SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2009)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.940

Cross-profiling for Java processors

Walter Binder[1,*,†], Martin Schoeberl[2], Philippe Moret[1] and Alex Villazón[1]

[1] *Faculty of Informatics, University of Lugano, CH-6900 Lugano, Switzerland*
[2] *Institute of Computer Engineering, Vienna University of Technology, Austria*

SUMMARY

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. The 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 article, 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 performance evaluation of new processor architectures before they are implemented. As a case study, we explore the performance impact of various processor design choices and optimizations, such as different cache sizes or pipeline organizations, and come up with an improved processor design that yields speedups of up to 40% on standard Java benchmarks. Copyright © 2009 John Wiley & Sons, Ltd.

Received 30 April 2009; Revised 13 August 2009; Accepted 17 August 2009

KEY WORDS: cross-profiling; embedded Java processors; bytecode instrumentation; platform-independent dynamic metrics; processor architecture design space exploration

1. INTRODUCTION

High-level, object-oriented programming models are becoming increasingly popular for the development of embedded and real-time systems, since they help to enhance the productivity and avoid certain kinds of programming mistakes. Because of its safety guarantees, Java [1] is an attractive

*Correspondence to: Walter Binder, Faculty of Informatics, University of Lugano, CH-6900 Lugano, Switzerland.
†E-mail: walter.binder@usi.ch

Contract/grant sponsor: Swiss National Science Foundation

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 addressed in the Real-Time Specification for Java [2]. While Java is still an emerging language for classic embedded systems (e.g. in the automotive and airborne domain), almost all mobile phones already contain a Java virtual machine (JVM) [3] to execute the so-called MIDlets.

JVMs 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 is, a JVM implemented in hardware) to directly execute bytecodes, such as the aJile processor [4], Cjip [5], or JOP [6]. In this article, we consider only embedded Java processors as cross-profiling targets.

Java processors may play an important role in future safety-critical applications, for which Java is currently being considered [7]. Safety-critical applications need to be certified and the reduction of code size is of primary importance, as the certification cost directly depends on the code size. Software certification is performed at the source code level and includes the whole software stack (operating system and libraries). A Java processor simplifies certification and hence helps to reduce cost, because it does not require an operating system and because only code in one language needs to be certified. Furthermore, worst-case execution time (WCET) for the safety-critical tasks must be known. A Java processor also simplifies WCET analysis of Java programs, as it can be performed at the bytecode level [8].

Performance evaluation of embedded system software is crucial in order to ensure that the created software executes satisfactorily 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 performance 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 [9] causes excessive overhead of up to factor 33 000 when compared with running the same Java application in 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) [10], 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) [11] provides detailed profiling data for each calling context, which helps to locate hotspots. However, the CCT may consume significantly more heap memory than the profiled application itself.

DOI: 10.1002/spe

In order to enable and ease the 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 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.

Cross-profiling not only helps analyze the performance of embedded Java software, it also allows estimating the performance of different Java processor designs without requiring these designs to be implemented. It is important to estimate the effects of architectural design choices on the performance of domain-specific applications in an early phase of the processor development [12]. With cross-profiling it is possible to collect evaluation data for realistically sized programs. That is, cross-profiling enables rapid design space exploration for embedded Java processors. Instead of implementing a new processor in hardware, it is sufficient to model its cycle consumption and to evaluate the performance on various workloads. Only the best performing processor model is afterwards implemented in hardware.

Our cross-profiling framework CProf is based on bytecode instrumentation in order to generate calling context cross-profiles with CPU cycle estimates for the target processor. It relies on a bytecode instrumentation framework [13], 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, the so-called *pointcuts* in aspect-oriented programming (AOP) terminology[‡] [14], 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 article, these pointcuts enable cross-profiling for Java processors.

This article builds on our prior work on cross-profiling [15–17], presenting the software architecture of the customizable cross-profiling framework CProf, as well as its specialization for the Java Optimized Processor JOP [6]. We choose JOP as cross-profiling target, because the CPU cycle consumption for the bytecodes is publicly 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 evaluate CProf with the JOP target model. Our results confirm that CProf yields accurate CPU cycle estimates with an error below 2% and causes reasonable overhead, orders of magnitude less than simulators.

As case study, we employ cross-profiling in order to quantitatively explore the performance impact of various processor design options and optimizations. Taking the current JOP architecture as baseline, we investigate the impact of different cache sizes and pipeline organizations, and come up with an improved processor design that yields speedups of up to 40% on standard Java benchmarks.

This article is structured as follows: Section 2 gives an overview of the JOP processor and describes CProf's execution time model. 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

[‡]In AOP, aspects specify *pointcuts* to intercept certain points in the execution of programs (so-called *join points*), such as method calls, field accesses, etc. *Advices* are executed *before*, *after*, or *around* the intercepted join points. Advices have access to contextual information of the join points.

DOI: 10.1002/spe

and measure the runtime overhead due to the cross-profiling. In Section 5 we describe our case study, exploring architectural enhancements for a Java processor with the aid of cross-profiling. Section 6 discusses the related work and Section 7 concludes this article.

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 first briefly describe JOP, which is the target processor for our evaluation, and discuss its method cache. Afterwards, we summarize our assumptions on the target execution time model.

2.1. The Java processor JOP

JOP [6] 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. Owing to JOP's accessible timing model, several WCET analysis tools support JOP [18–21].

We have chosen JOP [6] for our evaluation of the proposed cross-profiling and computer architecture exploration as it is: (a) a simple processor, (b) open-source, and (c) the execution timing is well documented. Furthermore, the JOP design is actually the root of a family of Java processors. Flavius Gruian has built a JOP compatible processor, with a different pipeline organization, with Bluespec Verilog [22]. The SHAP Java processor [23], although now with a different pipeline structure and hardware-assisted garbage collection, also has its roots in the JOP design.

JOP dynamically translates the 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 [18], which fully conforms to the aforementioned assumptions for cross-profiling.

The stack, which holds invocation frames, local variables, and the operand stack, is implemented as on-chip memory. This stack cache allows a non-stalling implementation of the microcode pipeline. Furthermore, the instructions are cached in the so-called method cache. Heap data and data in the class information area are not cached in our current configuration of JOP.

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 [24], was developed by Sun Microsystems. The most successful Java processor is the aJile processor [4] that was initially conceived as a platform for the Real-Time Specification for Java [2]. Another Java processor, Cjip [5], supports multiple instruction sets, and the JVM is implemented largely in microcode. Komodo [25] is a multithreaded Java processor intended as a basis for research on

DOI: 10.1002/spe

Table I. Execution time of simple bytecodes in cycles.

Instruction	Cycles	Function
iload_0	1	load local variable 0 on TOS
istore_0	1	store local variable 0
iconst_0	1	load constant 0 on TOS
dup	1	duplicate TOS
iadd	1	integer addition
isub	1	integer subtraction
swap	4	exchange TOS and TOS-1
ifeq	4	conditional branch

real-time scheduling on a multithreaded microcontroller. The follow-up project, jamuth [26], is a commercial version of Komodo.

2.2. Bytecode timing

Simple bytecodes (e.g. iadd, dup, or iload_0) execute in a single cycle on JOP. Slightly more complex bytecodes (e.g. dup_x1) are implemented in a short microcode sequence and executed in constant time. The execution time in clock cycles equals the number of microinstructions executed. As the stack is on-chip, it can be accessed in a single cycle. Table I shows example bytecode instructions, their timing, and their meaning (TOS is top-of-stack).

