Adaptive Optimization Using Hardware Performance Monitors

Mathias Payer

Supervising professor: Prof. T. Gross
Supervising assistant: Florian Schneider

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Laboratory for Software Technology
Institute of Computer Systems
Swiss Federal Institute of Technology
Zurich, Switzerland

February 2006

Adaptive Optimization Using Hardware Performance Monitors

Mathias Payer

Abstract

This thesis consists of two different parts. Namely a hardware interface to the Precise Event Based Sampling facilities of the Intel Pentium 4 processor and the corresponding libraries to propagate these samples into the Jikes Research Virtual Machine by IBM.

This interface makes it possible to sample events like cache or tlb misses and get the exact location in the bytecode of the corresponding method inside of the VM with all corresponding information like declaring class, references and so on. These samples may later be used for different optimizations.

The second part is a memory reordering technique at garbage collection time that uses the earlier collected samples to guide its decisions. All objects are separated into a hot and a cold space and every type gets a hotness assigned. Depending on its hotness the object is allocated in the hot or in the cold space.

The hot space then uses a copying garbage collector that orders the objects depending on the reference hotness.

Declaration

The work in this thesis is based on research carried out at the Laboratory for Software Technology, the Institute of Computer Systems, the Swiss Federal Institute of Technology, Switzerland. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Acknowledgements

First of all I want to thank Florian Schneider for his support, input and help. I could always count on his suggestions and his help when I needed it.

Second I want to thank Prof. Thomas Gross for his great lectures in Compiler Design. He taught me the basics of compilers and virtual machines and inspired me to dig deeper into this matter.

Additionaly I want to thank the Jikes RVM and the Perfmon2 mailing lists which were a great source of help. The questions were answered fast and accurate. I also saw that the people were really interested in my work and tried to help me whereever they could.

Thanks for all the support I received. This work would not have been possible without it.

List of Figures and Tables

List of Figures

4.1 PEBS control- and data-flow. 15

6.1 Bytecode distribution of different benchmarks for l2 cache misses. 34
6.2 Runtime benefit and l2 cache miss reduction for hot/cold mark and sweep collector. 38
6.3 Runtime benefit and l2 cache miss reduction for hot/cold copy and mark and sweep collector. 38

List of Tables

4.1 PEBS - ALL Sampling Format. 15

6.1 Overhead and samples per second for 2nd level cache misses (1/2). 35
6.2 Overhead and samples per second for 2nd level cache misses (2/2). 36
6.3 Number of samples and resolved percentiles for L2 misses. 36
6.4 Best intervals for specific benchmarks. 37
6.5 Relative execution time for 1st level cache misses and hot/cold copy collector (1/2). 39
6.6 Relative execution time for 1st level cache misses and hot/cold copy collector (2/2). 39

C.1 Command-line arguments. 55

Contents

Abstract ii

Declaration iii

Acknowledgements iv

List of Figures and Tables v

1 Introduction 1

2 Preliminaries 3

2.1 Intel Pentium 4 and event based sampling 3

2.2 Sun Java Environment [4] 4

2.2.1 Programming Language 4

2.2.2 Virtual Machine 5

2.3 IBM Jikes RVM [5] 5

2.4 Garbage Collectors 6

2.4.1 Reference Counting 6

2.4.2 copy-space 6

2.4.3 Mark & Sweep 7

2.4.4 Generational 7

3 Related work 8

3.1 Performance counters for (P-)EBS 8

3.2 Profiling for Virtual Machines 11

3.3 Garbage Collectors 12

4 Concept **14**

4.1 Interface for hardware performance monitors 14

4.1.1 Kernel interface 15

4.1.2 User-space library (libpebsi) 16

4.1.3 JNI Bindings 16

4.1.4 Sample handling 17

4.2 Hot cold garbage collector 17

4.2.1 Mark and sweep space 18

4.2.2 copy-space 18

5 Implementation **19**

5.1 Interface for hardware performance monitors 19

5.1.1 Kernel interface 20

5.1.2 User-space library (libpebsi) 21

5.1.3 JNI Bindings 24

5.1.4 Sample handling 25

5.2 Hot cold garbage collector 29

5.2.1 Mark and sweep space 30

5.2.2 copy-space 30

5.2.3 Changes in the scanning algorithm 31

6 Discussion and results **33**

6.1 Interface for hardware performance monitors 33

6.2 Hot cold garbage collector 37

6.3 Further work 40

Bibliography **41**

Appendix **43**

A Used benchmarks **43**

A.1 DaCapo Benchmarks 43

A.2 SPEC JVM98 Benchmarks 44

A.3 Other Benchmarks 45

B Changed files **46**

B.1 Changes inside the Jikes RVM 46

B.2 Files for libpebsi userspace library 50

B.3 Changes inside the Jikes MMTk (Memory Management Toolkit) 51

C Command line arguments **54**

Chapter 1

Introduction

Compilers are currently using two types of information for optimizing, profiles and low level information. Profiles are platform independent and very popular. A lot of research has already been done in this field. The other source of information comes from the low level hardware. This kind of data is relatively new and is still to be explored and used in compilers and virtual machines.

Most modern processors offer the possibility to sample or count performance related events. These monitors offer many different events like first or second level cache misses, DTLB misses, pipeline stalls and so on. Depending on the hardware implementation some events can be counted in parallel.

In a virtual machine these captured samples can the be used to optimize the running code and to focus these optimizations to the real hot spots. It is also possible to differ between events and optimize parts of the code for cache misses and other parts for pipeline stalls. The virtual machine can then directly benefit from these optimizations.

Because the Pentium 4 sampling mechanism is mostly implemented in hardware it is relatively cheap. The complete sample handling mechanism is done in microcode on the processor itself.

One of the tasks for this master thesis was to develop a library that transfers these samples into the IBM Jikes RVM (research virtual machine). The most important challanges were that the samples should be very precise and come with a low overhead. Using the hardware sampling mechanism it is possible to achieve very accurate samples. The precise location (IP) where the event happened is found.

The second task of this master thesis was to implement an optimization that uses the gathered profiles. Most of the gathered samples were from cache misses and showed that

bad object locality was a very big problem. The proposed new garbage collector separates the objects into hot objects and cold objects. Depending on the number of samples a type gets, it is either hot or cold. If objects are hot and are newly allocated, they will be copied into the hot space. This hot space is then handled by a copying garbage collector that reorders the objects depending on their locality and tries to bring often accessed objects and fields into the same cache line.

The next two chapters will show the preliminaries and related work like other performance monitoring tools and related papers. The concept chapter will focus on the general layout and concept of the PEBSI library and the sample handling inside the VM. Design criteria and special coding details are discussed in the implementation chapter.

The last chapter will focus on the discussion of this thesis, present some benchmarks and show possible further work.

Chapter 2

Preliminaries

This chapter presents all preliminary information that is needed throughout this thesis. There is information about the event based sampling in general and the Pentium 4's implementation in special.

Another important section is about the Java language. It has information about the source to byte-code compilers and about the virtual machines and their optimization potential. The virtual machine of choice for this thesis is the IBM Jikes RVM and it also is presented here.

The last section tries to give some details about garbage collection and the different used algorithms.

2.1 Intel Pentium 4 and event based sampling

Almost all modern processors have a interface to sample different events happening at the hardware level. The usual implementation is a counter that is incremented on every event and that may trigger an interrupt on overflow. Not every event is sampled because the data flow would be way too high. So one can specify a number n and set the counter to -n (0xffffffff - n). Then the counter is increased on every event and will trigger an overflow interrupt when it reaches the top (0xffffffff + 1 = 0).

When this overflow happens, the operating system executes a special interrupt handler that saves all needed data like processor registers and IP into an array of samples.

Depending on the application it is also often enough to just count the number of events. Therefore it is possible to read the number of events after a program has terminated.

The Pentium 4 also offers these hardware performance monitors and is able to count

many different events. So event based sampling by software would be possible on the P4 but there are two problems. Software and interrupt based sampling would be very inaccurate because the P4 has a very long instruction pipeline (from 20 to 31 stages) and depending on when the interrupt was triggered and delivered one could miss the exact location by a pipeline's length. The second problem is the speed of a software sampling solution. Some time is needed to save all needed data (and other stalls and misses may occur) and the interrupt must be handled by the operating system and the interrupt handler.

Thats why the Intel engineers developed a different solution. They included microcode based precise event based sampling on the CPU itself. So additionally to the existing HPM features it is possible to specify a memory region and to setup the processor to sample the events by itself.

Every time when the counter overflows the P4 will then execute a special microcode that saves all registers into the specified memory region. Another feature is to collect all sampled branch records in a special buffer. Please see [1] for a detailed description of the PEBS facilities.

These samples can then be read by the operating system and may be transferred to user-space where they can be used by normal programs.

The samples are often used for monitoring events like cache or TLB misses and to discover hotspots that are executed often. Using this information one is able to optimize the code (recompilation using a specific optimization, reordering the code and many others) and make the program faster. Another usage is for statistical calculations and benchmarking

2.2 Sun Java Environment [4]

Java really consists of two components. The programming language and the compiler that translates source-code into byte-code and the virtual machine that runs this byte-code.

2.2.1 Programming Language

Java is a strongly typed object oriented language, developed by Sun Microsystems in the early nineties. The biggest difference between Java and C or C++ is that Java is compiled to byte-code rather than machine code. This primitive byte-code is then run on a virtual

machine.

One advantage of the Java language is that memory management is completely handled by the virtual machine. The programmer just instantiates objects and uses them. As soon as they are no longer referenced the garbage collector will free them without any user interaction.

Compiler

There are many different compilers for the Java language that transfer the source-code to byte-code. The most prominent ones are the Sun Java Compiler which is included in the Sun Java Software Development Kit (Sun JSDK), the blackdown repackage for Linux and the IBM Jikes Java compiler.

Java compilers are a fairly easy task compared to C/C++ compilers because most of the optimization is done at runtime in the virtual machine. The compilers simply translate the source into the stack based byte-code without too many sophisticated optimizations.

2.2.2 Virtual Machine

The virtual machine executes Java byte-code. The basic principle of the byte-code is a stack based machine. Most virtual machines include profiling and different optimization levels. Depending on the sampled frequency different optimizations and recompilations of byte-code methods are applied.

The JVM also verifies the byte-code during the loading phase. This technique makes it possible to control the allowed instruction sequences in programs.

The VM forbids all access to raw memory or uncontrolled jumps by the executed byte-code.

