Method-Level Phase Behavior in Java Workloads

Andy Georges Dries Buytaert Lieven Eeckhout Koen De Bosschere

Department of Electronics and Information Systems (ELIS), Ghent University
St.-Pietersnieuwstraat 41, B-9000 Gent, Belgium
{ageorges,dbuytaer,leeckhou,kdb}@elis.ugent.be

ABSTRACT

Java workloads are becoming more and more prominent on various computing devices. Understanding the behavior of a Java workload which includes the interaction between the application and the virtual machine (VM), is thus of primary importance during performance analysis and optimization. Moreover, as contemporary software projects are increasing in complexity, automatic performance analysis techniques are indispensable. This paper proposes an off-line method-level phase analysis approach for Java workloads that consists of three steps. In the first step, the execution time is computed for each method invocation. Using an off-line tool, we subsequently analyze the dynamic call graph (that is annotated with the method invocations' execution times) to identify method-level phases. Finally, we measure performance characteristics for each of the selected phases. This is done using hardware performance monitors. As such, our approach allows for linking microprocessor-level information at the individual methods in the Java application's source code. This is extremely interesting information during performance analysis and optimization as programmers can use this information to optimize their code. We evaluate our approach in the Jikes RVM on an IA-32 platform using the SPECjvm98 and SPECjbb2000 benchmarks. This is done according to a number of important criteria: the overhead during profiling, the variability within and between the phases, its applicability in Java workload characterization (measuring performance characteristics of the various VM components) and application bottleneck identification.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: design studies, measurement techniques, performance attributes

General Terms

Measurement, Performance, Experimentation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.
Copyright 2004 ACM 1-58113-831-8/04/0010 ...$5.00.

Keywords

workload characterization, performance analysis, Java workloads, virtual machine technology

1. INTRODUCTION

The execution of a Java application involves a complex interaction between the Java code and the virtual machine (VM). Consequently the behavior that is observed at the micro-architectural level when executing Java workloads is not just a function of the application or the VM, but of the interaction between both. In addition to that, the applications themselves are growing in size and complexity and VM's are complex as well consisting of a number of subcomponents to drive the managed run-time system, such as the interpreter, compiler, optimizer, garbage collector, class loader, finalizer, thread scheduler, etc. As a result of that, understanding the behavior of a Java workload is non-trivial which increases the demand for automatic approaches to analyze Java workload behavior.

The purpose of this paper is to study method-level phase behavior in Java workloads. The notion of a phase is defined as a set of parts of the program execution with similar behavior which do not necessarily need to be temporally adjacent. The underlying assumption of method-level phase behavior is that phases of execution correspond to the code that gets executed. In particular, different methods are likely to result in dissimilar behavior and different invocations of the same method are likely to result in similar behavior. There has been some work in the literature that studies whether methods are of an appropriate granularity to detect program phase behavior [6, 17, 20]. Those studies did show that the method level is at least as good as lower levels (basic block level and loop level) especially for applications with lots of method calls in which each method is quite small [20]. And this is the case for object-oriented workloads, such as Java [24]. In other words, the granularity of method-level phases is course-grained enough to identify major phases, and at the same time is fine-grained enough to provide sufficient detail.

Our method-level phase behavior analysis is an off-line analysis that consists of three steps. In a first step, we determine how much time the Java workload spends in different portions or methods of the application. This is done by instrumenting all methods to read microprocessor performance counter values to track the amount of time that is spent in each method. The result of this first step is an annotated dynamic call graph. Using an offline analysis (step 2) we then determine the methods in which the application

component	subcomponent	description
memory hierarchy	L1 I-cache	64KB two-way set-associative, 64-byte lines, LRU replacement with next line prefetching
	L1 D-cache	64KB two-way set-associative, 8 banks with 8-byte lines, LRU write-allocate, write-back, two access ports 64 bits each
	L2 cache	256KB 16-way set-associative, unified, on-chip, exclusive
	L1 I-TLB	24 entries, fully associative
	L2 I-TLB	256 entries, four-way set-associative
	L1 D-TLB	32 entries, fully associative
	L2 D-TLB	256 entries, four-way set-associative
branch prediction	BTB	branch target buffer, two-way set-associative, 2048 entries
	RAS	return address stack, 12 entries
	taken/not-taken	gshare 2048-entry branch predictor with 2-bit counters
system design	bus	266MHz, 2.1GiB per second
pipeline stages	integer	10 cycles
	floating-point	15 cycles
integer pipeline	pipeline 1	integer execution unit and address generation unit also allows integer multiply
	pipeline 2	integer execution unit and address generation unit
	pipeline 3	idem
floating-point pipeline	pipeline 1	3DNow! add, MMX ALU/shifter and floating-point add
	pipeline 2	3DNow!/MMX multiply/reciprocate, MMX ALU and floating-point multiply/divide/square root
	pipeline 3	floating-point constant loads and stores

Table 1: The AMD Athlon XP microprocessor summary.

spends a significant portion of its total execution time with the additional constraint that one invocation of the method takes a significant portion of the total execution time as well. This is to avoid selecting methods that are too small. During a second run of the application (step 3), these selected methods are instrumented and performance characteristics are measured. Measuring these performance characteristics is done using the hardware performance counters. In this step, we measure a number of characteristics such as branch misprediction rate, cache miss rate, number of retired instructions per cycle, etc. As such, we obtain detailed performance characteristics for the major method-level execution phases of the Java application. In addition to the method-level phases, we also measure performance characteristics for major parts of the VM, such as the compiler/optimizer, the garbage collector, the class loader, the finalizer, etc.

There are several interesting applications for this work. First, for application programmers it is important to understand the behavior of the Java workload in all its complexity in order to optimize its performance. Using automatic techniques to characterize Java workloads can be helpful to identify performance bottlenecks with limited effort. Second, for VM developers, automatic workload characterization helps to get insight in how a Java application interacts with its VM, which allows improving the performance of the VM under development. Third, our approach also provides interesting insights into phase behavior. Detecting program execution phases and exploiting them has received increased attention in recent literature. Various authors have proposed ways of exploiting phase behavior. One example is to adapt the available hardware resources to reduce energy consumption while sustaining the same performance [6, 10, 26, 17]. Another example is to use phase information to guide simulation-driven processor design [25]. The idea is to select one single sample from each phase for simulation instead of the complete benchmark execution. On the software side, JIT compilers in VM's [3, 4] and dynamic optimization frameworks [5, 22] heavily rely on implicit phase behavior to optimize code. Fourth, in future work we plan to build a performance model to estimate Java performance. In such a model, performance models of different Java components need to be combined to form an overall performance model of the Java application. We believe that the granularity of method-level phases that are identified in this paper will be the right choice for this purpose.

This paper is organized as follows. The next section details on our experimental setup. Section 3 discusses our off-line approach for identifying method-level phase behavior. Section 4 discusses the statistical data analysis techniques that we have used to quantify the variability between and within phases. The results of our phase analysis are presented in section 5. Section 6 discusses related work. Finally, we conclude in section 7.

2. EXPERIMENTAL SETUP

In this section we discuss the experimental setup: our hardware platform, the use of performance counters, Jikes RVM in which all experiments are done, and the Java applications that are used in the evaluation section of this paper.

2.1 Hardware platform

In this paper we use the AMD Athlon XP microprocessor for our measurements, see Table 1. The AMD Athlon XP is a superscalar microprocessor implementing the IA-32 instruction set architecture (ISA). It has a pipelined microarchitecture in which up to three x86 instructions can be fetched. These instructions are fetched from a large predecoded 64KB L1 instruction cache (I-cache). For dealing with

the branches in the instruction stream, branch prediction is done using a global history (gshare) based taken/not-taken branch predictor, a branch target buffer (BTB) and a return address stack (RAS). Once fetched, each (variable-length) x86 instruction is decoded into a number of simpler (and fixed-length) macro-ops. Up to three x86 instructions can be translated per cycle.

These macro-ops are then passed to the next stage in the pipeline, the instruction control unit (ICU) which basically consists of a 72-entry reorder buffer. From this reorder buffer, macro-ops are scheduled into an 18-entry integer scheduler and a 36-entry floating-point scheduler for integer and floating-point operations, respectively. The 18-entry integer scheduler is organized as a collection of three 6-entry deep reservation stations, each reservation station serving an integer execution unit and an address generation unit. The 36-entry floating-point scheduler (FPU: floating-point unit) serves three floating-point pipelines executing x87, MMX and 3DNow! operations. In the schedulers, the macro-ops are broken down to ops which can execute out-of-order. Next to these schedulers, the AMD K7 microarchitecture also has a 44-entry load-store unit. The load-store unit consists of two queues, a 12-entry queue for L1 D-cache load and store accesses and a 32-entry queue for L2 cache and memory load and store accesses—requests that missed in the L1 D-cache. The L1 D-cache is organized as an eight-bank cache having two 64-bit access ports.

Another interesting aspect of the AMD Athlon microarchitecture is the fact that the L2 unified cache is an exclusive cache. This means that cache blocks that were previously held by the L1 caches but had to be evicted from L1, are held in L2. If the newer cache block that is to be stored in L1 previously resided in L2, that cache block will be evicted from L2 to make room for the L1 block, i.e., a swap operation is done between L1 and L2. If the newer cache block that is to be stored in L1 did not previously reside in L2, a cache block will need to be evicted from L2 to memory.

2.2 Performance counters

Modern processors are often equipped with a set of performance counter registers. These registers are designed to count microprocessor events that occur during the execution of a program. They allow to keep track of the number of retired instructions, elapsed clock cycles, cache misses, branch mispredictions, etc. Generally, there are only a limited number of performance counter registers available on the chip. On the AMD Athlon, there are four such registers. However, the total number of microprocessor events that can be traced using these performance counters exceeds 60 in total. As a result, these registers need to be programmed to measure a particular event. The events that are traced for this study are given in Table 2. These events are commonly used in architectural studies to analyze program execution behavior. For most of the analyzes done in this paper, we use derived performance metrics. These performance metrics are obtained by dividing the number of events by the number of retired instructions. As such, we use events that occurred per instruction. We deem this to be more meaningful than the often-used miss rates. For example, we will use the number of cache misses per instruction instead of the number of cache misses per cache access. The reason is that the number of cache misses per instruction relates more directly to performance than classical cache miss rate since it also incorporates the number of cache accesses per instruction. Thus, the performance metrics derived from the events shown in Table 2, include e.g. CPI (clock cycles per retired instruction), L1 D-cache misses per retired instruction, etc.

mnemonic	description
cycles	elapsed clock cycles during execution
ret_instr	retired instructions
L1-D-misses	L1 D-cache misses
L2-D-misses	L2 D-cache misses
L1-I-misses	L1 I-cache misses
L2-I-misses	L2 I-cache misses
L1-L-misses	L1 load misses
L1-S-misses	L1 store misses
L2-L-misses	L2 load misses
L2-S-misses	L2 store misses
I-TLB-misses	Instruction TLB misses
D-TLB-misses	Data TLB misses
br_mpred	branches mispredicted
res_stall	resource stalls

Table 2: Performance counter events traced on the AMD Athlon XP.