Access to object, array, and class fields depend on the timing of the main memory. As an example, we give the execution time of the bytecode getstatic. With r_{ws} wait states on the read access to the main memory, the execution time in cycles is computed as follows:

$$t_{getstatic} = 7 + r_{ws}$$

A complete list of all bytecode timings of JOP can be found in [27].

2.3. Method cache

JOP introduced a special instruction cache, the method cache [28], which caches whole methods. A method cache is also integrated in the embedded Java processor SHAP [29], and the CarCore processor [30] also uses method caches. Furthermore, it is considered in jamuth [26] as a time-predictable caching solution[§].

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 a memory with long latency but high bandwidth.

[§] Personal communication with Sascha Uhrig.

DOI: 10.1002/spe

Memory access time determines the cache load time on a miss. For the current implementation, the cache load time is calculated as follows: The wait state c_{ws} for a single word cache load is:

$$c_{ws} = \begin{cases} r_{ws} & : \quad r_{ws} > 1 \\ 1 & : \quad r_{ws} \leq 1 \end{cases}$$

On a method invoke or return, the respective method has to be loaded into the cache on a cache miss. The load time l is:

$$l = \begin{cases} 6 + (n+1)(1+c_{ws}) & : \quad \text{cache miss} \\ 4 & : \quad \text{cache hit} \end{cases}$$

where n is the size of the method in numbers of 32-bit words. As an example, the exact execution time for bytecode invokevirtual is:

$$t = 100 + 2r_{ws} + \begin{cases} r_{ws} - 3 & : \quad r_{ws} > 3 \\ 0 & : \quad r_{ws} \leq 3 \end{cases} + \begin{cases} r_{ws} - 2 & : \quad r_{ws} > 2 \\ 0 & : \quad r_{ws} \leq 2 \end{cases} + \begin{cases} l - 37 & : \quad l > 37 \\ 0 & : \quad l \leq 37 \end{cases}$$

On a method return, the caller method has to be found in the method cache or needs to be loaded. Therefore, the execution time of the return instruction depends on the method size of the caller. The execution time of bytecode return is:

$$t = 21 + \begin{cases} r_{ws} - 3 & : \quad r_{ws} > 3 \\ 0 & : \quad r_{ws} \leq 3 \end{cases} + \begin{cases} l - 9 & : \quad l > 9 \\ 0 & : \quad l \leq 9 \end{cases}$$

CProf supports the simulation of a method cache in a customizable way. The simulation provides the information about 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.

2.4. 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.

DOI: 10.1002/spe

3. CROSS-PROFILING

Bytecode instrumentation is a well-known technique for profiling [31,32]. 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 first describe our representation of the calling context. Second, we discuss how collected profiling data can be processed online and in a customized way. Third, we present our generic instrumentation techniques that enable cross-profiling for embedded Java processors. Fourth, we describe CProf's configuration for the JOP processor used in our evaluation.

3.1. Calling context tree

The CCT was first introduced by Ammons *et al.* in [11] as runtime data structure for calling context profiling. Each node in the CCT represents a calling context and stores the measured dynamic metrics for that calling context; it also refers to a unique identifier of the method in which the metrics were collected. Method identifiers convey class name, method name, method signature, and method size (in bytes).

The parent of a CCT node represents the caller's context, while the children nodes correspond to the callee methods. If the same method is invoked in distinct calling contexts, the different invocations are represented by distinct nodes in the CCT. In contrast, if the same method is invoked multiple times in the same calling context, the dynamic metrics collected during the executions of that method are stored in the same CCT node.

CProf instruments bytecode such that each thread creates a CCT while executing methods. Thanks to the generated code for CCT creation[¶], the instrumentation has access to both the caller's node and the callee's node in the current thread's CCT.

Figure 1 illustrates the CCT of an example cross-profile, assuming one invocation of method f(). The cross-profile was generated using a CProf configuration for the JOP target, which will be described in more detail in Section 3.4. In this example, each CCT node *N* stores the number of method invocations (by the same sequence of callers) and an estimate of the accumulated CPU cycles for the subtree rooted at node *N*.

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.

[¶] 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.

DOI: 10.1002/spe

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

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

void h() { return; }
```

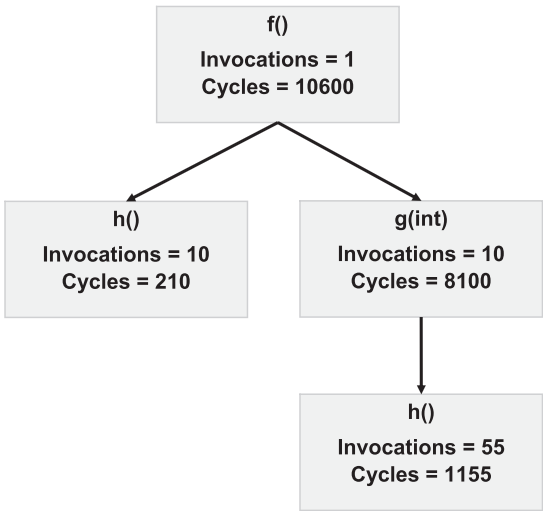

Figure 1. Example CCT, assuming method f() is invoked once. Each CCT node stores the number of method invocations (with the same call stack) and the accumulated cycle estimate for the subtree.

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 [33] 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 2 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

DOI: 10.1002/spe

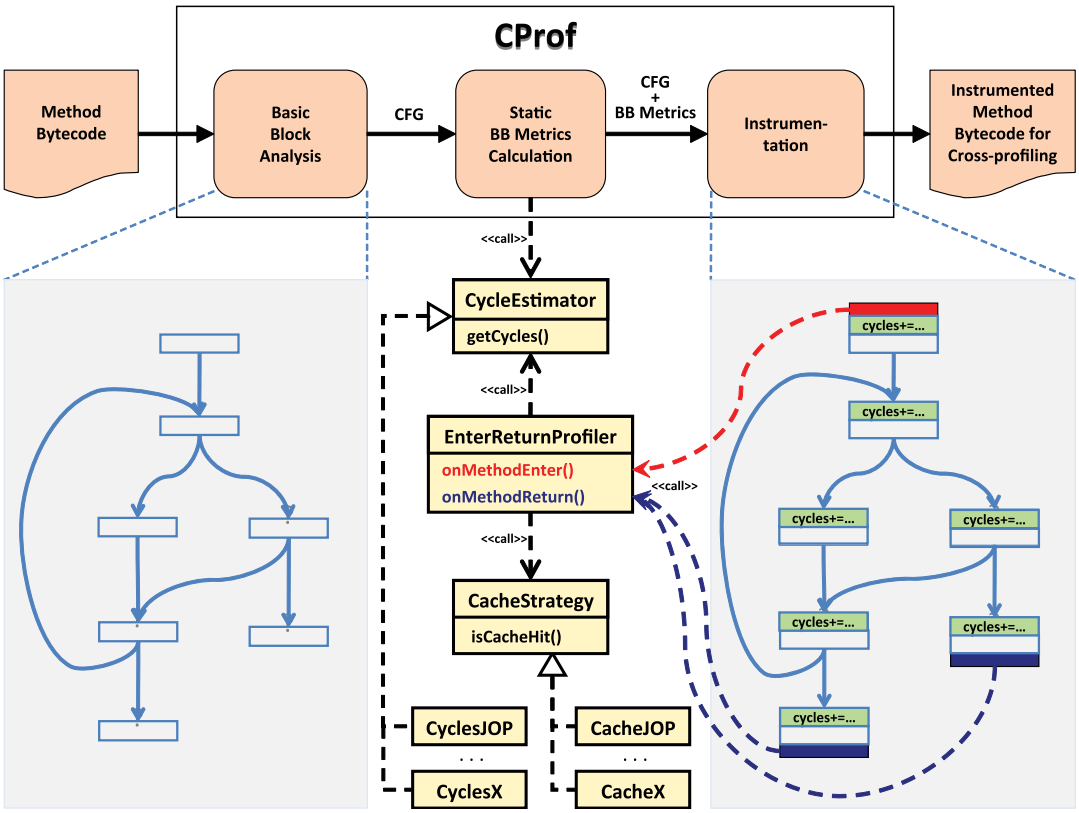

Figure 2. Overview of CProf.

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 returns, explicit throw of exceptions) 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 [34] and is related to the factored control flow graph (FCFG) [35]. In contrast, using the Precise BBA, each bytecode that potentially might throw an exception ends a BB.

When statically calculating metrics for a BB, method invocation and return bytecodes are ignored, as the metrics contribution of these bytecodes may vary depending on the state of the method cache and therefore are not statically known. The metrics contribution of these bytecodes is computed at runtime, when profiling method entry and return.

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 reduce the number of program locations where instrumentation code is inserted, resulting in lower cross-profiling overhead. As long as no exceptions are

DOI: 10.1002/spe

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 2 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 2). 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. Below we focus on the third issue.

Figure 2 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(...)[‖]. As arguments, these methods receive the caller and callee nodes in the CCT, as well as the invocation/return

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

DOI: 10.1002/spe

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 2 shows two concrete implementations, 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 about 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). That is, in certain cases the same method may be called by invokevirtual, invokeinterface, or invokespecial. However, as the CPU cycle consumption of the distinct method invocation bytecodes may differ, knowing that the concrete invocation bytecode can help improve the accuracy of cross-profiling.

As general solution, 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)**. 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. In contrast to CProf, prevailing AOP frameworks, such as AspectJ [36], provides mechanisms for intercepting abnormal method completion. However, because exception throwing is infrequent in embedded Java software, CProf currently lacks such a mechanism.

**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⟩.

DOI: 10.1002/spe

3.4. Customization for the JOP processor

While the software architecture of CProf allows customization and replacement of all components shown in Figure 2, the typical customization for a target processor that fits our execution time model will require only the implementation of the CycleEstimator and CacheStrategy interfaces.

