Cross-Profiling for Embedded Java Processors

Walter Binder
University of Lugano
Switzerland
walter.binder@unisi.ch

Martin Schoeberl
TU Vienna
Austria
mschoebe@mail.tuwien.ac.at

Philippe Moret Alex Villazón
University of Lugano
Switzerland
philippe.moret@lu.unisi.ch
alex.villazon@lu.unisi.ch

Abstract

Profiling is essential for finding execution time hot spots in applications. However, in embedded systems resources are usually scarce and profiling is not an option, although the detection and optimization of hot spots is particularly important in such resource-constrained systems. In this paper we propose cross-profiling for embedded systems equipped with a Java processor; the cross-profiles are collected in any standard Java environment, but represent the execution time metrics of the embedded target platform. We present a novel cross-profiler that relies on Java bytecode instrumentation and generates calling-context-sensitive cross-profiles with CPU cycle estimations for each calling context. Our cross-profiler reconciles platform-independence, portability, compatibility with standard Java runtime systems, complete bytecode coverage, moderate profiling overhead, and high accuracy of the generated cross-profiles.

Keywords: *Cross-profiling, embedded Java processors, bytecode instrumentation, platform-independent dynamic metrics*

1. Introduction

Java processors, such as the aJile processor [10] or JOP [20], have become an attractive choice for building embedded and real-time systems that are programmed in Java. Java processors implement the Java Virtual Machine (JVM) in hardware; their instruction sets correspond to the JVM bytecodes.

Concomitant performance evaluation is crucial for the development of embedded software that has to run on devices with limited computing resources. Furthermore, performance prediction based on existing benchmarks is essential for determining the effect of hardware optimizations before they are implemented.

However, today, profiling of embedded Java applications is a tedious and time-consuming task, requiring either a simulator of the target platform or deployment of the embedded application. Simulators can be extremely slow; e.g., the simulator ModelSim [13] causes excessive overhead of up to factor 33000. Hence, embedded Java applications are rarely profiled in an early phase of development. Moreover, there is little support for simulating the effects of new hardware optimizations or for predicting the performance of large-scale applications.

While the Java 2 Platform Standard Edition offers a dedicated profiling interface, the JVM Tool Interface (JVMTI), embedded Java systems often lack profiling support. Because of the resource constraints on embedded Java systems, CPU time and memory consuming profiling techniques are usually not applicable. For instance, calling-context-sensitive profiling [1] is an important technique for detecting execution time hot spots, but the generated data structure, the Calling Context Tree (CCT), can consume significantly more memory than the application being profiled itself.

To overcome these limitations, we introduce a novel technique for *platform-independent cross-profiling*. The cross-profiler is run in any standard Java environment (independently of hardware, operating system, and virtual machine), but generates profiles using the execution time metric of the embedded system. We call the environment running the cross-profiler the *host*, and we refer to the embedded system for which the profiles are collected as the *target*. The cross-profiler is not connected in any way to the target system.

The scientific contributions of this paper are our portable cross-profiling techniques that yield calling-context-sensitive profiles with CPU cycle estimations for an embedded target system, as well as our implementation of these techniques in the Java cross-profiler CProf, which targets the Java processor JOP [20]. Our cross-profiler is based on a new instrumentation mechanism that ensures full bytecode coverage, including the Java runtime system as well as any dynamically loaded bytecode. CProf is implemented in pure Java and compatible with standard Java runtime systems (JDK 1.5 or higher), and it supports customized dynamic processing of generated profiling data. Evaluation results confirm that our approach yields sufficiently accurate CPU cyle estimations and causes only moderate overhead.

The remainder of this paper is structured as follows: Section 2 gives an overview of Java processors and presents some details of the JOP processor. Section 3 explains our approach to platform-independent cross-profiling which is based on Java bytecode instrumentation. Section 4 discusses some of the challenges we encountered in the implementation. In Section 5 we evaluate the accuracy of the generated cross-profiles for some embedded Java benchmarks, compare our cross-profiler with simulators, and assess the overhead due to cross-profiling with several standard benchmark suites. Section 7 discusses the benefits and limitations of our approach and outlines some ideas for future work. Finally, Section 8 concludes this paper.

2. Embedded Java Processors as Cross-Profiling Target

In this Section we give an overview of Java processors as target systems for cross-profiling. Then, we describe the JOP processor that we have chosen to validate and evaluate our approach.

2.1. Java Processors

A Java processor is an implementation of the JVM in hardware. Java bytecode is therefore the native instruction set for the processor. The first Java processor, picoJava [14], was developed by Sun Microsystems. However, this processor was not a commercial success. The design model sources of picoJava are now open-source and are used as basis for improving various aspects of Java processors. A follow-up redesign, known as picoJava-II, is now freely available.

Java processors are mainly used for embedded systems where resources (memory and processing power) are limited. Unfortunately, profiling applications in such limited environments is difficult, if not impossible. Java processors are therefore ideal target systems for cross-profiling.

Several companies have built Java processors in an Application Specific Integrated Circuit (ASIC). The most successful one is the aJile processor [10] that was initially conceived as a platform for the Java real-time specification [4].

