# Dynamic Profile Guided Optimization in a VEE on IA-64

S T A F F A N   F R I B E R G

Examiner
Dr.-Ing. Christian Schulte

Master of Science Thesis
Stockholm, Sweden 2004

IMIT/LECS-2004-69

## Abstract

This thesis is the result of a Master thesis project at the Department of Microelectronics and Information Technology, at the Royal Institute of Technology, Sweden. The project was done at BEA Systems Stockholm Engineering in Sweden during 2004.

The goal of the project was to study and implement hardware sampling and profile guided optimizations in a virtual execution environment. Profile guided optimizations are available in several modern compilers. The large overhead of collecting the profile has limited the use of profile guided optimizations in virtual execution environments. This thesis presents how using hardware sampling and profile guided optimizations affect various aspects such as performance and optimization time.

The hardware sampling and profile guided optimization was implemented in BEA Weblogic JRockit, a Java virtual machine. To collect the profile information the hardware sampling capabilities on the Intel Itanium 2 processor and a Linux-based operating system was used.

The conclusion of this Master thesis is that the use of hardware sampling together with profile guided optimizations show encouraging results. The optimization time has significantly decreased and is 67% less when using profile information to guide optimizations. The tested applications show a performance gain at 4.7% on average when using hardware sampling and profile guided optimizations.

## Sammanfattning

Denna rapport är resultatet av ett examensarbete för Institutionen för Mikroelektronik och Informationsteknik på Kungliga Tekniska högskolan. Examensarbetet utfördes på BEA Systems Stockholm Engineering i Sverige under 2004.

Målet med examensarbetet har varit att studera och implementera hårdvarusampling och profilstyrda optimieringar i en virtuell exekveringsmiljö. Profilstyrda optimieringar används i flera moderna kompilatorer. Kostnaden i prestandaförlust för att samla in profilinformationen har tidigare hindrat användandet i virtuella exekveringsmiljöer. Rapporten presenterar hur användandet av hårdvarusampling och profilstyrda optimeringingar påverkar olika aspekter, till exempel prestanda och optimeringstid.

Hårdvarusamplingen och profilstyrda optimeringar implementerades i Javamotorn BEA Weblogic JRockit. För att samla in profilinformationen användes hårdvarusamplingsmöjligheterna på en Intel Itanium 2 processor och ett Linuxbaserat operativsystem.

Slutsatsen i rapporten är att hårdvarusampling tillsammans med profilstyrda optimeringar visar motiverande resultat. Optimeringstiden blir markant kortare och är 67% mindre när profilinformation utnyttjas i optimeringarna. De undersökta applikationerna uppvisar ökad prestanda på 4,7% i medel när hardvarusampling och profilstyrda optimeringar används.

## Acknowledgements

I would like to thank industrial supervisors Olof Lindholm and Peter Lönnebring for their help during this master thesis project at BEA. Their help during the project was invaluable.

I would also like to thank Mikael Vidstedt at BEA, for taking time to help me and answer all my questions about JRockit. Further I would like to thank the team at Intel working with hardware sampling, for helping me to understand and use the sampling framework.

Finally I would like thank Christian Schulte for making sure that the project moved along and for feedback on the written report.

# Contents

# Chapter 1

# Introduction

The main objective for compilers besides generating correct code, has always been to generate efficient code. The efficiency of the code is often measured by its size and execution speed. Compiler optimizations are mostly focused on reducing execution time. Common optimizations remove redundant code for example *dead code elimination* and *copy propagation*. Others like *loop invariant code removal* and *loop induction* move code from frequently executed regions (e.g. inside loops) to less frequently executed regions. Another loop optimization is *loop unrolling* where short loop bodies are duplicated to increase the size of optimizable regions [3, 25]. All optimizations mentioned above are different examples of *static optimizations*. Static optimizations only use information that can be extracted from the code at compile time.

Static analysis misses many optimization opportunities since it only has knowledge of the static information extracted from the source code. Static optimizations have for example trouble handling conditional statements and loops efficiently since it does not know the outcome of a conditional statement or the most common number of iterations in a loop. This information is known at runtime and can be collected to build a profile of the code. This profile can be used to optimize the code during a recompilation.

It has been shown that *profile guided optimizations* (PGO) give a clear performance improvement of the generated code [11, 12, 28]. PGO gives the compiler an opportunity to more efficiently do *inlining*, *trace scheduling* etc. Using a profile optimizations can be focused to optimize the most executed path through the code, which is where optimizations will have the largest effect.

PGO is used in modern compilers [16, 17] to take advantage of these opportunities. To create a PGO optimized binary the compiler first compiles a special instrumented version of the code. This code is then executed with typical input and a profile is collected during the execution. The code is then recompiled with the profile information available to create a binary using profile guided optimizations.

The problem with this approach is to find typical input to use when gathering the profile.

*Virtual Execution Environments*[1] (VEE) have become widely accepted and used during the last decade with programming environments such as Java and C#. A VEE helps programmers writing complex applications by having runtime services such as memory management, exception handling and security already available. VEE:s limit the platform dependent design issues otherwise visible to developers. The code executed by a VEE is usually in a platform independent format to allow execution on different systems without recompilation. The platform independent format is dynamically compiled or interpreted at runtime by the VEE.

Since the VEE is handling the whole process of executing a program it is possible for the VEE to collect profile information of a running application using real data. This profile information will be more accurate than when the code is executed with training input and it can also be updated if the input changes. This method, *dynamic profile guided optimizations* (DPGO) [1], can be used to do efficient optimizations and create an advantage for dynamic compilers in VEE:s.

Profiling has often been implemented in software because of limited hardware support. Implementing profiling in software has been associated with a large overhead where as much a one third of the execution time is spent profiling [7]. Recent years more efficient methods have been able to reduce this overhead to around 5% [6]. The large overhead has limited the use of DPGO since execution would be slowed down too much by profiling. With the Intel IA-64 architecture [21] new opportunities open for DPGO with hardware support for available in the architecture. The hardware is called *Performance Monitoring Unit* and can be used to do sampling of running programs. Using this hardware, profiling have been shown to give as little as 2% overhead [35], making it interesting to implement DPGO on IA-64.

This thesis describes how DPGO was implemented in BEA Weblogic JRockit, hereafter referred to as JRockit, on the Intel IA-64 architecture using the hardware sampling capabilities of the processor.

JRockit is a *Java Virtual Machine* (JVM) [23] focused on large scale server applications and its behavior is optimized for the characteristics of such applications [8]. The JVM is the VEE for the Java programming language [18]. Java is compiled to a machine independent format called *Java bytecode* which can be executed in a JVM.

To collect the profile information, a hardware sampling module was added to

[1]Virtual Execution Environments, otherwise known as managed runtimes, abstract machines, or virtual machines, has been chosen as the collective name for this type of programming environment.

the sampling framework in JRockit. Hardware sampling is available in JRockit on the Linux IA-64 platform. The sampling framework is able to collect the profile information sampled by the processor and map it to Java methods running in JRockit.

The most complicated part in implementing DPGO is to handle the profile information correctly. The profile information must be updated continously during the code generation. Many optimizations changes the flow of a method, this requires the profile to be updated to the reflect the new flow of the method. The requirement to update the profile information is not restricted to PGO:s, all operations that change the flow of a method must be able to manage the profile. A framework was designed to update the profile when the flow changes and allow quick access to profile information.

Several optimizations were implemented in JRockit to use the profile information. These include both static optimizations, that were changed to make use of the profile information, and new profile guided optimizations. Static optimizations that have been made more efficient with the use of profile information in JRockit include inlining and loop peeling. Added profile guided optimizations include basic block scheduling and branch hints.

The implementation is tested with several applications to determine the benefits of DPGO compared to static optimizations. The areas studied include the size of optimized code and time the optimizations require. The performance of the applications is measured to determine if any gains are visible with DPGO. The profile accuracy is also measured to check how accurate the profile is after all optimizations.

The results presented in the thesis show encouraging results for using DPGO and hardware sampling in a VEE. Both code size and optimization time has decreased. Optimizations time is less then half when using profile guided optimizations compared to static optimiziations. The performance of the tested applications show an average increase of 5%.

The rest of this thesis is organized as follows. In Section 2 different types of profiling is described. Section 3 presents several different Profile Guided Optimizations previously studied. Section 4 describes how JRockit is designed. In Section 5 the implemented hardware sampling and profile guided optimizations are presented. In Section 6 the experimental results are presented and discussed. Section 7 concludes the thesis and discusses future work.

# Chapter 2

# Profiling

Profiling is often focused on gathering information about the control flow of the code. This is not however the only information that can be profiled and used for efficient optimizations.