```
public class CacheJOP implements CacheStrategy {
   private final int cacheSize;
   private final int numberOfBlocks;
   private final int blockSize;
   private final MID[] cache;
   private int next;

   public CacheJOP(int cacheSize, int numberOfBlocks) {
      this.cacheSize = cacheSize;
      this.numberOfBlocks = numberOfBlocks;
      this.blockSize = cacheSize / numberOfBlocks;
      cache = new MID[numberOfBlocks];
      next = 0;
   }

   public synchronized boolean isCacheHit(MID mid) {
      // if mid is in the cache, return true
      for (int i = 0; i < numberOfBlocks; ++i) {
         if (cache[i] == mid) return true;
      }

      // store mid in the next slot
      cache[next++] = mid;
      if (next == numberOfBlocks) next = 0;

      // skip the blocks occupied by the method
      int msize = mid.getSize();
      int nbl = (msize + blockSize - 1) / blockSize;
      for (int i = 1; i < nbl; ++i) {
         cache[next++] = null;
         if (next == numberOfBlocks) next = 0;
      }

      return false; // cache miss
   }
}
```

Figure 3. CacheStrategy implementation with FIFO replacement.

DOI: 10.1002/spe

With respect to cross-profiling for the JOP processor, we use CProf's Default BBA. This choice is a reasonable trade-off 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 from the WCET analysis tool [18]. Furthermore, we implemented a CacheStrategy in the class CacheJOP that simulates the hardware method cache of the current JOP processor.

Figure 3 illustrates the CacheJOP implementation. It simulates an instruction cache with FIFO replacement. As CProf uniquely identifies each method by exactly one instance of type MID, reference comparison is sufficient to check whether a method is in the cache (i.e. there is no need to check the equality of MID instances, which would cause higher overhead). The cache implementation is straightforward. If the passed MID instance is in the array simulating the cache contents, a cache hit is returned (true). Otherwise, the MID instance is stored in the next position in the array, the slots needed to store the complete method body are overwritten, and a cache miss is returned (false). The method isCacheHit(...) is synchronized, because we simulate a single cache for all threads in the system.

4. EVALUATION

In this section, we evaluate CProf regarding cross-profile accuracy and runtime overhead. For the accuracy assessment, we compare CProf's cycle estimates, profiled on the host, with the actual CPU cycle consumption on JOP, measured on the target. 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[††] (JBE) and SPEC JVM98 [37]. JBE contains several micro benchmarks and three real-world applications [38], Kfl (a motor control system), Lift (a lift controller), and Udplp (a TCP/IP stack for embedded Java). We use only these three real-world applications in our evaluation. SPEC JVM98 consists of seven benchmarks, which cannot run on the target system (because of resource constraints and because JOP does not provide a file system), but represent larger workloads for the runtime overhead evaluation. We use SPEC 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

[††] http://www.jopwiki.org/JavaBenchEmbedded/.

DOI: 10.1002/spe

Table II. Benchmark execution time and cross-profiling result in clock cycles for JOP with a 4 kB/16 method cache.

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

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 Windows-based notebook, whereas the host environment for the overhead evaluation is a Linux Fedora Core 2 computer (Intel Pentium 4, 2.66 GHz, 1024 MB RAM). For the overhead measurements, we use the Sun JDK 1.7-ea-b24 HotSpot Server VM.

4.2. Accuracy of cross-profiles

To assess that our approach is sound, we compare the CPU cycle estimates from the generated cross-profiles with the actual CPU cycle consumption on JOP. With this experiment the accuracy of execution time estimation through cross-profiling is validated. In this experiment, JOP is clocked at 100 MHz in a low-cost FPGA and the memory access time is 2 clock cycles for reading a 32-bit word and 3 clock cycles for a 32-bit write access. JOP is configured with a 4 kB FIFO instruction cache organized in 16 blocks. We have used CProf with method cache simulation enabled to yield the best estimations. As the embedded applications are designed to not throw exceptions during runtime, the 'Default BBA' is sufficient for our cross-profiling application. Furthermore, the embedded applications do not allocate data during their mission phase. As a consequence, the garbage collector has no impact on the cross-profiling results.

The three embedded benchmarks Kfl, Lift, and Udplp are executed 10 000 times on JOP and the execution time is measured with a CPU cycle counter. The same benchmarks are profiled with CProf, and in the generated cross-profile the cumulative CPU cycle estimate of the benchmark harness (the method test()) is taken, effectively excluding the execution of startup code on the host. Table II shows the execution times on JOP and cross-profiling results in clock cycles. The last column shows the percent error of the cross-profiling estimates. For two benchmarks, the error is well below 1%, for Udplp the error is below 2%.

Although we use WCET values for the bytecode timings, the cross-profiling results underestimate the execution time. The inaccuracies are caused by differences in the Java class libraries on JOP, respectively, on the host, and by a simplified execution time models for some complex bytecodes (e.g. runtime-type checks and casts, floating point arithmetic, etc.). Those bytecodes are implemented in Java and their execution time also depends on the cache state. The WCET analysis tool [39] models those bytecodes as invocations of the static methods. For cross-profiling we model those bytecodes with an average case estimation. Nonetheless, for all measured benchmarks that run on the JOP hardware, the error in the CPU cycle estimates is below 2%[‡‡].

[‡‡] In prior work [16] we reported higher errors. Thanks to improved CPU cycle estimates for some bytecodes, we were able to reduce the error.

DOI: 10.1002/spe

Another reason for the underestimation, especially in the Udplp benchmark, is the write barrier code for incremental garbage collection on JOP. If the bytecodes putfield and putstatic access a reference field, they are substituted upon class loading by a special bytecode that contains the write barrier code. These special bytecodes are implemented in Java so as to ease data sharing with the garbage collector that is also programmed in Java. Consequently, these special bytecodes are slower than the versions for primitive data. The cross-profiler does not cover this difference and treats the execution time of putfield and putstatic as constant, independently of the field type.

4.3. Runtime overhead

Table III presents the results of our overhead evaluation for JBE and SPEC 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.

First, 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 2 but without the invocations of the EnterReturnProfiler. In this setting, we observe an overhead factor of 1.33–9.99. For 'mtrt', we experience the highest overhead. 'mtrt' is known to make extensive use of small methods [40], which makes the CCT maintenance expensive.

Second, we enabled the method entry and return pointcuts, but used a trivial 'no cache' strategy that always assumed a miss ('CProf ER, no cache'). In this setting, CProf produces cross-profiles for an embedded Java processor without method cache. We notice that the invocations of the EnterReturnProfiler increase the overhead factor by 0.21–4.19 (overhead difference between 'CProf ER, no cache' and 'CProf no ER').

