Cache-aware Cross-profiling for Java Processors

Walter Binder
Faculty of Informatics
University of Lugano
CH–6900 Lugano
Switzerland
walter.binder@unisi.ch

Alex Villazón
Faculty of Informatics
University of Lugano
CH–6900 Lugano
Switzerland
alex.villazon@lu.unisi.ch

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology
A–1040 Vienna
Austria
mschoebe@
mail.tuwien.ac.at

Philippe Moret
Faculty of Informatics
University of Lugano
CH–6900 Lugano
Switzerland
philippe.moret@lu.unisi.ch

ABSTRACT

Performance evaluation of embedded software is essential in an early development phase so as to ensure that the software will run on the embedded device's limited computing resources. Prevailing approaches 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 very slow. In this paper, we introduce a customizable cross-profiling framework for embedded Java processors, including processors featuring a method cache. The developer profiles the embedded software in the host environment, completely decoupled from the target system, on any standard Java Virtual Machine, but the generated profiles represent the execution time metric of the target system. Our cross-profiling framework is based on bytecode instrumentation. We identify several pointcuts in the execution of bytecode that need to be instrumented in order to estimate the CPU cycle consumption on the target system. An evaluation using the JOP embedded Java processor as target confirms that our approach reconciles high profile accuracy with moderate overhead. Our cross-profiling framework also enables the rapid evaluation of the performance impact of possible optimizations, such as different caching strategies.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.4 [**Performance of Systems**]: Measurement techniques; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*

CASES'08, October 19–24, 2008, Atlanta, Georgia, USA.

General Terms

Algorithms, Languages, Measurement, Performance

Keywords

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

1. INTRODUCTION

High-level, object-oriented programming models are becoming increasingly popular for the development of embedded and real-time systems, since they help enhancing productivity and avoiding certain kinds of programming mistakes. Because of its safety guarantees, Java [13] is an attractive language for developing embedded systems. Language safety means that the execution of programs proceeds according to the language semantics. For instance, types are not misinterpreted and data is not mistaken for executable code. The safety properties of Java depend on techniques such as strong typing, automatic memory management, dynamic bound checks, and bytecode verification. The Java Platform Micro Edition (JavaME) provides a subset of the language features and class library of the Standard Edition (JavaSE), suitable for embedded systems with limited computing resources. Special requirements for real-time systems are addresses in the real-time specification for Java [9].

Java Virtual Machines (JVMs) [20] tailored to embedded systems either interpret the application bytecode, employ just-in-time compilation on the embedded system, compile the embedded Java application (in the development environment) to native code for the embedded target system, or use a dedicated Java processor that directly executes bytecodes, such as the aJile processor [16], Cjip [17], or JOP [27]. In this paper, we consider only embedded Java processors as cross-profiling targets.

Performance evaluation of embedded system software is crucial in order to ensure that the created software manages on the target system's scarce resources. Concomitant performance evaluation is particularly important for embedded system software written in Java (or in any other high-level, object-oriented programming language), since the per-

formance impact of certain language features (such as type checks, bound checks, garbage collection, etc.) may not be directly apparent to the programmer.

Unfortunately, profiling of embedded Java applications is currently a tedious task that requires either deployment of the embedded application on the target platform or a simulator of that platform. However, the embedded target system may not be available in an early development phase. Furthermore, deployment and performance measurements on the target platform are time-consuming. Similarly, simulators can be prohibitively slow. For instance, we found that the simulator ModelSim [21] causes excessive overhead of up to factor 33000 (when compared to running the same Java application on a standard JVM on the same machine). Consequently, embedded Java applications are rarely profiled in an early development phase.

Whereas the JavaSE offers a dedicated profiling interface, the JVM Tool Interface (JVMTI) [30], embedded Java systems often lack profiling support. Because of the resource constraints on embedded Java systems, CPU time and memory consuming profiling techniques are often impossible. For example, the Calling Context Tree (CCT) [1] provides detailed profiling data for each calling context, which helps locating hot spots. However, the CCT may consume significantly more heap memory than the profiled application itself.

In order to enable and ease performance evaluation of embedded Java software that is intended to run on a Java processor, we introduce the customizable cross-profiling framework CProf. CProf is written in pure Java and runs on any standard JVM. It enables calling-context-sensitive cross-profiling of Java applications directly within the development environment, which we will call 'host' in the following, completely decoupled from the embedded target system. Nonetheless, the generated profiles show the execution time metrics of the target.

CProf uses bytecode instrumentation in order to generate calling-context-sensitive profiles with CPU cycle estimations for the target processor. It relies on a bytecode instrumentation framework [6], which ensures that each Java method is instrumented and therefore represented in the profile, including all methods in application classes and in the Java class library.

CProf identifies particular points in the execution of programs, so-called pointcuts in aspect-oriented programming (AOP) terminology [18], where the cycle estimate needs to be updated. Currently, CProf supports *method entry*, *method return*, and *basic block entry* as relevant pointcuts. As we will show in this paper, these pointcuts enable cross-profiling for Java processors.

