Analyzing Performance and Dynamic Behavior of Embedded Java Software with Calling-Context Cross-Profiling

Philippe Moret
Faculty of Informatics
University of Lugano
Switzerland
philippe.moret@usi.ch

Walter Binder
Faculty of Informatics
University of Lugano
Switzerland
walter.binder@usi.ch

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology
Austria
mschoebe@mail.tuwien.ac.at

Alex Villazón
Faculty of Informatics
University of Lugano
Switzerland
alex.villazon@usi.ch

Danilo Ansaloni
Faculty of Informatics
University of Lugano
Switzerland
danilo.ansaloni@usi.ch

ABSTRACT

Prevailing approaches to analyze embedded software performance either require the deployment of the software on the embedded target, which can be tedious and may be impossible in an early development phase, or rely on simulation, which can be extremely slow. We promote cross-profiling as an alternative approach, which is particularly well suited for embedded Java processors. The embedded software is profiled in any standard Java Virtual Machine in a host environment, but the generated cross-profile estimates the execution time on the target. We implemented our approach in the customizable cross-profiler CProf, which generates calling-context cross-profiles. Each calling-context stores dynamic metrics, such as the estimated CPU cycle consumption on the target. We visualize the generated calling-context cross-profiles as ring charts, where callee methods are represented in segments surrounding the caller's segment. As the size of each segment corresponds to the relative CPU consumption of the corresponding calling-context, the visualization eases the location of performance bottlenecks in embedded Java software, revealing hot methods, as well as their callers and callees, at one glance.

Keywords

Calling-context cross-profiling, Java processors, visualization of calling-context profiles

1. INTRODUCTION

Profiling of embedded Java applications is a tedious task that requires either a simulator of the embedded target platform or deployment of the application on that target. Both approaches have serious drawbacks. Simulation can be prohibitively slow. Software deployment and profiling on the target platform are time-consuming, too. Moreover, embedded Java systems often lack profiling support. In addition, because of resource constraints, some profiling techniques, such as calling-context profiling, cannot be applied on the target platform. Calling-context profiling is an important technique for locating performance problems in applications, since it yields detailed profiling data for each executed calling-context. The Calling Context Tree (CCT) [1] is a widely used datastructure for calling-context profiling, which stores dynamic metrics, such as CPU cycle consumption, for each calling-context.

In this tool demonstration, we promote cross-profiling [2, 3] for analyzing the performance of embedded Java software. The embedded software is profiled in any standard Java Virtual Machine (JVM) in a host environment, completely decoupled from the embedded target system. Nonetheless, the generated cross-profiles represent the execution time metric of the target system. The host environment is a typical machine for software development, providing sufficient resources for memory consuming profiling techniques, such as CCT construction. We present the customizable cross-profiler CProf[1] [2, 3], which generates CCTs with the number of method invocations and an estimate for the CPU cycle consumption on the embedded target for each calling-context.

As CCTs typically comprise a large number of calling-contexts, there is need for a condensed visualization that eases the location of performance problems. This tool demonstration presents a new visualization of calling-context cross-profiles as ring charts, where callee methods are represented in segments surrounding the caller's segment. In order to reveal hot methods, their callers, and callees at one glance, the visualization can size each segment according to a chosen dynamic metric.

2. CROSS-PROFILING WITH CPROF

Cross-profiling executes applications in a *host* environment and yields profiles that estimate dynamic metrics, such as CPU cycle consumption, for a different *target* environment. In the case of the JVM, both the host and the target environment execute the same instruction set, JVM bytecodes [5]. However, the JVM may be implemented in fundamentally different ways on the host and on the target. We run CProf in a typical environment for software develop-

[1] `http://www.inf.unisi.ch/projects/ferrari/`

PPPJ '09, August 27–28, 2009, Calgary, Alberta, Canada.

ment, comprising a standard PC with any state-of-the-art JVM implementation that relies on just-in-time compilation and supports dynamic optimization of executing applications. In contrast, the target environment is an embedded Java processor that implements most JVM bytecodes in hardware.

As cross-profiling target, CProf supports embedded Java systems where accurate CPU cycle estimates are available for most bytecodes and where instruction cache misses may happen only upon method invocation and return. Some recent Java processors, such as the Java Optimized Processor JOP [7], meet these requirements; JOP has a special instruction cache that caches whole method bodies.