Table III. CProf overhead with/without enter-return calls (ER), respectively, with/without cache strategy.

		CProf					
	Orig.	no ER		ER, no cache		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

DOI: 10.1002/spe

Third, we evaluated the real CProf configuration for JOP, using a cache strategy that simulates a method cache of 4 kB 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 2) was not optimized, which explains the high extra overhead. Overall, the execution time in this setting is still reasonable (below 70 ms for the JBE benchmarks); optimizing the cache strategy to reduce cross-profiling overhead is typically not worth the effort in practice.

5. CASE STUDY: CROSS-PROFILING FOR PROCESSOR ARCHITECTURE EXPLORATION

In this section we explore a new use-case for cross-profiling, evaluating the performance benefit of various possible architectural improvements for JOP. Cross-profiling gives as a tool for quick design space exploration without the need to actually implement the architectural changes. Only promising enhancements will be chosen for the next version of JOP.

For simple processor architectures without caches, the effects of different instruction timings can be evaluated by collecting dynamic instruction frequencies [41]. However, when instruction caches are integrated, the execution time of the whole program cannot be predicted with instruction frequencies anymore. The dependency of instruction timings on cache state (different timings for cache hit, respectively, miss) is hard to model statically. Therefore, we use cross-profiling with runtime cache simulation and cycle estimation.

Compared with the collection of instruction traces and post processing them with a processor simulation, cross-profiling has the benefit of aggregating the data during runtime. Accumulating the relevant information during profiling results in less data than collecting complete instruction traces. Furthermore, the combination of trace generation (executing the benchmark) and processor simulation in a single tool simplifies the exploration process.

5.1. Methodology

In order to evaluate the impact of an architectural change on JOP with the aid of CProf, the timing information about the bytecodes and the cache configuration are specified using corresponding CycleEstimator, respectively, CacheStrategy implementations. Afterwards, a set of benchmarks is cross-profiled; an overall cycle estimate is extracted from each cross-profile and compared with the baseline (i.e. the currently available version of JOP) so as to assess the achieved speedup.

For our case study, we use the 3 embedded benchmarks discussed in Section 4.1 (Kfl, Lift, and UdpIp), as well as 6 benchmarks from the SPEC JVM98 suite [37] (_201_compress, _202_jess, _209_db, _213_javac, _222_mpegaudio, and _228_jack). In the generated cross-profiles, the cumulative CPU cycle estimate of the benchmark harness is taken (method test() for the embedded benchmarks, respectively, method SpecApplication.main() for SPEC JVM98).

All chosen benchmarks are single-threaded. We exclude the multi-threaded SPEC JVM98 benchmark _227_mtrt from our evaluation, in order to avoid drawing any false conclusions because of possible inaccuracies in the cross-profiles caused by the different thread-scheduling on the cross-profiling host JVM and on the JOP target.

DOI: 10.1002/spe

In Section 4.2, we validated the soundness of cross-profiling with the three embedded benchmarks. Because the SPEC JVM98 suite requires more resources than provided by JOP, it is impossible to assess the accuracy of our cross-profiles for SPEC JVM98, since the benchmarks cannot be executed on JOP to collect exact execution times. Fortunately, for the evaluation of different architectural changes, the cross-profiling estimates need not be perfectly accurate [42]. We are interested in the relative performance differences between the cross-profiling runs for distinct architectures. Hence, we found it useful to include also some SPEC JVM98 benchmarks in this case study, in order to evaluate the performance impact of architectural changes on a larger set of workloads.

Our results represent performance differences in percent, using the following well-known formula [41], where p is the speedup in percent, t_{base} the base execution time, and t_{enh} the execution time of the enhanced architecture:

$$p = (t_{base}/t_{enh} - 1) \times 100$$

For example, with a speedup of 200% the enhanced architecture is three times faster. In addition, the average clocks per instruction (CPI) are given in tables where it is relevant. The CPI value is calculated by dividing the execution time in clock cycles by the dynamic instruction count.

In order to ease the performance comparison, we also compute the geometric mean of the measurements for all benchmarks. The geometric mean is calculated from the measured execution times and instruction count, and the other metrics (speedup, respectively, CPI) are computed from those values.

5.2. The baseline

In order to explore the performance benefit of different architectural enhancements to JOP, we first establish the baseline by measuring the performance of the current JOP design. Hence, we cross-profile the benchmarks using the current configuration of JOP with a 4 kB instruction cache, organized in 16 blocks, with FIFO replacement strategy. We will assess the effects of our architectural optimizations in comparison with this baseline. Table IV shows the execution time in clock cycles, the dynamic instruction count (IC), and the resulting clocks per instruction (CPI) of the benchmarks cross-profiled using the WCET cycle estimates for JOP.

An interesting result of this first evaluation is the significant difference with respect to the CPI values. The embedded benchmarks Kfl, Lift, and UdpIp, as well as compress, have a lower CPI than the other SPEC JVM98 benchmarks, where the higher CPI values result from a more object-oriented programming style (i.e. shorter methods, more frequent object allocation, etc.). Method invocation on JOP is expensive and shorter methods lead to a higher invocation frequency. Furthermore, floating point operations and operations on 64-bit integers are expensive as well. Those data types are avoided in the embedded applications.

For computer architects that work on in-order RISC pipelines, the CPI values may look excessively high. However, JVM bytecodes are often much more complex than RISC instructions. A JIT compiler will generate several RISC instructions for object-oriented bytecodes, such as field access or method invocation. In JOP the more complex bytecodes are mapped to microcode sequences for a RISC-style stack machine. The microcode instructions (except memory access) execute in a single cycle. Therefore, the CPI at the microcode level is about 1.

DOI: 10.1002/spe

Table IV. Cross-profiling results of the baseline in clock cycles (Time), dynamic instruction count (IC), and clocks per instruction (CPI).

Benchmark	Time (clocks)	IC	CPI
Kfl	5.02×10^7	1.09×10^7	4.62
Lift	5.26×10^7	1.13×10^7	4.66
UdpIp	1.11×10^8	2.06×10^7	5.39
_201_compress	9.28×10^{10}	1.25×10^{10}	7.44
_202_jess	2.91×10^{10}	1.74×10^9	16.72
_209_db	4.10×10^{10}	3.61×10^9	11.33
_213_javac	3.24×10^{10}	1.84×10^9	17.60
_222_mpegaudio	2.90×10^{11}	1.15×10^{10}	25.21
_228_jack	1.69×10^{10}	1.02×10^9	16.48
geo. mean	5.57×10^9	5.46×10^8	10.20

Table V. Influence of the instruction cache size on the performance relative to a 4 kB/16 cache.

Benchmark	1 kB/4 (%)	16 kB/64 (%)	64 kB/256 (%)
Kfl	−7.5	2.9	2.9
Lift	−3.6	0.0	0.0
UdpIp	−5.6	2.5	2.5
_201_compress	−7.4	0.0	0.0
_202_jess	−4.7	0.8	0.8
_209_db	−0.1	0.0	0.0
_213_javac	−30.8	10.1	11.9
_222_mpegaudio	−1.2	0.0	0.0
_228_jack	−15.8	5.4	11.0
geo. mean	−9.0	2.4	3.1

5.3. Variation of the instruction cache

