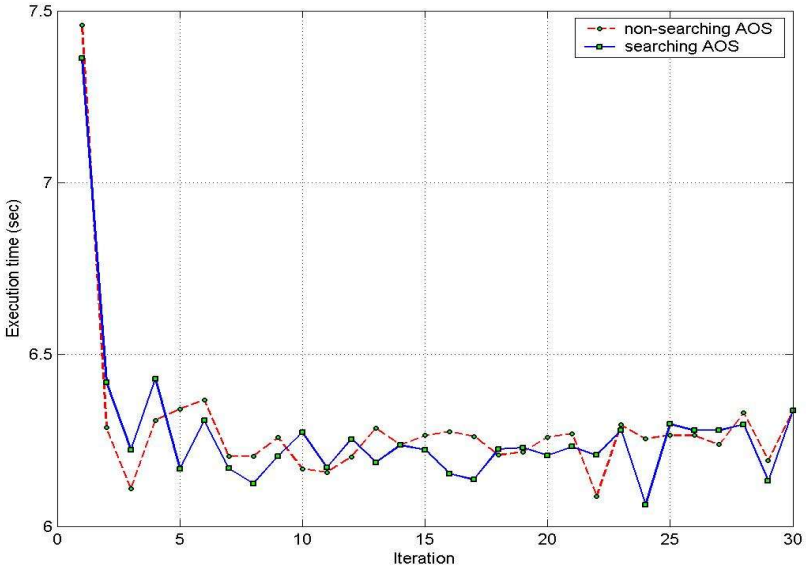

Figure 4.10: Per-method optimization benchmark type: mtrt, fitness threshold = 0.25, searching algorithm: GA

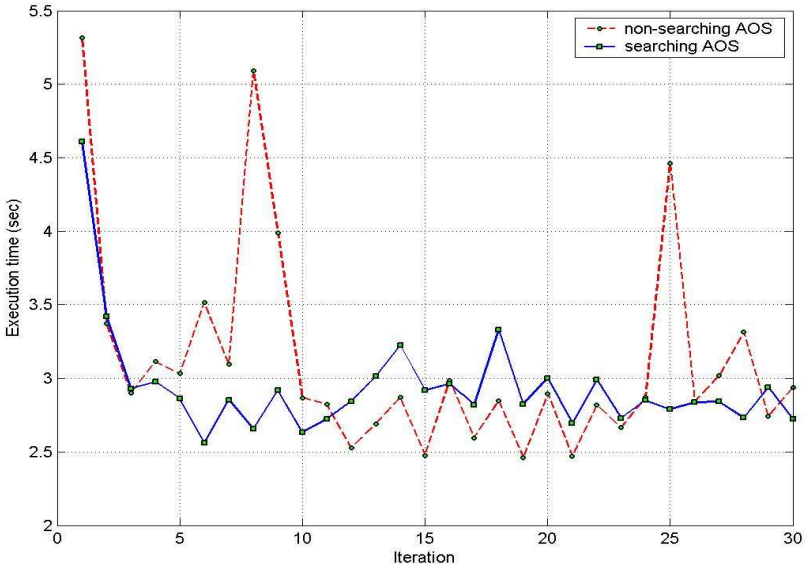

Figure 4.11: Per-method optimization benchmark type: javac, fitness threshold = 0.25, searching algorithm: GA

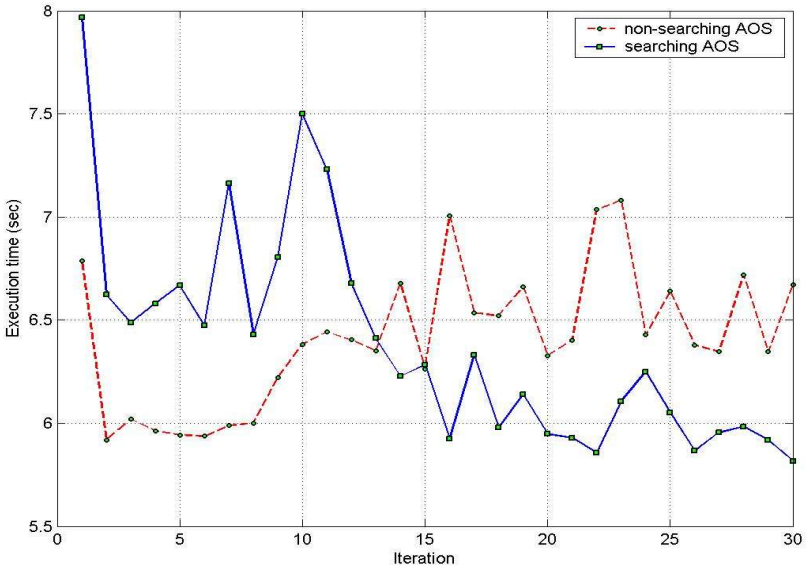

Figure 4.12: Per-method optimization benchmark type: mpegaudio, fitness threshold = 0.3, searching algorithm: GA

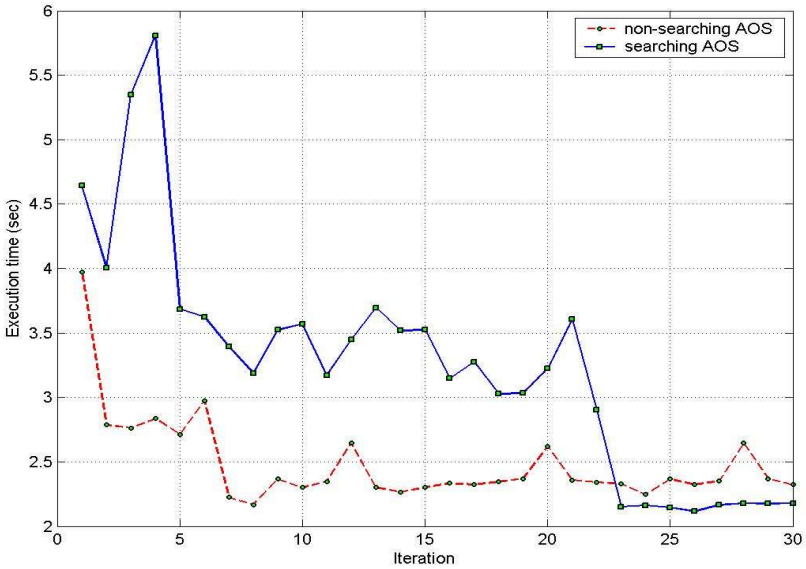

Figure 4.13: All-method optimization benchmark type: raytrace, fitness threshold = 0.25, searching algorithm: RMHC