To build a profile, information is gathered during the execution of the application of interest. The information is collected through either *instrumentation* or *sampling*. Instrumentation involves adding special instrumentation code in the application that stores the information [7]. Sampling collects information about an application by stopping it with a certain frequency and extracting the information [19].

## 2.1 Value Profiles

Value profiles [9] are used to identify the most common values for various variables. Some variables can be statically determined to be constants at compile time, this information can be used to do optimizations. The variables that can not be statically determined to be constants might during runtime only take one or a few values. If this behavior is discovered in a value profile it could be used to make similar optimizations as if the variable was statically determined to be a constant. Care must however be taken that the variable actually can take other values and these cases must also be handled. This can be solved by creating specialized versions of the code. Which version to use is selected depending on the value of the variable.

## 2.2 Memory Profiles

Higher processor speeds and multiprocessor systems have increased the demand for fast memory, which is one of the biggest bottlenecks in computer systems today. This problem has increased the demand for effecient memory optimzia-tions. Memory profiles describe how memory addresses are read and written by a program. These profiles can be used to make various memory optimizations.

One way to reduce this bottleneck is to use prefetching. Prefetching is similar to an ordinary load but it does not put the referenced memory in a register. The purpose of the prefetch instruction is to make sure that the referenced memory is in the processors memory cache. It is very difficult to determine where to put the prefetch in the code so the reference is available in the cache when the real load is executed. With a memory profile it is possible to determine where memory is referenced and if the reference causes a cache miss. This information can be used to decide when and where a prefetch should be inserted in the code [2].

Another possible optimization is to arrange how the data layout of a structure is ordered in memory. The structures should be ordered in such a way that re-ferring data structures are close, preferably on the same cache-line. This reduce cache misses and give better performance [22].

## 2.3 Control Flow Profiles

Control flow profiling is the most frequently used type of profiling and the one that has been studied most [4, 7, 11, 28, 36]. A control flow profile describes how the execution path flows through a *Control Flow Graph* (CFG) of the code. The CFG consists of a directed graph where the nodes are *basic blocks*. A basic block is a section of straight line code that does not contain any branches. When a basic block is entered all instructions in that basic block are executed in order. The basic blocks are connected by directed edges that represent the possible branches in the code. The control flow profile consists of a trace of the program execution. Since a complete trace of a program execution usually is very large the control flow profile is approximated using frequencies of sub paths.

There are several approximation methods but the most common are *node profiles*, *edge profiles* [28] and *path profiles* [7]. Node profiles describe the execution frequencies of the different basic blocks in the CFG. Edge profiles consist of the execution frequencies of each edge in the graph. Collecting edge profiles is done by analysis of the frequency of branches in the executing code and has similar overhead as node profiling. Edge profiles are superior to node profile since a node profile can always be calculated from an edge profile while the opposite is not always possible. Path profiles are complete paths through a section of the code. Path profiles contain more information than edge profiles about the execution. An edge profile can be determined from a path profile while the opposite is not true.

# Chapter 3

# Profile Guided Optimizations

This section describes various profile guided optimizations that have been developed. Many of them are modified static optimizations, which have been made more efficient with the use of profile information.

## 3.1 Static Optimizations

Static optimizations per definition do not use profile information. It is however possible to indirectly use profile information to guide static optimizations. Optimizing code that is seldom or never executed is a waste of resources. If the code is never executed it does not matter how fast it is after optimization.

Unnecessary optimizations are one of the biggest issues with static optimizations. Since static optimizations lack profile information it has to treat all the code as equally important. If profile information is available it is possible to concentrate the static optimizations to frequently executed code. The optimizations will still be static but indirectly they use the profile information to decide which sections of code to optimize. This will reduce compilation time since less amount of code is optimized. The performance of the code should not be affected since only code that is never or infrequently executed is left unoptimized.

### 3.1.1 Loop Unrolling

Loop unrolling is an efficient optimization that reduces the overhead of loop control code and exposes more instructions for optimizations in the loop block. The optimization is done by replicating the body of the loop so that each iteration in the new loop corresponds to several iterations in the original loop. The downside of the optimization is that register pressure and code size increase with the unroll factor[1]. Using a profile it is possible to determine if loop unrolling is beneficial. The profile can be used to guide how high the unroll factor should be depending on the common number of iterations in the loop [33].

### 3.1.2 Escape Analysis

Escape analysis tries to find memory structures in the code that are local to a method and therefore do not need to be allocated on the heap. Allocating this structure locally, on the stack or in registers, instead of on the heap minimizes memory management overhead since heap defragmentation is reduced and the structure does not need to be freed. Statically this can only be done if the entire method is analyzed and the structure is found to be local in the entire method.

Using profile information it is possible to find the common way through a method and then determine if an object is local with reference to this part of the code. If this is the case it is possible to allocate the object locally instead. Care has to be taken if the structure is not local outside the common path. Then extra code has to be inserted to move the structure to the heap if the execution enters a region where the structure is not local. Profile guided escape analysis is described by Whaley [34].

## 3.2 Inlining

When inlining, the called functions code is moved to the caller function and the call is removed. The benefits from inlining come from removing the call overhead. The call overhead consist of everything that is required to make a call, for example saving variables on the stack and doing the actual jump. Another benefit is the possibility to create a specialized version of the inlined method through the use of data properties at the call site and constant arguments.

### 3.2.1 Inlining Static Calls

Static method inlining is an efficient optimization that has been used for a long time. Static calls are calls where the destination is known at compile time and never change.

Static inlining heuristics are difficult to setup optimally. Inlining every call in a method is often bad since the called method might be very large and get called from several different methods. This would probably make it bad choice for inlining. Using profile information when inlining it is possible to derive better

[1]The unroll factor is the number of times a loop is unrolled

heuristics for determining which methods to inline and which to leave as calls. It has been shown that using a profile to guide inlining can give faster execution and reduced code size compared to static heuristics [5, 30].

### 3.2.2 Inlining Virtual Calls

Virtual calls are calls where the destination may be unknown until the call is executed. This is the case for calls to function pointers in C and non-fixed instance methods in object oriented languages such as Java and C++. A solution to this problem is to create specialized static calls to the methods and select different paths through depending on the method pointer or the object type. This type of optimization was first described by Chambers in [10].

A virtual call can either be *monomorphic*, with only one possible destination, or *polymorphic*, with several possible destinations. In statically compiled languages such as C++ some virtual calls can be determined to be monomorphic through analysis. This is the case if a method can be proven to only have one implementation. This can done in Java as well, but since dynamic class loading is possible the result might be invalidated later when new classes are loaded.

If one destination of a polymorphic call is inlined, execution of that inlined call must be guarded. The guards have to make certain that the inlined method is the correct method to execute. If it is not the correct method the original virtual call is made instead. Detlefs [15] compare *class test* [14] and *method test*, which are different algorithms for determining that the correct method is executed when doing virtual inlining. Class test compares the class of the method with the class of the object before the inlined code is executed. Method test compares the memory address of the call destination to be made with the address of the original method that is inlined.

Both methods have advantages and disadvantages: consider the case where class **B** inherits class **A**. Let **A** implement the method **m()** which **B** does not override. Suppose there exists a virtual call to **m()** that would be good to inline, and the actual method inlined is **A.m()**. The inlined code must be guarded to check that the correct method is executed. Using the class test to check the test will fail for all objects of type **B** and the inlined code will not be executed even though it is the correct method. If the method test is used instead the test will succeed for the objects of type **B** since it is known that the same method is called.

Inlining of virtual calls is difficult if no profile information is available and the call is polymorphic. The problem is to decide which of the called methods to inline. A call might be “almost monomorphic” if it is polymorphic but most calls go to the same destination. If the wrong method is inlined performance will suffer because of the added guard check. Using a profile of the program it is possible to determine which calls are almost monomorphic, and find the common destination.

## 3.3 Code Positioning

The number of branches in the executed code affects the performance. The processor has to fetch new instructions from memory before executing them. Normally the next instruction is available in the cache because a full cache-line is filled every time . If a branch is taken the destination instruction might not be in the cache because the instruction is placed too far away in memory from the branch to be automatically be prefetched by the processor. This result in a stall waiting for the memory read to be completed. Most modern processors do some prediction of branches to minimize the time for instruction fetching. Mispredicting a branch can take hundreds of clock cycles to handle before the correct instruction is fetched and executed.

Pettis and Hansen [27] has developed an algorithm for increasing the cache locality of code. They try to place methods that call each other close together in order to minimize page faults. They also do intra method optimizations such as basic block ordering and method splitting.

In basic block ordering they arrange the basic blocks so that blocks that often executed sequentially are arranged in that order in memory. This reduces the number of taken branches and therefore makes it more probable that the instructions to execute is already in the instruction cache.