Performance counters have several important benefits over other characterization methods: less slowdown since measurements happen at native execution speed, their ease of use, and their high accuracy compared to simulation-based and instrumentation-based approaches. However, there are also a number of issues that need further attention. First, measuring more than 4 events in our setup is impossible. As such, multiple runs are required to measure more than 4 events. Second, non-determinism can lead to slightly different performance counter values when running the same program multiple times. To address this issue, we measure each performance counter four times and use the average during analysis.

In this study, the performance counter values are accessed through the VM, see the next section. In turn, the VM makes use of the following tools: (i) the `perfctr`[1] Linux kernel patch, which provides a kernel module to access the processor hardware, and (ii) Performance API (PAPI) [7], a high-level library presenting a uniform interface to the performance counters on multiple platforms. The kernel patch allows tracing a single process, maintaining the state of the performance counters across kernel thread switches. The PAPI library presents a uniform manner for accessing the performance counters through the kernel module. Not all PAPI defined events are available on every platform, and not all native AMD events can be accessed through PAPI. However, for our purposes, it provides a sufficient set of events.

2.3 Jikes RVM

We use the Jikes Research Virtual Machine (RVM) [2, 3, 8] in this study. Jikes RVM is mainly targeted at server side applications. It is written (almost) entirely in Java. The Jikes RVM uses a compilation-only scheme for translating Java bytecodes to native machine instructions. The Jikes RVM comes with several compilation strategies: baseline, optimizing and adaptive. This paper uses the most advanced

[1] `http://user.it.uu.se/~mikpe/linux/perfctr/`

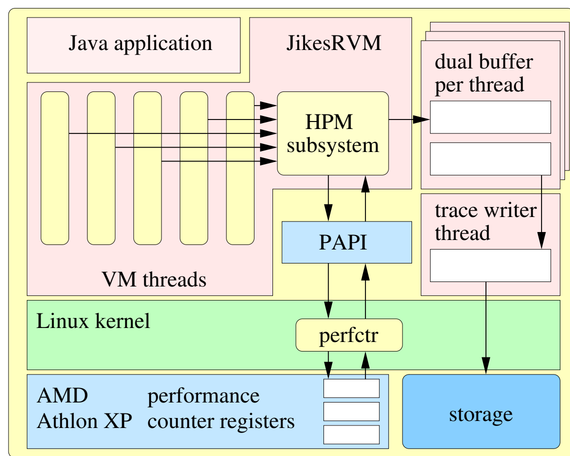


Figure 1: Overview of the Jikes RVM tracing system

compilation strategy, namely adaptive. In this scheme, Jikes RVM compiles each method on its first invocation using the baseline compiler and adaptively optimizes hot methods. Multiple recompilations are possible, each time using more optimizations. The Jikes RVM also supports different garbage collection mechanisms. The garbage collector in our experiments implements a generational garbage collection (GC) strategy with Mark-Sweep to clean the mature-object space, i.e. the CopyMS scheme. To build the Jikes RVM, we used (i) the Jikes[2] Java source-to-bytecode compiler (version 1.18) to compile the Jikes RVM source code class files, (ii) the Blackdown VM to build the boot image with the Jikes RVM optimizing compiler framework, and (iii) the GNU C and C++ compilers to compile the few C/C++ source files. The Jikes RVM itself is the CVS head (development) version from January 2004.

The threading system multiplexes n Java threads (application and VM) onto m native (kernel) threads that are scheduled by the operating system. A command line option specifies the number of kernel threads that are created by the Jikes RVM. Usually, there is one kernel thread used for each physical processor, also referred to as a virtual processor because multiple Java threads can be scheduled by the VM within the single kernel thread. In our setup, we have used a single virtual processor.

Current implementations of the Jikes RVM include support for hardware performance counters on both the IA-32 and PowerPC platforms. On the IA-32 platform, access to the processor hardware is done through the PAPI library (discussed above), see Figure 1. The Hardware Performance Monitor (HPM) subsystem of the Jikes RVM defines a set of methods to access the PAPI functions, such as starting, stopping, and resuming the counting of events, as well as reading the counter values. Keeping track of the events in the Jikes RVM is done as follows. Essentially, each Java thread keeps track of the performance counter events that occur while it is the executing thread on the virtual processor. Each time the VM schedules a virtual context switch, the removed thread reads the counter values, accumulates them with its existing values, and resets the counters. As such, a scheduled thread only observes counter values for the events that occur while it is executing. This mechanism for reading performance counter values is the standard implementation within the Jikes RVM. For a more detailed description on this, we refer to [28]. In section 3, we will detail how we extended this approach for measuring performance counter values on a per-method basis.

2.4 Java applications

We use the SPECjvm98 and SPECjbb2000 benchmark suites in this study. SPECjvm98[3] is a client-side Java benchmark suite consisting of seven benchmarks. For each of these, SPECjvm98 provides three inputs sets: `s1`, `s10` and `s100`. Contradictory to what their names suggest, the size of the input set does not increase exponentially. For some benchmarks, a larger input indeed increases the problem size. For other benchmarks, a larger input executes a smaller input multiple times. SPECjvm98 was designed to evaluate combined hardware (CPU, caches, memory, etc.) and software aspects (virtual machine, kernel activity, etc.) of a Java environment. However, they do not include graphics, networking or AWT (window management). In this study, we used the `s100` input set. The VM was set to use 64MiB[4] heap for SPECjvm98 benchmarks.

SPECjbb2000 (Java Business Benchmark)[5] is a server-side benchmark suite focusing on the middle-tier, the business logic, of a three-tier system. We have used a modified version of this benchmark, known as PseudoJBB, which executes a fixed amount of transactions, instead of running for a predetermined period of time. The benchmark was run with 8 warehouses. The VM's heap size was set to 384MiB.

3. METHOD-LEVEL PHASES

As stated in the introduction, Java applications are complex workloads due to the close interaction between the Java application and the VM on which it is running. To gain insight in how Java workloads behave we identify three major issues that need to be addressed. First, a distinction needs to be made between the Java application and the VM, i.e. the behavior of the application should be separated from the behavior of the VM. Second, the VM itself is a complex piece of software in which various components interact with each other to implement a managed run-time system. Several important VM components can be identified: class loader, compiler, optimizer, thread scheduler, finalizer, garbage collector, etc. As such, each of these components need to be profiled to extract performance characteristics. Third, the application itself may consist of several phases of execution. Previous work has extensively shown that applications exhibit phase behavior as a function of time, i.e. programs go from one phase to another during execution. Recall that we consider a phase to be a set of parts of a program execution that exhibit similar characteristics; and these parts of similar behavior do not need to be temporally adjacent. In this paper, we study method-level phase behavior, more in particular we consider a method including all its callees.

[2] http://www-124.ibm.com/developerworks/opensource/jikes/

[3] http://www.spec.org/jvm98/

[4] The notations KiB, MiB, and GiB used in this paper are SI standard notations for binary multiples, and denote 2^{10}, 2^{20} and 2^{30}, respectively. See also http://physics.nist.gov/cuu/Units/binary.html.

[5] http://www.spec.org/jbb2000/

This includes all the methods called by the selected method, i.e. all the methods in the call graph of which the selected method is the root. There are two motivations for doing this. First of all, the granularity of a method including its callees is not too small to introduce too much overhead during profiling. Second, the granularity is fine-grained enough to identify phases of execution. Previous work on phase classification [20] has considered various methods for identifying phases, ranging from the instruction level, the basic block level, the loop level up to the method level. From this research, the authors conclude that phase classification using method level information is fine-grained enough to identify phases reliably. Especially when the application has small methods. This is generally the case for applications written in object-oriented languages, of which Java is an example. The use of methods as a unit for phase classification was also studied by Balasabrumonian *et al.* [6] and Huang *et al.* [17].

It is important to state that the purpose of this paper is to identify method-level phase behavior using off-line techniques. This is particularly useful for Java and VM developers during performance analysis of their software. Method-level phase behavior allows them to improve their software since the performance characteristics that are obtained from the hardware performance monitors are linked directly to the source code of the application and virtual machine. For comparison, Sweeney *et al.* [28] read performance counter values on virtual context switches, and output the method ID of the method that is executing at that time to a trace file. In our methodology, we specifically link performance counter values to the methods. This is an important difference because the method executing at the virtual context switch is not necessarily the method that was executed during a major fraction of the time slice just before the virtual context switch. A second motivation for studying off-line phase behavior is that it can be used as a reference for dynamic (on-line) phase classification approaches. In other words, the statically identified phases can be used for evaluation purposes of dynamic phase classification techniques. Obviously, to identify recurring phases, static phase analysis has the advantage over dynamic phase analysis as it can look at the 'future' by looking ahead in the trace file. A dynamic approach on the other hand has to anticipate phase behavior and as such, can result in suboptimal phase identification. In addition, the resources that are available during off-line analysis are much larger than in case of on-line analysis, irrespective whether phase classification is done in software or in hardware. Yet another motivation for studying off-line method-level phase classification is for embedded and specialized environments in which the a priori information concerning the phase behavior in the Java application can be useful.

The following issues are some of the more specific goals we want to meet using our off-line phase analysis approach.

- We want to gather information from the start to the end of the program's execution. We want maximal coverage with no gaps.

- The overhead when profiling methods should be small enough not to interfere with normal program execution. This means that tracing all executed methods is not a viable option. Also, we want the volume of the trace data to be acceptable.