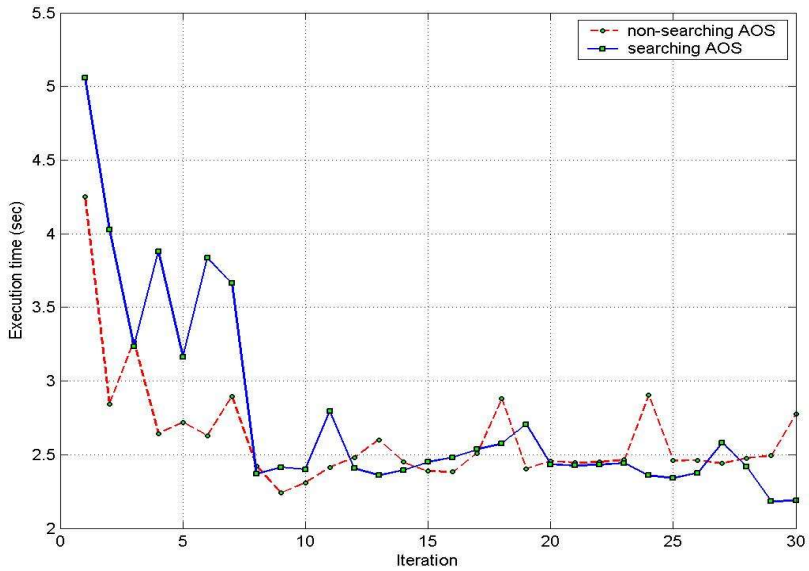

Figure 4.14: All-method optimization benchmark type: mtrt, fitness threshold = 0.25, searching algorithm: RMHC

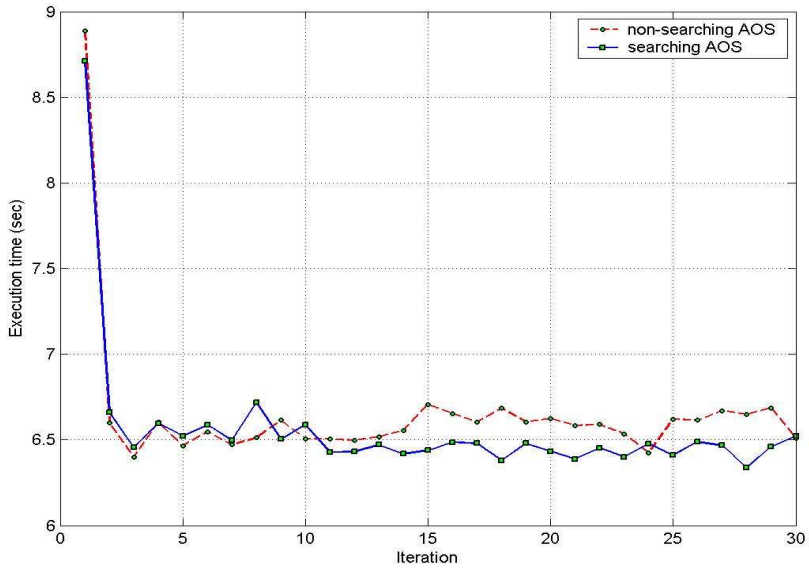

Figure 4.15: All-method optimization benchmark type: compress, fitness threshold = 0.25, searching algorithm: RMHC

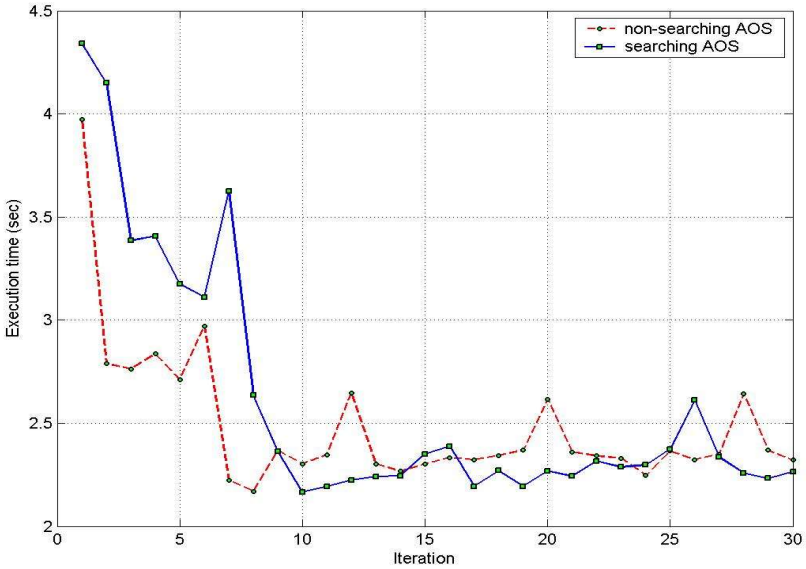

Figure 4.16: All-method optimization benchmark type: raytrace, fitness threshold = 0.25, searching algorithm: GA

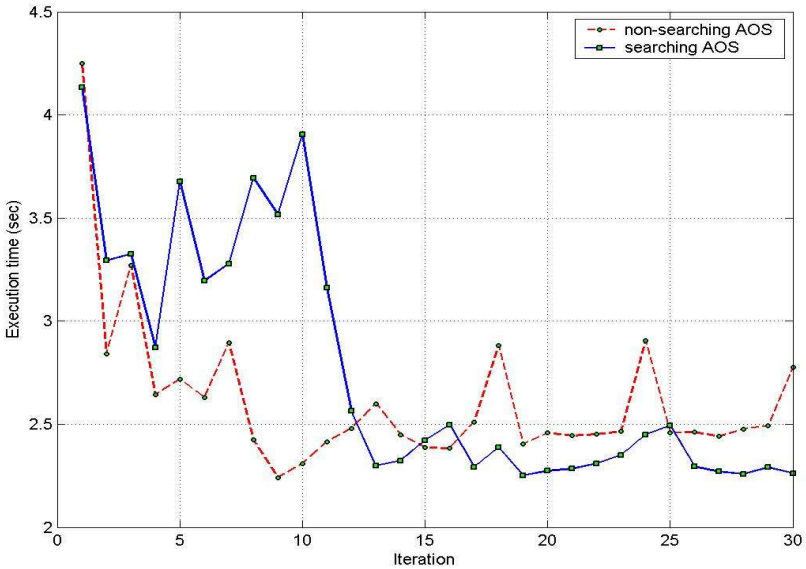

Figure 4.17: All-method optimization benchmark type: mtrt, fitness threshold = 0.25, searching algorithm: GA

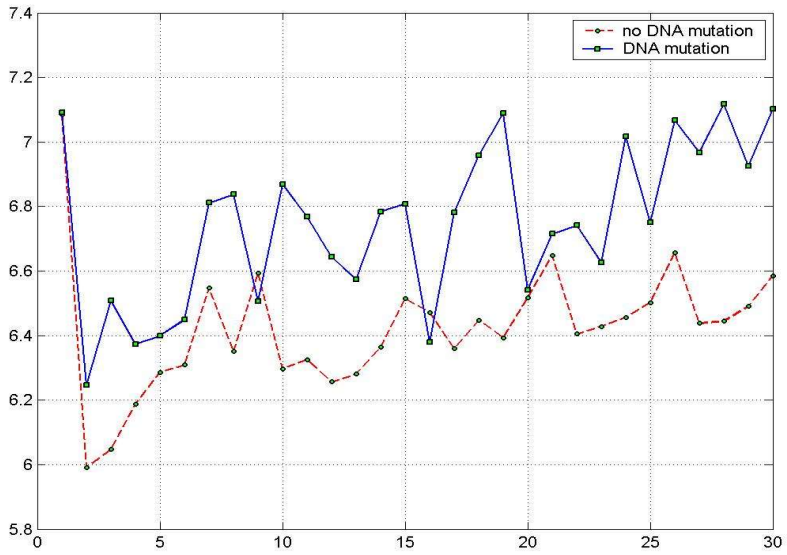

Figure 4.18: All-method optimization benchmark type: mpegaudio, fitness threshold = 0.3, searching algorithm: RMHC

three factors that need to be considered:

1. The speed of convergence: the search is started from a random point (In RMHC, we generate new DNA by performing a mutation operation that is a random process on the parent DNA. In Genetic Algorithm, we select the two parent DNA from the initial population randomly and the final mutation is also a random process). This will lead to different searching risk. The random point may be the first step of a good searching path (each step on this path will lead to better performance, and the fitness threshold can be reached in a few steps) . If so, the search can finish in a few iterations and converge to good DNAs, so the system (including the VM and the application) can gain better performance. If not, the adaptive optimization system will spend a long time searching and be difficult to converge, and this will result in overload of the system.