Method splitting splits the method in two parts: *Primary* and *Fluff*. Primary is the code that is executed normally in the method while the fluff is the code that is almost never executed. Method splitting makes the primary part of methods smaller and easier to fit within a page. When ordering the methods the primary parts are put on the same pages while the fluff parts are placed together on other pages. This gives the Operating System (OS) a possibility to always keep the pages with primary code in memory while swapping out the fluff pages.

## 3.4 Superblock

Basic blocks are normally short code segments with few instructions. This makes it difficult to do efficient instruction scheduling. To overcome this problem it is possible to use a structure called *superblock* [24].

A superblock differs from a basic block in that it allows branches out of the block inside of the superblock. It is however only allowed to branch to the beginning of the superblock. The execution of a superblock must start at the beginning of the block but may leave at one or more exit points. Basic blocks can rather easy be merge to create a superblock. If the merged basic block has entry points in the middle of the block these can be removed through tail duplication [11]. Tail duplication is done by copying all blocks that are below the first entry point in the merged block. Then all entry points to the middle of the merged block are changed to point to the duplicated basic blocks instead. The merged block are now treated as a superblock since it only has one entry point left at the beginning.

## 3.5 Memory Optimization

Using the memory profile described earlier (Section 2.2) it is possible to do some memory optimizations that can reduce execution time. The use of linked data structures results in many memory accesses that are hard to know at compile time and hard to optimize. Since the data can be widely spread in memory the referenced memory might not be available in the cache.

Adl-Tabatabai [2] describes an algorithm to minimize cache misses through prefetching. By sampling memory references it is possible to determine how the structures that refer to each other are positioned in memory. A *delta* is calculated between referring structures. The delta is the distance between two structures in memory. The delta is calculated for a type of structures and not individual structures. This delta is then used when a certain type of structure is read from memory to prefetch the normally referred structures. This algorithm works since most memory structures are allocated in the same order all the time and therefore end up in the same relative memory positions every time.

## 3.6 Profile Maintenance

Many optimizations change the CFG of the code and therefore have to update the profile at the same time. This is an important issue since optimizations are done sequentially and all optimizations need an accurate profile to work optimally. This is a difficult problem to discover since an inaccurate profile does not generate erroneous code instead the code will be not be optimal. An inaccurate profile can seriously cripple performance since the only the common path is optimized while the other paths sometimes are slowed down by moved code.

Wu [35] shows the importance of accurate profiles and shows how to update the profile after various optimizations. The optimization that contributes most to the loss of accuracy is inlining. A reason for this is the problem that the profile for a method is calculated over all calls and not for every specific call site. This may result in that a profile that is globally correct for a method is incorrect for a specific call site. Figure 3.1 illustrates an example of how the global and local profile might differ for a method.

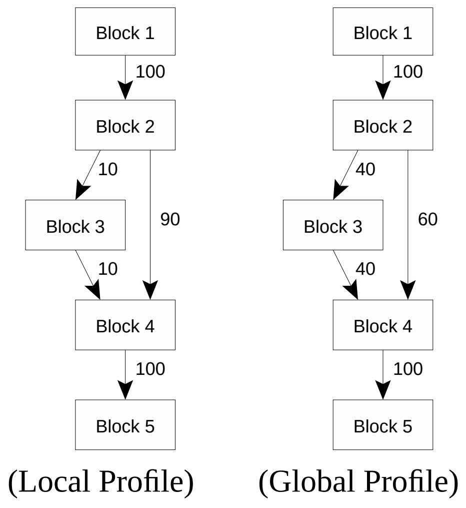


***Figure 3.1:*** *Graph showing the difference between the local profile of a specific call site and the global profile collected using calls from all call sites. This difference between the global and local profiles can result in that wrong optimizations are done. It is a problem since often only the global profile is available and the local profile is completely unknown.*

# Chapter 4

# JRockit

JRockit is a JVM focused on server applications. A server application typically has a long running time and demand high performance. Since the applications are long running, the startup time is of less importance than for client applications. Because of this and to achieve highest possible performance, JRockit always generates native code from the bytecode instead of interpreting it, as Sun [31] and IBM [20] does. Other demands of server applications are scalability and non-disruptiveness. To minimize disruptiveness in applications JRockit has several different garbage collectors. Which garbage collector to use is selected at runtime depending on the characteristics of the running application.

Optimization are time consuming and since many methods are only executed once or infrequently it is not necessary to do extensive optimization on all methods. Instead all methods are first compiled with few or no optimizations. When a method is later discovered to be frequently executed, hot, that method is *hotspotted* and recompiled with aggressive optimizations.

## 4.1 Software Sampling

JRockit only optimize and recompile methods that are frequently executed. These methods are found by sampling the running application. JRockit uses software sampling to find the frequently executed methods in the running application. The information is collected by iterating through the running Java threads in the JVM. Each thread is suspended and the context of the thread is read to find the current instruction pointer. Using the instruction pointer, JRockit finds which method a thread is currently executing and increases the hotness score of that method.

Suspending a thread, reading its context and then restarting it is associated with a large overhead. To get more information when a thread is already suspended JRockit looks at the call stack of the thread. The hotness scores of the methods found on the call stack are also increased. This approach usually gives a good result since optimizing a method higher up in the call chain expose possibilities to do inlining.

As stated earlier, it is costly to suspend the threads and read their context. JRockit therefore only looks at a fixed number of threads each time to avoid problems with applications using a large number of threads.

To minimize the overhead and unnecessary sampling JRockit has two different states of sampling called *active* and *passive*. Active sampling is a high frequency sampling that quickly finds methods to optimize but with a large overhead. Passive sampling reduces the overhead by decreasing the frequency of the sampling. However because of the lower frequency it will find methods slower. This is important for long running applications where sampling after a while becomes unnecessary, since the important methods already have been discovered and optimized. Which sampling mode to use is determined by looking at the ratio between samples in optimized and non-optimized code, and elapsed time since new code was generated.

## 4.2 Code Generation

Since JRockit always compile the bytecode to native code, JRockit is very similar to a traditional compiler and therefore contains the parts available in a compiler.

To represent a method compilers often use an *intermediate representation* (IR). The IR is a data structure that holds all the information about a method that is needed to compile and optimize it. The IR is an abstract representation of a method and makes code generation and optimization easier for the compiler. The IR represents all the instructions in the code and also the possible flow of execution in the method.

There are various ways to implement the IR, for example Appel [3] uses a tree structure to hold the instructions and flow of a method. In JRockit the IR consists of a graph that describes the flow of the method and nodes that contain the instructions. The graph is a CFG where the nodes are basic blocks and the edges describe the possible flow of the execution.

The code generation use three different levels of IR called HIR, MIR and LIR, which is short for High-, Middle- and Low-level IR. Figure 4.1 shows how the bytecode is translated to native code through a series of intermediate steps. For every step the IR is transformed to a format that more resembles native code.

The HIR is a very high level representation and is mostly used to ease the translation from bytecode. The platform independent MIR level is where most high level optimizations, such as inlining and dead code elimination, occur. The MIR is transformed to SSA-form [13] and back during optimizations. The SSA-

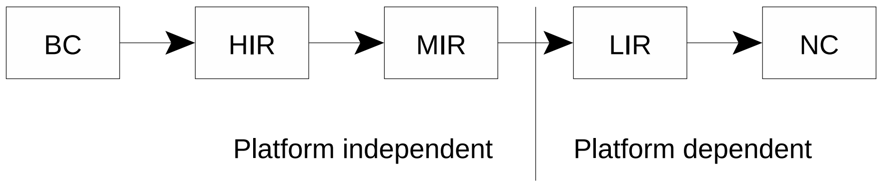


***Figure 4.1:*** *During code generation the Java bytecode (BC) is transformed to an intermediate representation. There exists three different levels of IR called High-, Middle- and Low-level IR. When the code generation reaches the LIR level the code is platform dependent and is then translated to native code (NC).*

form simplifies several optimizations. In LIR the IR has become platform dependent. The instructions in LIR are very similar to real native code instructions, which makes register allocation and low level optimizations easier.

JRockit has different IR levels since it is hard to accommodate the needs of the stages involved in code generation in one grand unified representation. Each representation is suitable for its specific stage and the optimizations of the code done in that stage.

# Chapter 5

# Implementation

A new type sampling is implemented in JRockit using the *hardware sampling* capabilities of Itanium 2. With this hardware it is possible to gather extensive information about individual methods in the application that can be used for profile guided optimizations. This section describes the hardware sampling and how this new information is used in JRockit to enable profile guided optimizations.