At the time when the Field Programmable Gate Array (FPGA) technology became large enough for the implementation of a processor, many projects in academia and industry started to use this technology for Java processors. An implementation of picoJava-II on a medium sized FPGA is described in [16, 17]. Komodo [11] is a multithreaded Java processor intended as a basis for research on real-time scheduling on a multithreaded microcontroller.

2.2. The Java Processor JOP

JOP [20] started as a research project for real-time Java. The main features of JOP are: (1) well known execution time of Java bytecodes; (2) good execution performance for embedded Java programs; (3) a design that can be implemented in small low-cost FPGAs. JOP is also in use in several commercial applications [19].

The main focus of the development of JOP is time-predictable execution of Java bytecodes. This feature simplifies the low-level part of worst-case execution time (WCET) analysis, a mandatory analysis for hard real-time systems. It also simplifies the timing model provided for cross-profiling.

JOP dynamically translates the CISC Java bytecodes to a RISC, stack based instruction set (the microcode) that can be executed in a 3-stage pipeline. The translation takes exactly one cycle per bytecode and is therefore pipelined (adding a fourth pipeline stage).

JOP contains a simple execution stage with the two topmost stack elements as discrete registers. No write back stage or forwarding logic is needed. The short pipeline (4 stages) results in short conditional branch delays and therefore helps avoiding any hard-to-analyze branch prediction logic or branch target buffer.

All microcode instructions have a constant execution time of one cycle. No stalls are possible in the microcode pipeline. Loads and stores of object fields are handled explicitly. The absence of time dependencies between bytecodes results in a simple processor model for the low-level WCET analysis [21, 18].

JOP has an optional instruction cache, the method cache, which caches whole methods. Only invoke and return instructions can result in a cache miss. All other instructions are guaranteed cache hits. In this paper, we do not consider the presence of a method cache.

JOP also contains a time predictable data cache for local variables and the operand stack. Access to local variables is a guaranteed hit and no pipeline stall can happen. Stack cache fill and spill is under microcode control and analyzable.

JOP has a good average case performance compared to other non real-time Java processors. Avoidance of hard-to-analyze architectural features results in a very small design.

3. Platform-independent Cross-profiling

In order to profile the CPU cycle consumption on the embedded target system, we firstly focus on techniques for collecting profiles on the host system using *platform-independent metrics* [6], and secondly explain a mechanism to map these platform-independent metrics to CPU cycle estimations on the target system. This approach ensures a maximum of profile independence from the host machine such that software developers may employ their preferred Java development environment for cross-profiling.

3.1. Collecting Platform-independent Profiles

Our profiling techniques are based on *bytecode instrumentation*. Every method that has a corresponding bytecode representation, including JDK methods as well as methods in dynamically loaded classes, is instrumented so as to generate profiling data at runtime.

Our profiling approach generates a *Calling Context Tree* (CCT) [1] for each thread [2]. While the depth of a CCT can be restricted, our CCTs are unlimited in depth; this is appropriate, since very deep call stacks (such as in recursions) are rare in embedded system software. We introduce an extra method[1] argument that represents the caller's context. Upon method entry, the callee looks up or creates its own calling context as a child node of the caller's context in the CCT. A calling context object uniquely identifies the callee method and stores the profiling data for the respective calling context. For each Java method, our instrumentation inserts a static final field to hold a method identifier object that stores the class name, method name, and method signature of the corresponding method. Since each thread builds its own CCT, there is no need for synchronizing each update of a node in the CCT.

Each CCT node stores the dynamic metrics to be measured. We focus on platform-independent metrics in order to avoid any dependencies on the specific platform used for profiling. Concretely, we keep track of the number of *method invocations* and the number of *executed bytecodes*.

While the method invocation counter is simply incremented upon method entry, the bytecode counter needs to be continuously increased as the bytecodes are being executed. Hence, we instrument all Java methods to augment the bytecode counter in the beginning of each basic block of code (BBC) with the number of bytecodes in the BBC. This approach is accurate if the execution of the first bytecode in a BBC implies the execution of all bytecodes in it. Exceptions can cause inaccuracies, if they are thrown by a bytecode that is not the last one in its BBC. In that case, we would count more bytecodes than are actually being executed. To remove this potential source of imprecision, our instrumentation algorithm offers an option to force all exception-throwing bytecodes to end a BBC. Consequently, the resulting BBCs tend to be very short and incrementing a bytecode counter in each BBC then causes higher overhead.

[1]In this paper, "method" stands for "method or constructor".

In order to regularly process the collected profiling data in a customizable way, our instrumentation ensures that each thread periodically invokes a user-defined profiler, passing the thread's CCT. A typical profiler may aggregate (in a synchronized way) the CCTs of all threads in a "global" CCT representing the activities of all threads. The invocation of the custom profiler is triggered when a dedicated, thread-local bytecode counter (which is not associated with any calling context) reaches a given threshold. Our instrumentation inserts code for incrementing and checking that counter in strategic locations in the program (method entry/return, begin of loop, etc.) so as to limit the number of bytecodes that can be executed between subsequent checks by a thread. This approach ensures the periodic activation of the profiler by each executing thread, independently of the underlying JVM scheduling.