Memory management is also handled completely by the virtual machine. It keeps track of active objects and collects no longer used ones during the garbage collection phase.

For a detailed overview and implementation details of JVMs have a look at [3].

2.3 IBM Jikes RVM [5]

The IBM Jikes Virtual Machine is a complete open source Java virtual machine that derived from an internal research project. This VM consists of a fast pattern matching baseline compiler that translates new byte-code into machine-code. When a method is

sampled often enough it is recompiled by the optimizing compiler. The optimizing compiler then goes through different stages and does an increasing amount of optimizations at every stage.

The memory management in the Jikes RVM is very flexible and detached from the rest of the VM. There are many different garbage collectors available. Starting from a simple copy-space collector up to a generational garbage collector with a nursery and a mark and sweep mature space.

This interface is extendable and can be changed easily. The memory interface has access to some VM internals through special classes and interfaces [6].

2.4 Garbage Collectors

Garbage collectors are needed for object oriented languages without explicit memory management. The runtime system takes care of memory allocation and deallocation. Objects are considered garbage and can therefore be collected if there are no references to them from live objects.

There are many different algorithms for garbage collection. All have their specific strengths and weaknesses. The most prominent and important ones are shortly presented here.

2.4.1 Reference Counting

Following the "No GC" collector this is the simplest possible garbage collector. Every object keeps track of how many objects it is referenced. If the reference count reaches zero then the object can be freed. The disadvantage is a big overhead as the runtime system has to check and change the counter at every reference assignment. Depending on the implementation this algorithm can also have problems with circular references.

2.4.2 copy-space

This algorithm divides the available memory into two different spaces. All new objects are allocated in one space. At collection time all life objects are copied to the other space. This is done using a tree traversal starting from a root set. All remaining objects after the copy process are garbage and can therefore be collected. Depending on the tree traversal algorithm this collector can increase data locality. The disadvantage is that all references

need to be updated as the addresses change.

2.4.3 Mark & Sweep

Mark and sweep is a tracing garbage collection algorithm. At garbage collection time all objects are considered white. Now all objects that are in the root set (accessible from a stack, machine registers, program counter and global objects) are marked as grey. As long as there are grey objects pick one from this list and mark it black, then mark all direct accessible objects from the target as grey.

Now all white objects can be collected and all black objects are alive. Using this algorithm we traverse all objects in the object space and check all references and test if the objects are alive. All garbage objects will be marked as white and can be collected in the next step.

This algorithm does not copy the objects so there is no need to update references, but the heap can be fragmented over time.

2.4.4 Generational

Generational garbage collectors are currently used in all important virtual machines. The heap is divided into different regions, starting with a nursery for very young objects and other spaces for aging objects. This algorithm uses the fact that most allocated objects die young. So there is no need to scan the complete heap for garbage. It is sufficient to scan the nursery. Live objects get older and are then propagated into the next space.

This technique is very efficient as only parts of the heap are collected and not all objects need to be scanned. A complete heap collection is only done if the available free memory in the mature space runs out.

The IBM Jikes RVM currently uses a two phased generational garbage collector with a mark and sweep mature space as default configuration for the production system.

Chapter 3

Related work

This chapter is about related work that inspired this thesis or was helpful in one way or another. There are different ways a virtual machine can gather information for the adaptive optimizations. Profiling is platform independent and already covered in the literature. Hardware performance counters are a relatively new field and bring direct hardware feedback into virtual machines.

The first part of this chapter presents different hardware sampling facilities and interfaces. This type of information is then used in a new type of virtual machine that does not only support profiling.

The last section will present garbage collectors that use this locality information to reduce the number of misses.

3.1 Performance counters for (P-)EBS

Currently multiple interfaces for hardware performance monitors exist as patches for the Linux kernel. All of them can be used for event counting and some may be used for precise event based sampling. It also depends on the implementation if they allow both global context sampling and virtual process sampling. Global sampling measures all running processes and the complete kernel time, whereas virtual process sampling includes virtual per-process counters. These per-process counters are set and read at every context switch. So only a process or process group with attached counters is sampled.

Brink and Abyss [7]

Brink and Abyss were one of the first performance measuring tools that supported the hardware performance monitors of the Pentium 4 processor. It was developed at the College of Engineering in Bucknell. Brink is the configuration parser that compiles a XML configuration into the specific settings for the hardware registers. Abyss then uses these data to setup the kernel module.

The software package comes with patches for the (antique) version 2.4.26 of the Linux kernel. This patch is then applied to the kernel source and adds the functionality for system wide sampling by adding a user-space interface to the HPM registers and by providing a PEBS overflow interrupt.

In the default implementation it is not possible to sample a single process. The extension by Flavio Pellanda supports virtualization and per process accounting. To implement these features the kernel saves all HPM information at every task switch and restores them when the process is resumed.

The drawbacks of this implementation were that the samples had to be parsed by a special user-space program and then copied sample per sample into the Jikes RVM. This is a very slow process and leads to many context switches.

Another problem is that the patches are no longer actively maintained. There has been no change since July 2004 and the traffic on the mailing list is very low. Brink and Abyss rely on a version of the Linux kernel that is no longer usable because a lot of newer hardware is not supported.

Perfctr [8]

Perfctr is a low level library that allows access to the hardware counters and events of many different processors, including the Pentium 4. But Perfctr itself has no built in support for PEBS. Additionally there is only very little documentation about the API, so one has to use the sources.

The basic structures for PEBS seem to be available, so it would be possible to extend Perfctr to include PEBS. The most dominant problem is, that Perfmon2 will supersede Perfctr as announced on the mailing list.

PAPI [9]

This is a user-space library that uses Perfctr for the low level hardware access. PAPI hides the hardware quirks from the user and tries to offer a constant interface on all platforms, if a similar event is supported.

Event based sampling is also supported, but only via software and is therefore imprecise and very expensive. PEBS itself is not supported for the Pentium 4.

Hardmeter [10]

Hardmeter is an extension of an old version of Perfctr that supports PEBS somehow. Unfortunately there is nearly no english documentation available and the project is no longer maintained.

PEBS is supported through a user level interface and multiple header files that provide the correct values for the performance registers.

This original version only supports a limited number of samples and will then shut down and stop collecting.

Hardmeter for recent kernels

To overcome the problems of the original Hardmeter version we ported Hardmeter to a more recent Perfctr version that could be used with a recent 2.4 kernel. A problem was that Hardmeter was very much out of date and could not be ported to a 2.6 kernel easily.

This version was also patched with a ring buffer that made it possible to use continuous sampling. But the old sources had some timer and thread synchronization issues that caused the sampling to stop after a limited amount of time. The problem was that an interrupt was not delivered and then the sampling stopped because the buffer was filled and the CPU entered an exception state.

A runtime interface for hardware performance monitors in a dynamic compilation environment [12], [13]

This is the preceding master thesis by Pellanda Flavio. He used an adaption of Brink and Abyss to copy the samples into the user-space. First of all Abyss reads the samples from the kernel interface and transfers the samples into user-space and will then copy them into a shared buffer. This buffer is then read by the Jikes JNI library that is used to transfer the samples one by one into the virtual machine.

Unfortunately this is a very slow process and imposes a large overhead on the runtime system. But the process was very reliable and if there were not too many samples the overhead was acceptable.

Perfmon2 [11]

HP and Intel are developing a library for hardware performance events and the corresponding kernel modules. The user-space library eases the usage of event counting over different hardware platforms. But it is also possible to use the kernel module by itself and attach to the system calls.

The kernel interface supports PEBS and the complete structure is very modular and extendable. The buffer format for the different sampling types is pluggable and can easily be exchanged.

This project is also very active and a new version is released about every month. The update cycle is normally quite easy as only small API changes take place. A drawback of this interface is that only very few documentation exists, but there are sample programs and together with the Intel documentation it is possible to program all different events.

The mailing list of this project is very responsive and it is possible to get a fast reply to a question. A problem with this interface may be the early stage of development, because there can still be changes to the system calls and to the whole interface.

Currently the project is being integrated into the developer version of the Linux kernel. This means that it is no longer needed to patch the kernel in the future because the Perfmon2 interface would already be included in every running kernel, provided that the kernel modules are activated and loaded.

3.2 Profiling for Virtual Machines

Profiling is an important feature of modern virtual machines. It is used to monitor and trace events that occur during run time. These events can then be tracked. The assembled costs are fed into the compiler and the run-time system. The adaptive optimization system then decides where it focusses.

The VM can generally profile every bytecode instruction, account the number, code and memory location of the event. Profiling for new instructions can be used to locate places with high memory usage. The profiler will also use timers to keep track of hot spots

in the code that use the most processor time. This execution frequency is then used to specify which methods need to be recompiled at a higher optimization level.

A profiler can work completely in the virtual machine and is therefore operating system independent.

3.3 Garbage Collectors

The garbage collector is a part of every virtual machine that traverses all available objects and marks them as alive or collects them. This traversal process can be used to optimize the object distribution in memory.

Many different garbage collectors have been proposed and offer an advantage for specific situations. The following two garbage collectors focus on memory and cache locality. They try to compact objects that are used after another, thereby minimizing the number of cache misses.

The Garbage Collection Advantage: Improving Program Locality [14]

This paper tries to improve cache and program locality using special forms of garbage collection. They differ between hot and cold objects. Whereas hot objects are optimized and placed close to each other during garbage collection.

In the paper they use software monitoring with an overhead between 0.6% and 3%. If an object type caused enough field accesses it is marked as hot and will then be optimized. Otherwise if the object has no hits for a longer period of time the object type can also cool down again and is no longer optimized. Using this technique the algorithm is able to detect phases where different types of objects are hot and need to be placed close to each other.

They also discuss that languages without strong type interface like C have a hidden performance cost. The memory management cannot move regions to improve cache locality because not all pointers are known. So systems like Java VMs may be able to improve their performance by reordering objects depending on the locality of the data.

Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times [15]

This paper tries to improve the locality of Java applications by object colocation and a locality based traversal of live objects.

They use a special allocator that places often referenced objects next to the newly allocated object by already allocating extra memory for future objects. Analysis for referenced objects is done on a per class basis. The garbage collector then assures that all relevant objects are moved in the right order to improve cache locality.

A drawback of this method is that they use a special profiling run to gather data. So without the profiles they are unable to detect the correct class layouts.

