Detecting Bottlenecks in Java Applications

D. Buytaert, A. Georges, L. Eeckhout, K. De Bosschere
Department of Electronics and Information Systems (ELIS), Ghent University
St.-Pietersnieuwstraat 41, B-9000 Gent, Belgium
{dbuytaer, ageorges, leeckhou, kdb}@elis.UGent.be

***Abstract*— We present MONITORMETHOD, a tool which helps Java programmers gain insight in the behavior of their applications. MONITORMETHOD instruments the Java application and relates hardware performance monitors (HPMs) to the methods in the Java application's source code. We present a detailed case study showing that linking microprocessor-level performance characteristics to the source code is helpful for identifying performance bottlenecks and their causes. In addition, we relate our work to a previously proposed time-based HPM profiling framework.**

I. INTRODUCTION

In almost all situations, it is desirable that programs show good performance, either by reacting immediately to user manipulations, or by finishing a computation within a reasonable time. Most of the time, programmers use profilers to analyze the performance behavior of their programs. As such, profilers are essential tools for the modern programmer. However, very few existing profilers exploit the presence of hardware performance monitors (HPMs) that are typically available on modern processors. Compared to conventional profilers, such monitors can help gain deeper insight in the performance behavior of a program. Conventional profiling tools sample the execution and determine in which methods most of the execution time is spent. Such methods are then selected for optimization by the programmer. While bad performance is often due to the use of a poorly performing algorithm, it remains impossible to relate the performance to the actual microprocessor events using conventional profilers. Therefore, HPMs are interesting, because they count events, such as cache misses, branch mispredictions, etc. They may give an indication *why* an algorithm exhibits bad performance behavior.

Recently, Sweeney *et al.* [6] presented a system that logs HPM values to a trace file at each virtual context switch of the Jikes RVM [1] and developed a tool for graphically exploring these traces. More recently, we studied method-level phases in Java workloads [3] using a mechanism that traces HPM values at phase entry and exit. A *method-level phase* is defined as a subtree of the application's call graph (such phases can be hierarchical). The toolset to identify those method-level phases is called MONITORMETHOD. The major difference between Sweeney *et al.*'s work and MONITORMETHOD, is that MONITORMETHOD collects HPM values for selected method calls, whereas Sweeney *et al.* collect HPM values at each virtual context switch. As such, Sweeney's method is unable to directly relate the observed events to a subtree in the call-graph. MONITORMETHOD allows for instrumenting user-selected methods as well as automatically determined method-level program phases.

In this work, we combine and compare the work of Sweeney *et al.* with MONITORMETHOD to trace HPM values at both virtual context switches and method calls. As such, we show how the method-level phases as identified using MONITORMETHOD relate to the time-dependent behavior of program execution as measured by Sweeney *et al*. By doing so, we strengthen our statement that MONITORMETHOD is a useful toolset to identify performance bottlenecks. In addition, by linking the HPM values to the source code we are able to point to the bottlenecks' causes. Our prototype implementation of MONITORMETHOD is built within IBM's Jikes Research Virtual Machine (Jikes RVM) and is evaluated on an IA-32 platform using the SPECjvm98 and SPECjbb2000 benchmarks.

II. EXPERIMENTAL SETUP

We have used the Jikes RVM [1] for our experiments. Jikes RVM is a research virtual machine for Java applications, written (for the most part) in Java itself. It offers several compilation modes, ranging from a simple, yet efficient baseline compiler, to a fully adaptive system, in which the optimizing compiler is used to recompile hot methods. Several garbage collection (GC) schemes can be used, such as generational GC, mark-and-sweep, copying GC, etc. The machine itself can currently run on two platforms (IA-32, and 32-bit PowerPC), and is being ported to a few other platforms. The virtual machine offers architecture-independent access to the HPMs, by using the PAPI toolkit on IA-32, and the IBM PMAPI toolkit on PowerPC machines. For our experiments, we used a modified version of the CVS head from January 2004.

The Jikes RVM uses a m-to-n threading system, i.e. the Java threads are mapped to a few OS threads. Each OS thread is running a so called Virtual Processor (VP). An instance of the VM usually instantiates a single VP per hardware processor in the computer. As such, Jikes RVM takes care of its own thread scheduling and the HPM values are gathered per VP.

