Design Space Exploration for Java Processors with Cross-Profiling

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

Walter Binder Philippe Moret Alex Villazón
Faculty of Informatics
University of Lugano, Switzerland
walter.binder@unisi.ch
philippe.moret@lu.unisi.ch
alex.villazon@lu.unisi.ch

***Abstract*—Most processors are used in embedded systems, where the processor architectures are diverse due to optimizations for different application domains. The main challenge for embedded system processors is the right balance between performance and chip size, which directly relates to cost. An early estimation of the performance for a new design is of paramount importance. In this paper we propose cross-profiling for that performance estimation, which can be accomplished very early in the design phase. We evaluate our approach in the context of a Java processor for embedded systems using cross-profiling on a standard desktop Java virtual machine. We explore the performance impact of various processor design choices and optimizations, such as different caches strategies or pipeline organizations, and come up with an improved processor design that yields speedups of up to 40% on standard Java benchmarks. Comparing the generated cross-profiles with the execution of benchmarks in real hardware confirms that our approach is sound.**

***Keywords*-processor architecture evaluation, embedded systems, cross-profiling, Java virtual machine**

I. Introduction

Today, most microprocessors in use are embedded processors. Embedded systems as well as the employed processor types are diverse, and architecture research for embedded systems is a very active area. It is important to estimate the effects of architectural design choices on the performance, with domain specific applications, very early in the development phase [1].

In this paper we propose cross-profiling for processor architecture evaluation. With cross-profiling it is possible to collect evaluation data for realistically sized programs even before the target hardware is available. Cross-profiling is also up to 33000 times faster than VHDL simulation [2].

We validate our approach in the context of a Java processor for embedded systems with cross-profiling on a standard desktop Java virtual machine (JVM) [3]. While Java is an emerging language for classic embedded systems (e.g., in the automotive and airborne domain), almost all mobile phones already contain a JVM to execute so-called MIDlets.

Java is also considered as future language for safety-critical applications [4]. 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 hardware-based JVM simplifies this process, as only code in one language needs to be considered for the certification. Furthermore, worst-case execution time (WCET) for the tasks needs to be known. A Java processor also simplifies WCET analysis of Java programs as it can be performed at the bytecode level [5]. All of the processor enhancements explored in this paper are still analyzable with respect to the WCET.

Cross-profiling is a technique that executes a program on a host platform, but collects profiling information for the target platform. Cross-profiling not only eases the detection of performance bottlenecks in embedded Java software in an early development phase without the need to deploy any software on the embedded device (which may not be available in an early stage of system development), but it also allows estimating the performance of different Java processor designs without requiring these designs to be implemented. 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 model is afterwards implemented in hardware.

One important advantage of our approach to exploring the design space of Java processors is the applicability of cross-profiling to standard workloads, such as the SPEC JVM98 [6] or DaCapo [7] benchmark suites. That is, there are lots of benchmarks that can be used for evaluating processor design options, whereas most of these standard workloads could not execute on an embedded system, for example, because of memory limitations or because of the lack of a file system.

In summary, the availability of accurate cross-profiling techniques allows us to quantitatively explore many different processor design options on a large number of standard workloads in a short period of time, and to spend implementation effort only on the most promising design alternative. In this way, cross-profiling significantly increases the productivity of the embedded processor architect. The original, scientific contributions of this paper are two-fold:

1) We apply cross-profiling as an efficient tool for Java processor architecture evaluation.
2) The soundness and benefits of our approach are demonstrated in the context of a Java processor for embedded real-time systems.

The remainder of this paper is structured as follows. Section II provides background information on Java processors and on our cross-profiling technique. In Section III we discuss related work, comparing our approach to processor architecture evaluation with other techniques. In Section IV the evaluation methodology is described. Section V assesses the accuracy of the generated cross-profiles and confirms the soundness of our approach. The evaluation of architectural enhancements in a Java processor is given in Section VI. The paper is concluded in Section VII.

II. Background

In this section we give an overview of the Java Optimized Processor (JOP) [8], which represents the basic processor architecture for the evaluation in this paper, and summarize our cross-profiling techniques [2], [9].