Chapter 4

Concept

It is necessary to define a clear design before something is implemented. In this master thesis there were two different aspects that needed to be defined. First of all a library is needed that can control the hardware performance monitors and is able to transfer the obtained samples into the virtual machine.

The second part is the "hot-and-cold" garbage collector that uses the additional information from these monitors.

The first part of this chapter will present the design decisions for the HPM library, the corresponding kernel interfaces and the user-space handling. Later the concept of the separating garbage collector is shown.

4.1 Interface for hardware performance monitors

As much as possible should be handled by the hardware itself to limit the overhead for collecting samples. Special care is taken that only samples of the selected process are collected because the hardware will not differ between different running tasks. Then the samples are handled by a special user-space library, that communicates with both the kernel module and the virtual machine. The samples are copied into the virtual machine process by JNI calls and are then handled by the PEBSI thread.

From this point on the samples can be used for optimization and to gather more information, like declaring class, the method which caused the sample or field and method references.

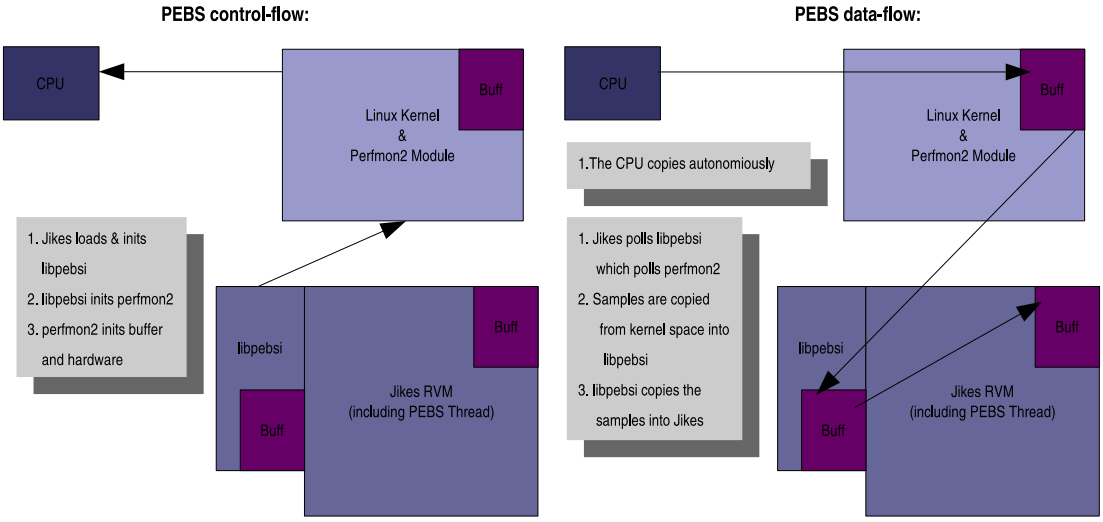

Figure 4.1: PEBS control- and data-flow.

4.1.1 Kernel interface

The kernel interface offers the possibility to set up and access the hardware performance monitors from the user and kernel-space. This service is available by a kernel module that plugs into a current developer kernel.

The kernel interface is then programmable to sample specific events. After allocation of a kernel buffer to store the samples, the PEBS hardware of the Pentium 4 is initialized. During the initialization special parameters like buffer size and the initial counter can be specified. It is generally not possible to collect every sample because the CPU would then be completely occupied with sample collection. So a special counter is initialized that increases at every event. Only when the counter overflows an interrupt is generated and a sample is taken. Then the counter is reset to the original value.

The performance monitor is suspended if the buffer is full and there is no more space available. This is possible if the user-space program is not collecting the samples fast enough. As soon as the buffer reaches the low mark the monitor is reactivated.

Samples are stored in a special format. The hardware supports two different sampling formats. EIP, where only the instruction pointers of the samples are saved and the ALL sampling format which takes a snapshot of the most important registers.

EFLAGS	EIP	EAX	EBX	ECX	EDX	ESI	EDI	EBP	ESP
0	4	8	12	16	20	24	30	32	36

Table 4.1: PEBS - ALL Sampling Format.

Another feature is counter fuzzying. A possible problem that can occur when handling loops is that all the samples are taken at the same position, because the loop causes exactly as many misses as the counter reset value. To avoid this problem the last bits of the counter size are randomized at every interrupt. The value will be in the same region but not exactly the same. This avoids misleading statistics.

As we are capturing events using a hardware interface, special care must be taken for multi tasking operating systems. Without special adjustments the hardware monitors would sample all running processes, including all kernel operations. Per process virtualization is introduced to limit the sampling only to desired processes. The kernel reads and saves all counter information before a context switch and restores them if the process starts again. This information is saved in an additional structure in the kernels process table.

4.1.2 User-space library (libpebsi)

PEBSI, the Precise Event Based Sampling Interface is a special user-space library that directly calls the exported Perfmon2 system calls without the use of the user-space Perfmon2 library.

All available PEBS events of the Pentium 4 are supported and are usable by a simple to handle interface. The developer can select the sampling interval, the size of the buffer in the kernel and the sampled event.

It initializes all the kernel structures and sets the values of the MSR registers. The offered functions are easy to use, only select the desired event, buffer size and counter. The setup routine will then issue the corresponding system calls and communicate with the kernel module.

The samples can then be read by another function call. This call copies the samples from kernel-space to user-space and saves them in a local buffer. To limit the number of slow transitions between the kernel context and user context as many samples as possible are copied and then relayed to the user program.

4.1.3 JNI Bindings

Another part of the PEBSI library are the JNI bindings. These definitions are needed to close the gap between the virtual machine and the libpebsi native code. Using these bindings the PEBSI library is pluggable into all available VMs that support JNI. The

virtual machine can load the shared library and use the defined functions to setup PEBS and start collecting samples.

These samples can then be used in a special evaluation thread. If bindings to the memory management and to internal structures are available the runtime system can also be instructed to start specific optimizations. A special collector thread for the Jikes RVM is implemented and uses the samples from the libpebsi library.

4.1.4 Sample handling

Samples are handled in a special system thread inside the virtual machine. This thread configures the desired PEBS event by JNI calls to the libpebsi library. The samples are then read and processed in a loop.

Samples are read into a buffer inside the VM. The thread then resolves the byte-code instruction that caused the sample and tries to gather more information. Some basic statistics and type information are also updated.

The PEBSI thread then collects other interesting information like field and method references. These information are then relayed to the memory management and to the optimizing compiler where they can be used.

4.2 Hot cold garbage collector

One of the biggest causes for the speed gap between object oriented languages and procedural languages is the added indirection for objects. Java programs have many references to other objects. When these references are followed a lot of cache misses occur. These misses take a lot of time to resolve.

If these cache misses could be reduced then the overall speed of the program would increase. This is possible if objects that are often accessed after each other reside in the same cache line. So only the first miss needs to be resolved and later objects are then already loaded in the same cache line. To achieve this locality the objects must be placed next to each other in main memory.

One possible way to do this is to separate hot and cold objects from each other. If all hot and often accessed objects are allocated in a special space, then the possibility that two objects are placed in the same cache line or on the same page and used after another are higher as if all objects are distributed randomly in the address space.

Additional information is needed to know if an object is hot or not. This information is gathered during the processing of the samples in the PEBSI thread. The heat of every declaring class is increased for every sample. The needed heat for a hot type is adjusted as more and more types get hot.

If a specific type did not trigger any samples during some time then it can get cold again and will then again be allocated in the cold space.

4.2.1 Mark and sweep space

This generational garbage collector allocates all objects in a nursery. If the objects survive the first collection they are either placed in the hot space or in the cold space. This is depended by the heat of the type and the current hotness limit.

As soon as an object is in one of these spaces, it is no longer moved. It will stay at the same memory position until it dies and is then deallocated by the garbage collector.

This collector uses the idea of a special allocator for hot types as presented in the paper *The Garbage Collection Advantage: Improving Program Locality* [14].

4.2.2 copy-space

This collector extends the idea from the mark and sweep space and also uses the garbage collector to relocate the hot objects depending on the heat of the field references presented in *Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times* [15]. Objects are first allocated in the nursery and then separated to the cold mark and sweep space or the hot copy-space.

During a garbage collection of the hot space objects are copied depending on the hotness of their fields. If an object is copied, then normally all fields are copied as they appear. This collector orders the fields depending on the number of sampled field references. So that the hottest field is copied first. This ensures that the hottest field of this object is close to the object itself and will lower the number of cache misses.

Chapter 5

Implementation

This chapter will focus on implementation details of the PEBSI environment for the Jikes RVM and the garbage collectors that use the samples. This environment consists of four layers.

The first layer is the kernel interface that handles interrupts and hardware details. A user-space library then uses the kernel system calls to offer an easy to use interface to the complicated hardware details. The next layer is a JNI interface that wraps the library calls and samples into Java-compatible methods and objects. The last layer is the sampling thread inside the VM that uses the JNI interface to sample data. This information is then used in the runtime system for optimization.

The second part of this chapter will describe the implementation of the two additional generational garbage collectors that implement an additional mature space for hot objects.

5.1 Interface for hardware performance monitors

This section will show how the different parts of the PEBSI system are implemented. The part about the kernel interface will highlight the different approaches that were taken to communicate with the hardware. Then the implementation of the user-space library is discussed, especially the mapping and coding of the different events. The mapping from C code and direct memory access to Java and objects are discussed in the JNI section. At last the section about sample handling tells how the samples are interpreted and used inside the VM.

5.1.1 Kernel interface

The final version of the kernel hardware interface uses the Perfmon2 kernel patch to access the hardware counters. This patch is actively maintained and extended. It also implements all needed functionality.

Two other tools were also evaluated, namely "Brink and Abyss" that was used in the preceding master thesis and Hardmeter, another PEBS tool that was developed at an university in Japan. Unfortunately both tools had their drawbacks and were therefore replaced by Perfmon2.

Hardmeter plus extension

Hardmeter is an extension of the Perfctr kernel patches. Perfctr implements access to hardware performance monitors of many different processors. It is actively maintained but only supports counting of events and no sampling.

The Hardmeter extension implements an overflow interrupt and is able to use the PEBS sampling feature of the Pentium 4. This patch was first released in 2003. Problems of this extensions are that only an old version of Perfctr is extended and that the development stopped in 2003. The basic version is therefore only available for an old Linux kernel. Additionally Hardmeter is only able to capture a fixed amount of samples and stops afterwards. The intended use was for sampling an attached thread and not for adaptive optimization.