We have used both the SPECjvm98 and SPECjbb2000 benchmark suites, but we only present the results for the `javac` benchmark from the SPECjvm98 suite due to space constraints.

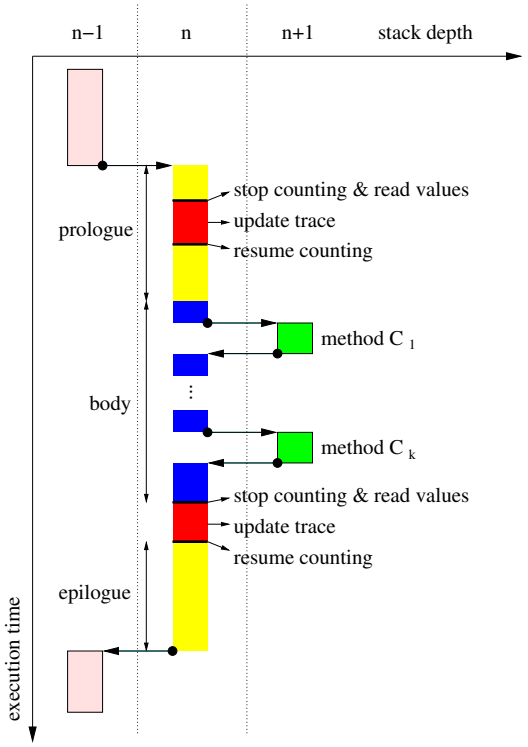


Fig. 1. Instrumenting the method calls to gather HPM values.

III. Profiling using HPMs

This section discusses two recently proposed Java application profiling mechanisms: the time-dependent behavior analysis by Sweeney *et al.* and the method-level phases in MonitorMethod.

A. Time-dependent behavior

As explained, Sweeney *et al.* use a time-based sampling mechanism in which a per-VP HPM record is captured every thread scheduler quantum. At each virtual context switch, they log the top-most method on the runtime stack. As such, the measured HPM values can be attributed to the Java threads, and to a lesser extend, to methods in the source code. However, they cannot relate the events directly to the call-graph.

B. Method-level phase behavior

MonitorMethod identifies and instruments method-level phases using the procedure discussed in [3]. We select a method as an execution phase if (i) its total execution time accross all invocations exceeds a threshold θ_{weight}, and (ii) its average execution time per call exceeds a threshold θ_{grain}. Of course, both θ_{weight} and θ_{grain} are relative to the total execution time of the program. These thresholds are chosen such that useful method-level phase behavior is obtained while keeping the instrumentation overhead of the selected methods small. As we have shown in [3], this means that our technique selects phase exhibit similar behavior within different execution instances of the same phase, and dissimilar behavior between different phases.

We distinguish threee steps in MonitorMethod. First, the execution time for each method call is measured. Second, we analyze the obtained execution time information, and identify a set of method-level phases, for a given pair of θ_{weight} and θ_{grain} thresholds. Third, once a set of method-level phases is determined, we instrument the execution to collect HPM values at entry and exit of each phase instance, see Figure 1. To this extend, we modified both the baseline and the optimizing compiler of Jikes RVM to instrument the prologue and epilogue of the selected methods. In order to combine and compare MonitorMethod with Sweeney *et al.*'s work, we instrumented the thread scheduler as well. The added instrumentation code reads the HPM values and writes them to a trace file. To minimize the incurred overhead, the latter is done through extensive buffering in C-code outside the VM.