3.2. Estimating CPU Consumption on the Target System

Our approach to cross-profiling for embedded Java processors takes advantage of the fact that on such systems, many bytecodes consume a well specified, constant number of cycles. Hence, we replace the bytecode counter in each calling context with a *cycle counter*, which we update in the beginning of each BBC with the sum of the estimated cycle consumption for the bytecodes in the BBC.

However, depending on the Java processor, certain bytecodes need special consideration, as their cycle consumption depends on concrete runtime parameters. For instance, in the case of the JOP processor, some bytecodes (e.g., runtime type checks and casts, object and array allocation, floating point arithmetic, division, etc.) are substituted by invocations of routines. Our cross-profiler currently estimates the cycle consumption of these bytecodes as constants. Despite of these simplifications and approximations, we will show in Section 5.2 that our cross-profiles are sufficiently accurate in practice.

Method invocation and return bytecodes require special treatment. Their cycle consumption may vary largely depending on the size of the callee respectively caller method. Because method invocation and return bytecodes are executed very frequently in typical object-oriented applications, any inaccuracy in the cycle calculation for these bytecodes must be avoided.

On the JOP processor, the cycles consumed by a method invocation depend on the size of the callee method (in bytes) and on the type of invocation (`invokestatic` for static methods; `invokespecial` for constructors, private methods, or methods in a superclass; `invokeinterface` for invocations on a variable where the declared type is an interface; `invokevirtual` for all other method calls). Similarly, the cycle consumption for method return depends on the size of the caller and on the return type. We store the method size within the method identifier objects (which are created by the instrumented static initializer in each class).

For JOP, the cycle estimates for invoke bytecodes are computed as follows (f is the same function for all four invoke bytecodes, while $\mathscr{K}_{static}$, $\mathscr{K}_{special}$, $\mathscr{K}_{interface}$, and $\mathscr{K}_{virtual}$ are four different constants).

cycles(invokestatic, $calleesize$) = $\mathscr{K}_{static} + f(calleesize)$
cycles(invokespecial, $calleesize$) = $\mathscr{K}_{special} + f(calleesize)$
cycles(invokeinterface, $calleesize$) = $\mathscr{K}_{interface} + f(calleesize)$
cycles(invokevirtual, $calleesize$) = $\mathscr{K}_{virtual} + f(calleesize)$

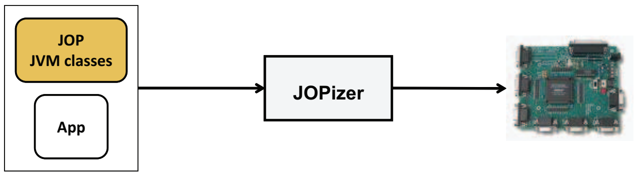
Because of polymorphic call sites, the callee of an invocation is not always statically known, and therefore $f(calleesize)$ cannot be computed at instrumentation time. Since computing $f(calleesize)$ at runtime on each method entry would cause high overhead, we follow a different approach. Upon instrumentation, we consider only the constant part ($\mathscr{K}_{static}$, $\mathscr{K}_{special}$, $\mathscr{K}_{interface}$, respectively $\mathscr{K}_{virtual}$) for calculating the cycle estimate of an invoke bytecode. In order to compensate for this loss of information, we use a dedicated profiler that computes the missing cycles before emitting the final profile upon program termination. This is possible because the CCT preserves the number of method calls for each calling context.

While traversing the CCT, the profiler regards pairs of caller and callee contexts. Each context contains a method identifier which provides, amongst others, information on the method size. The callee context stores the number of invocations by the corresponding caller ($\#calls$). Hence, the cycle counter in the caller context is increased by $\#calls * f(calleesize)$.

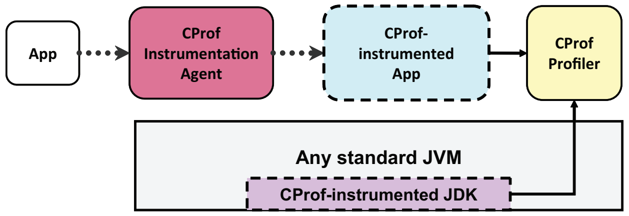
For return bytecodes, we follow a similar approach. Upon instrumentation, we consider the cycle estimate of all return bytecodes to be zero. The profiler then augments the cycle counter in the callee context by $\#calls * cycles(opcode, callersize)$, where $opcode$ denotes the return bytecode, which is uniquely determined by the method signature of the callee.

Note that for method invocation, the profiler generally could not uniquely reconstruct the concrete invoke bytecode, since there are cases where the same method may be invoked by `invokevirtual`, `invokeinterface`, or `invokespecial`. In contrast to return bytecodes, our solution exploits detailed knowledge of the JOP cycle estimation function for invoke bytecodes.

Our treatment of method invocation and return bytecodes is a simple solution that causes no extra runtime overhead. However, it has three limitations: (1) it disregards abnormal method completion (by throwing an exception), (2) it does not consider the impact of a method cache, and (3) the assumption that the differences in cycle consumption between the invoke bytecodes are constant may not hold for all Java processors. Fortunately, for the JOP processor, these limitations are not stringent; JOP currently does not support exception handling and the method cache is optional. Therefore, if the method cache remains disabled, our approach guarantees exact cycle counts for method call and return. Nonetheless, as outlined in Section 7, we are working on an extension for accurate method cache simulation that requires profiling method invocation and return separately at runtime; that approach will also handle abnormal method completion.