Our first architectural change is the variation of the method cache size. The method cache is split into cache blocks and caches whole methods. The replacement strategy is FIFO. Table V shows the performance differences relative to the standard configuration of JOP given in Table IV. The third and fourth columns show configurations with bigger caches of 16 and 64 kB with 64 and 256 blocks, respectively. A cache of 64 kB is uncommon in embedded processors, and the performance gain is quite small. To check whether the method cache has actually some performance enhancing effect, we also measured a smaller cache with 1 kB and 4 blocks. With this small cache the performance decreases considerably. Therefore, we conclude that, without any other changes in the processor architecture, a method cache of 4 kB with 16 blocks is a good design decision.

Four benchmarks do not benefit from a larger method cache at all. We conclude that for these benchmarks, the methods where most of the execution time is spent fit together into the cache, or most of the invoked methods are very short. In the latter case, the cache load time is hidden by the invoke instruction as cache loading is performed partially in parallel with microcode execution.

DOI: 10.1002/spe

Table VI. Faster invoke instructions with different cache sizes.

Benchmark	1 kB/4 (%)	4 kB/16 (%)	16 kB/64 (%)
Kfl	7.3	21.7	28.9
Lift	5.6	12.4	12.4
UdpIp	7.4	17.6	23.7
_201_compress	4.4	14.7	14.7
_202_jess	15.0	24.3	26.1
_209_db	14.8	14.9	14.9
_213_javac	−23.4	19.0	34.6
_222_mpegaudio	0.4	1.9	1.9
_228_jack	−2.7	21.0	30.8
geo. mean	2.6	16.2	20.5

It has to be noted that the cache size and organization (number of blocks) are configurable in JOP; hence, this variation can be easily explored in the FPGA for the embedded benchmarks. However, on-chip memory in an FPGA is very limited. Thus, for this experiment a big and expensive FPGA would be needed.

5.4. Faster method invocation and return

Invoke instructions for Java methods are complex. The number of arguments and local variables has to be determined, the stack frame manipulated, some state saved onto the stack, and a virtual method lookup has to be performed. This quite complex process is implemented in microcode on JOP and takes about 100 cycles. That number is not so uncommon, as the aJile processor takes about the same number of clock cycles for an invoke instruction [6].

We have investigated the microcode sequence for the invoke and return instructions and found several places where operations can be performed in hardware (e.g. bit manipulation to extract sub fields from the method dispatch data structure). With some hardware support, we assume that the number of cycles for the invoke and return instruction can be cut down by a factor of two. The performance impact of this optimization with cache sizes of 1, 4, and 16 kB is shown in Table VI. A size of 64 kB is not included as it performs similar to a 16 kB cache (see previous experiment). Furthermore, a 64 kB first-level cache is quite large, especially in resource-constrained embedded processors.

The third column of Table VI shows that the performance increases (except for _222_mpegaudio) between 12 and 24%, with a geometric mean of 16% for the standard cache size of 4 kB. Only _222_mpegaudio does not show any significant improvement; the reason is that most of the execution time is spent in just a few methods. The second and fourth columns show the performance changes with different cache sizes. The improved invoke instruction can compensate for the performance decrease due to a small 1 kB method cache, as seen by comparing the second column of Tables V and VI.

It is interesting to note that the cache size has now a higher impact on the performance than without the changed invoke instruction, as shown in the previous experiment. For example, the change from 4 to 16 kB results in a speedup of 10.1% for javac, the faster invoke instruction with 4 kB cache in a speedup of 19%, but the combined effect is a speedup of 34.6%. This effect can be

DOI: 10.1002/spe

explained by the fact that some cache load time is hidden by execution of microcode for the invoke instruction. That is, short methods have no cache load penalty on a miss and bigger caches do not help. When the invoke instruction itself is enhanced, less method load time can be hidden and larger caches help reduce the execution time. Therefore, both changes in the architecture result in more than a linear speedup. This result is also an argument for the dynamic approach of cross-profiling that takes cache influences into account.

5.5. Longer pipeline

The actual pipeline of JOP consists of four stages: bytecode fetch and translation to microcode addresses, microcode fetch, decode, and execute. The first pipeline stage is the limiting factor for the maximum clock frequency. Some experiments with the design showed that the split of bytecode fetch and microcode address mapping results in a 10% higher clock frequency. This additional pipeline stage results in an increase of the execution time of bytecode control instructions (branch and goto) by one cycle.

Table VII shows the resulting increase of CPI due to slower control instructions. The (negative) speedup is between -0.1% and -3.9%. Consequently, the increase of the maximum clock frequency will result in a faster architecture. With a 10% higher clock frequency, the longer pipeline results in a speedup of 8.5%.

It is interesting to note that the embedded benchmarks have a higher branch frequency than the SPEC JVM98 benchmarks. This is an indication that embedded applications have a more complex intra-procedural control flow and a simpler inter-procedural control flow, as they are less object-oriented.

5.6. Advanced instruction fetch

In the current design of JOP the bytecode instruction fetch is performed with 1 byte per cycle in order to achieve a time-predictable architecture. With an additional pipeline stage, fetching of complete bytecodes in a single cycle is possible. The optimization applies to all bytecodes that are longer than 1 byte and are implemented in microcode. Table VIII shows the performance increase when these bytecodes are fetched in a single cycle. The experiment also includes the penalty of the additional clock cycle for control instructions.

Table VII. Longer pipeline with slower control instructions.

Benchmark	Speedup (%)	CPI
Kfl	−3.9	4.81
Lift	−2.8	4.80
UdpIp	−1.8	5.49
_201_compress	−0.9	7.51
_202_jess	−0.7	16.84
_209_db	−1.0	11.44
_213_javac	−0.6	17.70
_222_mpegaudio	−0.1	25.25
_228_jack	−0.7	16.60
geo. mean	−1.4	10.35

DOI: 10.1002/spe

Table VIII. Single cycle bytecode fetch.

Benchmark	Speedup (%)	CPI
Kfl	3.6	4.46
Lift	6.3	4.39
UdpIp	8.0	4.99
_201_compress	10.2	6.75
_202_jess	3.0	16.23
_209_db	5.3	10.77
_213_javac	3.1	17.07
_222_mpegaudio	2.2	24.67
_228_jack	2.8	16.04
geo. mean	4.9	9.73

Table IX. Effect of the combination of all architectural enhancements.

	Speedup (%)		CPI	
	4 kB/16	16 kB/64	4 kB/16	16 kB/64
Kfl	27.0	34.8	3.64	3.43
Lift	20.4	20.4	3.87	3.87
UdpIp	28.8	36.2	4.19	3.96
_201_compress	28.3	28.3	5.80	5.80
_202_jess	29.0	30.9	12.96	12.77
_209_db	21.9	21.9	9.30	9.30
_213_javac	23.4	40.3	14.26	12.54
_222_mpegaudio	4.1	4.2	24.21	24.21
_228_jack	25.1	35.7	13.17	12.15
geo. mean	22.9	27.6	8.30	8.00

5.7. Combined effect

As a summary, the combined effect of all the mentioned architectural enhancements is given in Table IX. For a 16 kB method cache, the performance gain is between 20 and 40%, again with the exception of _222_mpegaudio. The geometric mean speedup is 28%. As before, it also includes the slower control instructions due to a longer pipeline. Therefore, the real speedup, with a clock frequency improvement of 10%, is up to 54% for the _213_javac benchmark and the geometric mean speedup is 40%.

6. RELATED WORK

In this section we describe the related work in two areas, (cross-)profiling, as well as computer architecture evaluation.