In order to estimate the number of executed CPU cycles on the target while profiling workloads on the host, CProf relies on bytecode instrumentation techniques to intercept particular "points" in the program execution. More precisely, CProf intercepts basic block (BB) entries in the code, as well as method entry and method return. Upon BB entry, a statically pre-computed CPU cycle estimate for bytecodes in the BB is added to a counter that keeps track of the number of CPU cycles that would be consumed on the target if the workload was executed there.

Method entry and return are specially instrumented, in order to model variable CPU cycle consumption for method invocation and return bytecodes. This enables the runtime simulation of method caches, where the execution time of method call respectively return depends on whether the callee respectively the caller method is in the cache. Upon method entry, the concrete invoke opcode (invokestatic, invokespecial, invokevirtual, invokeinterface), an identifier of the callee method, and the size of the callee method (in bytes) are used for CPU cycle estimation. Conversely, on method return, the return opcode, an identifier of the caller, and the size of the caller are used.

CProf supports the customization of CPU cycle estimation and method cache simulation through pluggable components. Hence, it eases experimentation with different bytecode performance models and cache strategies, which is a prerequisite for an effective processor design space exploration, as reported in this paper.

CProf yields calling-context-sensitive profiles, estimating CPU cycle consumption for each executed calling-context. It relies on the CCT [1] to store dynamic metrics separately for each calling-context. The CCT enables a detailed analysis of program performance and helps locate hotspots in the program code.

Regarding overhead, CProf significantly outperforms the target hardware [2, 3]. It runs standard Java benchmarks, such as SPEC JVM98 [8] or DaCapo [4], without any problems, although these benchmarks cannot execute on the target. Cross-profiling is several orders of magnitude faster than VHDL simulators. For more details on CProf, we refer to [2, 3].

3. ACCURACY OF CROSS-PROFILING

To assess that our approach is sound, we compare the CPU cycle estimates from the generated cross-profiles with the actual CPU cycle consumption on an embedded Java processor and compute the percent error of the estimates. For this experiment, we chose JOP [7] as target, because timing information is public available for all bytecodes.

Our evaluation is based on 3 embedded benchmarks, Kfl, Lift, and Udplp. As JOP's execution environment is a typical embedded system without a filesystem and with only 1 MB memory, standard Java benchmarks, such as SPEC JVM98 or DaCapo, cannot be executed on the target hardware. Hence, only embedded benchmarks can be used for assessing the accuracy achieved by CProf.

Benchmark	JOP	CProf	Error (%)
Kfl	5.023×10^7	5.021×10^7	−0.04
Lift	5.282×10^7	5.262×10^7	−0.38
UdpIp	1.132×10^8	1.111×10^8	−1.81

Table 1: Benchmark execution time and cross-profiling results in clock cycles for JOP with a 4KB/16 method cache

The benchmarks Kfl and Lift are based on real-world applications [6]. The benchmark Udplp uses a TCP/IP stack, implemented in Java, in order to simulate a UDP-based client/server application. Each embedded benchmark is executed 10000 times, and in the generated cross-profile the cumulative CPU cycle estimate of the benchmark harness (the method test()) is taken, effectively excluding the execution of startup code on the host.

In this experiment, JOP is clocked at 100 MHz in a low-cost FPGA and the memory access time is 2 clock cycles. It has a 4 KB FIFO instruction cache organized in 16 blocks. The three embedded benchmarks Kfl, Lift, and Udplp are executed on JOP and the execution time is measured with a CPU cycle counter. The same benchmarks are profiled with CProf.

Table 1 shows the execution times on JOP and cross-profiling results in clock cycles. The last column shows the percent error of the cross-profiling estimates. For two benchmarks, the error is well below 1%, for Udplp the error is below 2%. The observed inaccuracies in the cross-profiles generated by CProf are due to imprecise CPU cycle estimates for certain (complex) bytecodes and differences in the Java class libraries between the host and the target. Nonetheless, for all measured benchmarks that run on the JOP hardware, the error in the CPU cycle estimates is below 2%.[2]

4. VISUALIZATION OF CALLING-CONTEXT PROFILES

CProf supports user-defined profilers to process the collected profiling data. A typical profilers dumps the calling-context cross-profile in a text file upon JVM shutdown. As a textual representation of a calling-context cross-profile can be very large and cumbersome to analyze, we provide a novel visualization tool for CProf that represents calling-context cross-profiles as ring charts, where callee methods are represented in segments surrounding the caller's segment.