IV. Performance analysis

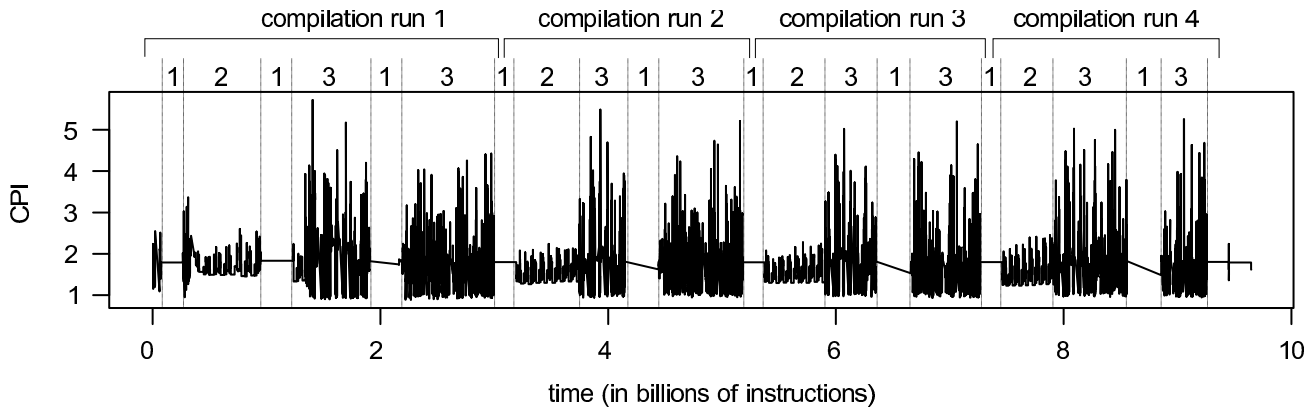
Performance analysis is done by an off-line tool that takes the final trace (obtained using the phase set) as input. The output of the analysis helps answer three fundamental questions programmers might ask when optimizing their application: (i) what are the application's bottlenecks, (ii) why do the bottlenecks occur, and (iii) when do the bottlenecks occur?

Figure 2 shows a graph that plots `_213_javac`'s cycles per instruction (CPI) over time. The vertical separators group phases in regions with similar performance characteristics. Note that `_213_javac` with the `-s100` input set compiles four times the same Java classes. Profile information captured at each virtual context switch is used to aggregate all profiling data into a single graph. To answer the first question (what is the bottleneck?), we ordered the phases by their CPI values as shown in the table of Figure 2. Methods whose CPI is worse than the average CPI, are potential bottlenecks. We have shown only the most interesting phases in the table. To answer the second question (why does the bottleneck occur?), one can investigate the corresponding metrics such as cache miss rates and the branch misprediction rate (BMP), see the table in Figure 2. Finally, to answer the last question (when does the bottleneck occur?), one can use region information to relate phases to the time behavior of an application, see also Figure 2.

Compared to the traces Sweeney *et al.* obtain, we can show more detailed information, and actually relate bottlenecks to the source code. Sweeney *et al.* can only relate bottlenecks to the moment they occur, i.e. the *when* part of the questions.

V. Related work

Dmitriev [2] 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



method / phase	region	time	CPI	L1-D	L1-I	L2-D	L2-I	BMP
SourceClass.check	3	7.06%	2.25	8.06	13.13	1.74	1.75	23.20
SourceClass.compileClass	3	26.03%	1.74	5.14	6.53	0.98	0.85	16.26
Garbage collector	1	28.99%	1.80	4.48	0.02	2.55	0.01	4.76
Parser.parseClass	2	22.68%	1.48	2.93	5.52	0.54	0.44	18.26
Benchmark average	n/a	n/a	1.67	4.28	4.48	1.32	0.66	13.26

Fig. 2. The graph and the table present some example phases in a `_213_javac -s100` run. The time is given as a percentage of the total execution time. The cache miss rates (L1-D, L1-I, L2-D and L2-I) and the BMP are given as the number of events per 1,000 instructions.

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.

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. Sherwood et al. [4], [5] 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 [4] and [5], respectively.

VI. Summary

We developed a system that bridges the gap between profilers for Java applications and HPMs by attributing performance characteristics to the source code. We compared our work with that of Sweeney *et al.* and demonstrated how it can be used to identify and analyze performance bottlenecks.

VII. Acknowledgments

Dries Buytaert is supported by a grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). Andy Georges is supported by the IWT in the CoDAMoS project, Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research—Flanders (Belgium) (FWO—Vlaanderen). This research was also funded by Ghent University.

References

[1] 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.

[2] M. 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.

[3] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in Java workloads. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*. ACM, October 2004.

[4] T. Sherwood, E. Perelman, G. Hamerly, and B. 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.

[5] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 336–349. ACM, 2003.

[6] 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 (VM'04)*. USENIX, May 2004.