The scientific contributions of this paper are the software architecture of the customizable cross-profiling framework CProf, as well as its specialization for the embedded Java processor JOP [27]. We choose JOP as our first cross-profiling target, because the CPU cycle consumption for the bytecodes is public available. Moreover, JOP is a recent architecture that includes an instruction cache, which caches whole methods. Therefore, cross-profiling for the JOP processor also requires simulating that instruction cache. We evaluated CProf with the JOP target model, varying different parameters of the cache. Our results confirm that CProf yields accurate CPU cycle estimations (with an error below 3.3%) and causes reasonable overhead, orders of magnitude less than simulators.

This paper is structured as follows: Section 2 describes CProf's execution time model and gives an overview of the JOP processor. Section 3 explains our cross-profiling framework CProf. In Section 4 we assess the accuracy of the generated cross-profiles for some embedded Java benchmarks and measure the runtime overhead due to cross-profiling. Section 5 discusses related work and Section 6 concludes this paper.

2. JAVA PROCESSORS AS CROSS-PROFILING TARGETS

There are several Java processors which follow a similar execution time model, and some implement a method cache. Since our cross-profiling approach is based on cycle estimates for bytecodes, we firstly summarize our assumptions on the target execution time model. Afterwards, we briefly describe JOP, which is the target processor for our evaluation, and discuss its method cache.

2.1 Assumptions on the Execution Time Model

Our cross-profiling approach targeting Java processors is based on the following assumptions:

- For most bytecodes, the CPU cycle consumption on the target can be accurately estimated by constants, independently of the context where these bytecodes occur.
- For method invocation and return bytecodes, the cycle consumption on the target may also depend on the size of the callee or caller method. Furthermore, the presence of a method cache may affect the cycle consumption of invoke/return bytecodes. As object-oriented programs tend to have rather small methods and method invocation/return bytecodes are expected to be executed frequently, we consider an accurate estimation of the cycle consumption essential for these bytecodes.
- In addition to invocation/return bytecodes, some other bytecodes, such as type checks, may not consume a constant number of cycles on the target. We assume that reasonable (though not always accurate) estimates are available for these bytecodes.

2.2 The Java Processor JOP

JOP [27] is an implementation of the JVM in hardware. As embedded systems are often also real-time systems with hard timing constraints, the main focus of the development of JOP has been on time-predictable bytecode execution. All function units, and especially the interactions between them, are carefully designed to avoid any time dependencies between bytecodes. This feature simplifies the low-level part of worst-case execution time (WCET) analysis, a mandatory analysis for hard real-time systems.

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. All microcode instructions

have a constant execution time of one cycle. No stalls are possible in the microcode pipeline. The absence of time dependencies between bytecodes results in a simple processor model for the low-level WCET analysis [28], which fully conforms to the aforementioned assumptions for cross-profiling.

Besides JOP, there are several other Java processors for embedded systems; the execution time modeling for those processors can be done in a similar way. The first Java processor, picoJava [22], was developed by Sun Microsystems. The most successful Java processor is the aJile processor [16] that was initially conceived as a platform for the Java real-time specification [9]. Another Java processor, Cjip [17], supports multiple instruction sets, and the JVM is implemented largely in microcode. Komodo [19] is a multithreaded Java processor intended as a basis for research on real-time scheduling on a multithreaded microcontroller. The follow-up project, jamuth [32], is a commercial version of Komodo.

2.3 Method Cache

JOP introduced a special instruction cache, the method cache [26], which caches whole methods. A method cache is also integrated in the embedded Java processor SHAP [23] and considered in jamuth [32] as a time-predictable caching solution.[1]

With a method cache, only invoke and return bytecodes can result in a cache miss. All other bytecodes are guaranteed cache hits. The idea to cache whole methods is based on the assumption that WCET analysis at the call graph level is more practical than performing cache analysis for each bytecode. Furthermore, loading whole methods also leads to better average case execution times for memory with long latency but high bandwidth.

CProf supports the simulation of a method cache in a customizable way. The simulation provides the information whether the invoked method or the method caller upon return will be a cache hit or a cache miss. On a miss, we calculate the cache load time with the given processor execution time model. The load time depends on the size of the method. However, on JOP, the cache loading is done in parallel with microcode execution in the core pipeline. Therefore, small methods do not add any additional latency to the invoke or return bytecodes.

3. CROSS-PROFILING

Bytecode instrumentation is a well-known technique for profiling [3, 4, 5]. While the work presented here leverages bytecode instrumentation-based profiling techniques that preserve calling context information, it introduces the instrumentation of certain low-level pointcuts, allowing for flexible, user-defined collection of dynamic metrics. Thanks to CProf's support for customization, we are able to create cross-profilers for Java processors with a minimum of development effort.