Figure 1(b) shows a conceptual representation of a calling-context cross-profile for the code sample in Figure 1(a). The cross-profile, which was generated by CProf using a cache simulator and cycle estimator for the JOP processor [7], represents one invocation of method f(). Each calling-context stores the number of method invocations and the aggregated CPU cycle consumption for the CCT subtree.

Figure 1(c) presents a ring chart visualization where all calling-contexts have the same weight. For instance, the segments representing the callees of f() (i.e., g(int) and h()) have the same size and completely surround the segment of f(). This representation gives a condensed view of the overall CCT, but does not convey the dynamic metrics collected for each context.

In order to ease locating performance problems, we support a different visualization, where each segment is sized according to a chosen dynamic metric. Figure 1(d) shows the corresponding ring chart, where segments are sized according to CPU cycle estimates.

[2]In prior work [3] we reported higher errors. Thanks to improved CPU cycle estimates for some bytecodes, we were able to reduce the error.

Here, the segments representing the calles of method f() have different size and do not completely surround the segment of f(). The execution of the callee g(int) consumes about 76% of the CPU cycles consumed by the overall execution of f(), whereas the callee h() (of method f()) contributes only little to the overall cycle consumption of f(). The part of the segment representing f() that is not surrounded by callee segments represents the CPU cycle contribution of f() excluding its callees.

Our tool not only supports different visualizations according to different dynamic metrics, it also allows, amongst others, for navigation in the CCT (i.e., selection of any calling-context to be displayed as root), for limitation of the CCT depth, and for marking calling-contexts with special properties (e.g., particular package, class, or method names).

Our visualization tool comes in two implementations. The first version is based on JavaScript and on the standard XML-based Scalable Vector Graphics (SVG)[3]. It enables the developer to analyze calling-context cross-profiles in any standard web browser. The drawback of the first implementation is its high memory consumption, preventing the visualization of large profiles. The second implementation, which offers many advanced features and will be presented in this tool demonstration, is a Java Swing application. It has been optimized for displaying very large profiles comprising millions of calling-contexts.

5. CONCLUSION

In this tool demonstration, we present CProf, a configurable cross-profiler for embedded Java processors. CProf yields calling-context cross-profiles, providing dynamic metrics, such as CPU cycle consumption, separately for each calling-context. We introduce a new visualization of calling-context cross-profiles as ring charts, where each calling-context corresponds to a ring segment. In order to reveal hot methods, their callers, and callees at one glance, ring segments can be sized according to a chosen dynamic metric.

Regarding limitations, CProf currently does not represent certain system activities of the target in the generated cross-profiles. For instance, the automated memory management on the target is not being simulated in the host environment.

With respect to ongoing research, we are working on techniques to simulate activities of the target runtime system on the host. Furthermore, we are exploring the possibility of cross-profiling for embedded Java systems that rely on just-in-time compilation (in contrast to Java processors). In addition, we are enhancing our visualization tools with a domain-specific language that allows a fine-grained selection, removal, or colorization of calling-contexts according to complex conditions, including also numerical bounds computed from selected dynamic metrics.

Acknowledgment

The work presented in this paper has been supported by the Swiss National Science Foundation.

6. REFERENCES

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96. ACM Press, 1997.

[2] W. Binder, M. Schoeberl, P. Moret, and A. Villazón. Cross-profiling for embedded Java processors. In *Fifth International Conference on the Quantitative Evaluation of SysTems (QEST-2008)*, pages 287–296, Saint-Malo, France, Sept. 2008. IEEE Computer Society.

[3] W. Binder, A. Villazón, M. Schoeberl, and P. Moret. Cache-aware cross-profiling for Java processors. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES-2008)*, pages 127–136, Atlanta, Georgia, USA, Oct. 2008. ACM.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, Oct. 2006. ACM Press.

