Locating Performance Bottlenecks in Embedded Java Software with Calling-Context Cross-Profiling

P. Moret W. Binder A. Villazón D. Ansaloni M. Schoeberl

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.

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. As original scientific contribution, this tool demonstration introduces 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. The Cross-Profiler CProf

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 [4], meet these requirements; JOP has a special instruction cache that caches whole method bodies.

CProf instruments the beginning of each basic block, method entry, and method return, in order to update a CPU cycle counter in the corresponding CCT node. Upon basic block entry, a statically pre-computed cycle estimate for the bytecodes in the basic block is added to the cycle counter. Method invocation and return bytecodes are treated specially, since their cycle consumption depends on the method size and on the state of the instruction cache. Upon method entry and return, CProf triggers the invocation of a user-defined cache simulator and cycle estimator. For more details on CProf, we refer to [2, 3].

3. 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-

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

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 [4], 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. 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).

4. 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.

This tool demonstration is accompanied by a methodological paper introducing cross-profiling for processor architecture design space exploration.

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI 1997*, 85–96.

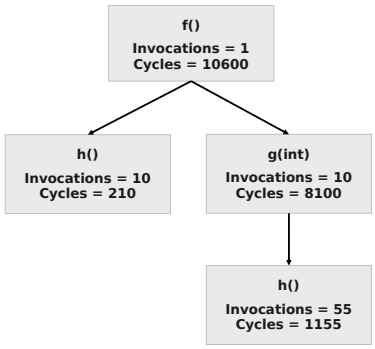
[2] W. Binder, M. Schoeberl, P. Moret, and A. Villazón. Cross-profiling for embedded Java processors. In *QEST 2008*, 287–296.

[3] W. Binder, A. Villazón, M. Schoeberl, and P. Moret. Cache-aware cross-profiling for Java processors. In *CASES 2008*, 127–136.

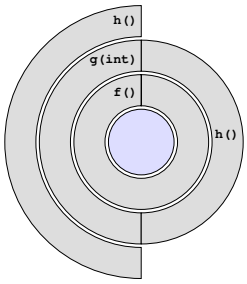
[4] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.


```
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; }
```

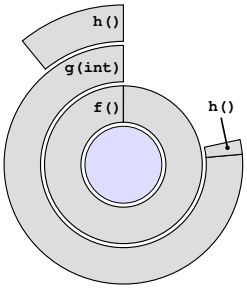
(a) Example code



(b) Generated CCT (conceptual representation)



(c) CCT visualization: calling-contexts with equal weight



(d) CCT visualization: calling-contexts weighted by estimated CPU cycle consumption

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