The most actual version was taken as a basis for a port and extension of the Hardmeter code. First of all Hardmeter was ported against an actual version of Perfctr for the latest 2.4 kernel version. Because of many changes in the kernel structures it would not have been possible to port the code to a 2.6 kernel version short of reimplementing the extension.

Another extension to the original code is a ringbuffer that makes continuous sampling possible. If the buffer is full, then sampling is stopped until the level drops under a specified threshold.

But due to continuing performance and runtime problems this path was abandoned. The implementation had some synchronization and locking problems that caused the hardware to stop sampling or raise a hardware exception.

Perfmon2

Perfmon is a software project that consists of kernel modules for different HPM architectures and the corresponding user-space libraries. It is maintained by HP. The kernel modules export low-level access to the monitors. The library then uses these exported monitors to implement platform independent monitoring.

The P4 kernel module also supports PEBS. It implements the overflow interrupt that is triggered every time the local sample buffer in the kernel is full.

This project is actively extended and maintained and will soon be included in the mainstream kernel. No kernel patch will be needed to use the perfmon features in the future.

The implementation is very reliable and is extendable by the direct access to the performance monitors. A library can directly use the kernel system calls to set up sample collecting for a specific process if the process owners match. Self monitoring is also possible. The samples are then read by other system calls.

The following system calls are available and used in the libpebsi library:

- *pfm_create_context*, *pfm_load_context*, *pfm_unload_context*
- *pfm_write_pmcs*, *pfm_write_pmds*, *pfm_read_pmds*
- *pfm_start*, *pfm_stop*, *pfm_restart*
- *pfm_create_evtsets*, *pfm_getinfo_evtsets*, *pfm_delete_evtsets*

A lot of features are already implemented and can be used in programs and libraries. An example is sample-counter randomization that changes the counter value at every interrupt by a random number the size of a specified bitmask.

5.1.2 User-space library (libpebsi)

This shared library is basically an independent software package. The software is written in C and comes with *Makefile*, *README* and *INSTALL* files. The main function is to hide the complexity of the hardware registers. It offers an easy to use, fast and dynamic way to setup an event type and to gather samples.

The usage pattern of the library can be split up into the following steps:

1. Loading the library

2. Initialization and event selection
3. Start of the collection phase
4. Main phase with an undefined number of calls to pause or resume collecting and to get available samples from the kernel module
5. End of the collection phase
6. Removal of the kernel structures and shutdown of the libpebsi library.

Initialization and event selection

Event sampling can be initialized by calling the exported method *setupCollecting*. The pointer event must lead to a string representation of the event. Supported events are listed in the appendix.

```
/**
 * Prepare the kernel interface and select the event
 * event - string representation of the event to sample
 * interval - every n-th sample is collected
 * buffersize - size of kernel buffer
 * debug - debug level (0 low - 9 max)
 */
int setupCollecting(const char* event, int interval,
    int buffersize, int debug);
```

This call programs all hardware monitors to sample the specified event. The allocated sample buffer is then mapped into user-space and everything is prepared to start sampling. An additional buffer is allocated only in user-space to keep samples. Using this additional buffer the hardware can collect more samples even if the user was not yet able to collect the last samples.

Start of collection phase

With the call *int startCollecting(void);* the processor monitors will start collecting samples. These samples are then written into the previously allocated buffer.

Main phase

The two functions *int pauseCollecting(void);* and *int resumeCollecting(void);* can be used to pause and resume the hardware. They use a system call to enable or disable PEBS, but this may take some time as write access to the performance monitors costs about 1'000 CPU cycles.

Using the method *getSamples* it is possible to copy already collected samples into a given buffer. This method will first check if there are some samples in the user-space buffer and copy them to the user.

If the specified buffer is larger, then the kernel buffer is polled and checked. These samples are then copied to the user-space buffer and to the user. The kernel buffer is reset after copying.

```
/**
 * This reads the samples out of the kernel buffer and
 * saves them in user space.
 * The samples are then accessible via the given
 * samplebuffer pointer
 */
int getSamples(int* samplebuffer, int count);
```

End of collection phase

Using the method *int stopCollecting(void);* the hardware performance monitors will stop collecting samples. The remaining samples can still be read because the kernel structures are not removed yet.

Shutdown and removal of the kernel structures

With a call to *int shutdownCollecting(void);* it is possible to shutdown the collection and to remove the internal kernel structures. The process state after this call is the same as before *setupCollecting*.

There is an example in *examples/monitor.c* which implements all details and samples a specific event.

5.1.3 JNI Bindings

The Java Native Interface bindings wrap the libpebsi functions so they can be used inside from a JVM. Strings are mapped to char arrays, Java method calls map to C functions, memory pointers become objects and basic types and arrays are transferred to their Java equivalents.

Basic methods mapping for initializing, starting and stopping the hardware performance monitors are easily mapped and need only minor adaption for exception handling and conversion of UTF strings into char arrays.

The method to get samples into the JVM lead to some difficulties. The very first implementation generated a new object for every sample and ordered these objects in an array and then returned this array to the JVM. This was a very slow process because JNI functions are handled with special care for memory management and the calls to generate new objects always had to cross the border into the VM code. Other problems were constructors and destructors that were called for every object and the additional load on the garbage collector. All these drawbacks together induced an runtime overhead of about 5% which was not tolerable, although this solution was already faster than delivering only one object per JNI call.

The next solution was an integer array that was preallocated inside the VM and then handed to the JNI function. A possible problem was a concurrent garbage collector that could move the array in memory during the sample copying which would lead to a random write somewhere in the VM's memory. So the array was pinned down by a JNI function and released after copying. The Jikes RVM implements the pinning process by copying the complete array to another temporary memory location. Upon release the array is copied back to the old location and overwrites the old data. This was a very slow process and lead to a high load for the garbage collector because the temporary array was deposed after use. This solution induced an overhead of about 3% which was still not tolerable.

As a final solution the pinning process is omitted and the samples are written directly to the jint array in VM space. This is only possible because the JNI function does not allocate any memory and will therefore not trigger the garbage collector. Although this violates the JNI specification, it works for the Jikes RVM and should also work for other virtual machines. The specification states that a JNI call may not run concurrently to system threads like the garbage collector. Therefore the array will not be changed during the write process.

The statement *System.loadLibrary("pebsi");* must be called to load the shared library *libpebsi.so* from the root directory of the Jikes RVM. The following methods are available in the Java virtual machine:

```
native void pebsiSetup(String event, int interval,
  int maxSamplesPerRound, int debug)
native void pebsiStartCollecting();
native void pebsiStopCollecting();
native void pebsiPauseCollecting();
native void pebsiResumeCollecting();
native int pebsiGetSamples(int addr, int len);
native void pebsiShutdown();
```

5.1.4 Sample handling

All the samples are collected and handled in a special system thread. This thread is started if the parameters for PEBSI sampling are set and an event is selected.

The main function of the PEBSI thread is to get samples, resolve the declaring method and gather information about the miss like field references, method references and statistics.

Fast method search

All compiled methods that are accessible in the VM are referenced in an unsorted array. The only information we get from our sample is the instruction pointer. The IP must now be mapped to a *VM_CodeArray*, but the Jikes implementation of this mapping does a linear scan through all available methods and checks if the IP is between beginning and end of the current method's VM_CodeArray.

This kind of lookup is very slow, especially if it is done for every sample. So part of this work was to find a structure that holds information about all methods that is extendable for new methods. The structure must also be aware of memory constraints.

Two different proposals were debated, a hash table and a tree like structure. It shortly became clear that the hash table was not the best solution. Some of the most important drawbacks were:

1. Generation of hashcodes for all method objects is too expensive. The calculation

inside the GNU Classpath is fairly complicated.

2. Rehashing if the table was not large enough and the collision detection are very CPU intensive. There are many insertions and deletions in the data-structure due to the optimization system and method recompilation.

3. The size of the large hash table needs a lot of memory. In a standard configuration there are over 18'000 methods for the spec JBB benchmark.

4. A complicated lookup process because the IP is somewhere inside the method and hash tables map to only one value. A hash table offers direct lookup of a key. This structure needs to return the method if the address is in the given address range.

So the hash table would need a lot of memory, updating and maintaining the data structure would be expensive and due to the hashing function in Jikes the lookup would not be very fast.

A better solution is a three-levelled tree structure with lazy allocation. This tree structure covers the complete 32 bit memory address space. The first level is an array with 256 entries. So methods are separated by their first 8 bits into different subtables. These subtables then each contain 4096 entries (12 bit). Every entry in a subtable then covers a page. About 2-3 methods will then be in such a page, as methods are on average about 2kb long. These methods are organized in an array. If a method covers multiple pages then it is inserted in every page it touches.

A lookup in this structure is very fast, because the address of the IP can directly be used to access the bytecode method. During a method lookup the address of the IP is shifted to access the first and second table. At last a linear scan in the page's methods brings back the correct method. Because there are many insertions an deletions the method table is unsorted. Therefore a binary search is not possible. The overhead for sorting would not be compensated by the benefit of the binary search.

```
VM_CompiledMethod searchFastMethod(int address)
{
  /* highest 8 bit */
  int firstLevel = address >>> 24;
  /* middle 12 bit */
  int secondLevel = (address << 8) >>> 20;
```

```
    VM_Codepage cp = codepageMapper[firstLevel].
      getCodepage(secondLevel);
    return cp.findMethod(address);
  }
```

This process is very fast because it is possible to directly access the correct page. An additional linear search in an array with very few elements is also not very expensive.

New methods can also be added easily. The lookup process is nearly the same. A second level object or a page is allocated if it does not yet exist. Not many second level objects are needed, as most code arrays are close together.

Bytecode resolving

The next step after resolving the method is to resolve the bytecode instruction. The bytecode map that was added during compilation can now be used to find the corresponding bytecode instruction. Normally only special garbage collection points and GCSpy methods have these mapping tables, but they are added for every method so that it is possible to resolve instruction pointers to bytecode instructions.

The declaring class of the method is resolved if it was possible to find the bytecode instruction. Depending on the bytecode different informations are gathered and inserted into the optimization system.

Field reference accounting

If the resolved bytecode needs a field reference the basic type of the reference and its hotness are resolved. As soon as the type is hot enough each of the type's field is analyzed. The fields are then reordered depending on their heat.

