Bottleneck Analysis in Java Applications using Hardware Performance Monitors

Dries Buytaert Andy Georges Lieven Eeckhout Koen 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

This poster presents MONITORMETHOD 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.

Categories and Subject Descriptors

C.4 [**Computer systems organization**]: Performance of systems

General Terms

Design, Performance

Keywords

Java, performance analysis, profiling, phase behavior

1. INTRODUCTION

Profilers are essential tools for programmers to analyze their programs' performance. Very few profilers exploit the hardware performance monitors (HPMs) that are typically available on modern processors. HPMs are interesting because they count microprocessor events such as cache misses, branch mispredictions and so on.

Sweeney *et al.* [2] presented a system that logs HPM values to a trace file at each virtual context switch and developed a tool for graphically exploring these traces. More recently, we studied method-level phases in Java workloads [1] using a similar HPM trace mechanism. A *method-level phase* is defined as a subtree of the application's call graph (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. MONITORMETHOD allows for instrumenting user-selected methods as well as automatically determined method-level program phases.

In this work, we combine and compare Sweeney *et al.*'s work 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.

2. 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.

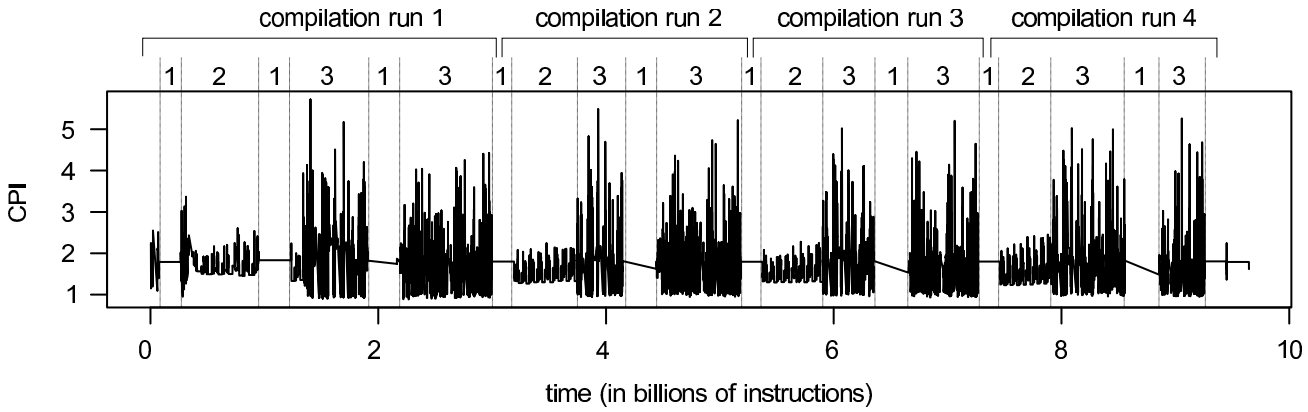
2.1 Time-dependent behavior

As explained, Sweeney *et al.* use a time-based sampling mechanism in which a per-processor HPM record is captured every thread scheduler quantum. At each 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.

2.2 Method-level phase behavior

MONITORMETHOD identifies and instruments method-level phases using the procedure discussed in [1]. For a method to represent a phase, its total execution time over all invocations must exceed a threshold θ_{weight} and its average execution time per call must exceed a threshold θ_{grain}. The thresholds are chosen such that useful method-level phase behavior is obtained while keeping the instrumentation over-

Copyright is held by the author/owner.
OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.



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

Figure 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.

head of the selected methods small. In [1], we have shown that this technique selects phases that exhibit similar behavior within a phase and dissimilar behavior between different phases.

MonitorMethod works in three steps. First, we measure the execution time for each method call. In a second step, this information is analyzed and method-level phases are identified for a given pair of θ_{weight} and θ_{grain}. Once a list of method-level phases is determined, we collect HPM values for each of them. To this extend, we modified both the baseline and the optimizing compiler 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.

3. PERFORMANCE ANALYSIS

Performance analysis is done by an off-line tool that takes the final trace 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 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. Due to space constraints the table depicts a subset of all phases only. 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.

4. 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.

5. 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.

6. REFERENCES

[1] 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 Press, October 2004.

[2] 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.