In the following, we firstly describe our representation of the calling context. Secondly, we discuss how collected profiling data can be processed online and in a customized way. Thirdly, we present our generic instrumentation techniques that enable cross-profiling for embedded Java processors. Fourthly, we describe CProf's configuration for the JOP processor used in our evaluation.

[1]Personal communication with Sascha Uhrig.

3.1 Calling Context Tree

We instrument bytecode such that each thread maintains a *Calling Context Tree* (CCT) [1]. Each calling context is represented by a node in the CCT, which holds a method identifier, a method invocation counter, and the set of callee contexts. In addition, CCT nodes may store various dynamic metrics that are collected for each calling context. Method identifiers convey class name, method name, method signature, and method size (in bytes).

CProf maintains a CCT for each thread. Thanks to the generated code for CCT management[2], the instrumentation has access to both the caller's node and the callee's node in the current thread's CCT.

3.2 Customized Processing of Profiling Data

Periodically, each thread invokes a user-defined profiler to process the thread's CCT. A typical profiler may aggregate the CCTs of all threads within a 'global' CCT representing the activities of all threads and output the 'global' CCT upon program termination (e.g., using a JVM shutdown hook). Alternatively, custom profilers may be used for online processing of the profiling data, such as for displaying continuous metrics.

The profiler interface has only three methods that must be implemented by the user-defined profiler: one method to initialize the profiler, a second method for registering the CCT root of each thread that executes profiling code, and a third method that gets periodically invoked by each thread to enable online processing of profiling data.

The latter method is invoked whenever a dedicated, thread-local bytecode counter reaches a user-defined threshold. We use some form of call/return polling [12] to increment and check the bytecode counter in strategic program locations, such as upon method entry or in the beginning of loops. This approach ensures the periodic activation of the profiler by each thread, according to the progress (measured as the number of executed bytecodes) the thread has made since its last invocation of the profiler.

3.3 Customized Collection of Dynamic Metrics

CProf allows customizing the way dynamic metrics are computed for each calling context. Figure 1 gives a high-level overview of CProf, showing the configurable and extensible parts of the system. Moreover, the figure illustrates the instrumentation of a sample method.

Instrumentation with CProf involves three phases, the basic block analysis (BBA), the static calculation of metrics for each basic block (BB), and the actual instrumentation.

3.3.1 Basic Block Analysis

The basic block analysis takes the bytecode of a method and returns a representation of the control flow graph (CFG). CProf provides the necessary abstractions to represent a CFG and the nodes in it. While the user may employ a custom BBA algorithm, CProf provides two predefined BBA algorithms, which we call 'Default BBA' and 'Precise BBA'.

In the Default BBA, only bytecodes that may change the control flow non-sequentially (i.e., jumps, branches, method

[2]Method signatures are extended so as to pass the caller's CCT node to the callee, and upon entry, the callee first looks up or creates its own node as a child of the caller's node.