DOI: 10.1002/spe

6.1. Profiling

Cross-profiling techniques have been used to simulate parallel computers [43]. As the host processor may have a different instruction set from the target processor, cross-profiling tries to match up the basic 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 are 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 [44] 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. The agent is implemented in native code using the JVM Profiler Interface (JVMPI) (the JVMPI [45] has been deprecated in JDK 1.5 and was replaced by the JVMTI [10]), 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.

There are several JVM hardware implementations, such as picoJava [24], aJile's JEMCore [4], Komodo [25], or FemtoJava [46]. Although simulation tools are available for the processor designs, profiling is not always possible and made only at the latest stage of development, that is, 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.

There is a large body of related work dealing with different techniques to generate CCTs [11,47–51], highlighting the importance of the CCT for calling context profiling. Generating complete CCTs reflecting every method call may cause high overhead, as stressed in [50,51], and much related work on calling context profiling has focused on efficiently approximating the CCT with sampling techniques [47,48,50].

Most prevailing approaches to calling context profiling depend on a modified JVM or on native code, limiting portability. For example, adaptive bursting [50] relies on the JVM Profiler Interface (JVMPI), the predecessor of the JVMTI [10], which requires profiling agents to be written in native code. Probabilistic Calling Context (PCC) [51] is based on a modified Jikes RVM [52], thus, preventing its use on other standard JVMs. In contrast, we strive for maximum portability and compatibility with standard JVMs thanks to our approach based on bytecode instrumentation, so as to generate calling context cross-profiles with CPU cycle estimates for the target processor. Nonetheless, while several other approaches, such as PCC, exclude the Java class library from profiling, our approach guarantees complete method coverage.

CProf performs complete cross-profiling, tracking all invoked methods. In contrast, sampling-based profilers are activated only periodically, in order to reduce profiling overhead [31,48,53]. The framework presented in [53] uses code duplication combined with compiler-inserted, counter-based sampling. A second version of the code is introduced which contains all computationally expensive instrumentation. The original code is minimally instrumented to allow control to transfer in and out of the duplicated code in a fine-grained manner, based on instruction counting. This approach achieves low overhead, as execution proceeds most of the time inside the lightly instrumented code portions. In [54], a hybrid approach combines timer-based and counter-based sampling.

DOI: 10.1002/spe

Such an approach reduces the overhead and increases the accuracy compared with pure counter-based sampling. For typical workloads to be executed on embedded Java systems, the overhead of complete cross-profiling using CProf is not an issue. Hence, we have not yet explored reducing cross-profiling overhead with sampling techniques.

In [31] a portable sampling profiler for standard JVMs is presented, which relies on bytecode counting. A profiling agent is periodically invoked in a deterministic way after the execution of a certain number of bytecodes. The profiling data structure generated by the sampling profiler can be regarded as a partial CCT covering only a subset of the executed calling contexts.

In [55] the authors present a fast partitioning algorithm based on profiles to remotely execute parts of an embedded Java application on a server so as to reduce the energy consumption on the embedded device. The partitioning algorithm is executed on the embedded device. Our cross-profiling approach helps in identifying and optimizing hotspots before the application is deployed.

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 (`http://www.aspectj.org/`) also works at the bytecode level and AspectJ provides pointcuts for method entry and return, but not for basic block entry. In [56] an extension to AspectJ uses control-flow analysis to determine loop pointcuts used to parallelize loops. Eos-T [57], 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.

The use of AspectJ for profiling is explored in [58], yielding mixed results. While it is possible to implement profilers concisely with aspects, unfortunately the AspectJ weaver does not provide support for comprehensive aspect weaving (it prevents weaving the Java class library), thus resulting in incomplete profiles. Even though the work presented in [59,60] removes this limitation, the AspectJ language itself lacks a number of pointcuts, such as those necessary to capture basic blocks as in CProf, thus limiting its applicability for cross-profiling. Some approaches, such as *abc* [61] or *Nu* [62], ease the extension of AspectJ with new pointcuts [63–65], or enable extensions using an intermediate language and explicit join points [66]. Unfortunately, these approaches do not support full method coverage, thus limiting their applicability to implement extensions for cross-profiling.

CProf is related to the profiler JP described in [32,67,68], which generates largely platform-independent profiles using the number of executed bytecodes as dynamic metric. In contrast to JP, CProf generates cross-profiles that estimate CPU cycles on an embedded target system. Hence, CProf can be regarded as a generalization of JP, which uses the constant cycle estimate of '1' for all bytecodes.

6.2. Computer architecture evaluation

Quantitative evaluation of computer architectures is mainly performed by simulation [42]. Skadron *et al.* argue that current simulation tools are built in an *ad hoc* manner and that tool development is error-prone. Furthermore, computer architecture research focuses on architectures where simulation models are available. The authors argue that research in simulation frameworks and benchmark methodologies is needed. In line with their argument, we introduce the new approach to computer architecture evaluation with cross-profiling.

Using benchmarks intended to evaluate real hardware for computer architecture simulation leads to impractical simulation time on cycle-accurate simulators [69]. The result is that usually only

DOI: 10.1002/spe

subsets of the benchmarks are used to reduce the simulation time. Yi *et al.* argue that besides standardizing the subsets, higher level abstractions in the simulation are a valuable option, especially in an early stage of design space exploration. Our cross-profiling approach follows their advice and simulates at the level of basic blocks, method entry, and return, instead of performing instruction-level simulation.

SimpleScalar [70] is a popular processor performance simulator. SimpleScalar contains, besides the simulator, the full tool chain (GCC compiler, libraries, assembler, and linker). SimpleScalar models a five-stage, out-of-order pipeline and is highly configurable. SimpleScalar models the microarchitecture, but not an entire system to run an operating system (OS). With our cross-profiling approach, we also omit the low-level functions of the OS, but include the rich standard Java class library.