The initial heat threshold is a parameter in the PEBSI thread. This threshold is adjusted during runtime. If there are many samples and hot types then the value is increased by a linear factor. Using this increasing threshold types can get cold again and will no longer be allocated in the hot area if they are no longer sampled. The PEBSI thread keeps only track about hot types and references, but does not use the collected data by itself.

Field reordering is handled by the PEBSI thread itself. If the type is above the threshold then the fields heat is increased by 1. The table is then scanned by a bubble sort to

move the corresponding field to the correct position. A bubble sort is fast enough because the fields are already ordered and the field reference will not move very far.

The memory management and garbage collector can later use this hotness information to traverse the object tree using the hotness rather than the linear order of the fields.

Method reference accounting

The sample counter of this reference is incremented if the bytecode instructions leads to a method reference. As soon as the reference's heat is high enough the method reference is added to a list of hot methods.

Statistics

Additional statistical data about the distribution of the bytecode instructions is collected if the compiler flag *RVM_WITH_PEBSI_STATISTICS* is set.

These statistics include information about the following measurements:

- Number of baseline compiled instructions
- Number of resolved baseline compiled instructions (that could be mapped to methods)
- Number of optimized compiled instructions
- Number of resolved optimized compiled instructions
- Number of static and dynamic field references
- Number of field references that are primitives (basic types like int, boolean and so on)
- Number of method references
- Number of static, special and virtual method invokes
- Number of loads and stores from and to the stack
- Number of heap array number (4 byte), heap array char (1 byte) and heap array reference loads and stores
- Number of object header acesses
- Number of unmatched bytecode instructions

5.2 Hot cold garbage collector

The main idea of this additional garbage collector is to separate often used and referenced object types from lesser used ones. One of the assumptions for this separation is that the often used types reference each other and are therefore all in the hot space. Upon pointer dereference these types can profit from their locality.

The memory management toolkit that is used in the Jikes RVM already offers many different garbage collectors. Basic collectors like a reference counting, a copy-space and a mark and sweep collector are implemented. Some more sophisticated collectors like a generational garbage collector with a nursery and a mature space are used in the default configuration. Whereas the mature space is either a copy-space or a mark and sweep space.

A new generational garbage collector is presented in this thesis. Both implementations extend the standard generational garbage collector already present in the MMTk. This toolkit offers an extendable collector system based on inheritance. There are many software parts that can be used in multiple collectors, there is a chain of collectors that extend each other. The most basic part is *org.mmtk.plan.Plan* which defines basic constants for all following collectors. One extension of *Plan* is *org.mmtk.plan.StopTheWorld* which is an abstract class that defines the core functionality of stop the world collectors. These collectors stop the complete runtime system during a collection and run uninterrupted in contrast to incrementing collectors which can be interrupted.

The collector *org.mmtk.plan.generational.Gen* is an implementation of a stop the world garbage collector. It defines basic properties including all methods and data structures of a nursery for generational collectors. This collector is extended by either the collector *org.mmtk.plan.generational.copying.GenCopy* that implements a copying mature space or by *org.mmtk.plan.generational.marksweep.GenMS* that implements a mark and sweep mature space.

The problem of this design is that our proposed garbage collector could not be implemented as an extension of *Gen* or *GenMS* because there are changes in the nursery and in the mature space. Both presented collectors use the code base of the implemented generational collector as basis and extend this code.

An additional allocator *ALLOC_HOT* is defined in the nursery. This allocator is then used in the sub classes to transfer hot objects into the hot mature space.

5.2.1 Mark and sweep space

This collector uses two mark and sweep mature spaces to keep older objects. The second (hot) space is implemented alongside the already existing space. Every place where the mature space is referenced is extended to include both mature spaces.

The following code shows the declaration of the two mature spaces and their corresponding descriptors. Both spaces need a trace for the marking phase during a collection.

```
MarkSweepSpace msSpace = new MarkSweepSpace("ms",
  DEFAULT_POLL_FREQUENCY, MATURE_FRACTION*0.7);
MarkSweepSpace msSpaceHot = new MarkSweepSpace("msh",
  DEFAULT_POLL_FREQUENCY, MATURE_FRACTION*0.3);

int MS = msSpace.getDescriptor();
int MSHOT=msSpaceHot.getDescriptor();

Trace matureTrace = new Trace(metaDataSpace);
Trace matureTraceHot =  new Trace(metaDataSpace);
```

5.2.2 copy-space

A mark and sweep space for cold types and a copy-space for hot objects are used in this collector. Both spaces are implemented alongside each other in the code. As the copy-space has different needs than the mark and sweep collector there are some other changes and extensions as well.

The different scanning and collection methods are extended to include both functionality for the mark and sweep space and for the copy-space. The code shows the definition of the two spaces. In this implementation the cold space takes 70% of the available memory and the hot space gets 30%.

```
MarkSweepSpace msSpace = new MarkSweepSpace("ms",
  DEFAULT_POLL_FREQUENCY, MATURE_FRACTION*0.7);
CopySpace matureSpaceHot0 = new CopySpace("csh0",
  DEFAULT_POLL_FREQUENCY, MATURE_FRACTION*0.15, false);
CopySpace matureSpaceHot1 = new CopySpace("csh1",
  DEFAULT_POLL_FREQUENCY, MATURE_FRACTION*0.15, true);
```

```
int MS = msSpace.getDescriptor();
int MSHOT0 = matureSpaceHot0.getDescriptor();
int MSHOT1 = matureSpaceHot1.getDescriptor();

Trace matureTrace = new Trace(metaDataSpace);
Trace matureTraceHot = new Trace(metaDataSpace);
```

5.2.3 Changes in the scanning algorithm

In the mark and sweep hot cold collector objects will stay in the hot space at the same position and will not be moved. The objects are only relocated once and may not be close together due to fragmentation. The mark and sweep space is not compacted and if objects die other objects can use their memory. This means that objects that are copied after each other may not be close to each other.

The copy-space allocates objects after another and if it is collected all reachable objects are copied into the second copy-space. This copying phase can be used to reorder the objects depending on the internal field reference's heat.

This functionality is implemented in the scanning engine of the MMTk and uses the available information of the PEBSI sample engine.

If PEBSI is enabled then all hot object types have a special ordered array with reordered references. These references are updated in the collector thread.

```
void scanObject(TraceLocal trace, ObjectReference object)
{
  MMType type = ObjectModel.getObjectType(object);
  int references = type.getReferences(object);
  if (type.isHot()) { /* hot types */
    for (int i = 0; i < references; i++) {
      Address slot = type.getSlot(object,
        type.pebsiFieldOrder[i]);
      trace.traceObjectLocation(slot);
      }
  } else { /* cold types */
```

```
      for (int i = 0; i < references; i++) {
        Address slot = type.getSlot(object, i);
        trace.traceObjectLocation(slot);
      }
    }
  }
```

Using this adaption of the scanning algorithm it is possible to enqueue and copy objects depending on their references. Fields that are referenced often are copied directly behind the parent object. This should lead to a reduction of cache misses.

Chapter 6

Discussion and results

This chapter provides a discussion of the concept and implementation and tries to reason the design decisions. The results of the benchmark runs are also presented. There are benchmarks of the PEBSI overhead and tables about the additional information that can be gathered with this interface. Another section covers the optimization in the memory management and provides tables about the changes of the execution time and number of cache misses.

6.1 Interface for hardware performance monitors

The overhead of the complete sampling interface is very small and it is easy to gather additional information. An easy to extend interface is offered and many different events can be sampled. The framework is also able to gather precise event statistics and knows exactly where the sample was taken. Because the hardware itself takes the samples the exact assembler instruction can be resolved and there is no imprecision from an interrupt handler or kernel interaction.

Only the collector thread inside the VM and the information resolving mechanism must be implemented to use the library in a different VM. In closed source virtual machines it is still possible to load the library in a user thread. But implementing optimizations will be difficult without access to the VM internals.

Randomization

For events that occur very frequently it is important to use randomization of the counter value. Using prime numbers is not always enough, randomization is a much better solution

to solve this problem. Upon every overflow the counter value is modified by a random 8 bit mask. Otherwise the program can get in lockstep with the execution. This can lead to biased and unusable samples.

As the Jikes RVM is a fairly large software package with many non-deterministic parameter like GC timer, AOS and so on, randomization is not that important. But it is a necessary extension for smaller code bases like C or C++ programs.

Distribution of event to bytecode type

This part shows the results of the benchmarks with additional statistics. Statistics about the method type (baseline or optimized compiled), number and type of references, number of the different load and invoke instructions and object header accesses are shown in the tables below.

General statistics about number of misses per benchmark and method and field references are always collected.

Figure 6.1 shows the bytecode distribution of different benchmarks. It can be seen that the benchmarks have very different access patterns and the cache misses happen at different locations. Depending on this information it should be possible to use specific optimizations to minimize these cache misses.

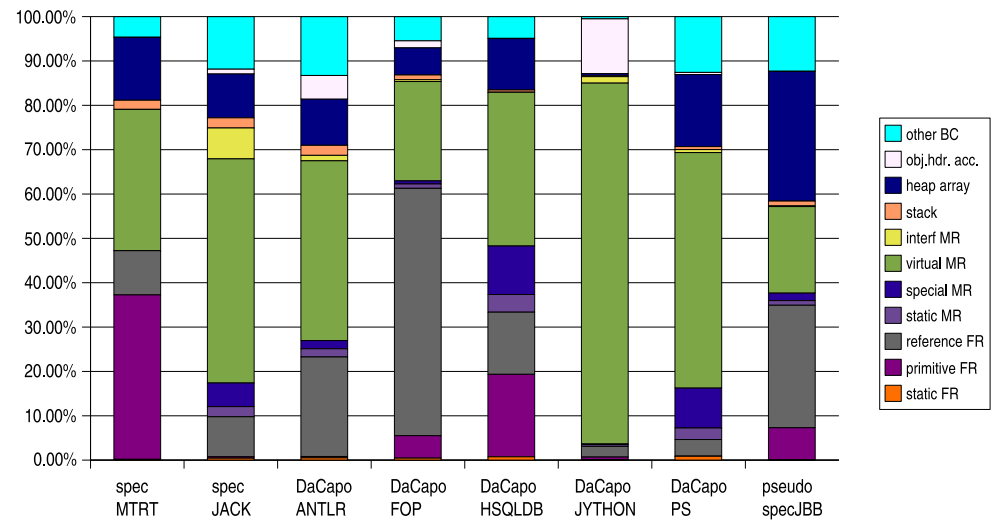

Figure 6.1: Bytecode distribution of different benchmarks for l2 cache misses.