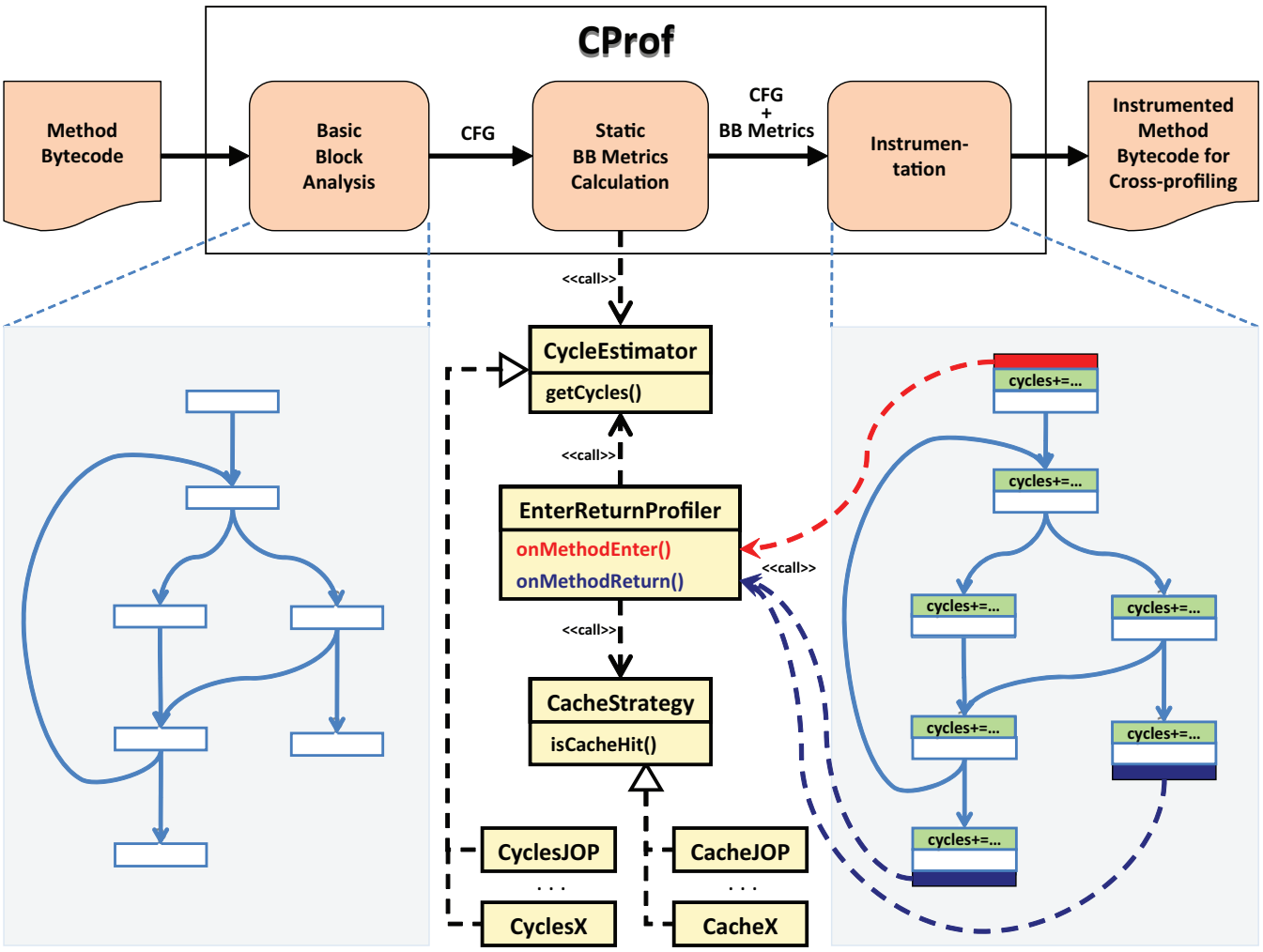

Figure 1: Overview of CProf

return, exception throwing) end a BB. Method invocations do not end BBs, because we assume that the execution will return after the call. This definition of BB corresponds to the one used in [7] and is related to the factored control flow graph (FCFG) [10]. In contrast, using the Precise BBA, each bytecode that might throw an exception ends a BB.

As we will explain below, CProf instruments the beginning of each BB, assuming that all bytecodes in the BB will be executed. The advantage of the Default BBA is that it creates rather large BBs. Therefore, the Default BBA helps reducing the number of program locations where instrumentation code is inserted, resulting in lower cross-profiling overhead. As long as no exceptions are thrown, the Default BBA does not cause any inaccuracies. However, in the case of an exception, CProf's assumption that all bytecodes in the BB would be executed is violated, if the exception-throwing bytecode is not the last one in its BB. The Precise BBA avoids this potential imprecision, but causes higher overhead because the resulting BBs are smaller.

3.3.2 *Static Basic Block Metrics Calculation*

Regarding the supported target processors for cross-profiling, we assume that the CPU cycle consumption of all bytecodes, apart from method invocation and return, can be estimated by a constant. This assumption allows us to statically compute the CPU cycle estimate for each BB, by summing up the CPU cycle estimates for the bytecodes in the BB (while ignoring method invocation/return bytecodes).

Figure 1 shows CProf's component for the static BB metrics calculation. While this component can be replaced by the user, the default implementation in our cross-profiling framework invokes a given `CycleEstimator` for each bytecode in a BB (see the UML class diagram included in Figure 1). The `CycleEstimator` is an abstraction providing the method `getCycles()` that takes, amongst others, a bytecode as argument and returns the corresponding CPU cycle estimate. We designed the cross-profiling framework so as to ease the plug-in of custom `CycleEstimator` implementations according to the cross-profiling target processor (`CyclesJOP` respectively `CyclesX` in the class diagram).

The statically computed BB cycle estimates are stored within the nodes of the CFG. That information is used afterwards by the instrumentation component for instrumenting the beginning of each BB.

3.3.3 Instrumentation

CProf's instrumentation component fulfills three roles: (1) it generates the code for maintaining the CCT, (2) it inserts the polling code that ensures the periodic invocation of a custom profiler, and (3) it injects the code that computes the CPU cycle estimation for the target. While the first two issues have been outlined before, we focus here on the third issue.

Figure 1 illustrates a sample CFG generated by a BBA algorithm on the left side, as well as the resulting CFG after instrumentation on the right side. The CFG is used to identify the method entry, method return, and basic block entry pointcuts.

For the basic block entry pointcut, CProf inserts a bytecode sequence in the beginning of each BB that increments a cycle counter (within the CCT node representing the executing method) according to the statically pre-calculated cycle estimate for the BB. Hence, the chosen BBA algorithm determines the program locations where this instrumentation happens.

The method entry and return pointcuts are instrumented with invocations to the `EnterReturnProfiler`, which provides the two methods `onMethodEnter()` and `onMethodReturn()`.[3] As arguments, these methods receive the caller and callee nodes in the CCT, as well as the invocation/return bytecode. As mentioned before, each CCT node refers to the corresponding method identifier, which in turn provides method-related information, such as the method size (before instrumentation).