- We want to gather as much information as possible. At a minimum, the collected information should be sufficiently fine-grained such that transitions in Java performance characteristics can readily be identified. Such transitions can be caused by thread switches, e.g. the garbage collector is activated, or because the application enters a method that shows different behavior from previously executed methods.

To meet the goals of this paper, we use the following off-line phase analysis methodology. During a first run of the Java application, we measure the number of elapsed clock cycles in each method execution. This information is collected in a trace file that is subsequently used to annotate a dynamic call graph. A dynamic call graph is a tree that shows the various method invocations during a program execution when traversed in depth-first order [1]. In a second step of our methodology, we use an off-line tool that analyzes this annotated dynamic call graph and determines the major phases of execution. The output of this second step is a Java class file that describes which methods are responsible for the major execution phases of the Java application. In the third (and last) step, we link this class file to the VM and execute the Java application once again. The Java class file that is linked to the VM forces the VM to measure performance characteristics using the hardware performance monitors for the selected methods. The result of this run is a set of detailed performance characteristics for each method-level phase.

3.1 Mechanism

This section details on how a Java workload is profiled in our methodology. We first discuss how the methods in the Java application are instrumented. This mechanism will be used during two steps of our methodology: when measuring the execution time of each method execution during the first run (step 1), and when measuring the performance characteristics of the selected methods (step 3). The only difference between both cases is that in step 1 we instrument all methods. In step 3, we only profile the selected methods. In the second subsection, we detail on what information is collected during profiling. Subsequently, we address profiling the components of the VM.

3.1.1 Instrumenting the application methods

Methods compiled by the VM compilers consist of three parts: (i) the prologue, (ii) the main body of the method, and (iii) the epilogue. The prologue and epilogue handle the calling conventions, pushing and popping the callee's stack frame, yielding at a thread switch, etc. The goal is to capture as many of the generated events during the execution of a method. To achieve this, we add our instrumentation to the method's prologue and to the beginning of the method's epilogue. Methods are instrumented on-line by all the Jikes RVM compilers, i.e. the baseline compiler as well as the optimizing compiler.

Extending the baseline compiler to instrument methods is quite straightforward. It involves emitting the machine code to call the instrumentation functionality in the Jikes RVM run-time facilities. Calling the instrumentation functionality is done using the Jikes RVM HPM API. Adding calls to baseline compiled methods introduces new yield points. As each yield point is a potential GC point (or *safe point*), it is necessary to update the stack maps accordingly. If not,

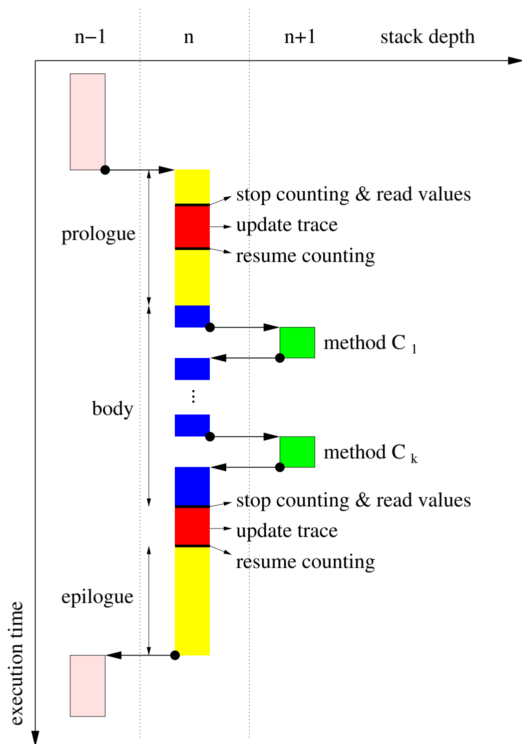


Figure 2: Tracing the performance counter events at the prologue and epilogue of a method.

referenced objects might not be reachable for the GC and risk being erroneously collected.

For the optimizing compiler, things are slightly more complicated. Optimizations are directed by the optimization planner, and involve multiple layers, from a high level representation to a machine code level representation. Our instrumentation involves adding an extra compiler phase to the compiler plan in the High Intermediate Representation (HIR) optimization set. Basically, we walk through the control flow graph of each method and add similar call instructions to the prologue and epilogue as we did for baseline compiled methods.

Next to the baseline and optimizing compiler, Jikes RVM also employs an on-stack replacement (OSR) scheme [16]. OSR allows the machine code of methods that are executing to be replaced by an optimized version of that machine code. This is especially useful for long-running methods, as the VM does not need to wait until the method finishes executing to replace the code that is currently executing with the newer optimized code. For this, certain OSR safe-points are available in the compiled code. At these points, the OSR mechanism can interrupt the thread executing the code and replace it with the optimized version. Our implementation also supports OSR.

Regardless of the compiler that was used to generate the executable machine code, we call our tracing framework as soon as the new frame for the callee has been established, see Figure 2. At this point, the thread executing the method updates its counter values, and suspends counting through the Jikes RVM HPM interface. As such, no events are counted during the logging of the counter values. When the trace values have been stored into a trace buffer, counting is resumed. To enhance throughput, the trace data is stored in a buffer. We maintain a per-thread cyclic list, which contains two 128KiB buffers. To ensure that these buffers are never touched by the garbage collector, they are allocated outside of the Java heap. Each time one of the buffers for a thread is full, it is scheduled to be written to disk, and the other buffer is used to store the trace data of that thread. A separate thread[6] stores the full buffer contents to disk in a properly synchronized manner. The same action sequence occurs at the method epilogue, just before the return value is loaded into the return register(s) and the stack frame is popped. When a method frame is popped because of an uncaught exception, we also log the counter values at that point. In summary, this approach reads the performance monitor values when entering the method and when exiting the method. The difference between both values gives us the performance metric of the method including its callees. For computing the performance metrics of the method itself, i.e. excluding its callees, the performance metrics of the callees need to be subtracted from the caller method.

From Figure 2, it can be observed that the events occurring before reading the counter values in the prologue and the events in the epilogue of a method are attributed to the calling method. However, this inaccuracy is negligible for our purposes.

In our methodology, we need to pay special attention to exceptions which can be thrown either implicitly or explicitly. The former are thrown by the VM itself whereas the latter are thrown by the program. In both cases, whenever an exception is thrown, control must be transferred from the code that caused the exception to the nearest dynamically-enclosing exception handler. To do so, Jikes RVM uses stack unwinding: stack frames are popped one at a time until an exception handler is reached. When a frame is popped by the exception handling mechanism, the normal (instrumented) epilogue is not executed, i.e. a mismatch in prologue versus epilogue. To solve this problem, we instrumented the exception handling mechanism as well to assure that the trace always contains records for methods that terminate because of an uncaught exception.

3.1.2 *Logging the trace data*

An instrumented run of our application results in multiple traces, one with the IDs for the compiled methods (baseline, optimized and OSR-compiled), and the others with the relevant counter information per thread. Each record in the latter requires 35 bytes at most, and provides the following information:

- A 1-byte tag, indicating the record type (high nibble) and the number of counter fields (1 up to 4) used in the record (low nibble). The record type denotes whether the record concerns data for a method entry, a method exit, a method exit through an exception, an entry into the compiler, a virtual thread switch, etc.
- Four bytes holding the method ID. This should prove more than sufficient for even very large applications.

[6] This is an OS-level POSIX thread, not a VM thread. This ensures that storing the trace data does not block the Virtual Processor POSIX thread on which the Jikes RVM executes.

- Eight bytes per counter field in the record. We can measure up to four hardware performance monitor values at a time.

It is possible to use a scheme in which the traces for each thread are written to a single file. In this case, we add extra synchronization to ensure the order of the entries in the trace is the same as the execution order. The disadvantage here is that there occurs a sequentialization during the profiling run, which can be bothersome when using multiple virtual processors on a multi-processor system. Also, in this case, each record will contain two extra bytes for the thread ID.

The total trace file size is thus a function of the number of method invocations, the number of virtual context switches and the number of traced events. Again, for clarification, the same structure is used for both the first step of our methodology (measuring execution times for each method) and the third step (measuring performance characteristics for the selected phases). However, for the first step we apply a heuristic so that we do not need to instrument all methods; this reduces the run-time overhead and prevents selecting wrapper methods as the starting point of a phase. A method is instrumented if the bytecode size of its body is larger than a given threshold (50 bytes), or if the method contains a backward branch, i.e. can contain a loop.

3.1.3 Instrumenting VM routines

As mentioned earlier, a VM consists of various components, such as class loader, compiler, optimizer, garbage collector, finalizer, thread scheduler, etc. To gain insight in Java workload behavior, it is thus of primary importance to profile these components. For most of these, this is easily done using the available Jikes RVM HPM infrastructure since they are run in separate VM threads. This is the case for the finalizer, the garbage collector and the optimizer (which uses six separate threads). To be able to capture the behavior of the compiler, we had to undertake special action since calling the compiler and optimizer is done in the Java application threads. In case of the baseline compiler, the compilation is done in the Java application thread itself. In case of the optimizing compiler, the method is queued to be optimized by the optimizer threads. These two cases were handled in our modified Jikes RVM implementation by manually instrumenting the `runtimeCompiler` and `compile` methods from `VM_Runtime`.

3.2 Phase identification

This section discusses how we identify phases using our off-line phase identification tool. Our tool takes a trace file with timing information about method calls and thread switches (see section 3.1.2) as input, analyzes it, and outputs a list of unique method names that represent the phases of the application.

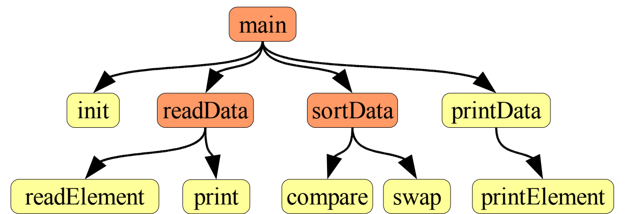
To select method-level phases, we use the algorithm proposed by Huang *et al.* [17] which requires two parameters, called θ_{weight} and θ_{grain}. The algorithm selects methods or phases as follows. Suppose the execution time of a program takes c_T clock cycles. Consider a method m for which the total execution time in this method including its callees equals $c_m = p_{\text{total}} \cdot c_T$ clock cycles, with $p_{\text{total}} < 1$. This is the total execution time consumed by this method over all its invocations. The average execution time consumed by this method (including its callees) equals $p_{\text{average}} \cdot c_m$. Method m will then be selected as a phase if the following conditions are met

$$p_{\text{total}} > \theta_{\text{weight}}$$
$$p_{\text{average}} > \theta_{\text{grain}}.$$

The basic idea of this definition is to select methods in which the program spends a substantial portion of its total execution time (this is done through θ_{weight}), and in which the program spends a sufficiently long period of time on each invocation (this eliminates short methods and is realized through θ_{grain}). As a result, the maximum number of selected methods equals $1/\theta_{\text{weight}}$ and the maximum number of method invocations profiled during our third step (measuring performance metrics using the hardware performance monitors) equals $1/(\theta_{\text{weight}} \cdot \theta_{\text{grain}})$.