(a) Running application on the target processor JOP



(b) Cross-profiling in the host environment

Figure 1. Executing an embedded application on the target system versus instrumentation-based cross-profiling on any standard Java runtime system.

4. Implementation Techniques

In the following we briefly discuss some of the challenges that we solved in our CProf implementation. In particular, we explain how we achieve full bytecode coverage of the instrumentation and how we ensure that the instrumented code works together with native code of the host JVM, which is not aware of the instrumentation.

4.1. Complete Bytecode Coverage

Figure 1 compares the execution of an embedded application on the JOP target system with our cross-profiling approach explained in the previous section. Details on application deployment for JOP (i.e., the JOPizer in Figure 1(a)) are not in the scope of this paper and can be found in [20].

Our instrumentation approach aims at complete bytecode coverage, i.e., every method that has a corresponding bytecode representation on the host JVM shall get instrumented. This requires that both the JDK classes as well as any dynamically loaded bytecode is instrumented. The latter requirement implies that instrumentation also happens at load-time, as some applications may dynamically generate classes at runtime (though this is not the case for typical embedded software). Load-time instrumentation has another benefit; it ensures that all loaded classes are instrumented in the same way and therefore helps avoiding a tedious, error-prone static instrumentation process that would be necessary after each compilation of the embedded application.

Following a fully portable approach, we avoid using the JVMTI because it requires writing platform-dependent, native code. Instead, we rely on the package `java.lang.instrument` that was introduced in JDK 1.5 and enables load-time instrumentation by a user-defined instrumentation agent written in Java. However, when the instrumentation agent starts execution, the JVM has completed bootstrapping and many classes have been loaded and linked. While these previously loaded classes can be redefined and replaced with instrumented versions by the agent, the `java.lang.instrument` API imposes strong restrictions on class redefinition. E.g., fields or methods cannot be added, method signatures cannot be modified, etc. Since our instrumentation requires the modification of method signatures (adding an extra argument to represent the reified calling context) and the insertion of static fields, and because we want to instrument all classes in a uniform way, class redefinition is not suited for our purpose.

We resort to a combination of static instrumentation, applied only once to the JDK classes used on the host, and load-time instrumentation for all other classes. In Figure 1(b), "CProf-instrumented JDK" refers to the statically instrumented JDK class library and "CProf Instrumentation Agent" denotes the agent that instruments all application classes at load-time. "CProf Profiler" is our cross-profiler that processes the calling-context-sensitive profiling data produced by each thread. The profiler is invoked by the instrumented code, both in JDK and in application classes. "CProf Profiler" employs a JVM shutdown hook to emit the cross-profile upon application termination [2].

Bootstrapping with an instrumented JDK, as well as load-time instrumentation within the same JVM process that runs the instrumented application are difficult problems [3]. We solved them by introducing "code-bypasses" in each Java method that allow resorting to the original method body. These code-bypasses, which can be controlled separately for each thread, are activated for bootstrapping, for load-time instrumentation, and for the execution of the profiler. Consequently, the JVM bootstrapping phase is not disrupted, and load-time instrumentation respectively profiler execution do not create artifacts in the generated profiles.

4.2. Compatibility with Native Code

Our instrumentation introduces an extra method argument to represent the reified calling context. The extra argument is passed from the caller to the callee. In order to ensure compatibility with Java method invocations by reflection or by native code (both kinds of callers cannot be changed by automated bytecode instrumentation techniques), we create "nativecode-to-bytecode wrapper" methods with the original signature that obtain the root node of the current thread's CCT (from a thread-local variable) and pass it to the corresponding instrumented method. Conversely, for each native method, we introduce a "bytecode-to-nativecode wrapper" method that takes the extra argument and simply invokes the corresponding native method without that argument. Bytecode-to-nativecode wrappers facilitate native method invocation by instrumented Java code.

A consequence of our treatment of native code issues is that the reified calling context is lost upon native method invocation. If native code calls back into Java code (through JNI, the Java Native Interface), the callee appears as a child of the root node in the thread's CCT. I.e., all callees of native code appear as siblings in the CCT. However, we found that this issue is not a major problem in practice; callbacks from native code to Java code are not frequent in the profiles we analyzed. Furthermore, this possible inaccuracy regarding the reified calling context does not affect the CPU cycle estimations. Although this limitation is of minor relevance in practice, we are considering techniques to preserve the reified calling context during native code execution as future work.

5. Evaluation

In this Section we evaluate our cross-profiling approach. Firstly, we describe the benchmarks for embedded Java systems we used. Secondly, we assess the accuracy of CProf's CPU cycle estimations obtained in the host environment, comparing the total cycle estimates in the cross-profiles with the actual CPU cycle consumption on the target processor JOP. Thirdly, we compare the benchmark execution time using CProf with two different simulators run on the same host environment. Fourthly, we evaluate the overhead of CProf on the host environment with additional standard benchmarks which cannot be executed on the embedded Java processor.