   In general, RMHC searching converges more quickly than GA searching. As observed in testing, RMHC algorithm converges in 4 10 iteration in average and GA algorithm converges in 6 12 iterations (it is not absolute,

GA searching may start on a good searching path and converge quickly). This phenomena is induced by 'hitchhiking' [Mit96]: once an instance of a higher-fitness DNA is discovered, its high fitness increases its probability for being selected and allows the DNA to spread quickly in the population. This slows the discovery of schemas[1] in the other loci in the DNA. In RMHC, the successive DNAs examined produce far from independent samples in each schema region: each DNA differs from the previous DNA in only one locus (the number of different loci between successive DNAs is determined by mutation rate, we choose one locus in general). It is the constant, systematic exploration, never losing what has been found, that gives RMHC the edge over the GA.

2. The fitness threshold: how to set the value of the fitness threshold in the new adaptive optimization system is a trade-off. Setting the fitness threshold with a very high value, the search will converge quickly, but the adaptive optimization system may not perform enough searching and the system may not gain better performance. Setting the fitness threshold with a very low value, the search may not converge, because the system may not reach such a high fitness. So, the fitness threshold should be selected carefully, then the runtime system can gain higher optimization and reduce the cost for searching. We selected different thresholds for different benchmark tests, most of the values are 0.25, but for mpegaudio, it is 0.3 (There are several functions that were called very frequently in this benchmark item, and they can not reach the fitness lower than 0.3 in most tests. So, we choose 0.3 as the fitness threshold and the system can converge with this threshold in most tests.) and it is 0.15 for jess (this benchmark item can converge easily with fitness value 0.25, but may not gain better performance, because the search stops earliy with such fitness value. So, we decrease the fitness value to 0.15, then the adaptive system can perform more search steps).

3. The local minimum: this is the most important factor that will influence the runtime system's performance. In any searching space defined in the field of machine learning, there should be one global minimum and several local minimum. Most of the searching processes converge to a local minimum

[1]Schemas are meant to be a formalization of the structural properties incorporated by adaptive system. The structural properties are hypothesized to give better performance in some environment .

(the DNAs in DNA tree). If the performance improvement got from this local minimum is counteracted by the cost of searching, the system can not gain better performance. This thesis presents two ideas to reduce the influence of local minima:

(a) Crossover on good leaf nodes in RMHC implementation: a good leaf node is a leaf node that didn't get back tracked in the DNA tree. It means that this node can lead to better performance. When a new DNA node is generated from a leaf node, the adaptive optimization system should check whether there are any good leaf nodes in the DNA tree; if so, it selects one of them randomly, then performs crossover on the parent node and selected node, and finally mutates the new node. This approach can combine the good schema got by different searching paths in the new DNA node, and hence reduce the influence of local minima on a single searching path.

    This approach is used in RMHC implement also. For GA implementation, all the DNA nodes share one population, the DNAs in the population are modified as the searching process going on. The high-fitness DNA has a larger probability to be selected, so the searching engine has the opportunity to get different high-fitness DNAs to generate new DNA.

(b) Fitness threshold decay: the searching process will stop when the recompiled methods' performance reach the fitness threshold. But the searching process may converge to a local minimum. The fitness threshold decay approach decreases the threshold by 0.05, when the searching process converges the first time. In this way, the adaptive optimization system will get an opportunity to restart the searching process and overcome the current local minimum to find a better optimization point.

Another issue that needs to be considered is the difference between per-method optimization and all-method optimization. As mentioned in chapter 3, all-method optimization has more searching risk than per-method optimization. The all-method optimization may require the methods to be recompiled with the same DNA. Studied from the testing, there are two issues that can lead to bad performance:

1. If there's not a DNA that is suitable for all method that need to recompiled, the searching algorithm will not converge. The searching process goes on all the iterations and the system will be overloaded. We can observe this on mpegaudio benchmark test (shown in Figure 4.18.

2. During the searching process, the search may reach a new DNA node that will lead to bad performance in future and set it as the current node (mentioned in chapter 3). All the following methods that need to be recompiled will use this current node's DNA to be recompiled, so this bad DNA will increase its influence to a larger scope. The performance of most of the methods recompiled with this DNA will be worse than before, they may require to be recompiled again (the adaptive optimization system generating more hot method events for these methods). All these scenarios will increase the system's work load and slow the speed of convergence, if the searching process can converge.

But on the other hand, the last issue above can also lead to better performance, if the new DNA node generated by the searching engine will lead to improvement in performance for most of the following methods that need to be recompiled. The searching process can then converge quickly. As there are many factors in the searching process, we can't point out which must be better than another in the two types of optimization.

4.3 Offline iteration test

Compared with online iteration, the Jikes RVM should be rebooted between every iteration in an offline iteration test. The search result should be preserved between every iteration, then one iteration can get the search result from its previous iteration. When a run finishes, Jikes RVM stores the search result (a DNA node) for each DNA tree into a file (the storing policy has been described in chapter 3). The stored DNA nodes will be reloaded in the next run and act as the root nodes in each DNA tree. The offline iteration is based on all-method optimization (mentioned in chapter 3), because it is easy to implement.

Subsection 4.3.1 gives the results of benchmark test on raytrace, compress, mtrt and javac. Subsection 4.3.2 analyzes the data.

4.3.1 Benchmark test

There are 15 iterations for each benchmark item. Here are the results for each benchmark item.

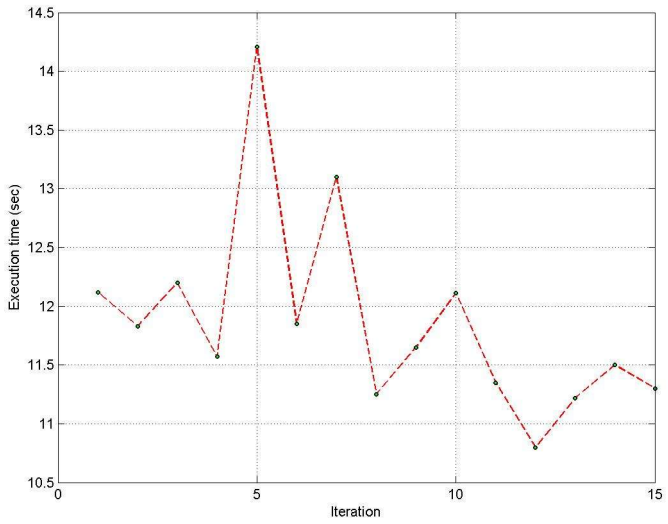

Figure 4.19: benchmark type: raytrace, searching algorithm: RMHC, fitness threshold: 0.25

Comparing the search result (an optimum DNA) to non-searching AOS, here is the results (shown in Figure 4.23), for each benchmark test (comparing the results of original configuration and the new DNA got by searching process).