## 5.1 Hardware Sampling

The hardware sampling is divided in layers to hide as much as possible of the underlying OS and processor from JRockit. This is to simplify porting of hardware sampling in JRockit to other OS:s and processors as profiling becomes available. Another reason is to have one profiling framework, independent of which type of sampling that is used.

### 5.1.1 Performance Monitoring Unit

The IA-64 architecture defines a *Performance Monitoring Unit* [21] (PMU) that allows monitoring of processor events. The architecture states that the PMU must implement at least four performance counters, each consists of two type of registers called *Performance Monitor Configuration* (PMC) and *Performance Monitor Data* (PMD). The PMC is used to configure how to monitor an event. The result is gathered in the PMD register or registers controlled by that PMC register. The PMU is only writable in privileged mode (kernel mode) but it is possible configure the PMU to make it readable for applications not running in kernel mode.

The Itanium processor implements the IA-64 architecture and the PMU, and several monitoring possibilities have been added to the PMU. The Itanium has added 6 PMC registers and 14 PMD registers to the ones required by the IA-64 architecture. The architecture requires only two events to be available, but the Itanium has extended this to approximately 230 different events. Important additions are *Address range checking, Event Address Registers* (EAR) and *Branch Trace Buffer* (BTB).

**Address range checking** Address range checking is used to limit the memory range in which events are captured. This can be used to sample only a specific application or a part of an application.

**EAR** The EAR registers are used to capture events related to memory usage. Two different EAR registers exist, one for instructions (I-EAR) and one for data (D-EAR). The D-EAR captures events for loads, stores and other instructions that use memory. D-EAR uses three PMD registers to record instruction address, memory address and memory latency for one event. D-EAR can be used to create memory profiles of an application. I-EAR instead record instructions address and memory latency when the processor fetches instructions.

**BTB** The BTB consists of four pairs of PMD registers that capture the address of a branch instructions along with the destination of those instructions. By using the PMC register for the BTB it is possible to configure which types of branches to capture. The BTB can for example be configured to sample only taken, not taken or mispredicted branches. With the four pairs of registers in the BTB it is possible to build path or edge profiles of the code.

### 5.1.2 Linux/IA64 kernel

Since the PMU is writable only when the processor is in kernel mode the OS must provide a system call if unprivileged programs are to be allowed to access it. In the Linux kernel[1] for IA-64 this support is called *perfmon* [26].

Perfmon consists of a single system call, `perfmonctl()` through which user level applications can register to receive all supported monitors. Through this system call it is possible for applications to register per-task or system-wide sampling. Perfmon is able to handle several distinct per-task sessions to be monitored in parallel. Perfmon also handles security aspects so that only the owner of a monitoring session is allowed to read the associated registers in the PMU.

[1]Perfmon is available in both 2.4 (perfmon-1.x) and 2.6 (perfmon-2.x) version of the Linux kernel. The API is slightly different between the two kernel versions.

### 5.1.3 RTMon

To minimize platform dependence, Intel, in a joint project with BEA, has developed an API called *RTMon* that lies between the OS kernel and the application. RTMon works as a producer of monitoring events. A consumer can register what type of information it is interested in and supply a buffer for storing the sampled data. The buffer is filled with the requested information and when the buffer is full the consumer is notified and can start to decode the data in the buffer.

### 5.1.4 Sample Processing

When a buffer is received in JRockit from RTMon, all collected samples are processed. Since the samples in the buffer are just a from- and a target-address for the sampled branches they need to be processed to limit the memory usage. Each sample is decoded and the method they belong to is looked up. For every unique sample a new counter is added to the corresponding method. This counter is then increased for every subsequent sample with the same address. For each sample the hotness score is increased in a similar fashion as with software sampling. When a method reaches a hotness score higher than a certain threshold the method is queued for optimizations.

## 5.2 Profile Decoration

Decorating the IR is the process of determining the probabilities and frequencies of the edges and basic blocks in the IR. This is done by mapping the sampled information to the branches in the IR. As shown in Figure 5.1 the IR gets decorated at the MIR level in the code generation. This is level was chosen since JRockit does not do any optimizations earlier. Decorating earlier would therefore only require more work managing a profile that is not used.

The IR only gets decorated when a method has been hotspotted and is recompiled with optimizations. Decorating the IR is one of the more complicated parts of the profiling process and involves translating the sampled instruction pointers to positions in the Java bytecode.

To enable this translation JRockit uses a mapping between the emitted machine instruction and the original Java bytecode it originates from. This mapping is called a *location* which is defined as a specific method and a *bytecode index*. A bytecode index is the offset of a specific bytecode instruction within a method. To limit the number of locations only certain types of bytecode instructions are given locations, for example control transfer instructions.

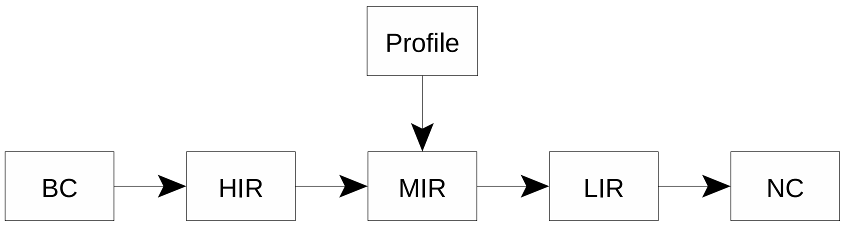


***Figure 5.1:*** *The IR is decorated in the MIR level. The reason the IR does not get decorated earlier is because that no optimizations are performed in the higher levels. Decorating the IR earlier would only impose more work since the profile would need to be managed longer without being used.*

A limitation with locations is that, there exist bytecode instructions that expand to several branches in native code. This makes it difficult to find which bytecode instruction that corresponds to the sampled machine instruction since several samples may correspond to the same bytecode instruction and location.

Figure 5.2 shows an example of an instruction that is expanded to two branches. If samples are collected for these branches it is impossible to separate them from each other using only locations.

To determine the probability for a specific branch the collected samples corresponding to that branch need to be found. In MIR the blocks are not yet scheduled and because of this it is impossible to know how the basic blocks are ordered in memory. This makes matching of branch samples with edges in the IR non-trivial.

To determine the probabilities for the edges in the IR in Figure 5.3, the samples for the two different outcomes of the corresponding branch must be found. First, the two samples with a from address corresponding to location 1 must be found. Location 1 is the last location in the basic block and corresponds to the branch. When the two samples have been found it must be determined which of the IR edges the samples corresponds to. The target address of the samples must correspond to the first location in one of the successor blocks, in this case either location 2 or 3. Using the count for the samples it is possible to calculate the probabilities for the two edges in the IR. If only one edge has received any samples the other edge is infrequently or never executed and should have zero probability.

Since all information is collected through sampling some parts of the IR may not have any samples at all while others contain many. Because of this there must exist a way to guess the outcome of branches that lack profile information. To solve this problem a simple algorithm has been implemented that give the branches equal probability, unless it is possible to assume that either edge should be taken infrequently. Examples of such branches are tests that check if an object reference is null or if either edge is connected to a basic block that throws a Java

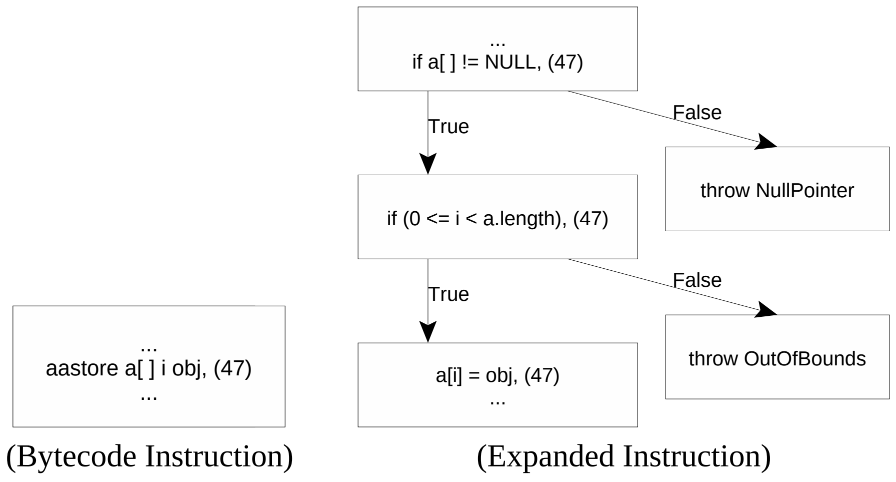


***Figure 5.2:*** *The figure shows how the bytecode instruction aastore, which stores a value at a specified index in an array is expanded. The single bytecode instruction, is expanded to two branches and the actual store all with the same bytecode index, 47. Since all instructions in the expanded version have the same location it is impossible to make any distinction between them when processing samples.*

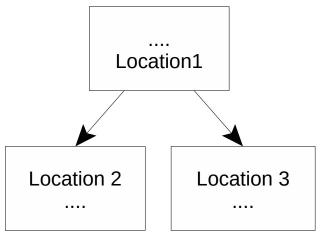


***Figure 5.3:*** *A simple IR with three different locations used to find samples that correspond to the edges in the IR in order to decorate it.*

Exception. These are both exceptional cases that normally do not occur and are therefore are given a low probability.