Another approach to architectural evaluation for designs implemented in an FPGA is shown in [71]. The execution time of a bytecode is artificially increased and a new design synthesized. The actual increase in the execution time can be used, with some transformations of Amdahl's law [41], to estimate the performance when the instruction timing is enhanced. The microcoded design of JOP simplifies the increase of the execution time; just no-operation instructions need to be inserted into the microcode sequences for the bytecode under evaluation. The turnaround time for these kinds of experiments is in the range of minutes. However, this approach is limited to benchmarks that can be executed on the target hardware. With cross-profiling it is possible to evaluate architectural changes with larger benchmark suites.

7. CONCLUSION

In this article 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 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 Optimized Processor JOP, which features a method cache. Using JOP as target platform, we have shown that our approach reconciles high cross-profile accuracy (error below 2%) and moderate overhead, which is an order of magnitude below the overhead caused by typical simulators.

As a case study, we have presented an approach to computer architecture evaluation for embedded Java processors using cross-profiling. With the aid of CProf, we evaluated the performance impact of various architectural enhancements to JOP using standard Java benchmarks. We have shown that the combination of several enhancements to the architecture of JOP will result in a speedup of about 40%. All the proposed changes are still time-predictable and will not defeat the intention of JOP to serve as a real-time Java processor.

DOI: 10.1002/spe

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

With respect to ongoing research, we are working on techniques to simulate activities of the target runtime system (e.g. garbage collection) on the host. Furthermore, we are investigating the simulation of data caches. In addition, we are exploring the possibility of cross-profiling for embedded Java systems that rely on just-in-time compilation (in contrast to Java processors).

REFERENCES

1. Gosling J, Joy B, Steele GL, Bracha G. *The Java Language Specification* (3rd edn) (*The Java Series*). Addison-Wesley: Reading, MA, 2005.
2. Bollella G, Brosgol B, Dibble P, Furr S, Gosling J, Hardin D, Turnbull M. *The Real-time Specification for Java*. Addison-Wesley: Reading, MA, U.S.A., 2000.
3. Lindholm T, Yellin F. *The Java Virtual Machine Specification* (2nd edn). Addison-Wesley: Reading, MA, U.S.A., 1999.
4. Hardin DS. Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine. *Proceedings of the Fourth International Symposium on Object-oriented Real-time Distributed Computing*. IEEE Computer Society: Silver Spring, MD, 2001; 53.
5. Imsys. *Im1101c* (*the Cjip*) *Technical Reference Manual/v0.25*, 2004.
6. Schoeberl M. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture* 2008; **54**(1–2):265–286.
7. Henties T, Hunt JJ, Locke D, Nilsen K, Schoeberl M, Vitek J. Java for safety-critical applications. *Second International Workshop on the Certification of Safety-critical Software Controlled Systems* (*SafeCert 2009*), York, U.K., March 2009.
8. Bate I, Bernat G, Murphy G, Puschner P. Low-level analysis of a portable Java byte code WCET analysis framework. *Proceedings of the 7th International Conference on Real-time Computing Systems and Applications*, December 2000; 39–48.
9. Mentor Graphic Inc. ModelSim. Web pages at: http://www.model.com/ [17 October 2009].
10. Sun Microsystems, Inc., JVM Tool Interface (JVMTI) version 1.1, 2006. Web pages at: http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html [17 October 2009].
11. Ammons G, Ball T, Larus JR. Exploiting hardware performance counters with flow and context sensitive profiling. *PLDI '97*: *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*. ACM Press: New York, 1997; 85–96.
12. Kozyrakis C, Patterson D. A new direction for computer architecture research. *Computer* 1998; **31**(11):24–32.
13. Binder W, Hulaas J, Moret P. Advanced Java bytecode instrumentation. *PPPJ'07*: *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*. ACM Press: New York, NY, U.S.A., 2007; 135–144.
14. Kiczales G, Lamping J, Menhdhekar A, Maeda C, Lopes C, Loingtier J-M, Irwin J. Aspect-oriented programming. *Proceedings of European Conference on Object-oriented Programming*, vol. 1241, Akşit M, Matsuoka S (eds.). Springer: Berlin, Heidelberg, New York, 1997; 220–242.
15. Binder W, Schoeberl M, Moret P, Villazón A. Cross-profiling for embedded Java processors. *Fifth International Conference on the Quantitative Evaluation of Systems* (*QEST-2008*), IEEE Computer Society: Saint-Malo, France, September 2008; 287–296.
16. Binder W, Villazón A, Schoeberl M, Moret P. Cache-aware cross-profiling for Java processors. *International Conference on Compilers*, *Architecture*, *and Synthesis for Embedded Systems* (*CASES-2008*). ACM: Atlanta, GA, U.S.A., October 2008; 127–136.
17. Schoeberl M, Binder W, Moret P, Villazon A. Design space exploration for Java processors with cross-profiling. *Proceedings of the 6th International Conference on the Quantitative Evaluation of Systems* (*QEST 2009*). IEEE Computer Society: Budapest, Hungary, 2009; 109–118.
18. Schoeberl M, Pedersen R. WCET analysis for a Java processor. *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems* (*JTRES 2006*). ACM Press: New York, NY, U.S.A., 2006; 202–211.
19. Harmon T. Interactive worst-case execution time analysis of hard real-time Systems. *PhD Thesis*, University of California, Irvine, 2009.
20. Bogholm T, Kragh-Hansen H, Olsen P, Thomsen B, Larsen KG. Model-based schedulability analysis of safety critical hard real-time Java programs. *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems* (*JTRES 2008*). ACM: New York, NY, U.S.A., 2008; 106–114.
21. Huber B. Worst-case execution time analysis for real-time Java. *Master's Thesis*, Vienna University of Technology, Austria, 2009.

DOI: 10.1002/spe

22. Gruian F, Westmijze M. Bluejep: A flexible and high-performance java embedded processor. *JTRES '07*: *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems*. ACM: New York, NY, U.S.A., 2007; 222–229.
23. Zabel M, Preusser TB, Reichel P, Spallek RG. Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures*, *Methods and Tools* (*DSD 2007*), Luebeck, Germany, August 2007; 59–62.
24. O'Connor JM, Tremblay M. picoJava-I: The java virtual machine in hardware. *IEEE Micro* 1997; **17**(2):45–53.
25. Kreuzinger J, Brinkschulte U, Pfeffer M, Uhrig S, Ungerer T. Real-time event-handling and scheduling on a multithreaded java microcontroller. *Microprocessors and Microsystems* 2003; **27**(1):19–31.
26. Uhrig S, Wiese J. Jamuth: An IP processor core for embedded Java real-time systems. *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems* (*JTRES 2007*). ACM Press: New York, NY, U.S.A., 2007; 230–237.
27. Schoeberl M. JOP *Reference Handbook*: *Building Embedded Systems with a Java Processor*. August 2009. ISBN 978-1438239699. CreateSpace.
28. Schoeberl M. A time predictable instruction cache for a Java processor. *On the Move to Meaningful Internet Systems 2004*: *Workshop on Java Technologies for Real-time and Embedded Systems* (*JTRES 2004*) (*Lecture Notes in Computer Science*, vol. 3292). Springer: Agia Napa, Cyprus, 2004; 371–382.
29. Preusser TB, Zabel M, Spallek RG. Bump-pointer method caching for embedded Java processors. *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems* (*JTRES 2007*). ACM: New York, NY, U.S.A., 2007; 206–210.
30. Metzlaff S, Uhrig S, Mische J, Ungerer T. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. *Proceedings of the 9th Workshop on Memory Performance* (*MEDEA 2008*). ACM: New York, NY, U.S.A., 2008; 38–45.
31. Binder W. Portable and accurate sampling profiling for Java. *Software*: *Practice and Experience* 2006; **36**(6):615–650.
32. Binder W, Hulaas J, Moret P, Villazón A. Platform-independent profiling in a virtual execution environment. *Software*: *Practice and Experience* 2009; **39**(1):47–79.