5.1. Embedded System Benchmark

We evaluate our cross-profiling solution with three benchmarks from the embedded benchmark suite JavaBenchEmbedded[2] (JBE). The original benchmark framework increases the iteration count for the benchmark exponentially until the program runs for at least one second. This mechanism adapts the benchmark iteration count to the performance of the target system and allows to benchmark very different platforms in about the same amount of time. However, this approach does not work for evaluating the cross-profiling accuracy, as we would measure different iteration counts on the two platforms. Therefore, we run each application benchmark with a constant iteration count of 10000.

The embedded benchmark suite contains several micro benchmarks and three real world applications. We are focusing here on the application benchmarks. The first application, *Kfl*, is taken from one of the nodes of a distributed motor control system. A simulation of both the environment (sensors and actors) and the communication system (commands from the master station) forms part of the benchmark, so as to simulate the real-world workload. The second application benchmark, *UdpIp*, is an adaptation of a tiny TCP/IP stack for embedded Java. This benchmark contains two UDP server/clients, exchanging messages via a loopback device. The third application benchmark, *Lift*, is a lift controller in an automation factory.

5.2. Accuracy of CProf Cycle Estimations

For a comparison of the accuracy of the proposed cross-profiling techniques, we ran each benchmark on the real hardware and used a clock cycle counter to measure the execution time on the target. Exactly the same program (compiled with the same Java compiler) was then run on the host environment with CProf.

Table 1 shows the execution time in clock cycles for JOP, as well as the cycle estimation obtained with CProf. Because we cannot obtain calling-context-sensitive CPU cycle consumption data from JOP, we summed up the cycle estimations of all calling contexts in the cross-profile and computed the percent error of the resulting total cycle estimate and the real cycle consumption on JOP.

The profiling estimation is quite close to the real execution time. The maximum error observed is below 5%. This accuracy is good enough for the purpose of cross-profiling: finding hot spots in the application code and getting information where the execution time is spent in the application.

The reasons for inaccuracies are (1) differences in the Java class libraries used on JOP respectively in the cross-profiling environment, and (2) a simplified execution time model for some bytecodes that are not implemented in microcode, but provided as Java methods. CProf currently assumes a constant execution time of 200 cycles for all of these bytecodes.

JOP also contains a special form of instruction cache, the method cache, where whole methods are cached. As we have not yet implemented the simulation of the cache in CProf, the former numbers in Table 1 are given with JOP configured with a cache for only a single method. Effectively, this corresponds to disabling the method cache.

The execution time for a configuration of JOP with a 4 KB method cache organized in 16 blocks is given in Table 2. We can see the influence of this performance enhancing feature on the execution time and the resulting overestimation of the cross-profiler. We consider implementation of the cache simulation as future work.

Benchmark	JOP	CProf	Error
Kfl	71.1×10^6	72.9×10^6	2.5%
UdpIp	137.8×10^6	139.0×10^6	0.8%
Lift	59.9×10^6	62.8×10^6	4.8%

Table 1. JOP cycle consumption versus CProf cycle estimates; JOP method cache disabled.

Benchmark	JOP	CProf	Error
Kfl	54.5×10^6	72.9×10^6	33.8%
UdpIp	117.0×10^6	139.0×10^6	18.8%
Lift	53.3×10^6	62.8×10^6	17.8%

Table 2. JOP cycle consumption versus CProf cycle estimates; JOP method cache enabled.

5.3. CProf versus Simulators

We compared the execution time of the benchmarks running on the target processor at 100 MHz (method cache disabled as in Table 1) with CProf and with two simulators. The first simulator, JopSim[3] (version 1.38), is a high-level simulation for JOP written in Java and not optimized for speed. JopSim is an interpreting JVM with simulation of JOP internal hardware (e.g., timer interrupts and I/O devices). The second simulator is the VHDL simulator ModelSim [13] (ModelSim SE 6.1e) that performs the whole processor simulation and is used to debug the hardware. CProf and the simulators were executed on an Intel Core2 Duo at 2.2 GHz running Windows XP.

Table 3 shows the wall clock execution time of the three benchmarks in different execution environments. The sec-

[2]`http://www.jopwiki.com/JavaBenchEmbedded`.

[3]JopSim is part of the source distribution of JOP: `http://www.jopdesign.com/simulation.jsp`

Benchmark	JOP	CProf	JopSim	ModelSim
Kfl	711 ms	14 ms	1360 ms	6h25'
UdpIp	1378 ms	53 ms	2719 ms	12h30'
Lift	599 ms	15 ms	1422 ms	5h30'

Table 3. Benchmark execution time in different environments; JOP method cache disabled.

Benchmark	JOP	JopSim	Error
Kfl	71.1×10^6	63.4×10^6	-11%
UdpIp	137.8×10^6	162.3×10^6	18%
Lift	59.9×10^6	120.9×10^6	102%

Table 4. JOP cycle consumption versus JopSim cycle estimates; JOP method cache disabled.

ond column shows the execution time on the embedded target. The time is, as expected, the same as given by the cycle count measurement (Table 1). To provide useful measurements with cross-profiling, we increased the problem size to 100000 iterations and averaged three benchmark runs. The third column shows the execution time results scaled back to 10000 iterations. As we can see from the execution times, cross-profiling on a fast host is an attractive, efficient option for performance estimation. Cross-profiling in a state-of-the-art host environment is about 25 to 50 times faster than executing the benchmarks in the target environment.