As stated above, sometimes only one of the edges out of a basic block has received any samples. For normal basic block with branches this poses no problem. However if the edge with no samples is a loop exit this means that the loop never exits. For some loops this is correct but for most loops this is not true. This problem occurs in loops that do many iterations before exiting, since there is a risk of all samples are collected from within the loop. To accommodate this problem, all loop exits are set to have a small, non-zero, probability.

The IR is at first decorated with branch probabilities for every edge in the IR. Using this information, basic block and edge frequencies are calculated using the algorithm described by Wu [36]. This enables the optimizations to find the frequently executed, *hot*, or infrequently executed, *cold*, basic blocks and edges in the IR. The frequency of a basic block is the sum of the frequencies of all the incoming edges. The frequency of an edge going out of a basic block is the frequency of that basic block multiplied with the probability of the of the edge itself. The equations below describe this relation.

$$\begin{aligned}
\mathrm{bfreq}(b_i) &= 1.0 && \text{(if } b_i \text{ is the start block)} \\
\mathrm{bfreq}(b_i) &= \Sigma_{b_p \epsilon \mathrm{pred}(b_i)} \mathrm{freq}(b_p \rightarrow b_i) && \text{(otherwise)} \\
\mathrm{freq}(b_p \rightarrow b_i) &= \mathrm{bfreq}(b_i)\mathrm{prob}(b_p \rightarrow b_i)
\end{aligned}$$

The equations are straight forward to solve for methods that do not contain loops. Then the equations can be solved by starting at the entry block[2] of the method and topologically walk the graph calculating the frequencies for edges and block as they are encountered. If the method contains a loop the equations become recursive since the frequency of an incoming edge to a basic block may depend on one of the outgoing edges from itself. Wu describes a fast algorithm that handles this and also is able to solve the equations for methods containing non-terminating loops.

The incoming frequency to a method is always 1.0, which makes it easy to determine if a block is hot or cold. A basic block or edge that has frequency 1.0, is executed one time on each entry of the method on average. If the frequency is even higher this shows how often for each entry in the method how many times that basic block is executed. The same is true if the block has a frequency less than 1.0, which implies that the block is not executed every time the method is

[2]The entry block in a method is the block where execution of the method always starts when the method is called.

entered. Using this information it is possible to set limits for when a basic block is hot or cold.

## 5.3 Profile Guided Optimizations

JRockit does several optimizations of the code in the different levels of IR. Most optimizations are made in the MIR stage. Each optimization can be performed several times, since when the IR is transformed by an optimization, new optimization opportunities that were not possible before might arise. This requires that JRockit is able to correctly manage the profile information so that it is still correct after the IR has been transformed. This update must be done for both profile guided and static optimizations.

Some of the static optimizations in JRockit have been modified to use the profile information available. New optimizations completely dependent on profile information has also been added to JRockit. These changes and additions to JRockits optimizations are described below.

**Hoist Loads** Only hoist loads from a basic block if the basic block to where the load is moved has a lower execution frequency than the basic block where it is currently located.

**Loop Peeling** Only do loop peeling on loops with high execution frequency. These are either loops that are entered often, do many iterations, or both.

**Load Prefetch** Do not add prefetches for loads in cold blocks, since most of the time it will be unnecessary. The effect of limiting the prefetches will hopefully be less instructions and reduced number of memory accesses.

### 5.3.1 Inlining

Inlining can increase performance significantly by removing the overhead of a call. The downside of inlining is code growth and longer optimization time because of the larger code. Using the profile information, inlining is restricted to calls that are frequently executed.

To determine which methods to inline, a score is calculated for each call. The score is a measurement on how profitable it would be to inline that call. To calculate the score, the function examines various aspects of a call, and returns a score depending on how these aspects affect inlining. The aspects considered at are for example number of constants in the call, loop depth, recursive calls and size of the called method.

The old static score function is still used since it performs well. Instead the score function is altered by adding profile knowledge. The first change is that all blocks that are cold are ignored and no score are calculated for calls in cold blocks. However inlining is still done for small methods for which the code to make the call is larger then the method called. If a call is not cold, the score for the call is calculated. The score is now not only affected by the static information but also the profile information. The higher the frequency of a call site is, the higher score it receives. These changes concentrate our inlining to the hot blocks of a method and limits the unnecessary inlining in cold blocks.

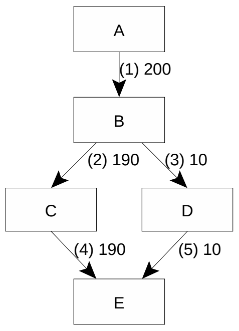
### 5.3.2 Basic Block Scheduling

One of the last steps in the code generation is the basic block scheduling. Basic block scheduling decides in which order the basic blocks will be placed in memory. An algorithm developed by Pettis and Hansen [27], simply called *Algo2* or *Bottom-up Positioning*, was implemented to do scheduling of basic blocks using profile information. The algorithm works by trying to build chains of basic blocks that are frequently executed in order. A chain is a linked list of basic blocks that also are connected in the IR. To determine how the blocks should be added to the chains, the edges are sorted by frequency in descending order. Going through all edges in order it looks at the two basic blocks connected by the edge.

To connect either of the basic blocks to an existing chain the other block must be first or last in a chain depending how the blocks are connected. To add the successor block to a chain, the predecessor block must be the last block in one of the available chains. If the predecessor is to be added, the successor block must be the first block in a chain. If either block is in the middle of an existing chain, the edge is skipped. If neither of the blocks are in a chain, a new chain is created with the two blocks. After all edges have been processed, some basic blocks might not have been added to a chain. Each of these basic blocks are now added to new chains containing only a single basic block.

The last step in the algorithm is to order the chains. In Algo2, this done by analyzing the precedence relations between the basic blocks in the chains. The objective is to minimize the number of backward jumps and it tries to place the chains in such a way that only forward jumps occur. This is not implemented in JRockit, instead the chains are ordered by finding the successor block that is connected with the highest frequency to a block in the last added chain. This will order the chains in the most frequently executed order and move less executed chains of blocks to the end of the method.

Figure 5.4 show an example of how the algorithm works when the IR in the figure is scheduled. The first step is to build the chains. The edge with the highest frequency is 1 so the first chain is A $\rightarrow$ B. The next edge is 2, since C is a successor



| | |
|---|---|
| Chain 1: | A → B → C → E |
| Chain 2: | D |

(IR) (Chains)

***Figure 5.4:*** *A decorated IR with edge frequencies. On the right are the chains that are created when scheduling the basic blocks with Pettis and Hansen's [27] algorithm.*

of B, C is added to the end of the first chain since B is the last block in that chain. In the same way E added to end of the first chain. D does not get added when going through edges 3 and 5 since both the successor and predecessor of D are in the middle of chain 1. The order of the chains is simple, since the start block must be first in a method the first chain is placed first, followed by the second chain.

### 5.3.3 Branch Prediction

The IA-64 branch instructions give the opportunity to set hints on branches. The hints are a guess of the outcome of a branch. Table 5.1 shows the hints available to branch instructions. These hints enable the processor to prefetch the code on the destination of the hinted outcome and start the execution before the evaluation is completed. Previously JRockit guessed how to place hints depending on the direction and type of branch. The new optimization use the profile information to determine the normal outcome of the branch and adds a dynamic taken or not-taken hint depending on the profile.

### 5.3.4 Profile Management

A very difficult problem when using PGO is to keep the profile updated and correct at all times. If the profile gets incorrect during a transformation of the IR it is hard or often impossible to recreate the lost information later. The profile has to be correct until the code has been generated and placed in memory, since the the profile information is used during the whole code generation. Optimizations, whether or not they depend on profile information, must update the profile if they

***Table 5.1:*** *The prediction hints available for branch instructions on IA-64. The table comes from Intel IA-64 Manual [21]*

| Hint | Strategy | Operation |
|---|---|---|
| spnt | Static Not-Taken | Ignore this branch, do not allocate prediction resources for this branch. |
| sptk | Static Taken | Always predict taken, do not allocate prediction resources for this branch. |
| dpnt | Dynamic Not-Taken | Use dynamic prediction hardware. If no dynamic history information exists for this branch, predict not-taken. |
| dptk | Dynamic Taken | Use dynamic prediction hardware. If no dynamic history information exists for this branch, predict taken. |

in some way change the flow of the IR. These changes might be adding or remove basic blocks, which might affect the probability and frequency for connected basic blocks.