33. Feeley M. Polling efficiently on stock hardware. *The 1993 ACM SIGPLAN Conference on Functional Programming and Computer Architecture*, Copenhagen, Denmark, June 1993; 179–187.
34. Binder W, Hulaas JG, Villazón A. Portable resource control in Java. *ACM SIG-PLAN Notices* 2001; **36**(11): 139–155. *Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming*, *Systems*, *Languages and Applications* (*OOPSLA'01*).
35. Choi J-D, Grove D, Hind M, Sarkar V. Efficient and precise modeling of exceptions for the analysis of Java programs. *Proceedings of the ACM SIGPLAN—SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM Press: New York, 1999; 21–31.
36. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An overview of AspectJ. *Proceedings of the 15th European Conference on Object-oriented Programming* (*ECOOP-2001*) (*Lecture Notes in Computer Science*, vol. 2072), Knudsen JL (ed.). Springer: Berlin, 2001; 327–353.
37. SPEC. The spec jvm98 benchmark suite. 5 Available at: http://www.spec.org/ [August 1998].
38. Schoeberl M. Application experiences with a real-time Java processor. *Proceedings of the 17th IFAC World Congress*, Seoul, Korea, July 2008; 9320–9325.
39. Huber B, Schoeberl M. Comparison of implicit path enumeration and model checking based WCET analysis. *Proceedings of the 9th International Workshop on Worst-case Execution Time* (*WCET*) *Analysis*. OCG: Dublin, Ireland, 2009; 23–34.
40. Gregg D, Power JF, Waldron J. A method-level comparison of the java grande and SPEC JVM98 benchmark suites. *Concurrency and Computation*: *Practice and Experience* 2005; **17**(7–8):757–773.
41. Hennessy J, Patterson D. *Computer Architecture*: *A Quantitative Approach* (4th edn). Morgan Kaufmann Publishers: Los Altos, CA, 2006.
42. Skadron K, Martonosi M, August DI, Hill MD, Lilja DJ, Pai VS. Challenges in computer architecture evaluation. *Computer* 2003; **36**(8):30–36.
43. Covington R, Dwarkadas S, Jump J, Sinclair J, Madala S. The efficient simulation of parallel computer systems. *International Journal in Computer Simulation* 1991; **1**:31–58.
44. ProSyst. JProfiler. Web pages at: http://www.prosyst.com/products/tools_jprofiler.html [17 October 2009].
45. Sun Microsystems, Inc., Java Virtual Machine Profiler Interface (JVMPI), 2000. Web pages at: http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/ [17 October 2009].
46. Beck AC, Carro L. Low power Java processor for embedded applications. *Proceedings of the 12th IFIP International Conference on Very Large Scale Integration*, Darmstadt, Germany, December 2003; 213–228.
47. Arnold M, Sweeney PF. Approximating the calling context tree via sampling. *Research Report*, IBM, July 2000; 1–10.
48. Whaley J. A portable sampling-based profiler for Java virtual machines. *Proceedings of the ACM 2000 Conference on Java Grande*. ACM Press: New York, 2000; 78–87.
49. Spivey JM. Fast, accurate call graph profiling. *Software*: *Practice and Experience* 2004; **34**(3):249–264.

DOI: 10.1002/spe

50. Zhuang X, Serrano MJ, Cain HW, Choi J-D. Accurate, efficient, and adaptive calling context profiling. *PLDI '06*: *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM: New York, NY, U.S.A., 2006; 263–271.
51. Bond MD, McKinley KS. Probabilistic calling context. *OOPSLA '07*: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object Oriented Programming*, *Systems and Applications*. ACM: New York, NY, U.S.A., 2007; 97–112.
52. Alpern B, Attanasio CR, Barton JJ, Burke B, Cheng P, Choi J-D, Cocchi A, Fink SJ, Grove D, Hind M, Hummel SF, Lieber D, Litvinov V, Mergen MF, Ngo N, Russell JR, Sarkar V, Serrano MJ, Shepherd JC, Smith SE, Sreedhar VC, Srinivasan H, Whaley J. The jalapeño virtual machine. *IBM Systems Journal* 2000; **39**(1):211–238.
53. Arnold M, Ryder BG. A framework for reducing the cost of instrumented code. *SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, U.S.A., 2001; 168–179.
54. Arnold M, Grove D. Collecting and exploiting high-accuracy call graph profiles in virtual machines. *CGO '05*: *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society: Washington, DC, U.S.A., 2005; 51–62.
55. Tallam S, Gupta R. Profile-guided Java program partitioning for power aware computing. *Eighteenth International Parallel and Distributed Processing Symposium* (*IPDPS*). IEEE Computer Society: Los Alamitos, CA, U.S.A., 2004; 156b.
56. Harbulot B, Gurd JR. A join point for loops in AspectJ. *AOSD '06*: *Proceedings of the 5th International Conference on Aspect-oriented Software Development*. ACM: New York, NY, U.S.A., 2006; 63–74.
57. Rajan H, Sullivan K. Aspect language features for concern coverage profiling. *AOSD '05*: *Proceedings of the 4th International Conference on Aspect-oriented Software Development*. ACM: New York, NY, U.S.A., 2005; 181–191.
58. Pearce DJ, Webster M, Berry R, Kelly PHJ. Profiling with aspectj. *Software*: *Practice and Experience* 2007; **37**(7): 747–777.
59. Villazón A, Binder W, Moret P. Aspect weaving in standard Java class libraries. *PPPJ '08*: *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*. ACM: New York, NY, U.S.A., 2008; 159–167.
60. Villazón A, Binder W, Moret P. Flexible calling context reification for aspect-oriented programming. *AOSD '09*: *Proceedings of the 8th International Conference on Aspect-oriented Software Development*. ACM: Charlottesville, VA, U.S.A., 2009; 63–74.
61. Avgustinov P, Christensen AS, Hendren L, Kuzins S, Lhoták J, Lhoták O, de Moor O, Sereni D, Sittampalam G, Tibble J. Optimising AspectJ. *PLDI '05*: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM: New York, NY, U.S.A., 2005; 117–128.
62. Dyer R, Rajan H. Nu: A dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. *AOSD '08*: *Proceedings of the 7th International Conference on Aspect-oriented Software Development*. ACM: New York, NY, U.S.A., 2008; 191–202.
63. Akai S, Chiba S, Nishizawa M. Region pointcut for Aspect J. *ACP4IS '09*: *Proceedings of the 8th Workshop on Aspects*, *Components*, *and Patterns for Infrastructure Software*. ACM: New York, NY, U.S.A., 2009; 43–48.
64. Bodden E, Havelund K. Racer: Effective race detection using AspectJ. *International Symposium on Software Testing and Analysis* (*ISSTA*). Seattle, WA, 20–24 July 2008. ACM: New York, NY, U.S.A., 2008; 155–165.
65. De Fraine B, Südholt M, Jonckers V. StrongAspectJ: Flexible and safe pointcut/advice bindings. *AOSD '08*: *Proceedings of the 7th International Conference on Aspect-oriented Software Development*. ACM: New York, NY, U.S.A., 2008; 60–71.
66. Rajan H. A case for explicit join point models for aspect-oriented intermediate languages. *VMIL '07*: *Proceedings of the 1st Workshop on Virtual Machines and Intermediate Languages for Emerging Modularization Mechanisms*. ACM: New York, NY, U.S.A., 2007; 4.
67. Binder W. A portable and customizable profiling framework for Java based on bytecode instruction counting. *Third Asian Symposium on Programming Languages and Systems* (*APLAS 2005*) (*Lecture Notes in Computer Science*, vol. 3780), Tsukuba, Japan. Springer: Berlin, November 2005; 178–194.
68. Moret P, Binder W, Villazón A. CCCP: Complete calling context profiling in virtual execution environments. *PEPM '09*: *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM: Savannah, GA, U.S.A., 2009; 151–160.
69. Yi JJ, Eeckhout L, Lilja DJ, Calder B, John LK, Smith JE. The future of simulation: A field of dreams. *Computer* 2006; **39**(11):22–29.
70. Austin T, Larson E, Ernst D. SimpleScalar: An infrastructure for computer system modeling. *Computer* 2002; **35**(2):59–67.
71. Schoeberl M. Architecture for object oriented programming languages. *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems* (*JTRES 2007*). ACM Press: Vienna, Austria, 2007; 57–62.

DOI: 10.1002/spe