The high-level simulation JopSim (fourth column) executes by a factor of two slower on the 2.2 GHz machine than the Java processor at 100 MHz. However, JopSim is a simple, high-level simulator intended to debug JOP related functions and has not been optimized for speed. JopSim also estimates the target execution cycles as shown in Table 4. However, the estimates are less accurate than those given by our new cross-profiler.

The VHDL simulation ModelSim (Table 3, fifth column) has been measured with a smaller problem size (1000 iterations) and scaled to 10000 iterations. VHDL simulation of the processor is a very time consuming approach. It is not very practical to estimate performance on the target system even for such small scale applications. One millisecond execution time on the target at 100 MHz needs about 32 seconds simulation time on a 2.2 GHz PC. However, VHDL simulation gives the greatest simulation details.

5.4. CProf Overhead

We evaluated the performance overhead introduced by the bytecode instrumentation and by the additional code inserted for cross-profiling. In this evaluation we only consider the overhead on the host environment where the cross-profiling is performed. For this purpose, we ran the JBE benchmarks with the same fixed number of iterations (10000) as in the previous experiments. In addition, since the goal of this evaluation is not measuring execution time in the target environment, but assessing the cross-profiling overhead in the host environment, we also included the standard SPEC JVM98[4] and DaCapo[5] benchmark suites, which provide much larger workloads. JVM98 consists of 7 benchmarks and DaCapo has 11 benchmarks. We ran the JVM98 suite with a problem size of 100 and DaCapo, version 'dacapo-2006-10-MR2', with its default workload size.

Our measurement platform for this overhead evaluation is a Linux Fedora Core 2 computer (Intel Pentium 4, 2.66 GHz, 1024 MB RAM). All the benchmarks were run in single-user mode (no networking) and we removed background processes as much as possible in order to obtain reproducible results. The test platform for this evaluation is therefore different from the one used in previous evaluations. The metric used for Dacapo and JVM98 is the execution time in seconds, whereas JBE gives the execution time in milliseconds. We present only the overhead factor between the execution time of the original unmodified benchmarks and the instrumented benchmarks running on an instrumented JDK. The benchmarks were instrumented dynamically whereas the JDK libraries were instrumented statically. We present measurements made with Sun JDK 1.7.0-ea-b21 using both the Hotspot Client VM ('client') and the HotSpot Server VM ('server').

Figure 2 shows the overhead factor as the median of 15 runs made within the same JVM process to attenuate the impact of load-time instrumentation. The overhead is of factor 1.72–13.86 in 'client' mode, respectively 1.33–9.99 in 'server' mode. For every benchmark suite, we also calculated the geometric mean giving an overhead factor of 4.16–5.71 for 'client' mode and of 3.27–4.29 for 'server' mode. We experienced the highest overheads with the 'udpip' benchmark of JBE and 'mtrt' of JVM98. For the 'updip' benchmark in 'client' mode, we observed that in the instrumented version, the percentage of interpreted methods (as opposed to the percentage of methods compiled by the HotSpot Client just-in-time compiler) significantly increased. The 'mtrt' benchmark is known to make extensive use of small methods [9], which explains the observed higher overhead. In general, we can see that the HotSpot Server just-in-time compiler is more effective in reducing the overhead due to our instrumentation. Overall, the overhead is moderate considering that profiling covers all bytecodes in application and JDK classes.

[4]`http://www.spec.org/osg/jvm98/`
[5]`http://www.dacapo-bench.org/`

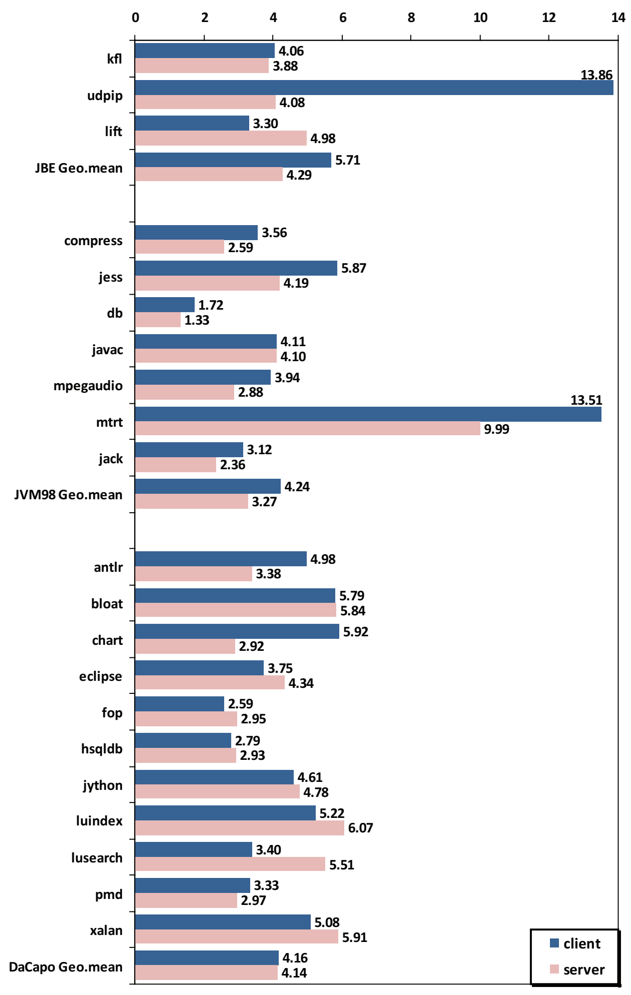


Figure 2. Cross-profiling overhead (slowdown factor) on Sun JDK 1.7.0, 'client' and 'server' mode.