Many transformations can be handled locally, such as splitting a basic block, where only the edge connecting the two blocks must be set to the correct probability and frequency. Others, like removing a predecessor from a basic block, require a recalculation of frequencies in all reachable basic blocks and edges. Changes to instructions inside basic blocks or movement of instructions to other blocks, do not require the profile to be updated. This is true for instructions not affecting the flow of the code. Examples of optimizations that affect the profile are:

**Inlining** The method that gets inlined must have a profile that is correct and scaled to the frequency of basic block containing the call site.

**Constant Branch Removal** If one edge is removed the probability of the other edge or edges is affected. Since the changed probability affects the frequency of the edge and successor block the frequency for the IR must be recalculated.

**IR Conversion** Going from one level of IR to the next usually involves transformation of instructions to a format closer to native code. This might add new basic block in the new IR depending on the type of instruction. The new blocks and their edges must have correct probability and frequency set.

# Chapter 6

# Results and discussion

## 6.1 Methodology

All the performance results presented in this thesis were obtained on an HP workstation zx2000, a single processor system with an Itanium 2, 900 MHz, processor. The OS used was Suse 9.0 [32] with Linux kernel version 2.6.5. To evaluate hardware sampling and the implemented profile guided optimization, SPECjbb2000 and SPECjvm98 [29] benchmarks and an BEA internally developed streaming XML-parsing benchmark, XMLStax, were chosen. SPECjvm98 is a test suite composed of eight separate benchmarks.

Each benchmark was executed several times each run to let the JVM hotspot and optimize as many methods as possible. Depending on optimization and type of sampling, different number of methods might be hotspotted and optimized.

First the individual parts of sampling and optimizations are tested. The software and hardware sampling is compared to see how they affect performance. Also the difference in ability to find frequently executed methods to optimize is studied. Then the implemented profile guided optimizations, are benchmarked to see how they impact optimization time, size of optimized code and performance of the benchmarks. The correctness of the profile information of the optimized methods is measured, in order to determine how the updated profile has been affected during code generaration and optimizations. Finally, the current version of JRockit with software sampling and static optimizations is compared to a version of JRockit with both hardware sampling and profile guided optimizations enabled.

## 6.2 Hardware Sampling

Table 6.1 shows the number of methods that were optimized in each of the benchmarks. The hardware sampling finds fewer methods to optimize than software

***Table 6.1:*** *The number of methods optimized when hotspotting with software (SW) and hardware (HW) sampling, respectively.*

| Benchmark | SW Sampling | HW Sampling |
|---|---|---|
| SPECjbb2000 | 159 | 177 |
| 201_compress | 13 | 7 |
| 202_jess | 52 | 43 |
| 209_db | 24 | 15 |
| 213_javac | 261 | 177 |
| 222_mpegaudio | 75 | 63 |
| 227_mtrt | 44 | 54 |
| 228_jack | 78 | 72 |
| XMLStax | 25 | 20 |

sampling. This is true for all but two cases, SPECjbb2000 and 227_mtrt, where more methods are optimized.

In Figure 6.1, the normalized optimization time spent for the hotspotted methods are shown. Optimizing methods found with hardware sampling require on average 37% of the time spent on optimizing methods found through software sampling. The time spent optimizing methods is lower when using hardware sampling, even in the two cases where more methods were optimized.

The performance difference between hardware and software sampling is shown in Figure 6.2. The applications show similar performance results on the various benchmarks independent of the sampling method. On average the benchmarks differ less than 0.3% in performance. The benchmarks 202_jess and 213_javac stand out with a performance difference around 10% between the two types of sampling.

Using hardware sampling instead of software sampling shows a slight decrease in performance compared to using the software sampling for most applications. However, using the hardware sampling fewer methods are normally optimized and the optimization time has been drastically shortened. This is achieved without using the more extensive information that is gathered through hardware sampling compared to software sampling. The plausible explanation for this is that hardware sampling produces a more accurate profile of a program than software profiling does.

One factor in decreasing the optimization time is that hardware sampling does not promote methods higher up in the call stack as the software sampling does. This limits the code growth when optimizing since less inlining can be done in

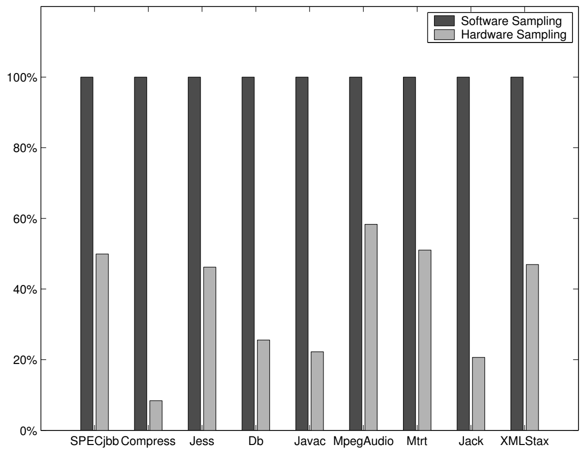


***Figure 6.1:*** *Normalized time spent on optimizing methods hotspotted with software and hardware sampling.*

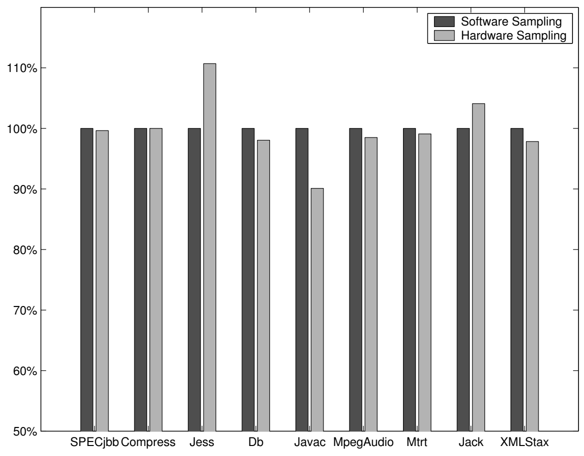


***Figure 6.2:*** *Normalized performance difference on the various benchmarks between hardware and software sampling.*

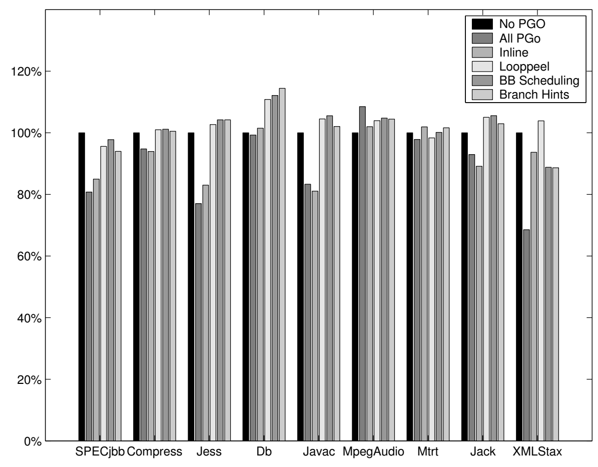


***Figure 6.3:*** *Comparison of the size of the code when using different PGO against using static optimization. PGO uses all implemented profile guided optimizations. Inline, LoopPeel, BB Schedule and Branch Hints use only that particular PGO while all other optimizations used are static.*

methods lower in the call stack. Optimization time does not scale linearly to growing code size, so much of the shorter optimization is most likely a result of less inlining.

The decreased performance of 213_javac probably comes from missing methods higher in the call chain where inlining has a large impact on performance. This has not been proved but should be the reason for the performance loss.

## 6.3 Profile Guided Optimizations

This section compares how different profile guided optimizations affect the size of the code, optimization time and performance. The compared profile guided optimizations are inlining, loop peeling, basic block scheduling and branch hints. They are both compared individually and combined against static optimizations. All versions of JRockit use hardware sampling and optimize the same methods.

Figures 6.3 and 6.4 show data for compilation overhead. Figure 6.3 shows the size of the optimized code expansion using different profile guided optimizations compared to only using static optimizations. In Figure 6.4, a comparison of the optimization times for the profile guided optimizations against static optimizations is shown. A comparison of performance is shown in Figure 6.5.