To illustrate the phase identification algorithm, consider the call graph in Figure 3. It depicts a call tree that is the result of analyzing the trace file of a fictive sort program. The sort program reads the data to be sorted, prints an intermediate status message to the screen, sorts the data, and finally prints the sorted data before terminating. For simplicity, abstract time units are used. The table in Figure 3 also shows the total time spent in each method, as well as the time spent per invocation.

To identify program phases, our tool first computes the total and average execution times spent in each method. For all methods, these times include the time spent in their callees. In order for a method to be selected as a program phase, its total execution time needs to be at least a fraction θ_{weight} of the program's total execution time, and the average execution time should take at least a fraction θ_{grain} of the program's total execution time on average. In our running example, $\theta_{\text{weight}} = 10\%$ and $\theta_{\text{grain}} = 5\%$, would select methods whose total execution time is more than 180 and whose average execution time is more than 90 — `main`, `readData` and `sortData`, respectively.



method name	information total time	time/call	calls
main	1800	1800	1
init	30	30	1
readData	300	200	1
readElement	200	4	50
print	30	30	1
sortData	1300	1300	1
compare	600	2	300
swap	500	2	250
printData	170	170	1
printElement	150	3	50

Figure 3: Phase identification example.

benchmark	configuration		overhead		number of phases		trace file size
	θ_{weight}	θ_{grain}	estimated	measured	static (total)	dynamic (total)	(4 counters)
compress	$8 \times 10^{-6}\%$	$6 \times 10^{-6}\%$	1.84%	1.82%	49 (54)	2,664 (19,726,311)	211KiB
jess	1.0 %	1.0%	1.22%	1.27%	10 (211)	23 (22,693,249)	68KiB
db	$8 \times 10^{-6}\%$	$6 \times 10^{-6}\%$	7.17%	5.61%	52 (57)	32,223 (1,484,605)	2590KiB
javac	$2 \times 10^{-2}\%$	$6 \times 10^{-3}\%$	2.61%	2.11%	29 (503)	9,864 (23,388,699)	871KiB
mpegaudio	$2 \times 10^{-2}\%$	$2 \times 10^{-3}\%$	10.75%	3.52%	23 (191)	40,064 (29,338,068)	753KiB
mtrt	$10^{-2}\%$	$10^{-3}\%$	24.68%	7.83%	30 (94)	88,719 (14,859,306)	2,680KiB
jack	1.0%	$10^{-2}\%$	3.98%	4.28%	18 (182)	2,528 (4,292,580)	244KiB
PseudoJBB	$2 \times 10^{-1}\%$	$2 \times 10^{-4}\%$	3.69%	6.65%	52 (381)	29,599 (16,224,804)	2,766KiB

Table 3: Summary of the selected method-level phases for the chosen θ_{weight} and θ_{grain} values: overhead (estimated versus real), the number of static and dynamic phases, and the size of the trace file.

4. STATISTICAL TECHNIQUES

To quantify the variability within phases, we use the Coefficient of Variation (CoV). We first measure the CoV for a given performance metric within a phase. This is defined as the standard deviation divided by the mean average value for that metric within the given phase. The overall CoV is then obtained by averaging over the various phases after weighting with the execution time spent in each of the phases. The smaller the CoV the less variability is observed within a phase.

For quantifying the variability within phases versus the variability between phases, we employ the ANOVA technique [18]. ANOVA allows us to verify whether the mean values for a given set of characteristics are significantly different for each phase by looking at the variability in the observations. In other words, our goal is to verify whether the variability is larger between the phases than within each phase. If this is the case, our methodology succeeds in identifying recurring phases in which the behavior is stable and similar. An ANOVA analysis is done as follows. First, the total variability in the observations X_{ij} (phase $i = 1 \ldots k$ and measurement $j = 1 \ldots n_i$ for phase i) can be expressed in terms of the deviations from the overall mean $\bar{X}$, i.e. $X_{ij} - \bar{X}$. Second, each measurement for a given phase i can be expressed in terms of the deviation from the mean for the phase $\bar{X}_i$, i.e. $X_{ij} - \bar{X}_i$. Finally, the mean for each phase i can be expressed in terms of the deviation from the overall mean, i.e. $\bar{X}_i - \bar{X}$. This can be expressed by the following formula

$$\left(X_{ij} - \bar{X}\right) = \underbrace{\left(\bar{X}_i - \bar{X}\right)}_{\tau_i} + \underbrace{\left(X_{ij} - \bar{X}_i\right)}_{\epsilon_{i,j}}$$

which is equivalent to

$$\sum_i \sum_j \left(X_{ij} - \bar{X}\right)^2 = \sum_i n_i \tau_i^2 + \sum_i \sum_j \epsilon_{i,j}^2$$

Clearly, if the mean values for the various phases are significantly different, the τ_i will be significantly larger than the $\epsilon_{i,j}$. If this is the case, the F-statistic

$$\frac{\sum_i n_i \tau_i^2/(k-1)}{\sum_i \sum_j \epsilon_{i,j}^2/(\sum_i n_i - k)}$$

that is used to compare both values with respect to all phases will yield a large value. The corresponding p-value will be smaller than 0.05 for a 95% level of significance or smaller than 0.01 for a 99% level of significance. Applying the ANOVA analysis in this paper is done using R [23], an S dialect.

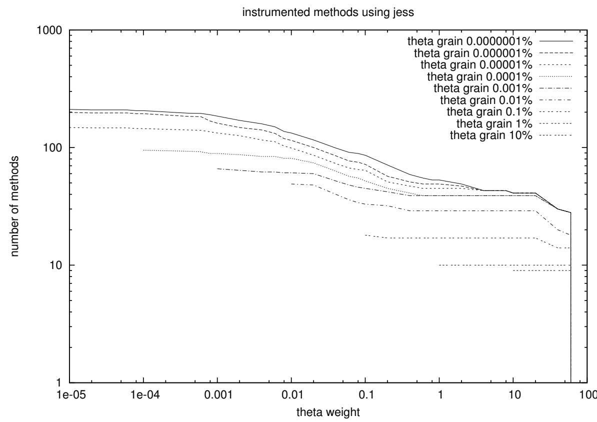
5. RESULTS

5.1 Identifying method-level phases

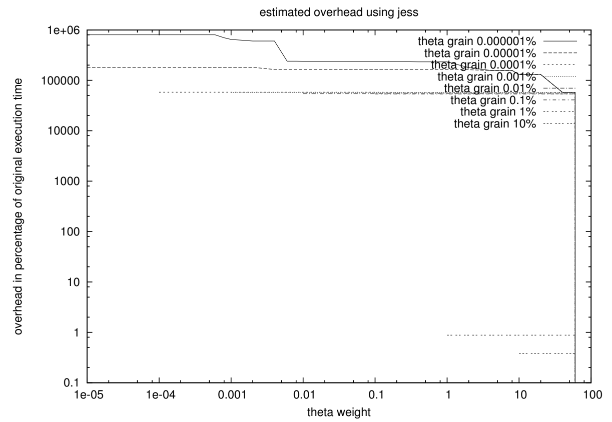
Tracing all methods at their entry and exit points is very intrusive. Thus, it is important to determine a set of method-level phases such that the incurred overhead is relatively low, but such that we still get a detailed picture on what happens at the level of the methods being executed. This is done by choosing appropriate values for θ_{weight} and θ_{grain} . These values depend on three parameters: (i) the maximum acceptable overhead, (ii) the required level of information, and (iii) the application itself. The off-line analysis tool aids in selecting values for θ_{weight} and θ_{grain} by providing an estimate for both the overhead and the information yielded by each possible configuration. The left column of Figure 4 presents the number of selected method-level phases as a function of θ_{weight} and θ_{grain} . The right column of Figure 4 shows the corresponding estimated overhead which is defined as the number of profiled method invocations (corresponding to method-level phases) divided by the total number of method invocations. Note that this is not the same as coverage, since selected methods also include their callees. The coverage is always 100% in our methodology. Figure 4 only presents data for three (jess, jack, and PseudoJBB) out of the eight benchmarks we analyzed because the remaining five benchmarks showed similar results. Clearly, the larger θ_{weight} , the fewer methods will be selected. The higher the value of θ_{grain} , the less short-running methods will be selected.

Using the plots in Figure 4 we can now choose appropriate values for θ_{weight} and θ_{grain} for each benchmark. This is done by inspecting the curves with the estimated overhead. We choose θ_{weight} and θ_{grain} in such a way that the estimated overhead is smaller than 1%, i.e. we want less than 1% of all method invocations to be instrumented. The results for each benchmark are shown in Table 3. Note that the user has some flexibility for determining appropriate values for θ_{weight} and θ_{grain} ; this allows the user to determine the number of selected method-level phases according to his interest.