6. Related Work

Cross-profiling has been used to simulate parallel systems [5, 8]. Since it is not always possible to use a host processor that has the same instruction set as the target processor, cross-profiling attempts to match up the BBCs on the host and the target machines and changes the estimation of the BBCs on the host machine to reflect the estimates on the simulated target machine. Our approach follows a similar principle, but using precise cycle estimations at the instruction-level, because both the target and the host instructions are JVM bytecodes.

A variety of dynamic metrics have been proposed for profiling, including bytecode metrics [6]. In [7], the *J tool is presented for the metrics computation. *J relies on the Java Virtual Machine Profiler Interface (JVMPI)[6], which is known to cause very high overhead and requires native code [12]. Our bytecode metrics computation minimizes the overhead and is implemented in pure Java.

Profiling JavaME applications is difficult because of the use of emulators, the lack of cross-profiling tools, and the limited resources and profiling support on these devices. ProSyst's JProfiler [15] uses a profiling agent running directly on the target device. The agent communicates through the network with the profiling front-end running on the Eclipse's IDE. Even though this approach enables accurate profiling, the agent is implemented in native code using the JVMPI, and therefore is limited to a reduced number of virtual machines and operating systems. In addition, the agent itself consumes resources on the target system which may have an impact on performance and may perturbate the measurements. Another drawback of this approach is that profiling requires deployment of the application on the target system, which is tedious and time-consuming. In contrast, our approach does not require deployment of the application. Cross-profiling is done totally independently of the target environment using portable, standard, and state-of-the-art Java technology and tools on the host environment. This allows profiling of applications in an early phase of development, without the overhead of application deployment and execution in the target environment.

7. Discussion and Future Work

In the following we discuss the strengths and limitations of our approach and outline our ongoing research on cross-profiling.

The main goals of our work are: (1) to provide a platform-independent cross-profiler for an embedded Java processor, (2) to ensure portability of our profiler and compatibility with standard JVMs, (3) to guarantee complete bytecode coverage in the cross-profiles, (4) to keep the cross-profiling overhead moderate such that also complex applications can be profiled, and (5) to achieve high accuracy in our CPU cycle estimations.

Our cross-profiler meets the first and second goal to a large extent, because it is implemented in pure Java and compatible with any standard JVM (JDK 1.5 or higher, since our cross-profiler relies on the `java.lang.instrument` API). For deterministic applications, cross-profiles are exactly reproducible, as long as the same Java class library is used. However, in practice many applications involve some non-determinism. For instance, if algorithms make use of the identity hashcodes of objects (e.g., hashtable operations), they may follow different execution paths if the hashcodes vary between different

[6]The JVMPI has been deprecated in Java 1.5 and has been replaced with the JVMTI.

runs of the program. Concurrency and the unknown thread scheduling contribute to the observed non-determinism, too. Therefore, multithreaded applications may be scheduled in a different way on the host platform used for cross-profiling respectively on the embedded target system.

Our approach to bytecode instrumentation ensures full coverage of any code that has a corresponding bytecode representation; hence, it fully meets the third goal. We are able to profile the execution of any Java method in the JDK (even in `java.lang.Object`) as well as any dynamically loaded or generated code.

Concerning our fourth goal, we have shown in Section 5.4 that even large benchmarks, such as the DaCapo suite, can be profiled within reasonable time. Although we inject instrumentation code for calling context reification, for cycle counting, and for periodically triggering a custom profiling agent, the experienced overhead remains moderate when compared with a JVMTI-based profiler that tracks each method call. In past work we evaluated the profiling agent "hprof" (which is part of many standard JDK releases and relies on the JVMTI) in its exact profiling mode and showed that it causes overhead up to factor 4000 [2]. The reason for such excessive overhead is that certain events signaled by the JVMTI, such as method entry or exit, prevent just-in-time compilation. In contrast, our instrumentation is completely transparent to the JVM, and the inserted instrumentation code gets optimized by the just-in-time compiler just as any application code. Because our approach treats the JVM as a "blackbox", we can always resort to the most recent JDK release with a state-of-the-art compiler for running our cross-profiler.

In Section 5.2 we have shown that for all measured embedded applications, the CPU cycle estimations in our cross-profiles are quite accurate with an error below 5%. Thus, we think that we have achieved the fifth goal to a large extend.

While inaccuracies due to application-inherent non-determinism cannot be avoided, there are still three sources of inaccuracy in our cross-profiles that we want to eliminate in our ongoing research: (a) differences in the Java class library, (b) simplistic cycle estimations for bytecodes that are substituted by routines in the embedded system, and (c) a simplified model of JOP's method cache.