4.3.2 Data analysis

The results of raytrace and compress have obvious improvement in performance. The results of mtrt and javac have little or even no improvement in performance. They didn't find the optimal DNA and need more search iterations.

The effect of the factors mentioned in subsection 4.2.2 (the speed of convergence, fitness threshold, and local minimum) is the same as online iteration.

In the figures for searching iterations, we can observe that there are great variations between each iteration and the performance of one iteration may be worse than its previous iteration. The reason could be explained by the difference between online and offline iteration. There are two differences that need to be considered:

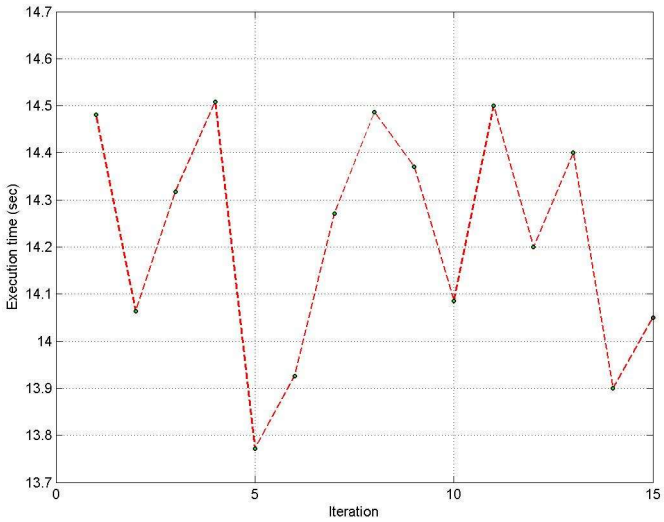

Figure 4.20: benchmark type: compress, searching algorithm: RMHC, fitness threshold: 0.25

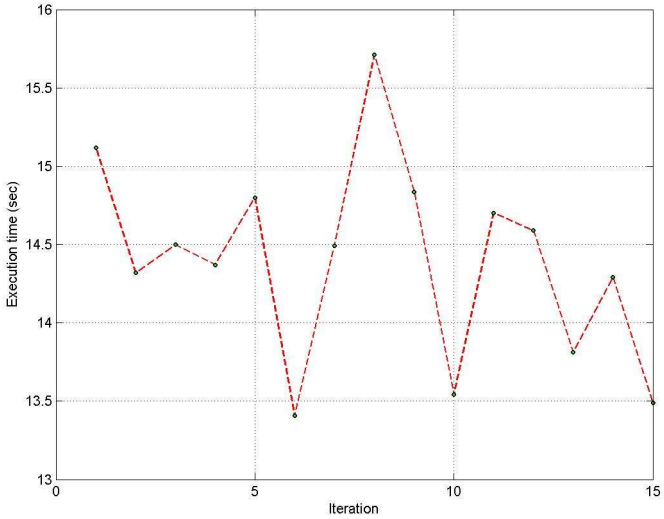

Figure 4.21: benchmark type: mtrt, searching algorithm: RMHC, fitness threshold: 0.25

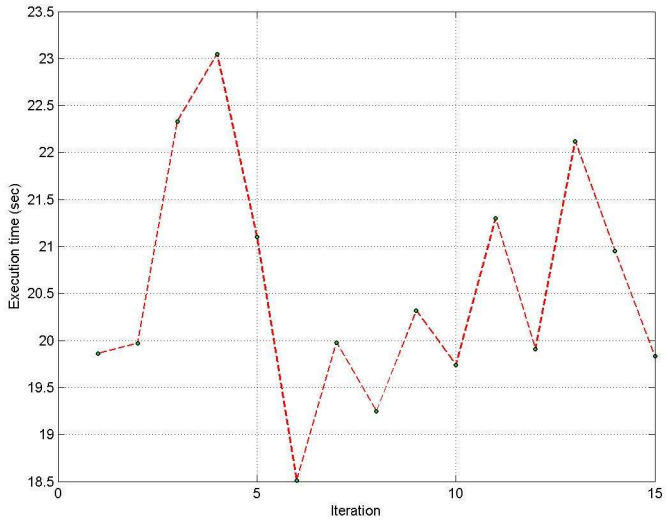

Figure 4.22: benchmark type: javac, searching algorithm: RMHC, fitness threshold: 0.25

1. For online iteration, the searching process will not stop until all iterations have been finished. But the searching process stops between each iteration for offline iteration.

2. Offline iteration uses the all-method optimization.

Because of the searching process stopping between each iteration and all-method optimization, the stored DNA node will act as the root node in the next iteration, then all the methods that need to be recompiled will be required to use this DNA when the new iteration is started. If this DNA leads to bad performance for most of the methods that need to be recompiled, the system performance will degrade, because the search starts from a bad DNA and this will cost more computing resource to find better DNAs. So, offline iteration has the same type of searching risk as online iteration with all-method optimization, and even larger than the latter's. Although we use a smart policy (described in chapter 3, we choose a good leaf node's parent node, this means that this node must always lead to better performance for some method, as the new DNA node was generated based on it) to choose the DNA node to be stored, it is likely to be true that many of the methods were compiled with earlier DNAs rather than the selected one. For online iteration, only the following methods that need to be recompiled will be influenced when the searching process finds a DNA that