All benchmarks could profit from optimization of virtual calls as all benchmarks have from 20% to 81% of their misses when a virtual call takes place. Using the PEBS infrastructure presented in this paper it is possible to get the corresponding methods and types

where the cache miss happened.

Most of the benchmarks can potentially profit from object reordering. On average 18% of the cache misses happen when a field reference is resolved. Some benchmarks like Jython use nearly no field references (or the references are not sampled because there are no cache misses) while others like FOP have more optimization potential. If an optimization is to use this information then it must limit itself on promising benchmarks. This can be decided at runtime using the statistical data from PEBS.

Some benchmarks like MTRT have only limited optimization potential because there are not many cache misses at all. Additionally a big part of the samples are from primitive fields like basic types.

Overhead for PEBSI sampling

The overhead for PEBSI sampling depends on the number of samples that are taken per second. But it is generally possible to sample enough information with a very small overhead.

Interval	DaCapoMTRT		DaCapoJACK		DaCapoANTLR		DaCapoFOP	
	Ovhd.	Smpl./s	Ovhd.	Smpl./s	Ovhd.	Smpl./s	Ovhd.	Smpl./s
5000	1.39%	180.62	0.38%	136.72	1.23%	222.85	2.15%	528.61
10000	0.55%	103.06	0.39%	82.05	-0.22%	132.96	2.24%	305.19
15000	1.12%	113.13	0.32%	51.46	0.00%	84.98	1.61%	211.44
25000	0.76%	37.14	0.43%	33.58	0.04%	52.05	1.54%	136.96
50000	0.79%	20.01	0.17%	15.84	0.02%	25.91	1.70%	66.96
100000	0.62%	10.31	0.23%	7.82	-0.44%	13.49	0.77%	33.09

Table 6.1: Overhead and samples per second for 2nd level cache misses (1/2).

As can be seen in the overhead tables 6.1 and 6.2 the overhead is between 2.8% and 0.2%. The tables show the overhead in percent to the production version of Jikes with a GenMS collector and no PEBS thread. The second column per benchmark shows the number of taken samples per second. The overhead for sufficient samples is at about 1%, but depends on the benchmark and the optimization. Both benchmarks ANTLR and HSQLDB are an exception. ANTLR has a very low overhead, this is most likely due to optimization of the Pentium 4 hardware itself. The overhead for HSQLDB is very unstable, most likely due to many parallel running threads. These threads are then

Interval	DC HSQLDB		DC JYTHON		DaCapoPS		pseudo JBB	
	Ovhd.	Smpl./s	Ovhd.	Smpl./s	Ovhd.	Smpl./s	Ovhd.	Smpl./s
5000	7.02%	268.37	1.03%	152.35	2.01%	664.18	2.85%	844.24
10000	6.89%	139.31	0.87%	83.21	1.55%	361.54	1.59%	451.74
15000	16.05%	108.93	0.66%	71.38	1.67%	250.06	1.40%	311.36
25000	6.81%	78.43	0.81%	47.58	1.88%	151.82	0.54%	201.6
50000	1.18%	36.51	0.87%	31.77	1.83%	77.64	0.73%	95.63
100000	13.02%	16.08	0.59%	10.77	1.82%	38.61	0.62%	48.96

Table 6.2: Overhead and samples per second for 2nd level cache misses (2/2).

scheduled non-deterministicly. The tables show the runtime overhead in percent against the original version without the PEBS thread. Every interval was measured eight times (after a warmup cycle) and averaged. These benchmarks only show the overhead, no optimization took place.

The benchmark pseudo spec JBB is a very good example because it runs for a long time and the results do not vary much over different runs. It shows that the overhead is linear dependent on the number of processed samples per second.

Unfortunately not all taken samples can be used. Table 6.3 shows the number of overall samples, resolved methods and resolved bytecode instructions. External methods, libraries and kernel calls are not resolvable, these samples are discarded.

	# Samples	Method resolved	BC instr. resolved
spec MTRT	2'481	1'884 (75.94%)	1'003 (64.49%)
spec JACK	2'704	1'975 (73.04%)	920 (60.98%)
DaCapo ANTLR	2'775	1'689 (60.86%)	1'252 (74.16%)
DaCapo FOP	23'067	18'225 (79.01%)	14'928 (81.91%)
DaCapo HSQLDB	25'670	20'115 (78.36%)	17'536 (87.18%)
DaCapo JYTHON	8'449	6'702 (79.32%)	5'751 (85.81%)
DaCapo PS	7'720	6'460 (83.68%)	1'920 (29.72%)
pseudo specJBB	221'558	154'204 (69.60%)	127'111 (82.43%)

Table 6.3: Number of samples and resolved percentiles for L2 misses.

The second data column of 6.3 shows the number of samples where a method was found. Some of the remaining samples are discarded if the corresponding bytecode instruction can

not be found. This is possible if the event happened in the method pro- or epilogue. The last data column of 6.3 shows the number of samples where both method and bytecode could be resolved. The percentage is relative to the number of resolved methods, because the bytecode instruction can only be resolved if the method is already determined.

On average 75% of the methods can be mapped from IP to the original method using the fast method lookup. The minimum of 61% is caused by the ANTLR benchmark that prints a lot of information to the console. These calls are all to libraries outside of the VM.

The bytecode instruction can be resolved of 71% of the remaining samples. The minimum of 29.72% is caused by the PS benchmark.

6.2 Hot cold garbage collector

As an application of the extended hardware profiling two new garbage collectors were implemented. These collectors use the available PEBS information to decide if an object is hot or not.

The first collector uses a mark and sweep space for the hot objects. This space is faster but has disadvantages regarding locality. Therefore the second collector uses a copy-space for hot objects and a special copy-order for the fields of hot objects to increase object locality.

	Best interval
spec MTRT	15'000
spec JACK	25'000
DaCapo ANTLR	10'000
DaCapo FOP	5'000
DaCapo HSQLDB	50'000
DaCapo JYTHON	15'000
DaCapo PS	25'000
pseudo specJBB	15'000

Table 6.4: Best intervals for specific benchmarks.

The standard generational garbage collector of the MMTk was taken as a basis. This basis was then extended to include the additional spaces. This might be the problem

that there is no real speed up. Due to limitations in the MMTk both spaces are collected simultaneously. If the hot and cold space could be collected separately then the collectors should do better. The new collectors each introduce a new space. This means that the available memory is shared by more spaces. If one of the mature spaces runs out of available memory then a full heap scan is started. This full heap scan then collects all spaces, including the two mature spaces.

The table 6.4 shows the intervals where the extended garbage collectors worked best. These intervals were used in the figures below that show the runtime benefit and the second level cache miss reduction.

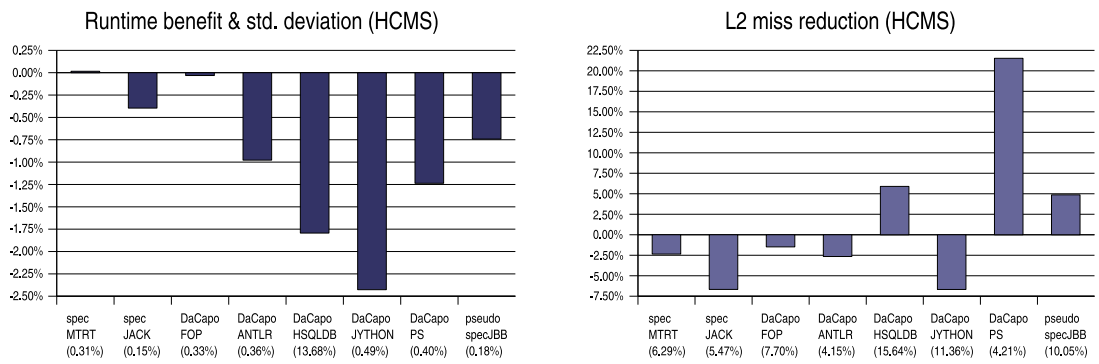

Figure 6.2: Runtime benefit and l2 cache miss reduction for hot/cold mark and sweep collector.

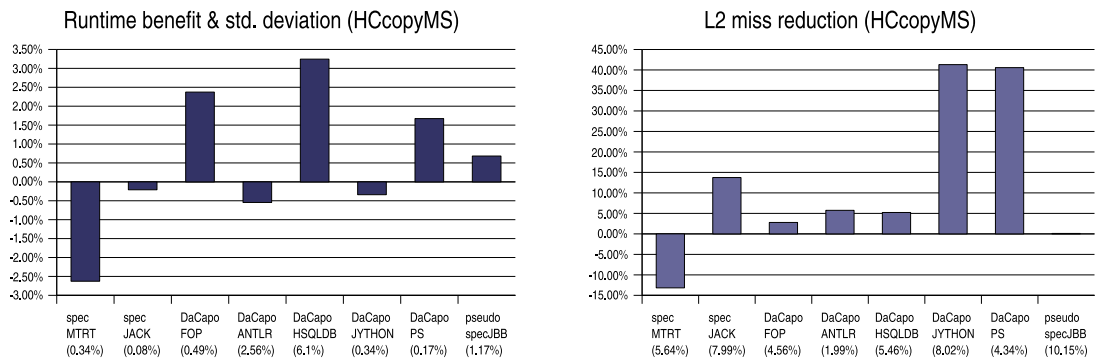

Figure 6.3: Runtime benefit and l2 cache miss reduction for hot/cold copy and mark and sweep collector.

The figure 6.2 shows the runtime benefit and l2 miss reduction of the hot/cold collector with two mature mark and sweep spaces. The figures show the reduction in percent against the production version from Jikes. The percentage beneath the name of the benchmark

shows the standard deviation. The maximum improvement can be seen at the MTRT benchmark which gives virtually no speedup with 0.02%. The worst degradation is caused by the JYTHON benchmark with 2.42%. The PS benchmark offers maximum benefit with a reduction of 21.53% of L2 misses and JYTHON performs worst with an increase of 6.68%. This collector gives a small average L2 cache miss reduction of 5.0%. But the additional overhead for sampling and the second mature space leads to an average runtime overhead of 1.0%.

Figure 6.3 shows the runtime benefit and l2 miss reduction of the hot/cold collector with a mature mark and sweep space and a mature copy-space. The figures show the reduction in percent against the production version from Jikes. The percentage beneath the name of the benchmark shows the standard deviation. Four benchmarks show an improvement in execution time. The average improvement is 1.0% with a maximum of 3.24% for HSQLDB and a degradation of 2.63% for MTRT. The overall number of misses can be reduced by 5.3% .