A. The Java Optimized Processor JOP

Several embedded Java processors are available from academia and industry [8], [10]–[14].

As all these processors execute Java bytecode, a JVM-based cross-profiling tool is a good match with a Java processor. We have chosen JOP [8] for our evaluation of the proposed computer architecture exploration. JOP is a simple processor, open-source, and 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 [15]. The SHAP Java processor [11], although now with a different pipeline structure and hardware assisted garbage collection, also has its roots in the JOP design.

JOP was designed as a real-time processor with time-predictable execution of Java bytecodes. The accurate cycle timing enabled the development of several worst-case execution time (WCET) analysis tools for embedded Java [16]–[19]. The timing model of JOP, which is used by the WCET analysis tools, simplifies our task of cross-profiling. JOP is implemented in a field-programmable gate array (FPGA). Therefore, adaption of the architecture for different domains is a valuable option.

JOP was the first Java processor featuring a dedicated instruction cache that caches whole method bodies, the so-called method cache [20]. A method cache ensures that instruction cache misses may occur only upon method invocation and return, easing WCET analysis and also cross-profiling. The Java processor SHAP [21] and the CarCore processor [22] also use method caches.

B. Cross-Profiling

Cross-profiling executes applications in a *host* environment and yields profiles that estimate dynamic metrics, such as CPU cycle consumption, for a different *target* environment. In the case of the JVM, both the host and the target environment execute the same instruction set, JVM bytecodes [3]. However, the JVM may be implemented in fundamentally different ways on the host and on the target. We run our cross-profiler in a typical environment for software development, comprising a standard PC with any state-of-the-art JVM implementation that relies on just-in-time compilation and supports dynamic optimization of executing applications. In contrast, the target environment is an embedded Java processor that implements most JVM bytecodes in hardware.

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

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

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

The cross-profiler yields calling-context-sensitive profiles, estimating CPU cycle consumption for each executed calling context. It relies on the Calling Context Tree (CCT) [23] to store dynamic metrics separately for each calling context. While the CCT enables a detailed analysis of program performance and helps locate hotspots in the program code, for processor architecture design space exploration we are primarily interested in the total CPU cycle estimation for given workloads. This information can be easily obtained from the generated cross-profiles by summing up the CPU cycle consumption for the calling contexts in the CCT subtree of interest.

Regarding overhead, our cross-profiler significantly outperforms the target hardware [2], [9]. It runs standard Java benchmarks, such as SPEC JVM98 [6] or DaCapo [7], without any problems, although these benchmarks cannot execute on the target. Cross-profiling is several orders of magnitude faster than VHDL simulators.

C. Prior Work

This paper builds on our prior work on cross-profiling. In [2] we introduced cross-profiling for embedded Java processors. The approach presented in [2] is based on constant cycle estimates for all bytecodes. As a major limitation, it does not take the presence of hardware caches into account, which requires a runtime simulation of the cache, resulting in different cycle estimates depending on cache hit or miss.

In [9] we extend our cross-profiler into the customizable cross-profiling framework CProf,[1] which supports pluggable CPU cycle estimation models, as well as pluggable method cache simulation strategies (e.g., for the current JOP processor, a FIFO method cache simulation is used). The cycle consumption of method invocation and return is computed at runtime depending on the cache state.

In this paper, we leverage the CProf framework for exploring the design space of embedded Java processors, in order to select the most promising hardware optimizations for the next version of the JOP processor.

Another approach to architectural evaluation for designs implemented in an FPGA is shown in [24]. 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 [25], 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 those 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.

III. Related Work

Quantitative evaluation of computer architectures is mainly performed by simulation [26]. 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.

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

Using benchmarks intended to evaluate real hardware for computer architecture simulation leads to impractical simulation time on cycle-accurate simulators [27]. The result is that usually only 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.

For simple processor architectures without caches, the effects of different instruction timings can be evaluated by collecting dynamic instruction frequencies [25]. 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.

The probably most popular processor performance simulator is SimpleScalar [28]. 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.