Comparing the effect of the different profile guided optimizations, it is clear that inlining has the largest effect on optimization time and code size. In most

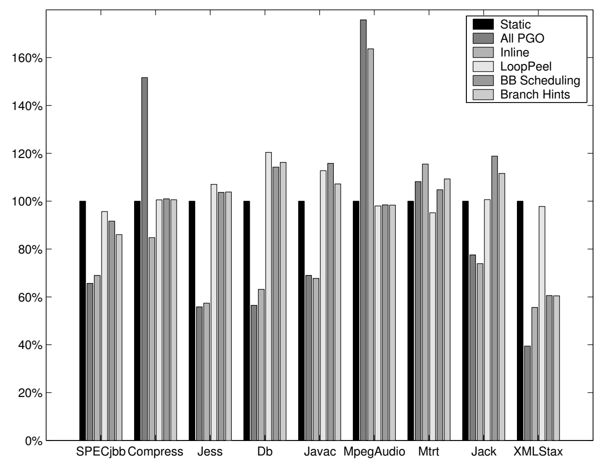


***Figure 6.4:*** *The amount of time spent optimizing code using different PGO compared to using static optimization. Inline, LoopPeel, BB Schedule and Branch Hints use only that particular PGO while all other optimizations used are static.*

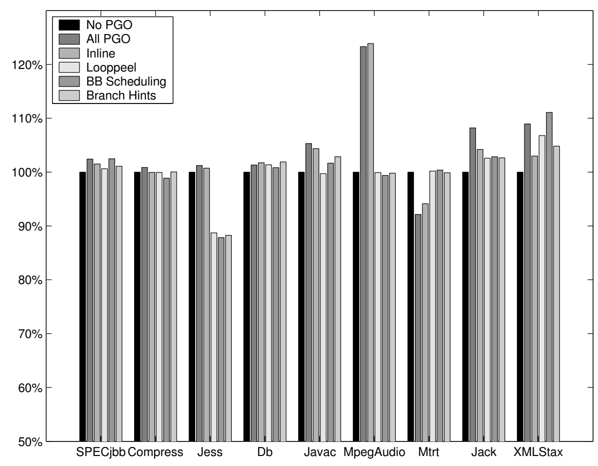


***Figure 6.5:*** *Comparison of the benchmark results when using different PGO compared to using static optimization. Inline, LoopPeel, BB Schedule and Branch Hints use only that particular PGO while all other optimizations used are static.*

cases it also has the largest effect on performance as well. The other profile guided optimizations do not affect either the size or time in a large extent, which is not unexpected. Using PGO, there is on average a 4.9% performance increase on the benchmarks, compared to only using static optimization. There are three cases that stand out, 201_compress, 222_mpegaudio and 227_mtrt.

Figure 6.4 shows a large increase in optimization time for 201_compress, but since the total optimization time is around 500 ms for the benchmark, this is of little interest.

The large increase in optimization time for 222_mpegaudio due to inlining is more interesting. The increase is around 13 seconds and comes from one single method named `q.o`[1]. The profile guided inlining decides to inline two calls to the same method inside a loop in the `q.o` method. The inlined method is large which explains the increased optimization time. The size also is the reason that the static inline function decides not to inline the two calls. When these calls are inlined other optimizations are able to remove several memory reads and array bounds checks. These optimizations results in a 25% increase in performance compared to the static optimizations.

The benchmark 227_mtrt shows a performance loss of around 6% when using profile guided inlining. It is one single method, `Scene.RenderScene`, that causes the performance loss. The problem is that inlining makes the profile inaccurate. A part of the method is thought to be infrequently executed while in reality it is. Because of the inaccurate profile, wrong decisions are made about optimizations which results in a performance loss. This shows how important it is that the profile is correct during all optimizations to be able to make correct decisions. This is even more evident when looking at the performance when all PGO:s are enabled. This gives even lower performance since all the other PGO:s also use the inaccurate profile and make wrong decisions.

## 6.4 Profile Correctness

To measure the correctness of the profile the code was split into three different sections when it was placed in memory. These parts where *hot*, *cold* and *medium*. A hot block is a block that is executed on average 0.9 times or more each time a method is entered. A cold block is executed 0.1 times or less for each method entry. The medium blocks are the blocks with a frequency between these extremes.

The optimized code was then sampled and the collected samples were analyzed to determine how the execution was divided between the different sections of code. Looking at how the samples in the optimized code are distributed be-

[1]The benchmark 222_mpegaudio uses obfuscated code, therefore the awkward name.

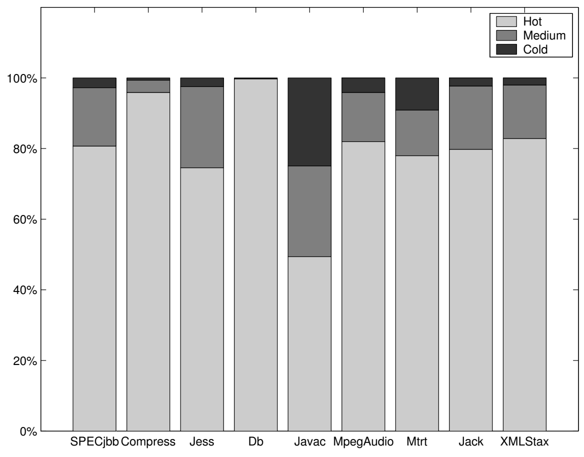


***Figure 6.6:*** *The figure shows how execution was divided between the different types of blocks for each benchmark.*

tween the sections makes it possible to see if the profile of the optimized code was correct. The benchmarks were executed with hardware sampling and all profile guided optimizations enabled.

Figure 6.6 shows how execution was divided between the different types of blocks. On average 80% of the excution time was spent in hot blocks and less than 5% was spent in cold blocks. There are two benchmarks that deviate from this, 213_javac and 227_mtrt. 213_javac has the largest deviation and spend around 50% of the execution in either cold or medium block.

That 213_javac executes more medium and cold blocks depends on the benchmark characteristics. The methods contain many branches with equal probability to be taken or not. If a method contains a few consecutive branches with equal probability for either outcome the frequency along a path in the method will quickly become small. The frequency is halved for each branch it passes along a path. That explains why there exists many medium and cold blocks in 213_javac that get executed.

Since the aim of PGO is to make the common case fast, they have less impact on methods where there is no clear common case. Since much of the execution is spent in what the profile said were cold blocks which are ignored by profile guided optimizations, the performance of the benchmark is less than optimal. Despite this problem 213_javac shows increased performance over static optimizations as shown in Figure 6.5.

The benchmark 227_mtrt has an inaccurate profile, with around 10% of the execution is in unoptimized cold code. The performance which was decreased by 8%, clearly shows the importance of using accurate profile information.

## 6.5 Hardware Sampling and Profile Guided Optimization

This section compares the current JRockit with software sampling and static optimizations to a JRockit with the hardware sampling and profile guided optimizations described in this thesis.

Table 6.2 shows how many methods that were optimized by the two different sampling and optimizations strategies. The number of optimized methods depends more on the sampling than on optimizations and are similar to the numbers in Table 6.1, which compares software and hardware sampling.

Figure 6.7 shows the difference in size of the optimized code. In Figure 6.8 the optimization time is compared. The performance difference is shown in Figure 6.9.

Comparing software sampling and static optimization with hardware sampling and profile guided optimizations, the latter in most cases show improvement in all categories measured. The size of the optimized code is significantly smaller than when using static optimizations. Using PGO the time spent on optimizations is on average 32% of the time used with static optimizations. The performance gain when using PGO is on average 4.7%. This clearly shows the advantage of using PGO to reduce optimization time and increase the performance of an application running in a JVM.

Two benchmarks mentioned earlier, 213_javac and 227_mtrt, do however show decreased performance. The performance 213_javac has increased with the use of profile guided optimizations but is still lower than when using software sampling and static optimizations. This depends on the hardware sampling as shown earlier. In the 227_mtrt benchmark, the performance loss is due to inaccurate profile as discussed earlier.