While a custom `EnterReturnProfiler` can be provided by the user, our default implementation first determines whether the method to be loaded (i.e., the callee for `onMethodEnter()`, respectively the caller for `onMethodReturn()`) is in the cache. To this end, CProf provides the `CacheStrategy` abstraction with a boolean method `isCacheHit()` that takes as argument a method identifier. The user has to provide an appropriate implementation of the `CacheStrategy` for the target processor. The UML class diagram in Figure 1 shows two concrete implementation, `CacheJOP` and `CacheX`.

After consulting the configured `CacheStrategy`, the default implementation of `EnterReturnProfiler` invokes the `CycleEstimator` to compute the cycles consumed by the method invocation/return. In addition to a bytecode, `getCycles()` also takes information on the method size and whether the method was found in the cache (for bytecodes other than method invocation/return, this extra information is meaningless and ignored by implementations of `getCycles()`). The cycle estimate is added to the cycle counter in the appropriate CCT node (i.e., in the caller node for method invocation, respectively in the callee node for method return).

For the method return pointcut, the instrumentation component exactly knows the return bytecode and passes it as argument to `onMethodReturn()`. However, for the method entry pointcut, the instrumentation component cannot always determine which invocation bytecode will be used by the caller. A method that is also declared in an interface may be called with the `invokevirtual` bytecode or with the `invokeinterface` bytecode. Furthermore, one use of `invokespecial` is to access a superclass' version of a method (this mechanism is used to compile Java's `super()` construct). I.e., in certain cases the same method may be called by `invokevirtual`, `invokeinterface`, or `invokespecial`.

As general solution to this problem, we can pass the information regarding the method invocation bytecode from the caller to the callee as an extra method argument. For invocations of static methods, private methods, and constructors, the extra argument is not needed, because the invocation bytecode is statically known (both when instrumenting the caller method(s) and the callee method).[4] Note that in general, the method entry pointcut cannot be implemented by instrumenting the caller, because of polymorphic call sites (where the method identifier of the callee would not be known). In contrast, the method identifier of the caller is always known to the callee, since the caller passes its CCT node to the callee.

Another issue is abnormal method completion through an exception. In this case, `onMethodReturn()` is not invoked. A general solution to this issue would be the introduction of another pointcut (corresponding to a method `onMethodAbnormalCompletion()` in `EnterReturnProfiler`), which could be implemented by an inserted exception handler. However, as such an instrumentation would cause higher cross-profiling overhead, we have not yet implemented such a pointcut.

3.4 Customization for the JOP Processor

While the software architecture of CProf allows customization and replacement of all components shown in Figure 1, the typical customization for a target processor that fits our execution time model will require only the implementation of the `CycleEstimator` and `CacheStrategy` interfaces.

With respect to cross-profiling for the JOP processor, we use CProf's Default BBA. This choice is a reasonable tradeoff between cross-profiling accuracy and overhead.

Regarding the customized processing of cross-profiling data, we created a simple profiler that aggregates the CCTs of all application threads into a global structure, but disregards the CCTs of system threads. The profiler keeps weak references to application threads, associated with the corresponding CCT root nodes, in order to collect the CCTs of threads that terminate during cross-profiling. Upon JVM shutdown, the CCTs of the remaining application threads are collected, before the final cross-profile is emitted.

We implemented a `CycleEstimator` in the class `CyclesJOP`, which is an adapter to the JOP cycle estimation API provided by the class `com.jopdesign.wcet.WCETInstruction`[5]. Furthermore, we implemented a `CacheStrategy` in the class `CacheJOP`, which has less than 50 lines of Java code and simulates the hardware method cache of the current JOP processor.

[3] In AOP terminology, `EnterReturnProfiler` corresponds to an aspect class, and the methods `onMethodEnter()` and `onMethodReturn()` are related to advices.

[4] The instrumentation component can statically determine whether a callee is private, since it processes all methods within the same class file, and private methods can only be called in the defining class. Constructor invocations are identified by the special method name `<init>`.

[5] `http://www.opencores.org/cvsweb.shtml/jop/`

In Figure 1, the classes `CyclesX` and `CacheX` are meant to illustrate that CProf can be easily adapted to another Java processor X, provided that X fits into our execution time model.

4. EVALUATION

In the following we evaluate CProf regarding cross-profile accuracy and runtime overhead. For the accuracy assessment, we compare CProf's cycle estimates measured on the host with the actual CPU cycle consumption on JOP. For the runtime overhead evaluation, we measure the execution time of CProf on the host and analyze the different sources of overhead.

4.1 Benchmarks and Evaluation Settings

To evaluate our cross-profiling approach, we selected two benchmark suites, the embedded benchmarks JavaBenchEmbedded[6] (JBE) and SPEC JVM98[7]. JBE contains several micro benchmarks and three real-world applications (a motor control system, a tiny TCP/IP stack for embedded Java, and a lift controller); we use only the real-world applications, run with a constant iteration count of 10000. JVM98 consists of seven benchmarks, which are not aimed to run on the target system, but provide larger workloads for the runtime overhead evaluation. We use JVM98 with a problem size of 100.