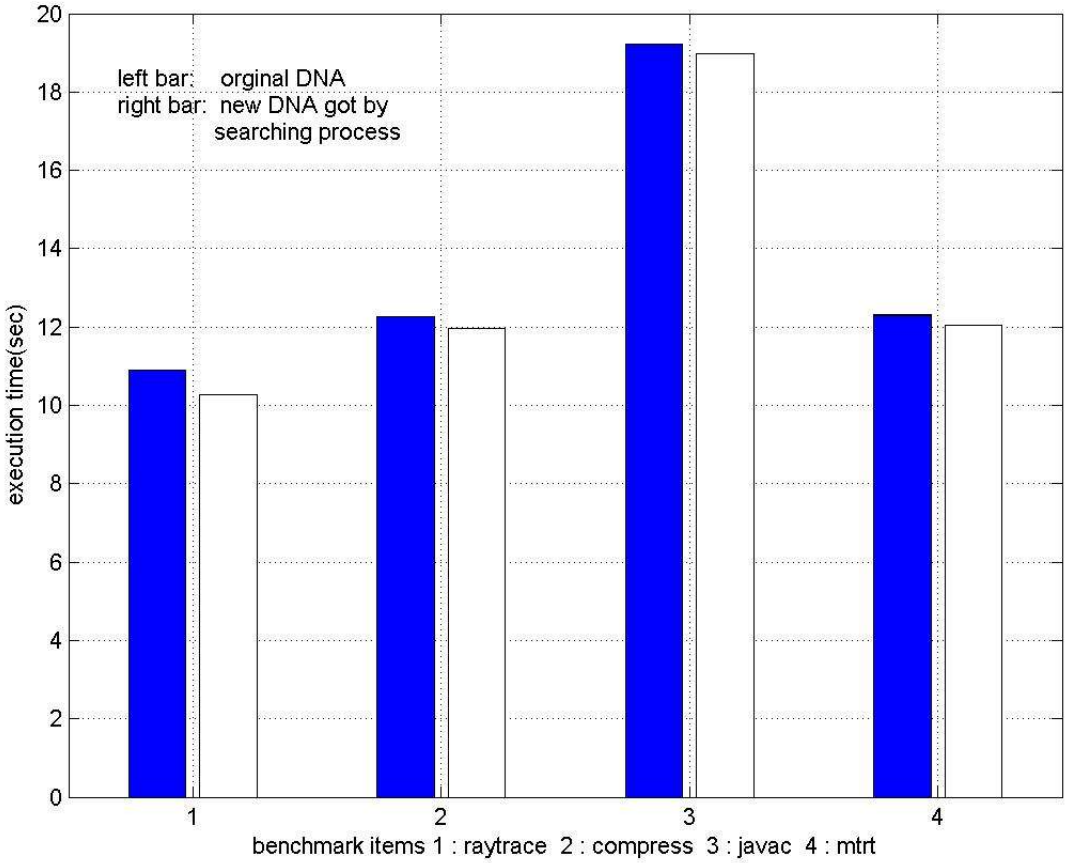

Figure 4.23: Comparing the Performance

will lead to bad performance, and this searching risk can be fixed quickly (if the searching process can converge), because the searching process is going on, the adaptive optimization system will try a new DNA when a method with bad performance needs to be recompiled.

4.4 Summary

This chapter has evaluated the implementation of the Jikes RVM adaptive optimization system with intelligent optimization algorithms.

The experimental results have shown that the new searching mechanism in the adaptive optimization system can work correctly. Both RMHC implementation and GA implementation can converge to an optimal point where the system (VM and application) could gain better performance than before.

Chapter 5

Conclusion

This thesis presents the design and implementation of adaptive optimization system (AOS) with intelligent optimization algorithms in the Jikes RVM Java Virtual Machine, evaluates its performance with benchmark tests and analyzes the results.

A summary of the thesis and its contribution is given in section 5.1, followed by a critical review of the overall approach in section 5.2. A brief description of future work is given in the 5.3.

5.1 Summary

Chapter 1 described the demand for applying intelligent optimization algorithms within the Jikes RVM adaptive optimization system to check whether or not it is suitable for improving the runtime performance.

Chapter 2 gave the background knowledge related to the research work in this thesis. It included the organization of Jikes RVM Java Virtual Machine, the architecture and working mechanism (Multi-level optimizing strategy) of Jikes RVM adaptive optimization system, the concept of adaptive optimization, and introduction of Genetic Algorithm and RMHC Algorithm.

Chapter 3 described the design of the tree-based searching engine and its two optimization strategies (per-method optimization and all-method optimization), gave the details about how to map the Genetic Algorithm and the RMHC Algorithm onto this searching engine to perform searching at runtime with these intelligent algorithms, including the searching mechanism and profiling mechanism. Some issues in engineering were also given here, including encoding the

DNA, implementing the mutation operation, implementing fitness-proportionate selection, etc. Finally, two issues related to testing (online / offline iteration) were described.

Chapter 4 gave the experimental results of some benchmark tests, compared the difference between the RMHC implementation and the GA implementation, using per-method optimization and all-method optimization, with online and offline iteration. Possible reasons were given to explain these differences.

The overall contributions of this thesis are the new adaptive optimization system with intelligent optimization algorithms (RMHC and GA). This new adaptive optimization system enables the Jikes RVM to perform more intelligent adaptive optimization and gain improvement in performance. In addition, we evaluated the results of benchmark test and analyzed the factors that affect the searching process.

To construct an efficient adaptive optimization system, there are three important factors that need to be considered:

1. An efficient profiling mechanism

   Here 'efficient' means that the mechanism must be precise for evaluating and doesn't cost much computing resource. But these two issues are a trade-off in practical implementation; if we want to get a more precise profiling mechanism, we need to implement a fine-grain profiling mechanism and this will increase the system load (i.e. if we need count the calls for each method precisely, we need to add a small code stub at the method's prologue or epilogue, and this will result in obvious influence on runtime performance). As mentioned in chapter 2, Jikes RVM uses call stack sampling to implement a profiling mechanism for hot method recompilation. It is a trade-off, because the sampling is performed at thread switching and doesn't cost much. But this also decreases the precision; there may be several methods that were executed during one interval of thread switching. So, we can not evaluate the performance of recompiled methods precisely. This problem may be solved by the aid of hardware implementation in future. i.e. the micro-processor can be expanded to add some special registers for performance monitoring.

2. An efficient searching algorithm and related implementation of the algorithm The searching algorithm is the basic mechanism for the evolution of

an adaptive optimization system. Its efficiency is determined by two factors defined in machine learning, namely search space and searching bias.

- search space: is a space which contains all the possible hypotheses that will be evaluated by the searching process. The size of search space will affect the speed of search. It is easy to understand that the searching process can reaches its aim faster in a smaller search space. Selecting a suitable search space will be helpful for implementing an efficient adaptive optimization.
- searching bias: can be comprehended as a regulation for searching. This concept is mainly used in supervised learning. A stronger bias will increase the speed of the searching process, and a weaker one will decrease the speed (i.e. if there's no bias, the searching process has to try all the hypotheses in the searching space).

In this thesis, we choose the RMHC algorithm and Genetic Algorithm as the searching algorithms which are unsupervised learning algorithms, so we need only to consider the size of the search space. The search space is limited to all the possible combinations of the optimizing compiler's parameters in corresponding optimizing. Supervised learning was not applied within the implementation, because the performance of a recompiled method can not be evaluated before it finished running.

3. An efficient mechanism to change the behavior of the system itself In Jikes RVM, the behavior was changed by recompiling some methods in the application or the virtual machine itself, so the performance for binary codes can be improved. There are several independent threads that are used for adaptive recompilation; this approach reduces the effect on the application's performance.

5.2 Critique

One criticism is that the new adaptive optimization system preserves the Multi-level optimizing strategy. One reason for this is to reduce the search space mentioned in last section. In the current design, the search space is limited in the scope corresponding to an optimizing level; the searching for a new DNA that will

be applied to the recompiled methods in one optimizing level will be performed in the search space corresponding to that optimizing level. A larger search space would result in more or larger DNAs and, therefore, become expensive to process. Another reason that we limit the size of search space is that the larger search space will also increase the searching risk, because searching may jump from one optimizing level to another.

In the original version of the adaptive optimization system, there is a static configuration table in VM_CompilerDNA that collects the performance improvement rate for each compiler optimization. The searching engine can find the most suitable optimizing level for the methods that need to be recompiled, so the searching needn't jump to a lower optimizing level. e.g. suppose one method recompiled with opt level 1 needs to be recompiled again, the searching engine generates a new DNA that has opt level 0 and uses this DNA to recompile that method, the new recompiled method is very unlikely to gain any improvement on performance or may be worse than it before, so the system wastes the computing resource for recompilation. Jumping from low opt level to a higher opt level (this opt level is higher than the opt level selected by the multi-level strategy) may be feasible, but in a practical test, recompilation with such a higher opt level may not result in better performance and is quite expensive to perform. (As observed in testing, most of the searching concentrates in opt level 1; there is no obvious improvement in performance for most of methods that have been recompiled with opt level 1 and will be recompiled with opt level 2, but need more computing resource than opt level 1 does). So we didn't choose this scheme.