***Table 6.2:*** *The number of methods optimized when using software sampling with static optimizations and hardware sampling and profile guided optimizations.*

| Benchmark | Static opt. | PGO |
|---|---|---|
| SPECjbb2000 | 159 | 170 |
| 201_compress | 13 | 7 |
| 202_jess | 52 | 41 |
| 209_db | 24 | 17 |
| 213_javac | 261 | 175 |
| 222_mpegaudio | 75 | 67 |
| 227_mtrt | 44 | 55 |
| 228_jack | 78 | 68 |
| XMLStax | 25 | 18 |

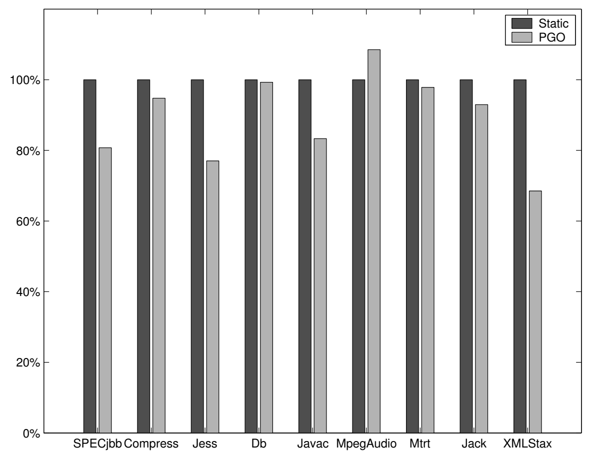


***Figure 6.7:*** *Difference in code size of optimized code when using hardware sampling and PGO compared to software sampling and static optimizations.*

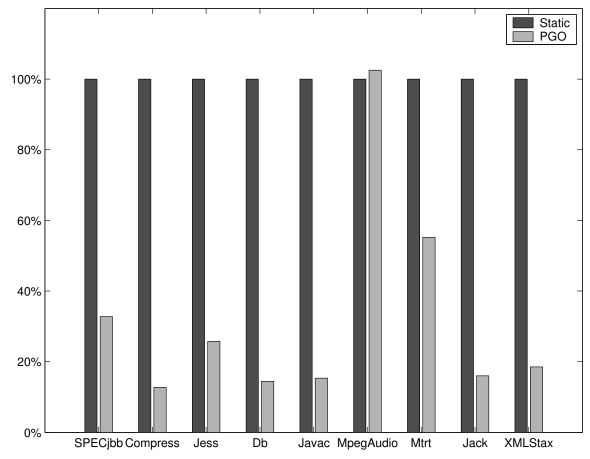


***Figure 6.8:*** *Time spent optimizing code using hardware sampling and PGO compared to using software sampling and static optimizations.*

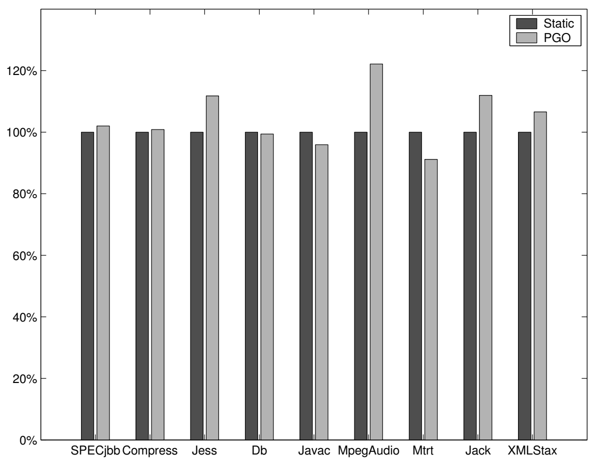


***Figure 6.9:*** *Performance of the benchmarks using hardware sampling and PGO compared to software sampling and static optimizations.*

# Chapter 7

# Conclusion

In this thesis a comparison between software sampling with static optimizations and hardware sampling with profile guided optimizations in a JVM on the Itanium processor was presented. The results show encouraging support for using hardware sampling on Itanium. The hardware sampling gathers a more accurate profile of hot methods in the running application. The information collected is also more extensive than what is possible with software sampling. Using this information to build profiles for single methods and do profile guided optimizations creates possibilities for further performance gains.

The performance gain on the benchmarks used in this thesis was on average 4.7% and clearly shows the advantage of hardware sampling and profile guided optimization. At the same time the size of the optimized code is 10% smaller than when using static optimizations. The optimization time decreased even more and is 67% less when using profile guided optimization.

While the opportunities with hardware sampling and PGO is clear, care must be taken to ensure that the collected profile is correct during the optimization. An inaccurate profile can lead to performance loss due to wrong optimization decisions. The problem is that there is no obvious way to verify that the profile is correct during optimization.

The results in this thesis clearly show the advantages of using hardware sampling and PGO on Itanium in a VEE. The use of hardware sampling enables collection of profile information with low overhead. The information collected enables the use of PGO that gives higher performance and less optimization time than using static optimizations.

## 7.1 Future Work

The evaluation of hardware sampling and profile guided optimizations in this thesis shows several new interesting areas to study. Especially concerning the problems with incorrect profiles, unsampled areas in methods and importance of finding methods higher in the call chain.

Using the branch samples it is possible to find the calling method when looking at the return branch. Is it beneficial to use this information to increase the count of methods higher in the call chain and do optimizations on these as well?

If a method contains a loop that does many iterations, many of the samples will be in that loop. In some cases a method can be hotspotted with samples only collected in a frequently executed loop. This results in a method that only contains samples for a small part of a method and several branches rely on static guessing. How is it possible to make sure that a method gets samples evenly spread through out a method? A problem is that some parts of a method might never get executed and should never receive samples. How is it possible to determine when to stop sampling?

The problem with inaccurate profiles could be solved by allowing sampling of the optimized method and then do a reoptimization of that method. The sampling information collected will give a local profile for inlined methods and also about the benefits of other optimizations. When should a method be selected for reoptimization? How should the mapping of native code and bytecode for the optimized method look to be able to extract any data?

# Bibliography

[1] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, B. Murphy, and M. Serrano. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 07(01):19–31, 2003.

[2] A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 267–276. ACM Press, 2004.

[3] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java 2nd edition*. Cambridge University Press, 2002.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, 2000.

[5] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, pages 52–64. ACM Press, 2000.

[6] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179. ACM Press, 2001.

[7] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

[8] BEA Systems. *BEA WebLogic JRockit: Java for the Enterprise, White Paper*. BEA Systems, 2003.

[9] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 259–269. IEEE Computer Society, 1997.

[10] C. Chambers and D. Ungar. Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 146–160. ACM Press, 1989.

[11] P. P. Chang, S. A. Mahlke, and W. mei W. Hwu. Using profile information to assist classic code optimizations. *Software - Practice and Experience*, 21(12):1301–1321, 1991.

[12] R. Cohn and P. G. Lowney. Design and analysis of profile-based optimization in compaq's compilation tools for alpha. *The Journal of Instruction-Level Parallelism*, 2:1–25, 2000.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[14] J. Dean. *Whole-Program Optimization of Object-Oriented Languages*. PhD thesis, University of Washington, Seattle, Washington, 1996.

[15] D. Detlefs and O. Agesen. Inlining of virtual methods. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 258–278. Springer-Verlag, 1999.

[16] C. Dulong, R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr. An overview of the Intel IA-64 compiler. *Intel Technology Journal*, Q4:15, 1999.

[17] GNU. *GNU Compiler Collection: GCC Manual, version 3.4.3*. GNU, 2004.

[18] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[19] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126. ACM Press, 1982.

[20] IBM. *IBM JVM Diagnostics Guide, 1.4.2*. IBM, 2004.

[21] Intel Corporation. *Intel IA-64 Processor Hardware Developer's Manual*. Intel Corporation, 2000.

[22] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.*, 25(4):500–548, 2003.

[23] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[24] W. mei W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing, Kluwer Academic Publishers*, pages 229–248, 1993.

[25] R. Morgan. *Building an optimizing compiler*. Digital Press, 1998.

[26] D. Mosberger and S. Eranian. *IA-64 Linux kernel: design and implementation*. Prentice-Hall PTR, 2002.

[27] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27. ACM Press, 1990.

[28] Y. N. Srikant and P. Shankar, editors. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2002.

[29] Standard Performance Evaluation Corporation. The SPECjbb2000 and SPECjvm98 benchmarks are available at `http://www.spec.org`.

[30] T. Suganuma, T. Yasue, and T. Nakatani. An empirical study of method inlining for a java just-in-time compiler. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 91–104. USENIX Association, 2002.

[31] Sun Microsystems. *Java 1.4.1 HotSpot Virtual Machine, White Paper*. Sun Microsystems, 2002.

[32] Suse Linux. Information about Suse Linux Enterprise Server 9 is available at `http://www.novell.com`.

[33] K. Vaswani and Y. N. Srikant. Dynamic recompilation and profile-guided optimizations for a .net jit compiler. In *Proceedings of the IEE Software Special on Rotor .NET*, volume 150, pages 296–302. IEE Publishing, 2003.

[34] J. Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 166–179. ACM Press, 2001.

[35] Y. Wu. Accuracy of profile maintenance in optimizing compilers. In *Interaction between Compilers and Computer Architectures, 2002. Proceedings. Sixth Annual Workshop*, pages 27–38. IEEE Computer Society, 2002.

[36] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 1–11. ACM Press, 1994.