The Java class library available on the embedded system is often different from the version in the cross-profiling environment. In our case, JOP uses a modified version of GNU Classpath[7], whereas in our measurements we used a standard Sun JDK for cross-profiling. As GNU Classpath now supports the `java.lang.instrument` API, we may use a JVM based on GNU Classpath for cross-profiling in the future. In this way, we can reduce inaccuracies when profiling the execution of JDK methods.

[7]`http://www.gnu.org/software/classpath/`

Several bytecodes (e.g., runtime type checks and casts, object and array allocation, floating point arithmetic, division, etc.) consume a variable number of cycles on embedded Java processors. These bytecodes are substituted with routines that depend on runtime parameters (e.g., object type, array dimensions, etc.). Currently, we estimate the cycle consumption of these bytecodes with a constant, causing inaccuracies in the cross-profiles. To solve this issue, we will instrument the affected bytecodes so as to simulate the routines that would be executed on the embedded target platform. This simulation will not actually execute these routines, but only compute the cycles that would be consumed when executing them.

One major limitation of our current cross-profiler is its inability to take cache behavior into account. Recent Java processors have a method cache that helps reducing the cycles consumed by method invocation and return. We are currently working on an extension mechanism for our cross-profiler that allows intercepting method call and return so as to simulate a custom method cache and compute the actual call/return cycles accordingly. With this additional feature, our cross-profiler will also become a valuable tool for quickly evaluating new caching strategies on large sets of benchmarks, before implementing the most effective strategy in hardware. However, depending on the complexity of the cache simulation, the extra overhead can be significant.

Finally, we want to show the general applicability of our approach by supporting additional Java processors as cross-profiling target.

8. Conclusion

In this paper, we introduced a novel technique for cross-profiling based on Java bytecode instrumentation. Our approach ensures platform-independence, portability, and compatibility with standard Java runtime systems. It enables calling-context-sensitive cross-profiling of embedded applications (which normally execute in resource-constrained target environments) in any standard Java host environment, completely decoupled from the target system.

As case study, we evaluated our approach with the Java processor JOP as target system. The generated cross-profiles are sufficiently accurate for many practical purposes; the error is below 5% for the embedded benchmarks we profiled. Our performance evaluation also shows that the overhead caused by bytecode instrumentation is reasonable, considering that the cross-profiler is written in pure Java and hence is completely portable.

To sum up, we are promoting new profiling techniques (and their implementation in the cross-profiler CProf) for accurate performance analysis of embedded Java applications that can be easily used by the software developer within the preferred development environment. Because there is no need to deploy the embedded application on the target system and because the profiling causes only moderate overhead, our approach is already applicable before the target system exists and allows cross-profiling of large-scale applications and benchmarks that could not run on the target system because of resource constraints.

Acknowledgements

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

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. A portable and customizable profiling framework for Java based on bytecode instruction counting. In *Third Asian Symposium on Programming Languages and Systems (APLAS 2005)*, volume 3780 of *Lecture Notes in Computer Science*, pages 178–194, Tsukuba, Japan, Nov. 2005. Springer Verlag.

[3] W. Binder, J. Hulaas, and P. Moret. Advanced Java Bytecode Instrumentation. In *PPPJ 2007 (5th International Conference on Principles and Practices of Programming in Java)*, pages 135–144, Lisbon, Portugal, 2007. ACM Press.

[4] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[5] R. Covington, S. Dwarkadas, J. Jump, J. Sinclair, and S. Madala. The efficient simulation of parallel computer systems. *International Journal in Computer Simulation*, 1:31–58, 1991.

[6] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, Nov. 2003.

[7] B. Dufour, L. Hendren, and C. Verbrugge. *J: A tool for dynamic analysis of Java programs. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 306–307, New York, NY, USA, 2003. ACM Press.

[8] S. Dwarkadas, J. R. Jump, R. Mukherjee, and J. B. Sinclair. Execution-driven simulation of shared-memory multiprocessors. In *MASCOTS*, pages 83–86, 1993.

[9] D. Gregg, J. F. Power, and J. Waldron. A method-level comparison of the Java Grande and SPEC JVM98 benchmark suites. *Concurrency and Computation: Practice and Experience*, 17(7–8):757–773, 2005.

[10] D. S. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 53–59. IEEE Computer Society, 2001.

[11] J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and T. Ungerer. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems*, 27(1):19–31, 2003.

[12] S. Liang and D. Viswanathan. Comprehensive profiling support in the Java virtual machine. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS-99)*, pages 229–240, Berkeley, CA, May 3–7 1999. USENIX Association.

[13] Mentor Graphic Inc. ModelSim. Web pages at `http://www.model.com/`.

[14] J. M. O'Connor and M. Tremblay. picoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, 1997.

[15] ProSyst. JProfiler. Web pages at `http://www.prosyst.com/products/tools_jprofiler.html`.

[16] W. Puffitsch. picoJava-II in an FPGA. Master's thesis, Vienna University of Technology, 2007.

[17] W. Puffitsch and M. Schoeberl. picoJava-II in an FPGA. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 213–221, Vienna, Austria, September 2007. ACM Press.

[18] M. Schoeberl. A time predictable Java processor. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006)*, pages 800–805, Munich, Germany, March 2006.

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

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

[21] M. Schoeberl and R. Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.