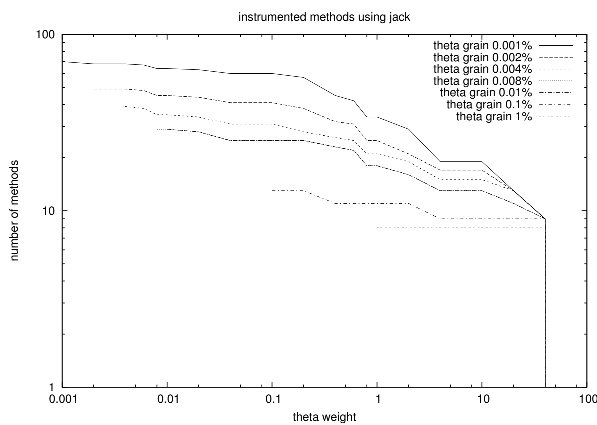
So far, we have used an estimated overhead which is defined as the number of profiled method invocations versus the total number of method invocations of the complete program execution. To validate these estimated overheads, i.e. to compare with the actual overheads, we proceed as fol-



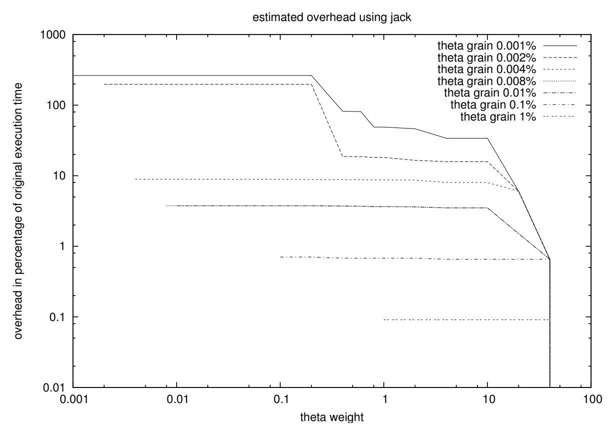
(a)



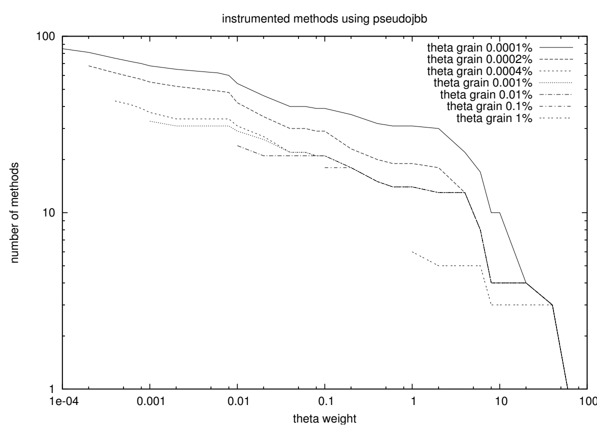
(b)



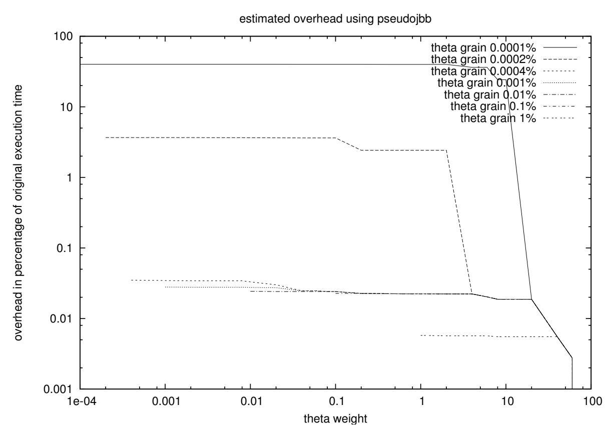
(c)



(d)



(e)



(f)

Figure 4: Estimating the overhead as a function of θ_{weight} and θ_{grain} for jess (a,b), jack (c,d), and PseudoJBB (e,f). The figures on the left present the number of selected method-level phases; the figures on the right present the estimated overhead.

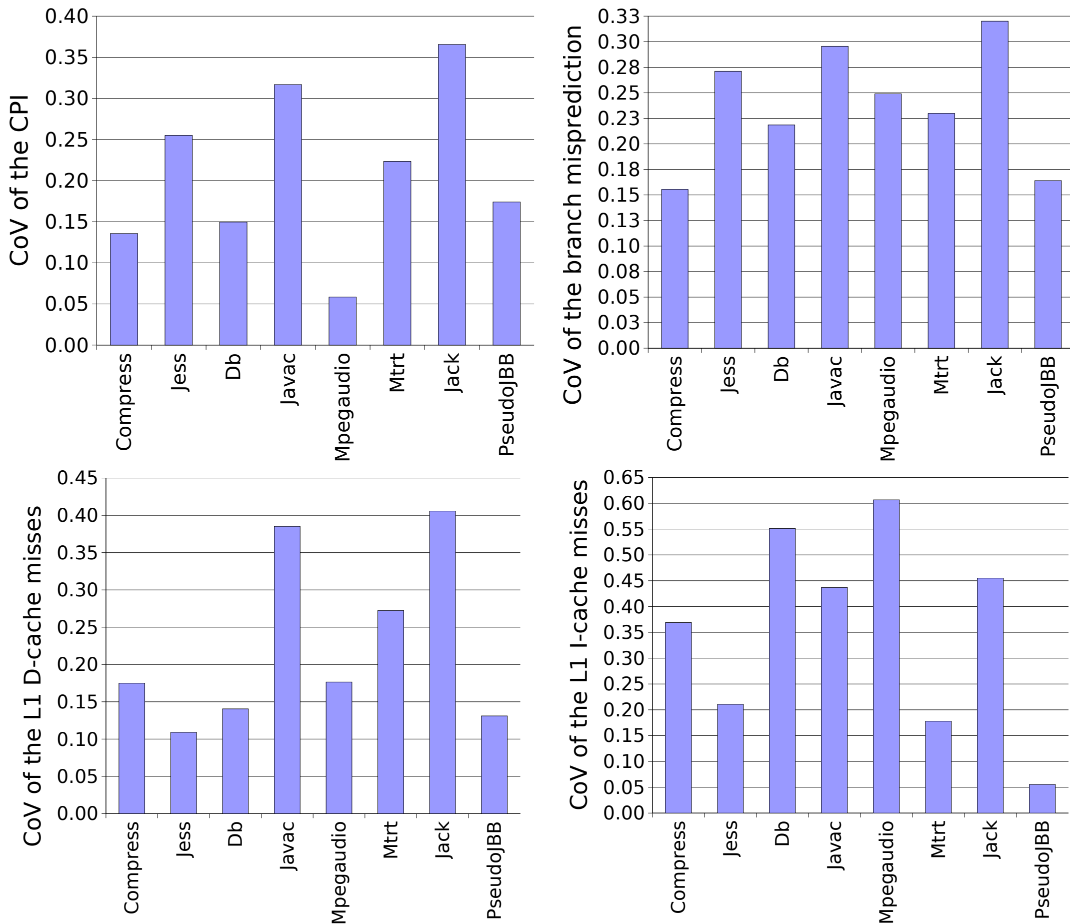


Figure 5: Accumulated weighted CoV values for the various benchmarks for four characteristics: (a) CPI, (b) branch mispredictions (c) L1 D-cache misses, and (d) L1 I-cache misses.

lows. We measure the total execution time of each benchmark (without any profiling enabled) and compare this with the total execution time when profiling is enabled for the selected methods. The actual overhead is defined as the increase in execution time due to adding profiling. Measuring the wall clock execution time is done using the GNU/Linux `time` command. Table 3 compares the estimated overhead and the actual overhead. We observe from these results that the actual overhead is usually quite small and mostly tracks the estimated overhead very well. This is important since determining the estimated overhead is more convenient than measuring the actual overhead. In two cases the estimate is significantly larger than the measured overhead, i.e. for mpegaudio and for mtrt.

For completeness, Table 3 also presents the number of static method-level phases as well as the number of phase invocations or dynamic phases. For reference, the total number of static methods as well as the total number of dynamic method invocations are shown. The rightmost column in Table 3 presents the file size of the trace file obtained from running the Java application while profiling the selected method-level phases. Recall that besides application method records, the trace also contains data regarding thread switches, GC, and compiler activity.

5.2 Variability within and between phases

The purpose of this section is to evaluate the variability within and between the phases. Our first metric is the Coefficient of Variation (CoV) which quantifies the variability within a phase, see section 4. Figure 5 shows the CoV for the various benchmarks over various characteristics (CPI, L1 D-cache misses, L1 I-cache misses, and branch mispredictions). We observe that for the CPI and the branch mispredictions, the CoV is quite small. This indicates that the behavior within a phase is quite stable for these characteristics. The I-cache behavior on the other hand, is not very stable within the phases for mpegaudio. This can be due to the low I-cache miss rate for the mpegaudio for which a similarly small variation exists. Indeed, a small variation in absolute terms, e.g. 0.0001 versus 0.0002, can result in large variations in relative terms (100% in this example). As such, we do not consider this as a major problem since analysts do not care about code having very low cache miss rates. Furthermore, if unstable behavior for a given characteristic is undesirable, θ_{weight} and θ_{grain} can be adjusted so that more phases are selected and less variability will be observed within the phases.

To get a better view on the performance characteristics within and between phases, we use boxplots. Figure 6 shows