A architecture description languages (ADL) enables an integrated approach to computer architecture evaluation. A single specification of the architecture can be used to automatically generate tools, such as compiler backends [29], assemblers, and linkers; even hardware descriptions can be synthesized. A simulation tool based on an ADL is presented in [30]. Although the reported simulation speed peaks at 800 MHz for a 5-stage MIPS R2000 core, the average simulation frequency is only 47 MHz on a 2200 MHz AMD Athlon processor.

Cross-profiling techniques have been used to simulate parallel computers [31]. As the host processor may have a different instruction set than 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.

IV. Evaluation Methodology

The cross-profiler CProf relies on the same bytecode timing information of JOP that is also used for WCET analysis [16]. In JOP most bytecodes are implemented in microcode, either as single microcode instructions or as short sequences of microcode instructions. For these bytecodes, the execution time is known cycle-accurately. A few bytecodes (e.g., the new bytecode for object allocation) are implemented as special static methods in Java; we call them *system methods*. Some system methods make use of certain low-level JOP primitives to directly access memory locations. The WCET analysis tool models the execution time for the bytecodes implemented as system methods by replacing the bytecodes with invocations of the corresponding system methods. This is possible, because the WCET analysis tool abstractly interprets code, but does not actually execute it. However, because we cannot execute system methods on a standard JVM (because of the usage of low-level JOP primitives), our cross-profiler relies on CPU cycle estimates for the bytecodes implemented as system methods. Nonetheless, the cross-profiler achieves good accuracy, because the bytecodes suffering from possibly imprecise CPU cycle estimates are executed relatively infrequently. To evaluate the impact of an architectural change on JOP, the timing information of the bytecodes and the cache configuration are specified accordingly.

Our evaluation is based on 3 embedded benchmarks, Kfl, Lift, and UdpIp, as well as on 6 benchmarks from the SPEC JVM98 suite [6].

The embedded benchmarks Kfl and Lift are based on real-world applications [32]. The benchmark UdpIp uses a TCP/IP stack, implemented in Java, in order to simulate a UDP-based client/server application. Each embedded benchmark is executed 10000 times, and in the generated cross-profile the cumulative CPU cycle estimate of the benchmark harness (the method test()) is taken, effectively excluding the execution of startup code on the host.

From SPEC JVM98, we consider the benchmarks _201_compress, _202_jess, _209_db, _213_javac, _222_mpegaudio, and _228_jack. As with the embedded benchmarks, in the generated cross-profiles the startup code is excluded and the cumulative CPU cycle estimate of the method SpecApplication.main() is used.

All chosen benchmarks are single-thread. 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.

JOP's execution environment is a typical embedded system without a filesystem and with only 1 MB memory. In contrast to the 3 embedded benchmarks, the chosen SPEC JVM98 benchmarks cannot be executed on the target hardware. Hence, only the embedded benchmarks can be used for assessing the accuracy of our cross-profiling approach to processor design space exploration. Nonetheless, for assessing the impact of architectural enhancements, we use all benchmarks.

Table I
BENCHMARK EXECUTION TIME AND CROSS-PROFILING RESULTS IN CLOCK CYCLES FOR JOP WITH A 4KB/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

Our results represent performance differences in percent, using the following well-known formula [25], 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. Also 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 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.

V. ACCURACY OF CROSS-PROFILING

To assess that our approach is sound, we compare the CPU cycle estimates from the generated cross-profiles with the actual CPU cycle consumption on 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. It has a 4 KB FIFO instruction cache organized in 16 blocks. The three embedded benchmarks Kfl, Lift, and UdpIp are executed on JOP and the execution time is measured with a CPU cycle counter. The same benchmarks are profiled with CProf.

Table I 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 UdpIp the error is below 2%. The observed inaccuracies in the cross-profiles generated by CProf are due to imprecise CPU cycle estimates for certain (complex) bytecodes and differences in the Java class libraries between the host and the target. Nonetheless, for all measured benchmarks that run on the JOP hardware, the error in the CPU cycle estimates is below 2%.[2] For our purposes, the accuracy of CProf's cross-profiles suffices.

[2]In prior work [9] we reported higher errors. Thanks to improved CPU cycle estimates for some bytecodes, we were able to reduce the error.