The tables 6.5 and 6.6 show the optimization potential for first level cache misses using the generational hot/cold garbage collector with a mark and sweep mature space for cold objects and a copy-space for hot objects.

Interval	DaCapoMTRT	DaCapoJACK	DaCapoANTLR	DaCapoFOP
25000	3.72%	1.51%	2.01%	2.96%
50000	2.21%	0.82%	1.23%	1.46%
100000	1.06%	0.66%	0.83%	1.57%

Table 6.5: Relative execution time for 1st level cache misses and hot/cold copy collector (1/2).

Interval	DC HSQLDB	DC JYTHON	DaCapoPS	pseudo JBB
25000	-3.51%	1.09%	4.98%	2.90%
50000	6.72%	0.62%	3.22%	1.99%
100000	0.77%	1.02%	2.01%	1.83%

Table 6.6: Relative execution time for 1st level cache misses and hot/cold copy collector (2/2).

It can be seen in the tables 6.5 and 6.6 that the overhead generally increases. This means that the hot/cold garbage collector and field reordering are not usable in this

implementation to reduce the first level cache misses. The overhead depends linearly on the interval, but the garbage collector is not able to reduce the cache misses because the objects are most likely too large.

6.3 Further work

This thesis offers a very good framework for hardware based performance monitoring and precise event based sampling. It is possible to use many different events with varying sampling interval and buffer.

Further work is needed for additional optimizations that use more of the available information. Information about the sampled methods is still unused and could be fed back into the adaptive optimization system to recompile these methods with specific optimizations.

Other additional options include garbage collectors that compact and reorder these elements depending on a more sophisticated algorithm. The presented garbage collector uses at most the available field reference information and nothing else.

Another idea is a change in the AOS of Jikes. The PEBSI thread could gather extensive statistics. The AOS then could use this statistics on a per type or per method basis to select the best possible optimizations depending on the type statistics.

The information about the different load and store types could also be used for more optimizations. Maybe in combination with the method information to guide the adaptive optimization system. Currently methods are recompiled at different levels with a fixed increasing number of optimizations per level. Using these information it could be possible to only select optimizations that are promising. Other optimizations without potential would not be applied, saving compilation time.

Bibliography

[1] **IA-32 Intel Architecture Software Developer's Manual, Volume 3B: System Programming Guide** (document #253669 / January 2006).
http://developer.intel.com/design/pentium4/manuals/index_new.htm

[2] **IA-32 Intel Architecture Optimization Reference Manual**
(document #248966 / January 2006).
http://developer.intel.com/design/pentium4/manuals/index_new.htm

[3] Tim Lindholm, Frank Yellin: **The JavaTM Virtual Machine Specification**, September 1996 / 1999.
http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html
http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

[4] **Sun Microsystems Java Environment**
http://java.sun.com

[5] **The JikesTM Research Virtual Machine's Homepage**
http://jikes.sourceforge.net/

[6] **The JikesTM Research Virtual Machine User's Guide (2.4.2)**
November 21, 2005
http://jikesrvm.sourceforge.net/userguide/HTML/userguide.html

[7] **Brink and Abyss:** Pentium 4 Performance Counter Tools For Linux
Kernel driver and user-space program for Pentium 4 hardware performance monitors.
http://www.eg.bucknell.edu/ bsprunt/emon/brink_abyss/brink_abyss.shtm

[8] **Perfctr:** Linux Performance Counters Driver
Low level kernel driver for hardware performance monitors of multiple architectures.
http://sourceforge.net/projects/perfctr/

[9] **PAPI:** Performance Application Programming Interface
High level user-space library for performance counters, uses Perfctr.
http://icl.cs.utk.edu/papi/

[10] **Hardmeter:** Memory profiling tool
Hardmeter is an extension of an old Perfctr version that implements PEBS for Pentium 4 processors.
http://sourceforge.jp/projects/hardmeter

[11] **Perfmon2** kernel interface
Perfmon tries to offer HPM facilities for multiple platforms, including PEBS for Pentium 4 processors. It consists of a kernel driver and a user-space library, but the kernel driver can also be used stand alone.
http://sourceforge.net/projects/perfmon2/
http://www.hpl.hp.com/research/linux/perfmon/perfmon.php4
Quick overview: http://lwn.net/Articles/164807/

[12] **Using Platform-Specific Performance Counters for Dynamic Compilation** by Florian Schneider and Thomas Gross, October 2005.

[13] **A run-time interface for hardware performance monitors in a dynamic compilation environment** by Pellanda Flavio, supervisors: Prof. Thomas Gross, Florian Schneider, March 2005.

[14] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, Perry Cheng. **The Garbage Collection Advantage: Improving Program Locality**.
http://cs.anu.edu.au/ Steve.Blackburn/pubs/papers/oor-oopsla-2004.pdf

[15] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, Jaswinder Pal Singh. **Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times**.
http://www.research.ibm.com/people/g/gupta/oopsla02.pdf

[16] **DaCapo Benchmark Suite**
http://osl-www.cs.umass.edu/DaCapo/gcbm.html

Appendix A

Used benchmarks

Multiple benchmarks are used to test changes in the virtual machine. Most of the benchmarks are from two of the bigger benchmarks suits. Some tests are from the DaCapo benchmarks that build upon popular open source software. Others come from the spec jvm 98 benchmarks that widely known.

Jikes uses an adaptive optimization system (AOS) to recompile methods that are executed often. This AOS is non deterministic and can lead to different results. This system is disabled and a predefind compilation plan is used for all benchmarks of a given type. So every run of a specific benchmark uses the same parameters for the AOS.

All benchmarks were calculated on an Intel Pentium 4 "Northwood" with 2.66 GHz and 512KB second level cache. The available memory was 1024MB and the VM used a maximum of 256MB, without a specific initial heap size.

The adaptive system needs some warmup time until the VM can profit from the optimizations. Some adaptions were made to cover these limitations. These changes are listed below.

A.1 DaCapo Benchmarks

This benchmark suite [16] consists of a set of open source applications that are used in production systems. All of these applications have a non-trivial memory load and profit from optimizations in the memory management system.

In this thesis the benchmarks with the version number beta051009 are used. Due to restrictions of the Jikes RVM not all benchmarks could be run. The AWT and XML libraries are missing or only partly implemented in Jikes.

DaCapo ANTLR

Parses some grammar files and generates a parser and lexical analyzer for each input. This benchmark is used unchanged with the large set of input data.

DaCapo FOP

This benchmark takes an XSL-FO file and parses and formats it. Then it generates a PDF file using these XSL transformations. The benchmark is unchanged and the large set of input data is used.

DaCapo HSQLDB

Builds a JDBC-like in-memory database and executes a number of transactions. A model of a banking application is used. The source is unchanged and the large set of input data is used.

DaCapo JYTHON

Offers a python interpreter and runs a series of programs against it. The benchmark runs as-is with the large set of input data.

DaCapo PS

Reads a PostScript file and interprets the stack based language. The large input data is used in the unchanged benchmark.

A.2 SPEC JVM98 Benchmarks

This benchmark from the Standard Performance Evaluation Corporation benchmarks the performance of Java Virtual Machines. The available benchmarks cover a wide area of optimization potential.

jvm98 227 mtrt

Is a raytrace benchmark running in two concurrent threads. It renders the file input/test-test.model with a resolution of 600 by 400.

jvm98 228 jack

This benchmark is a Java parser generator with a lexical analyzer. It runs 4 times with default parameters.

jvm98 pseudo jbb

Pseudo jbb is a variation of the spec jbb benchmark. In contrast to the original benchmarks which runs for a given time and returns the number of processed transactions, the pseudo jbb benchmark runs a given number of transactions. This makes it easier to measure time differences. No changes have been made to the source-code of this benchmark.

A.3 Other Benchmarks

One of the artificial test benchmarks of this thesis is MatrixMultiplication which multiplies two matrices A and B and saves the result in C.

Appendix B

Changed files

This chapter will present an overview over all changed files for the PEBS extension. Using this chapter it should be possible to adapt the current version to another sampling mechanism or a newer version.

B.1 Changes inside the Jikes RVM

The following list provides a description of all changes inside the Jikes RVM. All changes inside the RVM can be removed if the switch **RVM_WITH_PEBSI** is unset.

All files that are marked with (*) were newly added to Jikes. All other files were expanded.

- *bin/jconfigure*
  This file now includes the three switches **RVM_WITH_PEBSI**, if PEBSI is to be compiled and the PEBSI thread should be activated and started during the boot-phase, **RVM_WITH_PEBSI_DEBUG**, if PEBSI should include debug information and **RVM_WITH_PEBSI_STATISTICS**, if generous statistics should be collected.

- *config/i686-pc-linux-gnu.payerm*
  This file handles the defaults for the Jikes build process like local paths for binaries and the like.

- *config/build/BaseAdaptiveGenHCMS*
  Defines a Virtual Machine with the adaptive optimization system, where the boot image is baseline compiled and the special Hot-Cold mark and sweep generational

garbage collector is used. This configuration can be prepared by the command '**jconfigure BaseAdaptiveGenHCMS**'.

- *config/build/BaseAdaptiveGenHCCopyMS*

  Same setup as above, but the hot space uses a copy collector rather than a second mark and sweep collector. This configuration can be prepared by the command '**jconfigure BaseAdaptiveGenHCCopyMS**'.

- *config/build/FastAdaptiveGenHCMS*

  Same configuration as *BaseAdaptiveGenHCMS*, but the boot image is compiled with the optimizing compiler at the highest optimization level. Execute '**jconfigure FastAdaptiveGenHCMS**' to use this setup.

- *config/build/FastAdaptiveGenHCCopyMS*

  Defines the same configuration as *FastAdaptiveGenHCMS*, but the hot space uses a copy collector rather than a second mark and sweep collector. Use '**jconfigure FastAdaptiveGenHCCopyMS**' to configure this environment.

- *config/build/gc/GenHCMS*

  This file sets which class is used as the generational hot and cold collector with two mature mark and sweep spaces. The collector is defined in the package named *org.mmtk.plan.hcgenerational.hcmarksweep.GenHCMS*.