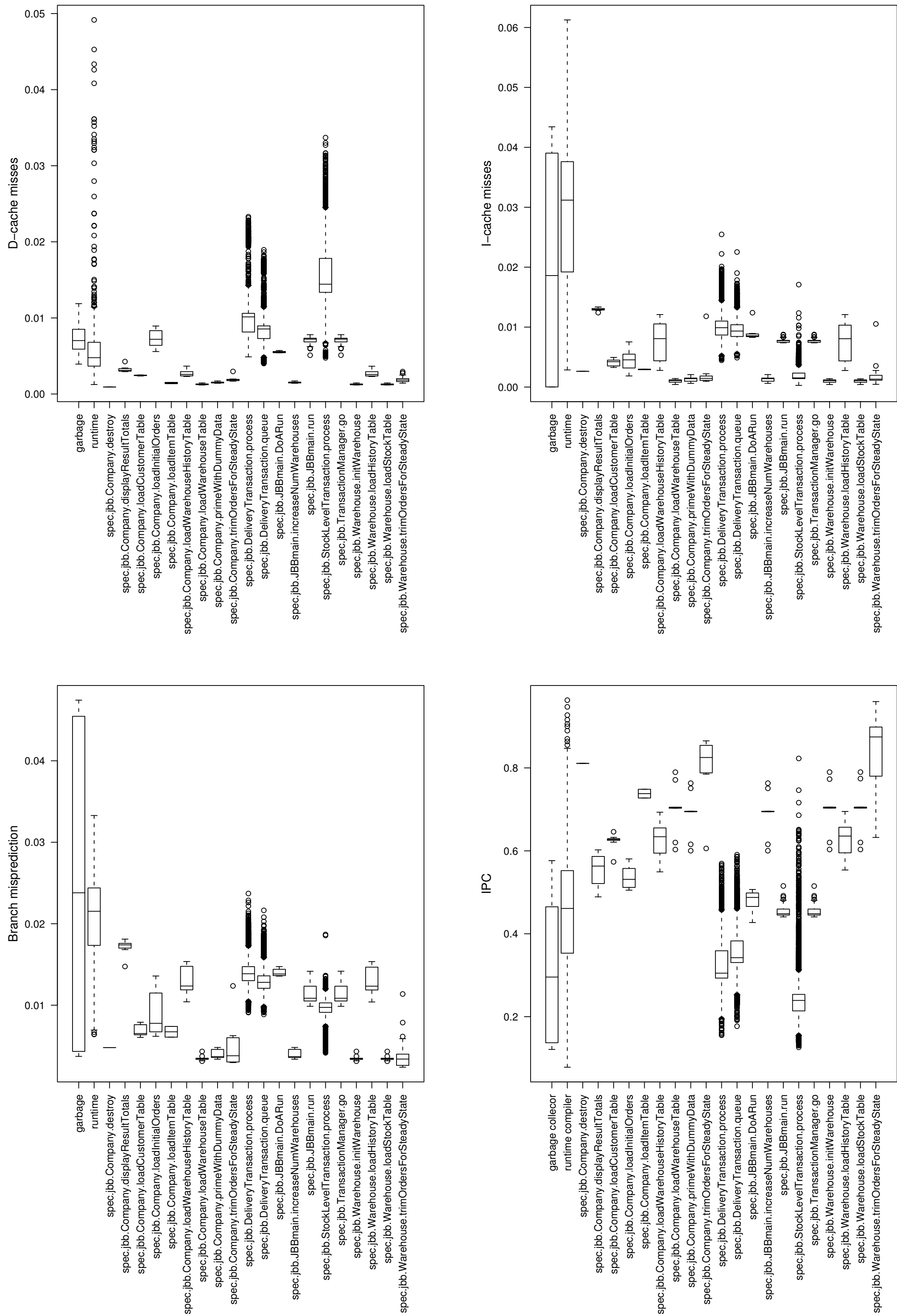


Figure 6: Boxplots showing the distribution for the phases of PseudoJBB on the following characteristics: (a) L1 D-cache misses, (b) L1 I-cache misses, (c) branch mispredictions, and (d) IPC.

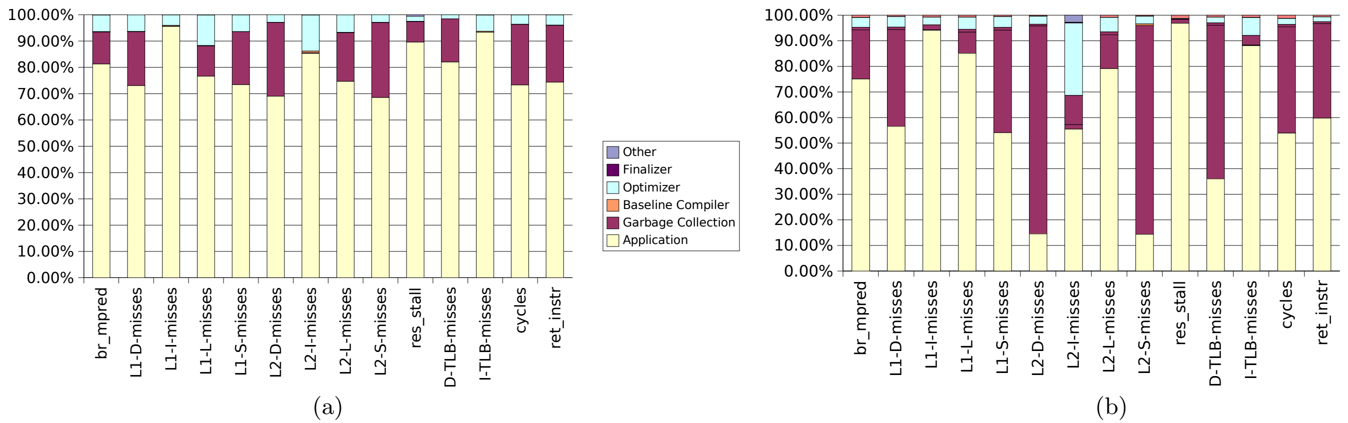


Figure 7: Performance characteristics for the application versus the VM components for PseudoJBB (a) and jack (b).

benchmark	configuration		ANOVA results		
	θ_{grain}	θ_{weight}	F-value	degrees of freedom	p-value
compress	$8 \times 10^{-6}\%$	$6 \times 10^{-6}\%$	37.94	(48, 2640)	$< 10^{-16}$
jess	1.0 %	1.0%	1.74	(10, 664)	0.067
db	$8 \times 10^{-6}\%$	$6 \times 10^{-6}\%$	43.03	(49, 32339)	$< 10^{-16}$
javac	$2 \times 10^{-2}\%$	$6 \times 10^{-3}\%$	117.34	(20, 10749)	$< 10^{-16}$
mpegaudio	$2 \times 10^{-2}\%$	$2 \times 10^{-3}\%$	495.95	(26, 102894)	$< 10^{-16}$
mtrt	$10^{-2}\%$	$10^{-3}\%$	39.23	(26, 89004)	$< 10^{-16}$
jack	1.0%	$10^{-2}\%$	63.56	(14, 2934)	$< 10^{-16}$
PseudoJBB	$2 \times 10^{-1}\%$	$2 \times 10^{-4}\%$	288.24	(19, 32206)	$< 10^{-16}$

Table 4: Results for ANOVA comparing the means for the observed characteristics.

the performance characteristics for the various phases for PseudoJBB. On the X axis of this graph, we display all the phases. The Y axes represent the various metrics that were measured: IPC, L1 I-cache miss rate, L1 D-cache miss rate and the branch misprediction rate. For each phase, we display the mean value as the middle of the rectangular box. The borders of each box represent the standard deviation and the individual points above and under these boxes represent outliers, i.e. not within one standard deviation from the mean. This figure clearly shows that the performance characteristics can be quite different between phases. The variability within each phase on the other hand, is usually small. Notable exceptions to this are the GC and VM compiler.

We now quantify the variability between the phases versus the variability within the phases in a more rigorous way. This is done using an ANOVA test, see section 4. In Table 4, we show the *maximum* p-value per benchmark over all characteristics. Recall that the lower p, the better. In our results, p is smaller than 10^{-16} for nearly all benchmarks, from which we conclude that the mean values for each characteristic for the various phases are different at a significance level that almost reaches certainty. This means that the variability between the phases is significantly larger than the variability within the phases which proves that our off-line technique reliably identifies phases with dissimilar inter-phase behavior and similar intra-phase behavior. For jess however, the p value reported in Table 4 is larger than for the other benchmarks. The higher p-values are due to the I-TLB and D-TLB miss rates, which do not show that much variability between the phases.

5.3 Analysis of method-level phase behavior

A programmer analyzing application behavior will typically start from a high-level view of the program. Two of the first things one wants to analyze are where the time is spent, and whether potential bottlenecks are due to the application or the VM. In the first subsection, we look at the high-level behavior of Java applications and compare it with the behavior of the VM (GC, compiler, etc.). Once the high-level behavior has been understood, the next logical step is to investigate parts of the application into more detail. The subsequent subsection shows how the programmer can use the information collected by our framework to gain insight about the low-level behavior of his program, and how our data can help identify and explain performance bottlenecks.

5.3.1 VM versus application behavior

Figure 7(a) shows the number of events occurring in the application versus the VM. This is done here for PseudoJBB. We observe that most of the events occur in the application and not in the VM. Indeed, the total program execution spends 73% of its total execution time in the application; the remaining 27% is spent in the VM. The time spent in the VM is partitioned in the time spent in the various VM components: compiler, optimizer, garbage collector, finalizer and others. We observe that the most dominant part

benchmark	application	garbage collector	finalizer	compiler		other
				baseline	optimizing	
compress	89.136%	9.377%	0.006%	0.205%	0.696%	0.580%
jess	56.835%	39.641%	0.003%	0.919%	1.914%	0.688%
db	92.211%	6.991%	0.002%	0.128%	0.455%	0.213%
javac	65.463%	28.987%	0.008%	0.940%	3.618%	0.984%
mpegaudio	85.000%	7.999%	0.002%	0.559%	5.821%	0.620%
mtrt	65.802%	28.039%	0.005%	0.485%	4.687%	0.982%
jack	53.905%	41.556%	0.015%	0.941%	2.317%	1.265%
PseudoJBB	73.348%	22.974%	< 0.001%	0.091%	3.532%	0.063%

Table 5: The time spent in the application and the different VM components.

of the VM routines is due to the optimizer (3.5%) and the garbage collector (23%). This graph reveals several interesting observations. For example, although the optimizer is responsible for only 3.5% of the total execution time, it is responsible for a significantly larger proportion of the L1 D-cache load misses (6.3%) and L2 I-cache misses (13.7%). The garbage collector on the other hand, accounts for significantly more L2 D-cache misses (28%) than it accounts for execution time (21%). Another interesting result is that the garbage collector accounts for a negligible fraction of the L1 and L2 I-cache and I-TLB misses. This is due to the fact that the garbage collector has a small instruction footprint while accessing a large data set.

Figure 7(b) presents a similar graph for jack. The percentage of the total execution time spent in the application is 54%. Of the 46% spent in the VM, 41.5% is spent in the garbage collector, 2.3% in the optimizer, 0.9% in the baseline compiler and 1.3% in other VM routines, such as the thread scheduler, class loader, etc. These results confirm the specific behavior of the garbage collector previously observed for PseudoJBB: low L1 and L2 I-cache and I-TLB miss rates and high L2 D-cache and D-TLB miss rates (due to writes). The baseline compiler and the optimizer show high L2 I-cache miss rates.

Table 5 presents the time spent in the application versus the time spent in the VM components for the SPECjvm98 and PseudoJBB benchmarks. The time spent in the application varies between 54% and 92% of the total execution time; the time spent in the garbage collector varies between 7% and 42% and the time spent in the optimizer varies between 0.4% and 5.8%. The execution time in the other VM components is negligible. We conclude that Java workloads spend a significant fraction of their total execution time in the VM, up to 46% for the long-running applications included in our study. For short-running applications, for example SPECjvm98 with the `s1` input set, this fraction will be even larger. It is interesting to note that the three benchmarks (compress, db, and mpegaudio) for which the total execution time spent in the application is significantly larger than the average case (89%, 92% and 85%, respectively), were denoted as 'simple' benchmarks by Shuf *et al.* [27].