Table II
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

Although we use WCET values for the bytecode timings, the cross-profiling results underestimate the execution time. It has to be noted that most bytecodes on JOP have a constant execution time. Therefore, their WCET values equal the best-case execution time. One possibility for the underestimation, especially in the UdpIp 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.

It has to be noted that for the evaluation of different architectural changes, the cross-profiling estimates need not be perfectly accurate [26]. We are interested in the relative performance differences between the cross-profiling runs for distinct architectures.

VI. COMPUTER ARCHITECTURE EVALUATION

In this section we evaluate the performance benefit of various possible architectural improvements for JOP using CProf. In Section VI-A the performance baseline is established with the current JOP design. As a simple enhancement, in Section VI-B the instruction cache size is increased and the resulting speedup is measured. In the following subsections more advanced architectural improvements are measured, such as faster method invocation (Section VI-C), pipeline reorganization (Section VI-D), and optimized bytecode fetch (Section VI-E). In Section VI-F we show that combining all aforementioned architectural enhancements yields a 40% faster in-order Java processor than the current JOP design. Finally, in Section VI-G we explore the theoretical limit of the performance of an in-order pipelined Java processor.

A. 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 II 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.

B. Variation of the Instruction Cache

The first and quite simple change in the architecture 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 III shows the performance differences relative to the standard configuration of JOP given in Table II.

The third and fourth columns show configurations with bigger caches of 16 KB 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.

Table III
INFLUENCE OF THE INSTRUCTION CACHE SIZE ON THE PERFORMANCE RELATIVE TO A 4KB/16 CACHE

Benchmark	1KB/4 (%)	16KB/64 (%)	64KB/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

Table IV
FASTER INVOKE INSTRUCTIONS WITH DIFFERENT CACHE SIZES

Benchmark	1KB/4 (%)	4KB/16 (%)	16KB/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

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.

It has to be noted that the cache size and organization (number of blocks) is configurable in JOP, so 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.

C. *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 [8].

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 can cut down the number of cycles for the invoke and return instruction by a factor of two. The performance impact of this optimization for different cache sizes is shown in Table IV.

The measurements with cache sizes of 1 KB, 4 KB, and 16 KB are shown in Table IV. 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 IV 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 third and fourth column 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 Table III and Table IV.

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 KB 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 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.

D. *Longer Pipeline*

The 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 V 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.

Table V
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

Table VI
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

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.

E. Advanced Instruction Fetch

In the current design of JOP the bytecode instruction fetch is performed with one byte per cycle in order to achieve a time-predictable architecture. However, with an additional pipeline stage an optimization of the bytecode fetches from the method cache will be possible to provide single cycle bytecode fetch despite the variable instruction length. This optimization is still time-predictable.

The optimization applies to all bytecodes that are longer than one byte and are implemented in microcode. Table VI 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.

F. Combined Effect

As a summary, the combined effect of all mentioned architectural enhancements is given in Table VII. 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%.

G. Theoretical Limit

One interesting question is the theoretical limit of an in-order Java processor without instruction folding. Instruction folding, as implemented in picoJava [12], can map certain frequent sequences of simple bytecodes to a single RISC-like instruction. Instruction folding in picoJava reduces the CPI by 15 to 29% [33]. However, instruction folding consumes a lot of chip resources due to the folding unit and the five-port register file and can limit the maximum clock frequency.[3]

For the theoretical limit experiment, we model a realistic but very advanced architecture. For bytecodes that do not access heap memory, we assume a single cycle execution time. To achieve this value, even for floating point operations, a long pipeline will be needed. The resulting ideal processor has a cache hit rate of 100% and a perfect branch prediction. With the prefect branch prediction we assume a branch or goto execution time of 2 cycles – the branch/goto instruction is executed in the decode stage.

The ideal architecture will still need several cycles for some bytecode instructions, e.g., for the object-oriented instructions, such as field access, object creation, and virtual method dispatch. Table VIII shows the assumed cycle count for more complex bytecodes. The numbers are derived from the fact that the bytecodes need more than one access to heap memory. For a write operation, we account an additional clock cycle as the hit detection has to be performed before the actual write. The invoke instruction also includes the additional cycles for the return instruction.