Another criticism is the precision of the sampling mechanism. As call stack sampling can not capture all the method calls at runtime, we may not evaluate the performance of the methods that need to be recompiled precisely. The recompiled methods' performance is a very important factor in the new adaptive optimization system; it will determine whether or not to go on with the current searching direction and whether or not to stop the searching. If the searching engine gets the wrong performance data, there are two bad consequences:

- The searching may be directed in the wrong direction that will waste more computing resource and slow the speed of convergence.
- The searching stops early before it has converged, and then the system doesn't reach an optimal point even a local minimum.

The last criticism is the machine noise; this issue will affect the testing and generate the variation in testing results. To avoid such problems, we can stop other applications (i.e. mail client, document editer, etc) and some system services to reduce the effect of system scheduled operations (the best solution should be using single user mode that has least noise, but this approach wasn't used in the benchmark test, because the RVM system is stored on network disks).

5.3 Future work

In this thesis, using intelligent optimization algorithms to perform adaptive optimization has been proved to be a feasible approach to improve the Jikes RVM runtime performance. Currently, the new adaptive optimization system only concentrates on finding an optimal combination of optimizing compiler's parameters. But this approach can be expanded to such areas:

- Parallelize system optimization: in a parallel execution environment, there are more heuristics that need to be explored to optimize. i.e. we may need to explore a combination of loop interchange, loop distribution, loop reversal and loop fusion, that can be used to optimize a parallel application. A dynamic parallel compilation with adaptive optimization enables development of single-source application which runs on widely differing target multi-processor platforms and gains good performance on such platforms.

  In the JAMAICA project, Jikes RVM has been ported to a chip multiprocessor environment, and we can develop more optimizing compilation phases for parallelizing optimizations in the VM and then use the adaptive optimization mechanism to optimize the application running on such a platform.

- Adaptive optimization for the operating system: the adaptive optimization function can enable the operating system to adapt to different hardware configurations and different types of application. i.e. For the different platforms that have different numbers of processors, the adaptive OS will use different scheduling strategies to maximize the use of all processors and gain the best performance; for the different types of application (some of them concentrate on computing and others may concentrate on I/O operations), the OS will also change the scheduling strategy to assign more

processor resource to the application that concentrates on computing. All the strategy selection should be driven by adaptive optimization (using an intelligent algorithm to find an optimal strategy at runtime). For future research, Java OS may be increasingly used on server side applications (especially in multi-processor environment) or even client side. The dynamic parallel compiler with adaptive optimization functions mentioned above is very suitable to be used as the just in time (JIT) compiler for the Java OS to perform dynamic compilation. The Java OS can also perform dynamic recompilation (just like Jikes RVM) to change its behavior at binary code level.

- Adaptive optimization for large business application that has complex mechanism: the large business application often encapsulates large amount of business rules, and these rules can also be changed dynamically. So, adaptive optimization should be suitable for keeping the system working efficiently, and most such applications are constructed on Java platforms. Applying a Java Virtual Machine with adaptive optimization system to them should be a good choice.

Finally, the items above are some possible applications and research fields of adaptive optimization with intelligent algorithms. To apply adaptive optimization to such research fields and applications, we still need to consider the issues for constructing an efficient adaptive optimization system (described in 5.1).

The project has explored the foundation of online adaptive optimization with intelligent algorithms (applied two unsupervised learning algorithms (RMHC and GA) on the searching engine and evaluated the performance of the new adaptive optimization system) . The machine learning algorithms are suitable for adaptive optimization. To improve the performance of online adaptive optimization, we still need to find some approaches to improve the searching efficiency and reduce the cost in future.