5.3.2 Application bottleneck analysis

Profilers provide a means for programmers to perceive the way their programs are performing. Our technique provides an easy way to the programmer to gain insight about the performance of their application at the micro-architectural level. That is, hardware performance counters can be linked to the methods in the source code. Conventional methods on the other hand, are much more labor-intensive and error-prone for the following reasons: (i) the huge amount of data gathered from a profiling run, and (ii) the presentation of this huge amount of data usually prevents quick insight.

This section shows how our technique can help answer three fundamental questions programmers might ask when optimizing their application: (i) what is the application's bottleneck, (ii) why does the bottleneck occur, and (iii) when does the bottleneck occur?

To answer the first question (what is the bottleneck?), we compile a list of phases with the highest CPI values. The methods with the highest CPI are most likely to represent a bottleneck. To answer the second question (why does the bottleneck occur?) we investigate the corresponding HPM counters of the bottleneck phase(s). To answer the last question (when does my bottleneck occur?) we can plot the CPI over time.

Table 6 presents the major bottleneck phases for both the SPECjvm98 benchmarks and for PseudoJBB. Due to space constraints Table 6 depicts a subset of all phases only: we show the phases of which the total execution time takes more than 1% or 2% of the program execution time and of which the CPI is above the average CPI, or which otherwise display bad behavior for a shown characteristic. This table shows the percentage of the total execution time that is spent in each phase, the average CPI in each phase, the cache miss rates and the branch misprediction rate. This information can be helpful in identifying why these phases suffer from such a high CPI. For example, high D-cache miss rates suggest that the programmer should try to improve the data memory behavior for the given phases. We can make the following interesting observations—these are just a few examples to clarify the usefulness of linking microprocessor level information to source-code level methods.

- The `Compressor.compress` method in compress suffers from high D-cache miss rates. Optimization of the data memory behavior can be achieved by applying prefetching.

- From all the benchmarks, `mtrt` has a method with the most mispredicted branches: `Scene.RenderScene`. This method contains two nested loops, iterating over all pixels in the scene to be rendered. Inside the loop there are a number of conditional branches and a call to e.g. `Scene.Shade`. In turn, the latter shows bad branch behavior due to numerous (nested) tests that are conducted to decide on the color of the pixel that is being rendered. This behavior can be optimized by

phase	time	CPI	L1-D	L1-I	L2-D	L2-I	br_mpred
compress							
garbage collector	9.3773%	1.7778	4.04	0.02	2.59	0.01	4.39
Compressor.compress	58.3916%	1.7447	22.91	0.01	4.36	< 0.01	7.28
Decompressor.decompress	25.2042%	0.9242	2.53	< 0.01	0.12	< 0.01	4.83
benchmark average	n/a	1.4830	12.25	0.10	2.01	0.04	5.69
jess							
Jesp.parse	1.1021%	1.8701	4.52	5.91	1.04	1.71	14.55
garbage collector	39.6413%	1.7647	4.02	0.02	2.58	0.01	4.33
Rete.Run	53.8732%	1.1796	4.92	0.51	0.45	0.05	4.51
benchmark average	n/a	1.3959	4.66	0.68	1.12	0.17	4.73
db							
Database.shell_sort	85.5593%	5.1134	26.42	0.02	18.01	0.01	4.87
Database.remove	4.5821%	2.7155	11.33	0.10	6.28	0.06	1.36
garbage collector	6.9912%	1.7989	3.92	0.03	2.45	0.01	4.23
Database.set_index	2.3873%	1.5749	5.74	0.08	3.38	0.04	0.08
benchmark average	n/a	3.9847	4.71	0.05	3.13	0.01	1.07
javac							
SourceClass.check	7.0644%	2.2501	8.06	13.13	1.74	1.75	23.2
SwitchStatement.check	1.4973%	1.9129	7.91	13.21	0.76	0.49	22.87
garbage collector	28.9874%	1.8059	4.48	0.02	2.55	0.01	4.76
SourceClass.compileClass	26.0361%	1.7408	5.14	6.53	0.98	0.85	16.26
Assembler.collect	1.8285%	1.6994	4.96	4.58	1.23	0.52	13.64
Parser.parseClass	22.6849%	1.4789	2.93	5.52	0.54	0.44	18.26
ConstantPool.write	5.3863%	1.0231	1.20	0.19	0.33	0.08	6.26
benchmark average	n/a	1.6747	4.28	4.48	1.32	0.66	13.26
mpegaudio							
garbage collector	7.9994%	1.7795	4.15	0.03	2.62	0.01	4.71
lb.read	20.7281%	0.8602	1.41	1.17	0.01	< 0.01	6.38
t.O	75.1119%	0.8430	0.82	0.49	< 0.01	< 0.01	2.29
benchmark average	n/a	0.8157	1.07	0.47	0.03	0.02	3.25
mtrt							
Scene.RenderScene	1.9640%	2.3249	12.32	18.74	0.21	0.25	35.98
garbage collector	28.0391%	1.7829	3.73	0.02	2.40	< 0.01	4.47
Scene.GetLightColor	23.7782%	1.4919	9.74	3.29	0.68	0.03	8.95
Scene.Shade	36.2558%	1.3496	7.21	2.84	0.42	0.05	11.42
Scene.ReadPoly	2.4275%	1.2909	1.52	3.32	0.12	0.10	8.88
benchmark average	n/a	1.5389	8.27	2.66	1.03	0.07	7.18
jack							
Jack_the_Parser_Generator._Jack3_1	2.34092%	1.9655	5.84	5.19	0.57	0.22	11.53
garbage collector	41.5561%	1.7741	4.04	0.02	2.59	0.01	4.36
Jack_the_Parser_Generator.production	1.87112%	1.7039	4.38	7.41	0.51	0.73	15.16
Jack_the_Parser_Generator.jack_input	2.8412%	1.6014	3.59	6.26	0.38	0.56	13.69
Jack_the_Parser_Generator.expansion_choices	20.5098%	1.5546	4.46	7.54	0.22	0.23	15.94
Jack_the_Parser_Generator.java_declarations_and_code	19.4081%	1.3648	2.70	5.1	0.11	0.09	12.64
Jack_the_Parser_Generator_Internals.db_process	2.78693%	1.2737	2.61	1.61	0.59	0.28	4.74
ParseGen.build	2.6689%	1.1157	2.27	0.37	0.69	0.06	2.41
benchmark average	n/a	1.5976	3.83	3.58	1.19	0.24	9.58
PseudoJBB							
DeliveryTransaction.process	2.7597%	3.0722	8.74	9.95	6.45	2.61	17.32
garbage collector	22.9744%	2.1581	5.76	0.03	3.59	< 0.01	4.35
TransactionManager.go	57.9074%	2.1219	6.77	7.77	2.91	0.75	11.08
benchmark average	n/a	2.046	6.02	5.02	2.69	0.57	9.13

Table 6: Interesting methods from the SPECjvm98 and SPECjbb2000 (as observed in PseudoJBB) benchmark suites. The L1 and L2 I-cache miss rates, L1 and L2 D-cache miss rates and the branch misprediction rate are given as the number of events per 1,000 instructions.

changing the code layout to improve the branch predictability.

- Poor I-cache behavior can be observed for e.g. the `expansion_choices` method in `jack`.
- For the SPECjvm98 benchmarks, the GC shows a very consistent behavior, with a CPI that remains around 1.77. Also, GC shows a very good I-cache behavior both on L1 and L2. This due to the fact that GC (i) usually can take quite some time, hence the initial cache misses can be made up for by a longer execution time, and (ii) GC code is usually quite compact.

6. RELATED WORK

The first subsection details on related work done on characterizing Java workloads. In the second subsection, we discuss phase classification and detection techniques.

6.1 Java workload characterization

Cain *et al.* [9] characterize the Transaction Processing Council's TPC-W web benchmark which is implemented in Java. TPC-W is designed to exercise the web server and transaction processing system of a typical e-commerce web site. They used both hardware execution (on an IBM RS/6000 S80 server with 8 RS64-III processors) and simulation in their analysis.

Karlsson *et al.* [19] study the memory system behavior of Java-based middleware. To this end, they study the SPECjbb2000 and SPECjAppServer2001 benchmarks on real hardware as well as through simulation. For the real hardware measurements, they use the hardware counters on a 16-processor Sun Enterprise 6000 multiprocessor server. They measure performance characteristics over the complete benchmark run and make no distinction between the VM and the execution phases of the application.

Luo *et al.* [21] compare SPECjbb2000 versus SPECweb99, VolanoMark and SPEC CPU2000 on three different hardware platforms: the IBM RS64-III, the IBM POWER3-II and the Intel Pentium III. All measurements were done using performance counters and measure aggregate behavior.

Eeckhout *et al.* [15] measure various performance counter events and use statistical data analysis techniques to analyze Java workload behavior at the microprocessor level. One particular statistical data analysis technique that is used in that paper is principal components analysis which allows to reduce the dimensionality of the data set. This reduced data set allows for easier reasoning. In that work, the authors also measured aggregate performance characteristics and made no distinction between phases of execution.

Dufour *et al.* [14] present a set of architecture-independent metrics for describing dynamic characteristics of Java applications. All these metrics are bytecode-level program characteristics and measure program size, the intensiveness of various data structures (arrays, pointers, floating-point operations), memory use, concurrency, synchronization and the degree of polymorphism.

Dmitriev [12] presents a bytecode-level profiling tool for Java applications, called JFluid. During a typical JFluid session, the VM is started with the Java application without any special preparation. Subsequently, the tool is attached to the VM, the application is instrumented, the results are collected and analyzed on-line, and the tool is detached from the VM. The instrumentation is done by injecting instrumentation bytecodes into methods of a running program. In JFluid, the user needs to specify which call subgraph, called a 'task' by Dmitriev, from an arbitrary root method is to be instrumented. This method has two major differences with our approach: (i) we do not operate at the bytecode level but at the lower microprocessor level, and (ii) we provide a means to automatically detect these 'tasks'. This relieves the user from manually selecting major tasks of execution.