[5] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[6] M. Schoeberl. Application experiences with a real-time Java processor. In *Proceedings of the 17th IFAC World Congress*, Seoul, Korea, July 2008.

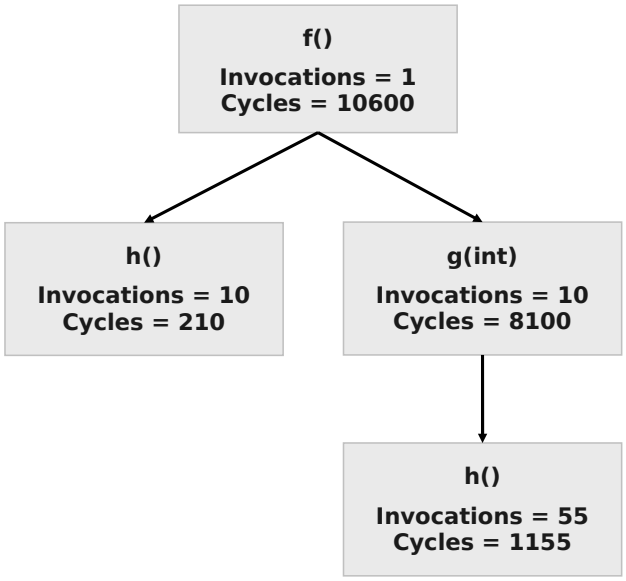
[7] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[8] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Web pages at `http://www.spec.org/osg/jvm98/`, 1998.

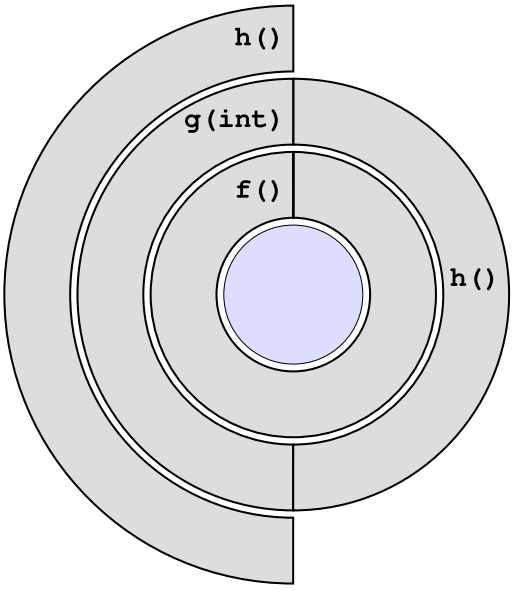
[3] `http://www.w3.org/Graphics/SVG/`

```
void f() {
   for (int i = 1; i <= 10; ++i) {
      h();
      g(i);
   }
}

void g(int i) {
   for (int j = 1; j <= i; ++j)
      h();
}

void h() {
   return;
}
```

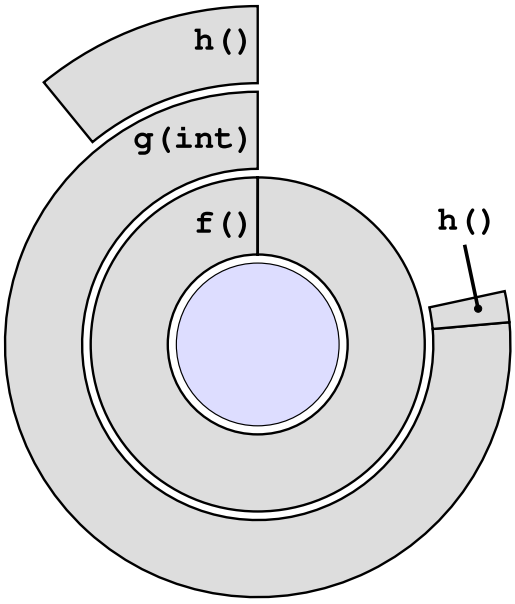
(a) Example code



(b) Generated CCT (conceptual representation)



(c) Calling-contexts with equal weight



(d) Calling-contexts weighted by estimated CPU cycle consumption

Figure 1: Example calling-context cross-profile and its visualization (assuming method f() is invoked once)