Appendix A

A.1 RMHC Class Diagrams

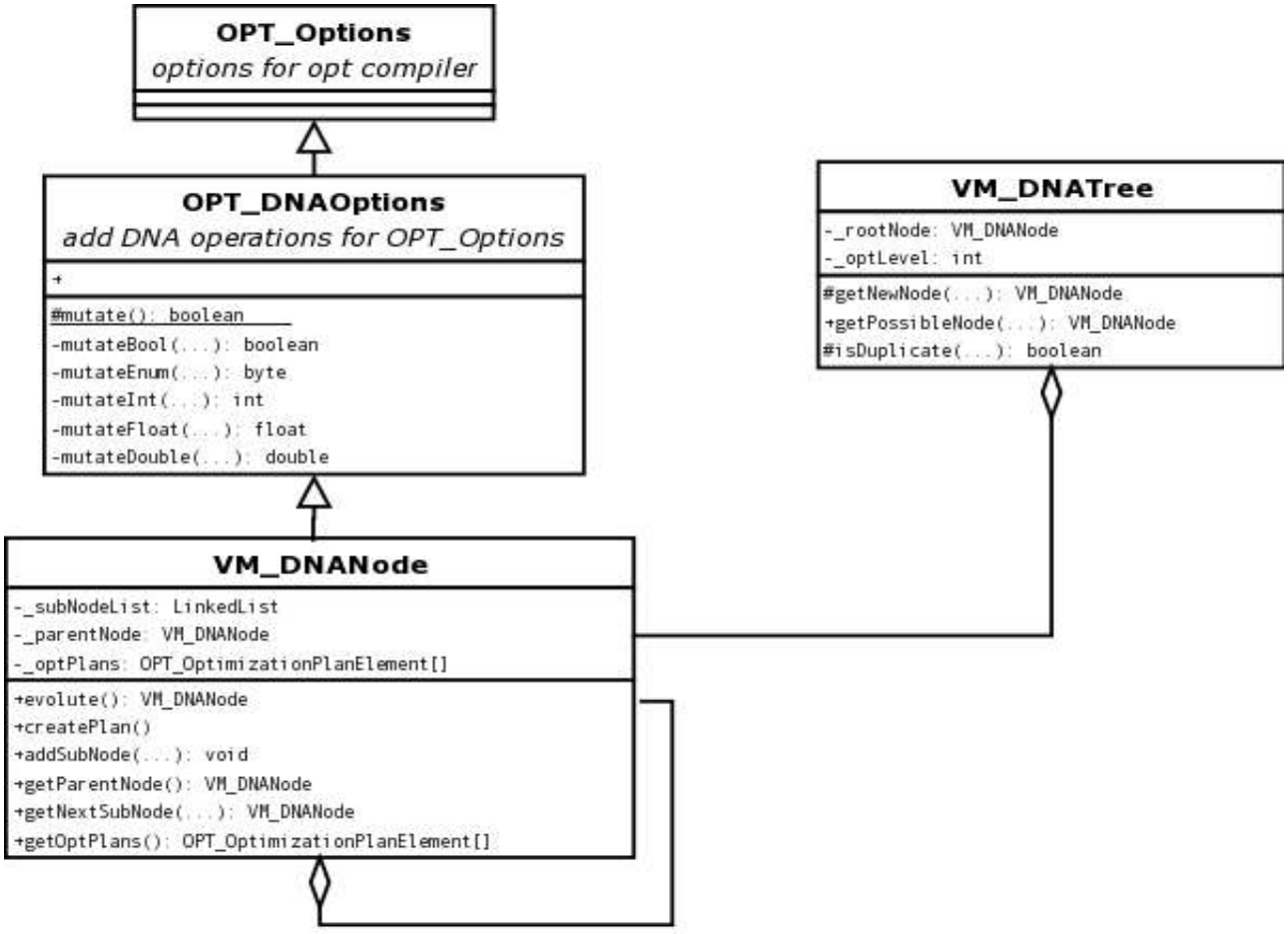

Figure A.1: Class Diagram for RMHC implementation 1

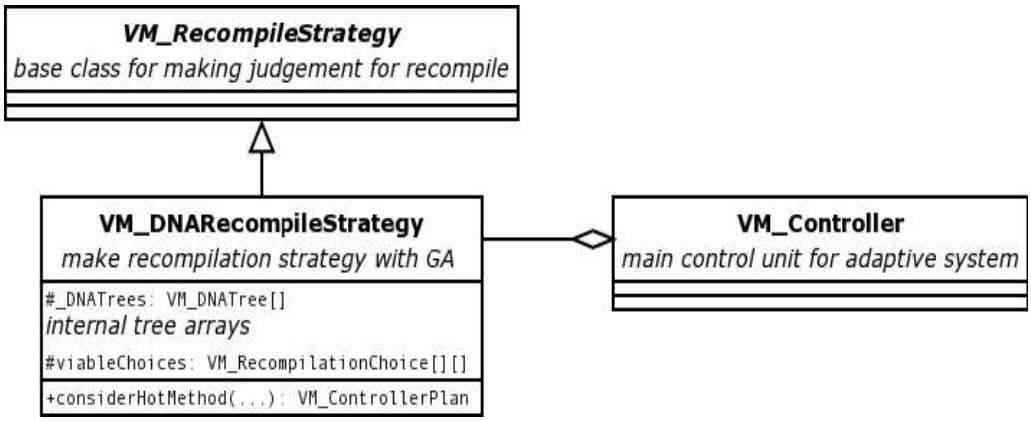

Figure A.2: Class Diagram for RMHC implementation 2

A.2 GA Class Diagrams

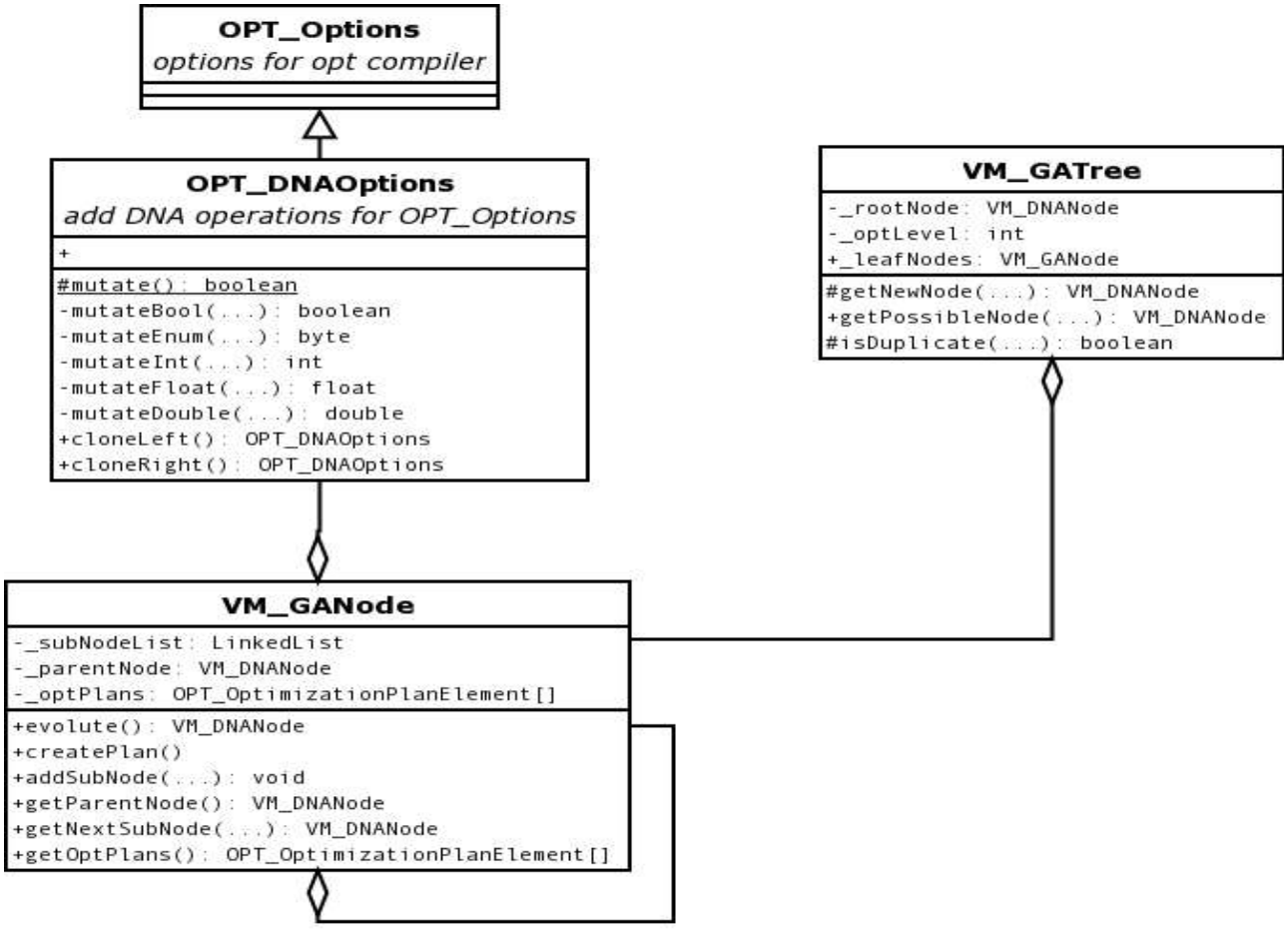

Figure A.3: Class Diagram for GA implementation 1

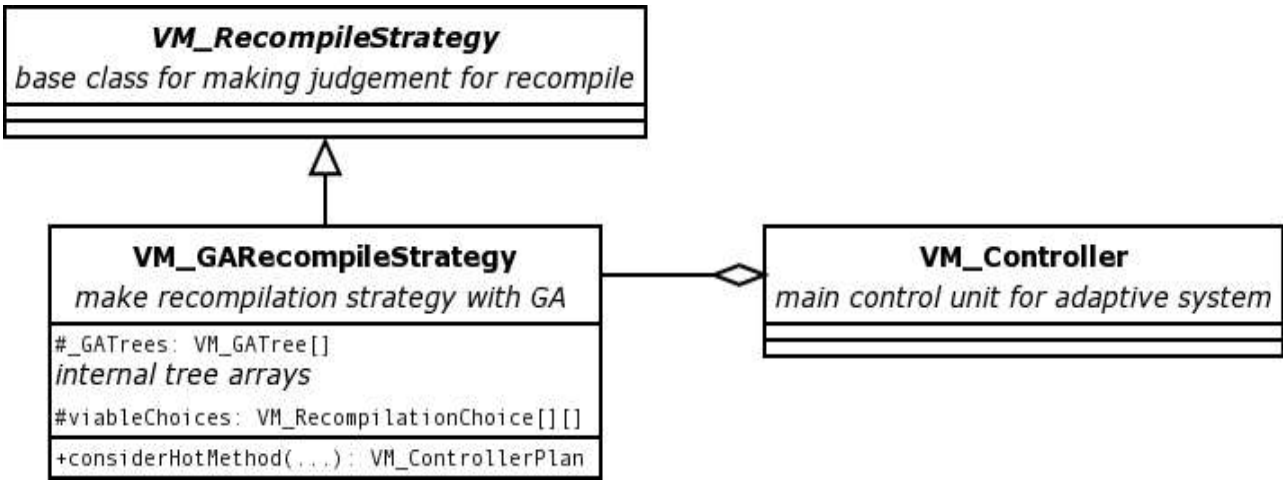

Figure A.4: Class Diagram for GA implementation 2

Appendix B

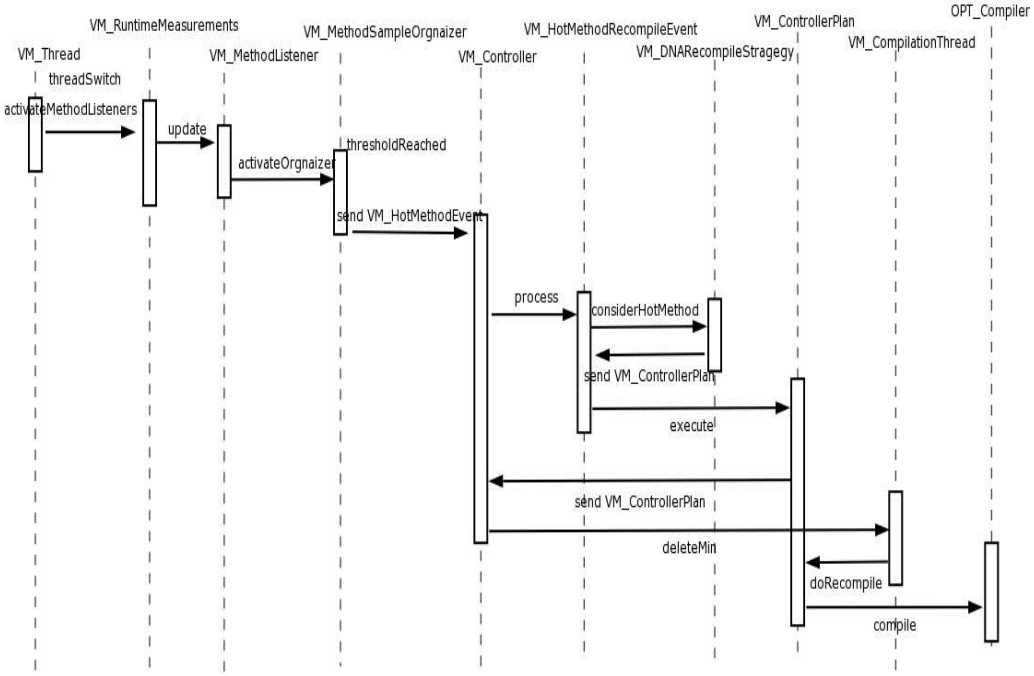

Figure B.1: Working Flow for Adaptive Optimization System

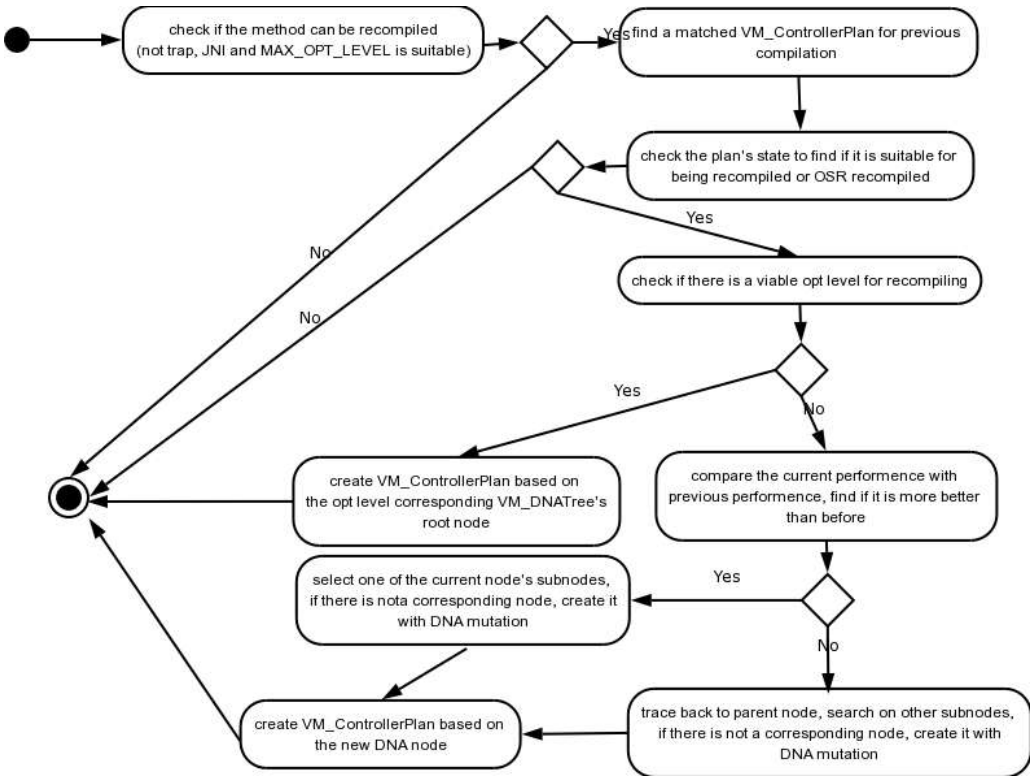

Figure B.2: Working Flow for Determining Hot Methods

Bibliography

[AAea00] B. Alpern, C. Attanasio, and et al. The Jalapeno virtual machine. *The IBM Systems Journal*, 39(1), 2000.

[ABC+00] Bowen Alpern, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark Mergen, Janice C. Shepherd, and Stephen Smith. Implementing Jalapeno in Java, 2000.

[AFG+00] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, 2000.

[AFG+03] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines, 2003.

[AHR02] M. Arnold, M. Hind, and B. Ryder. Online feedback-directed optimization of java. 2002.

[A.P98] A.P.Nisbet. Gaps: A compiler framework for genetic algorithm (ga) optimised parallelisation. *Poster Presentation HPCNE98*, (1), 1998.

[BCF+99] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno dynamic optimizing compiler for Java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, United States, June 1999. ACM.

[BS04] McKinley K.S Blackburn S.M., Chen P. Oil and water? high performance garbage collection in java with mmtk. *ICSE 2004, 26th International Conference on Software Engineering*, (1), 2004.

[Mit96] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT, 1996.

[SAMO02] M. Stephenson, S. Amarasinghe, M. Martin, and U. O'Reilly. Meta-optimization: Improving compiler heuristics with machine learning. Technical Report MIT-LCS-TM-634, 2002.

[SP95] Russel S and Norgig P. Artificial intelligence a modern approach. 1995.

[www98] www.spec.org/osg/jvm98/. Spec jvm 98. 1998.