For comparison, measured numbers obtained with the just-in-time compiler CACAO [34] on the fast in-order MIPS CPU YARI are shown in the last column [35]. Two bytecode timings are missing in the table, as these have not been measured in [35]. The invoke instruction also includes the cycles for the corresponding return instruction. The CACAO/YARI combination, implemented in the same FPGA as JOP, can be clocked 20% slower than JOP, but is up to 2.8 times faster than JOP (our baseline). However, the resulting design consumes twice the chip resources of JOP and, due to the compiler overhead, needs 20 times more main memory for the execution of a *Hello World* example.

Table IX shows the resulting performance and CPI values for the ideal in-order Java processor. The CPI value for all benchmarks is very similar with a geometric mean of 1.5. The speedup of the embedded benchmarks is between a factor of 3.3 and 3.9. The SPEC JVM98 benchmarks, which

[3]picoJava is about 8 times larger than JOP and can be clocked at about half the frequency of JOP in the same implementation technology.

Table VII
EFFECT OF THE COMBINATION OF ALL ARCHITECTURAL ENHANCEMENTS

	speedup (%)		CPI	
	4KB/16	16KB/64	4KB/16	16KB/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

Table VIII
CYCLES FOR CONTROL AND OBJECT-ORIENTED INSTRUCTIONS, FOR THE IDEAL JAVA PROCESSOR, AS WELL AS FOR CACAO/YARI

Instruction	Ideal	CACAO/YARI
branch/goto	2	3–4
getfield/getstatic	2	3–5
putfield/putstatic	3	
invoke	5	12-15
xaload	3	9
xastore	4	

Table IX
PERFORMANCE OF THE IDEAL IN-ORDER JAVA PROCESSOR

Benchmark	speedup (%)	CPI
Kfl	232.3	1.39
Lift	228.9	1.42
UdpIp	289.2	1.39
_201_compress	403.2	1.48
_202_jess	924.9	1.63
_209_db	654.2	1.50
_213_javac	1000.4	1.60
_222_mpegaudio	1595.0	1.49
_228_jack	924.7	1.61
geo. mean	581.3	1.50

had a higher CPI in all other experiments, are between 5 and 17 time faster than our baseline. For the benchmark with the highest speedup (_222_mpegaudio), the high speedup results from the fact that the ideal processor executes float and long operations in a single cycle.

VII. CONCLUSION

In this paper we have applied cross-profiling as an approach to computer architecture evaluation for embedded Java processors. We used the cross-profiling tool CProf for standard Java benchmarks and evaluated the performance impact of various enhancements to the Java Optimized Processor JOP. 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.

The availability of accurate bytecode-level execution time information for JOP greatly simplified the architecture evaluation. Regarding ongoing research, we are investigating the automated extraction of bytecode execution timings for embedded Java systems (e.g., jamuth, aJile, Cjip, and CACAO/YARI). The resulting timing models can then be used for cross-profiling or for a first estimation of the WCET.

This methodological paper is complemented by a tool demonstration paper of the cross-profiler CProf [36].

Acknowledgment: We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “FERRARI – Framework for Efficient Rewriting and Reification Applying Runtime Instrumentation” (SNF Project No. 200021-118016/1).

REFERENCES

[1] C. Kozyrakis and D. Patterson, “A new direction for computer architecture research,” *Computer*, vol. 31, no. 11, pp. 24–32, Nov 1998.

[2] W. Binder, M. Schoeberl, P. Moret, and A. Villazon, “Cross-profiling for embedded Java processors,” in *Proceedings of the 5th International Conference on the Quantitative Evaluation of SysTems (QEST 2008).* St Malo, France: IEEE Computer Society, September 2008.

[3] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1999.

[4] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek, “Java for safety-critical applications,” in *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, Mar. 2009.

[5] I. Bate, G. Bernat, G. Murphy, and P. Puschner, “Low-level analysis of a portable Java byte code WCET analysis framework,” in *Proc. 7th International Conference on Real-Time Computing Systems and Applications*, Dec. 2000, pp. 39–48.