Since the accuracy evaluation of CProf aims at calculating CProf's cycle estimate independently from the execution time on the host, we use a standard desktop computer for this purpose. In contrast, for the runtime overhead evaluation, in order to obtain reproducible results, we execute the benchmarks on an isolated host in single-user mode (no networking), where we removed background processes as much as possible. The host environment for the accuracy assessment is a Linux Debian computer (Intel Core2 Duo, 2.33GHz, 2048MB RAM), whereas the host environment for the overhead evaluation is a Linux Fedora Core 2 computer (Intel Pentium 4, 2.66GHz, 1024MB RAM). For all measurements, we use the Sun JDK 1.7-ea-b24 HotSpot Server VM.

4.2 Accuracy

In order to gather reference data for the accuracy assessment, we ran JBE on the real hardware (JOP processor, 100MHz, 1MB RAM) and used a clock cycle counter to measure the execution time.

Table 1 compares the CPU cycle consumption on the target ('JOP') with the total cycle estimates in the cross-profiles obtained on the host ('CProf'). The total cycle estimates are the sum of the cycle estimates in all calling contexts of the cross-profiles. We varied the method cache size respectively the number of blocks, and observed a maximum error of 3.28%. The inaccuracies are caused by differences in the Java class libraries on JOP respectively on the host, and by a simplified execution time model for some bytecodes (e.g., runtime type checks and casts, floating point arithmetic, etc.).

In order to compare the achieved cross-profiling accuracy with a simulator, we also obtained cycle estimates with JopSim[8], a high-level simulator for JOP written in Java. JopSim is an interpreting JVM with simulation of JOP internal hardware (e.g., timer interrupts and I/O devices); its main purpose is to help debugging JOP-related functions. We found that JopSim's cycle estimates are far less accurate than those obtained with CProf. E.g., we observed an error of up to 102% for the JBE benchmarks with a 1KB/16 cache setting, which is extremely high compared to an error of -2.54% obtained with CProf for the same setting.

4.3 Runtime Overhead

Table 2 presents the results of our overhead evaluation for JBE and JVM98. Each measurement represents the median of 15 runs of the benchmark within the same JVM process. The column 'Orig.' shows our reference measurements, running the benchmarks without CProf on the host. The columns 'ovh' present overhead factors for each setting. For each benchmark suite, we also show the geometric mean of the measurements.

We evaluated CProf in three different configurations so as to separate the distinct sources of overhead.

Firstly, we evaluated the overhead due to the inserted code that creates the CCTs and performs computations on the BB pointcuts, but without the method entry and return pointcuts ('CProf no ER'). This corresponds to the instrumented CFG of Figure 1 but without the invocations of the `EnterReturnProfiler`. In this setting, we observe an overhead factor of 1.33–9.99. For 'mtrt', we experienced the highest overhead. 'mtrt' is known to make extensive use of small methods [14], which makes the CCT maintenance expensive.

Secondly, we enabled the method entry and return pointcuts, but used a trivial 'no cache' strategy that always assumed a miss ('CProf ER, no cache'). We notice that the invocations of the `EnterReturnProfiler` introduces an additional overhead of factor 0.21–4.19 (overhead difference between 'CProf ER, no cache' and 'CProf no ER').

Thirdly, we evaluated the real CProf configuration for JOP, using a cache strategy that simulates a method cache of 1KB with 16 blocks ('CProf ER+cache'). The experienced overhead of factor 2.94–47.54 is more than twice the overhead in the previous setting. In contrast to CProf's instrumentation, the custom cache strategy (`CacheJOP` in Figure 1) was not optimized, which explains the high extra overhead. Overall, the execution time in this setting is still reasonable (below 70ms for the JBE benchmarks); optimizing the cache strategy to reduce cross-profiling overhead is typically not worth the effort in practice.

We also executed the JBE benchmarks with the JopSim simulator (cache enabled) and observed overheads of factor 690–10500 depending on the host platform. Still, the overheads caused by VHDL simulators, such as ModelSim [21], can be orders of magnitude higher. However, such simulators are not intended to be used for cross-profiling.

5. RELATED WORK

Cross-profiling techniques have been used to simulate parallel computers [11]. Since it is not always possible to use a host processor that has the same instruction set as the target processor, cross-profiling tries to match up the basic

[6] `http://www.jopwiki.org/JavaBenchEmbedded/`
[7] `http://www.spec.org/osg/jvm98/`
[8] JopSim is part of the JOP source distribution. `http://www.jopdesign.org/`

Table 1: Accuracy of CProf's cycle estimates with different method cache size and number of blocks.

