

# Spotting Java Performance Bottlenecks

A. Georges, D. Buytaert, L. Eeckhout

Promoter(s): K. De Bosschere

*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.

*Keywords*—Java, Bottlenecks, Hardware Performance Monitors

## 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. 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, we studied method-level phase behavior in Java workloads [3] where we traced HPM values at phase entry and exit. We defined a phase to be a subtree in the dynamic call-tree of the application. As such, we can directly relate hardware performance behavior to the call-tree, and thus to the application’s source code. This allows us to find execution phases that show worse than average behavior and behave as potential bottlenecks causing congestion in the execution flow. We deem our tool, called MONITORMETHOD, to be quite useful in detecting and pinpointing performance bottlenecks. We have built a proof-of-concept version of our tool into IBM’s Jikes Research Virtual Machine (RVM).

## II. EXPERIMENTAL SETUP

For evaluation purposes, we have used the Jikes RVM [1] during our experiments. Jikes RVM is a research virtual machine for Java applications, written (for the most part) in Java itself. 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 due to space constraints we only present the results for the `213_javac` benchmark from the SPECjvm98 suite.

## III. PROFILING USING HPMS

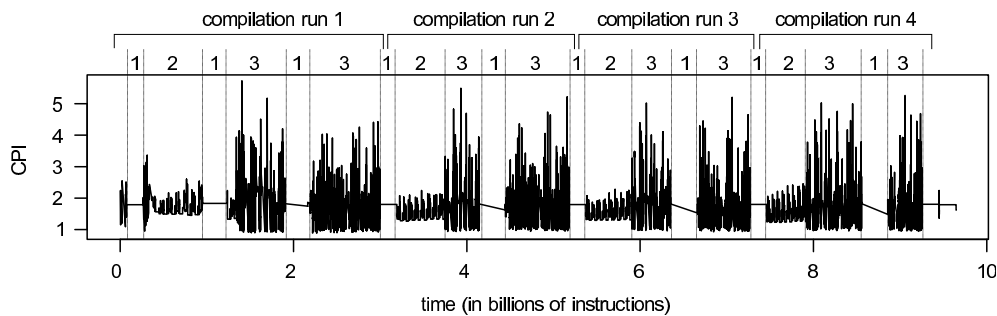
MONITORMETHOD identifies and instruments method-level phases using the procedure discussed in [3]. We select a method as an execution phase if (i) its total execution time across all invocations exceeds a threshold  $\theta_{weight}$ , and (ii) its average execution time per call exceeds a threshold  $\theta_{grain}$ . Of course, both  $\theta_{weight}$  and  $\theta_{grain}$  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 phases that exhibit similar behavior within different execution instances of the same phase, and dissimilar behavior between different phases.

We distinguish three steps in the approach used by 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  $\theta_{weight}$  and  $\theta_{grain}$  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. 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. We instrumented the thread scheduler as well, to gather information at each thread switch [5]. 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?

A. Georges, D. Buytaert, and L. Eeckhout are with the Department of Electronics and Information Systems (ELIS), Ghent University, St.-Pietersnieuwstraat 41, B-9000 Gent, Belgium, email: {ageorges, dbuytaert, leekhou, kdb}@elis.UGent.be



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. 1

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.

Figure 1 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 1. Methods whose CPI is worse than the average CPI, are potential bottlenecks. We have only shown 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 1. 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 1.

## V. RELATED WORK

Sweeney *et al.* extended the Jikes RVM to measure HPM events [5]. 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, because they only answer the *when* part of the questions raised in section IV.

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] use basic block vectors (BBVs) to identify phases.

## 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. Hamslerly, 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] 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.