[6] SPEC, "The spec jvm98 benchmark suite," Available at http://www.spec.org/, August 1998.

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

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

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

[10] S. Uhrig and J. Wiese, "jamuth: an IP processor core for embedded Java real-time systems," in *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*. New York, NY, USA: ACM Press, 2007, pp. 230–237.

[11] M. Zabel, T. B. Preusser, P. Reichel, and R. G. Spallek, "Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture," in *Prceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, Aug. 2007, pp. 59–62.

[12] J. M. O'Connor and M. Tremblay, "picoJava-I: The Java virtual machine in hardware," *IEEE Micro*, vol. 17, no. 2, pp. 45–53, 1997.

[13] 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*. IEEE Computer Society, 2001, p. 53.

[14] Imsys, "Im1101c (the Cjip) technical reference manual / v0.25," " 2004.

[15] F. Gruian and M. Westmijze, "Bluejep: a flexible and high-performance java embedded processor," in *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*. New York, NY, USA: ACM, 2007, pp. 222–229.

[16] 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)*. New York, NY, USA: ACM Press, 2006, pp. 202–211.

[17] T. Harmon and R. Klefstad, "Interactive back-annotation of worst-case execution time analysis for Java microprocessors," in *Proceedings of the Thirteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, August 2007.

[18] T. Bogholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K. G. Larsen, "Model-based schedulability analysis of safety critical hard real-time Java programs," in *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)*. New York, NY, USA: ACM, 2008, pp. 106–114.

[19] B. Huber, "Worst-case execution time analysis for real-time Java," Master's thesis, Vienna University of Technology, Austria, 2009.

[20] 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)*, ser. LNCS, vol. 3292. Agia Napa, Cyprus: Springer, October 2004, pp. 371–382.

[21] T. B. Preusser, M. Zabel, and R. G. Spallek, "Bump-pointer method caching for embedded java processors," in *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*. New York, NY, USA: ACM, 2007, pp. 206–210.

[22] S. Metzlaff, S. Uhrig, J. Mische, and T. Ungerer, "Predictable dynamic instruction scratchpad for simultaneous multithreaded processors," in *Proceedings of the 9th workshop on Memory performance (MEDEA 2008)*. New York, NY, USA: ACM, 2008, pp. 38–45.

[23] 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*. ACM Press, 1997, pp. 85–96.

[24] M. Schoeberl, "Architecture for object oriented programming languages," in *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*. Vienna, Austria: ACM Press, September 2007, pp. 57–62.

[25] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[26] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai, "Challenges in computer architecture evaluation," *Computer*, vol. 36, no. 8, pp. 30–36, Aug. 2003.

[27] J. J. Yi, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, and J. E. Smith, "The future of simulation: A field of dreams," *Computer*, vol. 39, no. 11, pp. 22–29, 2006.

[28] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002.

[29] F. Brandner, D. Ebner, and A. Krall, "Compiler generation from structural architecture descriptions," in *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES 2007)*. ACM, 2007, pp. 13–22.

[30] F. Brandner, A. Fellnhofer, A. Krall, and D. Riegler, "Fast and accurate simulation using the llvm compiler framework," in *1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*, Paphos, January 2009.

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

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

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

[34] A. Krall and R. Grafl, "CACAO – A 64 bit JavaVM just-in-time compiler," in *PPoPP'97 Workshop on Java for Science and Engineering Computation*, G. C. Fox and W. Li, Eds. Las Vegas: ACM, Jun. 1997.

[35] F. Brandner, T. Thorn, and M. Schoeberl, "Embedded JIT compilation with CACAO on YARI," in *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*. Tokyo, Japan: IEEE Computer Society, March 2009.

[36] P. Moret, W. Binder, A. Villazon, D. Ansaloni, and M. Schoeberl, "Locating performance bottlenecks in embedded Java software with calling-context cross-profiling," in *Proceedings of the 6th International Conference on the Quantitative Evaluation of SysTems (QEST 2009)*. Budapest, Hungary: IEEE Computer Society, September 2009.