Cache size / blocks	Kfl JOP	Kfl CProf	Kfl %Err	UdpIp JOP	UdpIp CProf	UdpIp %Err	Lift JOP	Lift CProf	Lift %Err
1 KB / 4	58,600,769	60,137,802	**2.56 %**	123,670,263	123,650,110	**-0.02 %**	55,270,135	56,750,036	**2.61 %**
1 KB / 8	57,433,890	58,679,094	**2.12 %**	122,280,263	121,110,110	**-0.97 %**	55,270,135	57,029,997	**3.09 %**
1 KB / 16	56,821,578	56,071,896	**-1.34 %**	120,100,263	117,120,110	**-2.54 %**	55,160,135	54,800,220	**-0.66 %**
2 KB / 4	58,600,769	60,137,802	**2.56 %**	123,670,263	123,650,110	**-0.02 %**	55,270,135	56,750,036	**2.61 %**
2 KB / 8	57,142,061	58,679,094	**2.62 %**	121,500,263	121,110,110	**-0.32 %**	55,160,135	57,029,997	**3.28 %**
2 KB / 16	56,361,230	56,071,896	**-0.52 %**	117,040,276	117,120,110	**0.07 %**	53,280,314	54,800,220	**2.77 %**
4 KB / 4	58,600,769	60,137,802	**2.56 %**	123,670,263	123,650,110	**-0.02 %**	55,270,135	56,750,036	**2.61 %**
4 KB / 8	57,142,061	58,679,094	**2.62 %**	121,500,263	121,110,110	**-0.32 %**	55,160,135	57,029,997	**3.28 %**
4 KB / 16	54,512,933	56,071,896	**2.78 %**	117,040,276	117,120,110	**0.07 %**	53,280,250	54,800,220	**2.77 %**

Table 2: CProf overhead with/without Enter-Return calls (ER), respectively with/without cache strategy.

	Orig.	CProf no ER		CProf ER, no cache		CProf ER + cache	
JBE	[ms]	[ms]	**ovh**	[ms]	**ovh**	[ms]	**ovh**
kfl	2.87	11.16	**3.89**	15.80	**5.51**	39.04	**13.60**
udplp	4.39	17.93	**4.08**	31.17	**7.10**	67.06	**15.28**
lift	1.82	9.11	**5.01**	13.41	**7.37**	23.29	**12.80**
Geo.mean	2.84	12.22	**4.30**	18.76	**6.60**	39.36	**13.85**
JVM98	[s]	[s]	**ovh**	[s]	**ovh**	[s]	**ovh**
compress	5.68	14.69	**2.59**	19.70	**3.47**	40.74	**7.17**
jess	1.47	6.16	**4.19**	8.07	**5.49**	20.81	**14.16**
db	13.71	18.22	**1.33**	21.12	**1.54**	40.30	**2.94**
javac	3.79	15.54	**4.10**	17.42	**4.60**	32.33	**8.53**
mpegaudio	2.48	7.15	**2.88**	9.23	**3.72**	19.22	**7.75**
mtrt	1.16	11.59	**9.99**	16.45	**14.18**	55.15	**47.54**
jack	3.48	8.20	**2.36**	10.09	**2.90**	18.91	**5.43**
Geo.mean	3.31	10.82	**3.27**	13.68	**4.13**	30.05	**9.08**

blocks on the host and on the target machines, changing the estimates on the host to reflect the simulated target. Our approach follows a similar principle, but uses precise cycle estimates at the instruction-level, because both the target and the host instructions are JVM bytecodes.

Profiling embedded Java 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 [24] uses a profiling agent running directly on the target device. The agent communicates through the network with the profiling front-end running within the Eclipse IDE. Although this approach enables accurate profiling, the agent is implemented in native code using the JVM Profiler Interface (JVMPI)[9], and hence is limited to a reduced number of virtual machines and operating systems. Furthermore, the agent itself consumes resources on the target system which may perturbate measurements. Another drawback is that profiling requires deployment of the application on the target system. In contrast, CProf runs independently from the target platform, using state-of-the-art Java technology on the host.

[9]The JVMPI [29] has been deprecated in JDK 1.5 and was replaced by the JVMTI [30].

Reference [31] presents a fast partitioning algorithm based on profiles to remotely execute parts of an embedded Java application on a server so as to reduce energy consumption on the embedded device. The partitioning algorithm is executed on the embedded device. Our cross-profiling approach helps identifying hot spots before the application is deployed.

There are several JVM hardware implementations, such as picoJava [22], aJile's JEMCore [16], Komodo [19], or FemtoJava [2]. Even though simulation tools are available for the processor design, profiling is not always possible and made only at the latest stage of development, i.e., on the actual processor. Moreover, most current profilers rely on the JVMPI or JVMTI, which are not well supported by many embedded Java systems. Our portable approach for cross-profiling avoids this problem.

Cross-profiling for an embedded Java processor was introduced in [8]. In contrast to the solution presented here, the cross-profiler in [8] was not configurable and supported only a single target processor. Furthermore, it lacked the method return pointcut and was incapable of simulating a method cache.