- *config/build/gc/GenHCCopyMS*

  Defines the class which is used for the garbage collector with a cold mark and sweep space and a hot copy-space. The collector is declared in the class file named *org.mmtk.plan.hcgenerational.hccopymarksweep.GenHCCopyMS*.

- *src/vm/VM.java*

  Some hooks to the boot process are added so that the method tables are synced for fast method lookup. The PEBSI thread is also started in this file.

- *src/vm/compilers/compiledCode/VM_CompiledMethod.java*

  A sample counter and the corresponding methods are added. This counter is increased for every sample this method caused.

- *src/vm/compilers/compiledCode/VM_CompiledMethods.java*

  In this file all the compiled methods are handled. A special lookup table was added

that implements a nearly O(c) lookup of an IP to method. Methods are organized in a three level lookup table that uses codepage mappers and codepages to map in between.

- *src/vm/compilers/compiledCode/VM_Codepage.java (*)*
  This file implements a memory page and holds all methods inside a memory page. There are normally not more than 3-8 methods located in a page.

- *src/vm/compilers/compiledCode/VM_CodepageMapper.java (*)*
  In the current implementation there are at most 2^8 codepage mapper which each will hold 2^{12} codepages.

- *src/vm/compilers/optimizing/ir/conversions/mir2mc/*
  OPT_ConvertMIRtoMC.java
  Hooks to generate extended bytecode maps are added.

- *src/vm/compilers/optimizing/vmInterface/services/*
  VM_OptCompiledMethod.java
  An additional BC map is added for every bytecode instruction and not only for GC points. There is also a get method for these BC maps.

- *src/vm/classLoader/VM_BytecodeStream.java*
  A special method is added that returns the field reference index inside the bytecode stream as a number rather than only the reference. This may be used for further optimization inside the bytecode finder.

- *src/vm/classLoader/VM_FieldReference.java*
  An additional sample counter and the corresponding methods per field reference are added. This class now also implements the "comparable"-interface to sort the most sampled field references for the additional statistics.

- *src/vm/classLoader/VM_MethodReference.java*
  An additional sample counter and the corresponding methods per method reference are added. This class now also implements the "comparable"-interface to sort the most sampled method references for the additional statistics.

- *src/vm/classLoader/VM_Type.java*
  An additional sample counter and the corresponding methods per type are added.

This information is needed by the memory management and the Pebsi sampling thread. Two additional methods (**public final short[] getPebsiFieldOrder()** and **public final int[] getPebsiFieldCounts()**) that get information from the memory management into the Jikes sampling thread are added. It is also possible to define a type as hot or cold. These changes are then propagated into the memory management.

- *src/vm/memoryManagers/JMTk/vmInterface/MM_Interface.java*
  A small change so that methods are always allocated in the immortal space is added. Otherwise the addresses of the methods would change and this would lead to a resource intensive rescanning of all active methods for the fast lookup table as the table works with absolute addresses.

- *src/vm/utility/VM_CommandLineArgs.java*
  Adds command line switches for PEBSI. These values are evaluated in the class file **VM_Pebsi.java**. If the VM is compiled without support for PEBSI and a PEBSI argument is specified then an error message is printed.

- *src/vm/pebsi/VM_BytecodeFinder.java (*)*
  This class locates the bytecode instruction for every sample and tries to resolve different information for the corresponding bytecode, like method reference, field reference or load and store statistics.

- *src/vm/pebsi/VM_OptBCMap.java (*)*
  Is indirectly used by **OPT_ConvertMIRtoMC.java** and offered directly by the file **VM_OptCompiledMethod.java**. These bytecode maps are then used by **VM_BytecodeFinder.java** to locate machine-compiled methods to actual bytecode.

- *src/vm/pebsi/VM_PebsiException.java (*)*
  Defines a hardware exception that is thrown if there are problems with the libpebsi native interface.

- *src/vm/pebsi/VM_Pebsi.java (*)*
  This class declares the PEBSI-Thread inside the VM. It controls the collection and distribution of samples and informs the optimization system of hot types and hot spots.

B.2 Files for libpebsi userspace library

The libpebsi library is basically an independent software package. The low level library includes Makefile, README and INSTALL files. These files describes the bindings to the perfmon library and kernel modules and how to install all components.

Additional to the JNI bindings there are two examples, a self sampling matrix multiplication and a program that counts specific events for external programs.

The complete library was written from scratch and can be used either in a Java VM using the JNI bindings or in any other programming language that can load shared libraries.

- *src/pebsijni.h*

  Includes the bindings and definitions for the JNI functions. These functions can then be used inside the JVM.

- *src/pebsijni.c*

  Implements the JNI functions and uses the library itself. Maps all the Java calls to the library calls and boxes basic types into objects.

- *src/pebsi.h*

  Defines all PEBSI methods and bindings to perfmon. Some default values like DEBUG and DEBUG_LEVEL are also defined in this file.

- *src/pebsi.c*

  Implementation of the low level functions to control the PEBS interface.

- *examples/monitor.c*

  An example program that implements PEBS self monitoring. The collected samples are printed to stdout.

- *examples/extmonitorcount.c*

  Takes an external program as parameter and counts the number of events and prints this number to stdout after the child exits. This sample only uses counters and no PEBS.

- *examples/Makefile*

  Makefile to build and run the examples. Be aware that the monitor example needs

the **LD_LIBRARY_PATH** environment variable set. The paths inside this file must be adjusted to the directory layout.

- *libpebsi.so*

  This is the actual shared library that is built from the sources. It can be installed into the Jikes RVM home directory and can then be loaded into the RVM.

- *README*

  Readme file describing the libpebsi package and the used components.

- *INSTALL*

  This file includes instructions how to download the kernel patch and install the perfmon2 patches. There is also some information about installation of the shared library.

- *Makefile*

  General makefile for the libpebsi library. The paths inside this file must be adjusted to the directory layout.

B.3 Changes inside the Jikes MMTk (Memory Management Toolkit)

These changes represent all the additions to the memory management toolkit that Jikes uses. This toolkit defines different garbage collectors, how they interact and work.

This thesis introduces two new garbage collectors, the **hcmarksweep** collector and the **hccopymarksweep** collector. Both collectors are generational garbage collectors with a nursery space and two mature spaces. There is a hot and a cold mature space. If an object is sampled more times than a threshold then the object will be moved into the hot space, otherwise it will be moved into the cold space.

- *src/org/mmtk/plan/hcgenerational/hcmarksweep/*
  GenHCMSMatureTraceLocal.java
  GenHCMSLocal.java
  GenHCMSConstraints.java
  GenHCMS.java

  These files define the mature space of a generational garbage collector. The mature

space consists of two parts, a hot and a cold mature space. Both mature spaces follow the "mark-and-sweep" principle. These files contain mostly code from the standard generational collector, but are extended by a second mature space.

- *src/org/mmtk/plan/hcgenerational/hccopymarksweep/*
  GenHCCopyMSMatureTraceLocal.java
  GenHCCopyMSLocal.java
  GenHCCopyMSConstraints.java
  GenHCCopyMS.java
  Defines another "hot-and-cold" garbage collector. The cold mature space contains a mark-and-sweep collector and the hot space is handled by a copy-space collector. These files contain mostly code from the standard generational collector, but are extended by a second mature space.

- *src/org/mmtk/plan/hcgenerational/*
  GenNurseryTraceLocal.java
  GenMatureTraceLocal.java
  GenLocal.java
  GenConstraints.java
  hcgenerational/Gen.java
  These files are the basis for the two extensions **hccopymarksweep** and **hcmarksweep**. The nursery and some shared methods are defined in these files. Most of the code is copied from the nursery of the standard generational garbage collector, but extended for a two-spaced mature space.

- *src/org/mmtk/policy/*
  CopySpaceHC.java
  CopyLocalHC.java
  This package defines a special copy-space that is used in the **hccopymarksweep** collector. There is an additional check that tests if the object is still hot. If it gets cold then it is moved to the cold-mature-space.

- *src/org/mmtk/utility/scan/MMType.java*
  This class is extended by a field order that is defined by the number of samples and the hotness of the references. The array is then accessible from the internal VM context by a get method.

- *src/org/mmtk/utility/scan/Scan.java*

  The objects are scanned and enqueued using the additional information from the collector thread.

Appendix C

Command line arguments

The PEBSI extensions support many different command line arguments. Only the mandatory option **event** must be used, all other options are optional.

Multiple options can be concatenated. The event must be first and they must be entered in the form *-X:pebsi:<option>=<value>:<option>=<value>:...*.

The event option supports all events that are exported by the **libpebsi** library. Currently the following events are supported (use *event=EVENTNAME*):

- **Precise front-end events:**

memory_load	Memory loads
memory_store	Memory stores
memory_move	Memory loads and stores

- **Precise execution events:**

packed_sp	packed single-precision uop retired
packed_dp	packed double-precision uop retired
scalar_sp	scalar single-precision uop retired
scalar_dp	scalar double-precision uop retired
64bit_mmx	64bit SIMD integer uop retired
128bit_mmx	128bit SIMD integer uop retired
x87_fp	floating point instruction retired
x87_simd_memory_moves	x87/SIMD store/moves/load uop retired

- **Precise replay events:**

l1_cache_miss	1nd level cache load miss
l2_cache_miss	2nd level cache load miss
dtlb_load_miss	DTLB load miss
dtlb_stor_miss	DTLB store miss
dtlb_all_miss	DTLB load and store miss
mispred_branch	Tagged misspredicted branch
mob_load_replay	MOB (memory order buffer) causes load replay
split_load	replayed events at the load port
split_store	replayed events at the store port

There are many other options that can be used to fine-tune the operation of the PEBSI thread. If the interval should be set explicitly without automatic adpation then the two options *interval=NUMBER:autointr=false* must be combined.

interval	Set the capture interval. Only every n-th sample is recorded. This option depends on autointerval. If autointerval is true, then this value is only a startingpoint for the adaptive optimization. If interval is 0, then PEBSI will be disabled!
autointr	Setting for the adaptive optimization of the sampling interval. Default: true
buffer	Set the number of samples the kernel buffer will store. Default: 1000
status	Displays status information about identified methods when the VM is shut down. If the additional flag status=verbose is set, then additional information about methods, methodrefs and fieldrefs will be printed. Default: not shown
dump	Dumps all collected samples into the file 'raw_samples.out' if set to true. Default: false
dumpfile	If set, then all pebsi output is redirected into the specified file. Default: " (output to console)
debug	Toggle debug level. Default: 0, max.: 9

Table C.1: Command-line arguments.