Sweeney *et al.* [28] present a system to measure microprocessor level behavior of Java workloads. To this end, they generate traces of hardware performance counter values while executing Java applications. This is done for each Java thread and for each microprocessor on which the thread is running. The latter can be useful in case of a multiprocessor environment. The infrastructure for reading performance counter values used by Sweeney *et al.* is exactly the same as the one used in this paper—using HPM in the Jikes RVM—except for the fact that our measurements are done on an IA-32 ISA platform opposed to the PowerPC ISA platform. Sweeney *et al.* read the performance counter values on every virtual context switch in the VM. This information is collected for each virtual processor and for each Java thread, and written in a per virtual processor record buffer. Sweeney *et al.* also present a tool for graphically exploring the performance counter traces. The major difference between the work by Sweeney *et al.* and this paper, is that we collect performance counter values on a per-phase basis as opposed to the timing-driven approach of taking one sample on every virtual context switch. The benefit of measuring performance counter values on a per-phase basis is that performance counter values can be easily linked to the code that is executed in the phase. We believe this is particularly useful for analysis in general, and for application and VM developers in particular. Moreover, our approach is more general than the approach by Sweeney *et al.* since the information we obtain can be easily transformed to behavioral information over time. This can be done by ordering our information on a time-line basis. Additionally, Sweeney *et al.* use an on-line approach, while we essentially perform an offline analysis.

6.2 Program phases

Several techniques that have been proposed in the recent literature to detect program phases divide the total program execution in fixed intervals. For each interval, program characteristics are measured during program execution. When the difference in program characteristics between two consecutive intervals exceeds a given threshold, the algorithm detects a phase change. These approaches are often referred to as *temporal techniques.* The proposed temporal techniques all differ in what program characteristics need to be measured over the fixed interval. Balasubramonian *et al.* [6] compute the number of dynamic conditional branches executed. A phase change is detected when the difference in branch counts between consecutive intervals exceeds a threshold. This threshold is adjusted dynamically during program execution to match the program's execution behavior. Dhodapkar and Smith [11] use the instruction working set or the instructions executed at least once. Since representing a complete working set is impractical, especially in hardware, the authors propose working set signatures which are lossy-compressed representations of working sets. Work-

ing set signatures are compared using a relative signature distance. A program phase change is detected when the relative signature distance between consecutive intervals exceeds a given threshold. Sherwood et al. [25, 26] use basic block vectors (BBVs) to identify phases. A BBV is a vector in which the elements count the number of times each static basic block is executed in the fixed interval. These BBVs are weighted by the number of instructions in the given basic block. A phase change is detected when the Manhattan distance between two consecutive intervals exceeds a given threshold. They consider both static and dynamic methods for identifying phases in [25] and [26], respectively. The purpose of their static phase classification approach was to identify equally behaving intervals throughout a program execution so that one single representative interval for each phase can be used for efficient simulation studies. In a follow-up study, Lau *et al.* [20] study several structures for classifying program execution phases. They study approaches using basic blocks, loops, procedures, opcodes, register usage and memory address information. In contrast to the previously mentioned approaches which all use micro-architecture-independent characteristics—i.e. the metrics are only dependent on the instruction set architecture (ISA) and not on the micro-architecture—Duesterwald *et al.* [13] use micro-architecture-dependent characteristics to detect phases. The metrics used by them are the instruction mix, the branch prediction accuracy, the cache miss rate and the number of instructions executed per cycle (IPC). These metrics are measured using performance counters over fixed intervals of 10 milliseconds.

Next to temporal phase detection approaches, there exist a number of approaches that do not use fixed intervals. Balasubramonian et al. [6] consider procedures to identify phases. They consider non-nested and nested procedures as phases. A non-nested procedure is a procedure that includes its complete call graph, i.e., including all the methods it calls, as is done in this paper. A nested procedure does not include its callees. They concluded that non-nested procedures are better performing than nested procedures. Huang *et al.* [17] also use procedures to identify phases. The method used in our work to identify method-level phases of execution—using θ_{weight} and θ_{grain} —is based on the approach proposed by Huang *et al.* Next to this static approach, they also propose a hardware-based call stack mechanism to identify program phase changes. This paper differs from the one by Huang *et al.* for at least three reasons. First, we explore the technique for detecting phases in more detail by quantifying the overhead and coverage as a function of θ_{weight} and θ_{grain} . Huang *et al.* chose fixed $\theta_{\text{weight}} = 5\%$ and $\theta_{\text{grain}} = 1{,}000$ cycles in their experiments. Second, we study Java workloads whereas Huang *et al.* studied SPEC CPU2000 benchmarks. Java workloads provide several additional challenges over C-style workloads because of the managed run-time environment. Third, the focus of the work by Huang *et al.* was on exploiting phase behavior for energy-efficient computing. The focus of our work is on using phase behavior to increase the understanding during whole-program analysis

7. CONCLUSIONS AND FUTURE WORK

Java workloads are complex pieces of software in which the Java application closely interacts with the VM. In addition, the applications themselves are becoming increasingly complex due to the ever-increasing processing power of current microprocessor systems. Because of this, automatic tools for characterizing such software systems are becoming paramount during performance analysis.

The purpose of this paper was to study method-level phase behavior of Java applications. In other words, our goal was to identify methods (including their callees) that exhibit similar behavior within the phase and dissimilar behavior between the phases. The phase analysis framework presented in this paper consists of three steps. In the first step, we measure the execution time for each method invocation using hardware performance monitors which are made available through the Jikes RVM HPM API. The second step analyzes this information using an off-line tool and selects a number of phases. These phases are subsequently characterized in the third step using performance counters. This characterization includes measuring a number of microprocessor events such as cache miss rates, TLB miss rates, branch misprediction rates, etc. for each selected phase.

Using this framework, we investigated the phase behavior of both the SPECjvm98 and SPECjbb2000 benchmark suite. In a first set of experiments, we have compared the characteristics of the Java application versus the various VM components. We concluded that Java workloads spend a significant portion of their total execution time (up to 46%) in the VM, more specifically in the garbage collector. In addition, the VM exhibits a significantly different behavior from the Java application, and this can vary widely over different applications. In a second set of experiments, we have focused on the method-level phase behavior of the Java application itself. We have shown that our phase analysis technique is capable of reliably discriminating method-level phases since a larger variability is observed between the phases than within the phases. We have also shown that the overhead incurred during profiling is small.

Particularly novel compared to existing work is the fact that our framework can link the microprocessor-level information to the methods in the Java application's source code. This provides a way for programmers to identify performance bottlenecks automatically which can guide them while optimizing their software.

In future work we will extend the ideas presented in this paper as several future research directions open up. First, we will explore on-line phase identification. Detecting and selecting phases at run-time is interesting for program analysis because we could eliminate the required training run. A second future direction is the development of a framework capable of both visualizing and replaying the profile information captured by our tool. Such a framework could result in new contributions that help gain deep understanding of program behavior of Java applications. A third possible direction is to develop techniques that are capable of exploiting the collected phase information. The insights obtained in this paper can be beneficial for VM, garbage collector and compiler optimizations. Furthermore, the information could lend itself to predict the behavior of Java applications based on the method-level phase behavior.

Acknowledgments

Andy Georges is supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) in the CoDAMoS project, and Dries Buytaert is supported by an IWT grant. Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research—Flanders

(Belgium) (F.W.O.—Vlaanderen). This research was also funded by Ghent University.

8. REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[2] B. Alpern, B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[3] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA) 2000*, pages 47–65. ACM, 2000.

[4] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of Java. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA) 2002*, pages 111–129. ACM, 2002.

[5] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation (PLDI) 2000*, pages 1–12. ACM, 2000.

[6] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 245–257. ACM, 2000.

[7] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[8] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141. ACM, 1999.

[9] Harold W. Cain, Ravi Rajwar, Morris Marden, and Mikko H. Lipasti. An architectural evaluation of Java TPC-W. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 2001.

[10] Ashutosh S. Dhodapkar and James E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 233–244. IEEE Computer Society, 2002.

[11] Ashutosh S. Dhodapkar and James E. Smith. Comparing program phase detection techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 217. IEEE Computer Society, 2003.

[12] Mikhail Dmitriev. Selective profiling of Java applications using dynamic bytecode instrumentation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, 2004.

[13] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 220. IEEE Computer Society, 2003.

[14] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 149–168. ACM, 2003.

[15] Lieven Eeckhout, Andy Georges, and Koen De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA) 2003*, pages 169–186. ACM, 2003.

[16] Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the international symposium on Code generation and optimization*, pages 241–252. IEEE Computer Society, 2003.

[17] Michael C. Huang, Jose Renau, and Josep Torrellas. Positional adaptation of processors: application to energy reduction. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 157–168. ACM, 2003.

[18] R. A. Johnson and D.W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, fifth edition, 2002.

[19] Martin Karlsson, Kevin E. Moore, Erik Hagersten, and David A. Wood. Memory system behavior of Java-based middleware. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 2003.

[20] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. Structures for phase classification. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, 2004.

[21] Yue Luo, Juan Rubio, Lizy Kurian John, Pattabi Seshadri, and Alex Mericas. Benchmarking internet servers on superscalar machines. *Computer*, 36(2):34–40, 2003.

[22] Matthew C. Merten, Andrew R. Trick, Ronald D. Barnes, Erik M. Nystrom, Christopher N. George, John C. Gyllenhaal, and Wen mei W. Hwu. An architectural framework for run-time optimization. *IEEE Transactions on Computers*, 50(6):567–589, 2001.

[23] R Development Core Team. *R: A language and*

environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria, 2003. ISBN 3-900051-00-3.

[24] Ramesh Radhakrishnan, N. Vijaykrishnan, Lizy Kurian John, Anand Sivasubramaniam, Juan Rubio, and Jyotsna Sabarinathan. Java runtime systems: Characterization and architectural implications. *IEEE Transactions on Computers*, 50(2):131–146, 2001.

[25] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)*, pages 45–57. ACM, 2002.

[26] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 336–349. ACM, 2003.

[27] Yefim Shuf, Mauricio J. Serrano, Manish Gupta, and Jaswinder Pal Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 194–205. ACM, 2001.

[28] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of the Third Virtual Machine Research and Technology Symposium*, pages 57–72. USENIX, 2004.