Related to aspect weaving in AOP, our approach is based on the customized instrumentation of three low-level pointcuts, method entry, method return, and basic block entry.

The AspectJ weaver[10] also works at the bytecode level and AspectJ provides pointcuts for method entry and return, but not for basic block entry. In reference [15] an extension to AspectJ uses control-flow analysis to determine loop pointcuts used to parallelize loops. Eos-T [25], an aspect-oriented version of C#, also supports low-level pointcuts to enable selective branch coverage profiling. Our approach shows that low-level pointcuts at the basic block level are well suited for cross-profiling.

6. CONCLUSION

In this paper we presented CProf, a customizable cross-profiling framework for embedded Java processors. CProf is completely portable and runs on any standard JVM. It relies on bytecode instrumentation to collect calling-context-sensitive cross-profiles for a given target processor. Instrumentation takes place at certain low-level pointcuts, concretely on method entry, on method return, and on basic block entry.

CProf has been designed for customization and extension. All involved algorithms (i.e., basic block analysis, static basic block metrics calculation, and instrumentation) are provided as pluggable components. The default implementations of these components are also configurable in a flexible way, regarding the cycle estimation for bytecodes and the simulation of a method cache. Hence, CProf can be easily customized for Java processors that conform to CProf's general execution time model.

For our evaluation, we configured CProf to yield cycle estimates for the Java processor JOP, which also features a method cache. Using JOP as target platform, we have shown that our approach reconciles high cross-profile accuracy (error below 3.3%) and moderate overhead, which is orders of magnitude below the overhead caused by typical simulators.

We are currently using CProf to evaluate the impact of different cache strategies (different size, organization, and replacement strategy), as well as to determine the impact of performance optimzations of individual bytecodes. While there is a lack of large benchmark suites for embedded Java, our cross-profiling approach allows us to use also standard Java benchmarks (e.g., SPEC JVM98, SPEC JBB2005, DaCapo, etc.) in order to gather more statistics on the impact of certain processor optimizations.

Regarding future work, we are extending the set of pointcuts supported by CProf to enable a more precise estimation for bytecodes where a constant approximation is inappropriate. Hence, CProf is evolving towards a low-level, aspect-oriented programming environment for profiling and cross-profiling. Moreover, we want to explore whether our approach can be generalized to Java-based embedded systems that use interpretation respectively compilation instead of a Java processor.

Acknowledgements

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

[10] `http://www.aspectj.org/`

7. 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] A. C. Beck and L. Carro. Low power Java processor for embedded applications. In *Proceedings of the 12th IFIP International Conference on Very Large Scale Integration*, December 2003.

[3] 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.

[4] W. Binder. Portable and accurate sampling profiling for Java. *Software: Practice and Experience*, 36(6):615–650, 2006.

[5] W. Binder and J. Hulaas. Flexible and efficient measurement of dynamic bytecode metrics. In *Fifth International Conference on Generative Programming and Component Engineering (GPCE-2006)*, pages 171–180, Portland, Oregon, USA, Oct. 2006. ACM.

[6] 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.

[7] W. Binder, J. G. Hulaas, and A. Villazón. Portable resource control in Java. *ACM SIGPLAN Notices*, 36(11):139–155, Nov. 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[8] 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)*, Saint-Malo, France, Sept. 2008. IEEE Computer Society Press.

[9] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, Reading, MA, USA, 2000.

[10] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Proceedings of the ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 21–31. ACM Press, 1999.

[11] 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.

[12] M. Feeley. Polling efficiently on stock hardware. In *the 1993 ACM SIGPLAN Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 179–187, June 1993.

[13] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, 2005.

[14] 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.

[15] B. Harbulot and J. R. Gurd. A join point for loops in AspectJ. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 63–74, New York, NY, USA, 2006. ACM.

[16] 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.

[17] Imsys. Im1101c (the cjip) technical reference manual / v0.25, 2004.

[18] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[19] 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.

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

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

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

[23] T. B. Preusser, M. Zabel, and R. G. Spallek. Bump-pointer method caching for embedded Java processors. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 206–210, New York, NY, USA, 2007. ACM.

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

[25] H. Rajan and K. Sullivan. Aspect language features for concern coverage profiling. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 181–191, New York, NY, USA, 2005. ACM.

[26] M. Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

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

[28] 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.

[29] Sun Microsystems, Inc. Java Virtual Machine Profiler Interface (JVMPI). Web pages at `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/`, 2000.

[30] Sun Microsystems, Inc. JVM Tool Interface (JVMTI) version 1.1. Web pages at `http://java.sun.com/javase/6/docs/technotes/guides/jvmti/index.html`, 2006.

[31] S. Tallam and R. Gupta. Profile-guided Java program partitioning for power aware computing. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, page 156b, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[32] S. Uhrig and J. Wiese. jamuth: an IP processor core for embedded Java real-time systems. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 230–237, New York, NY, USA, 2007. ACM Press.