JCSE Journal of Computing Science and Engineering

Regular Paper
Journal of Computing Science and Engineering,
Vol. 6, No. 1, March 2012, pp. 26-38

Time-Predictable Java Dynamic Compilation on Multicore Processors

Yu Sun
Mathworks, Boston, MA, USA **sunyu@siu.edu**

Wei Zhang*
Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, VA, USA
wzhang4@vcu.edu

Abstract
Java has been increasingly used in programming for real-time systems. However, some of Java's features such as automatic memory management and dynamic compilation are harmful to time predictability. If these problems are not solved properly then it can fundamentally limit the usage of Java for real-time systems, especially for hard real-time systems that require very high time predictability. In this paper, we propose to exploit multicore computing in order to reduce the timing unpredictability that is caused by dynamic compilation and adaptive optimization. Our goal is to retain high performance comparable to that of traditional dynamic compilation, while at the same time, obtain better time predictability for Java virtual machine (JVM). We have studied pre-compilation techniques to utilize another core more efficiently, pre-optimization on another core (PoAC) scheme to replace the adaptive optimization system (AOS) in Jikes JVM and the counter based optimization (CBO). Our evaluation reveals that the proposed approaches are able to attain high performance while greatly reducing the variation of the execution time for Java applications.

Category: Embedded computing

Keywords: Performance; Reliability; Time predictability; Java virtual machine; Multicore processors; Adaptive optimizations

I. INTRODUCTION

Following its success and popularity on servers and desktops domains, recently Java has been increasingly used in programming for embedded and real-time systems. However, as Java was not originally designed for real-time use, some of Java's attractive features for portability or performance actually makes Java computing not time predictable. If these problems are not well solved, then they can severely limit its usage in the real-time domain. For instance, dynamic compilation and adaptive optimization are two techniques to boost the performance of Java programs but they are generally unpredictable in terms of the execution time. An unpredictable dynamic compilation/optimization may cause the real-time tasks to miss their deadlines. It is certainly unacceptable that in a real-time system that uses Java, especially in a hard real-time or safety-critical system, designers will have no idea about when the compilation/optimization occurs or how long it takes.

Open Access http://dx.doi.org/10.5626/JCSE.2012.6.1.26 http://jcse.kiise.org

This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

Received 2 February 2012, **Revised** 14 February 2012, **Accepted** 17 February 2012
*Corresponding Author

Copyright © 2012. The Korean Institute of Information Scientists and Engineers
pISSN: 1976-4677 eISSN: 2093-8020

A simple solution to address this problem is to disable those features in real-time Java, which has been proposed in a few earlier studies. For instance, scoped memory [1-4] which is manually controlled by developers like in C/C++, was proposed to take the place of automatic memory management. Instead of dynamic compilation, interpretation or ahead-of-time (AOT) compilation [5-7] is used to generate native code from Java byte code. Although this kind of solutions satisfies the restriction of real-time systems very well, the downside is also obvious. For instance, the performance is often degraded, the platform independence and the easiness of development are also badly compromised.

Another way to attack this problem is to modify those features in order to make them more time-predictable and at the same time retaining performance or even achieving better performance (note that the latter becomes possible by the exploitation of new architectural features such as multicore computing which is the focus of this paper). A series of valuable studies have been done in the field of memory management. Several deterministic garbage collectors [8-13] were developed and tested. With those collectors, the worst-case time spent on garbage collection can be bounded. However, no such work has been done for dynamic compilation and optimization and they are harmful to time predictability.

The advancement in computer architecture provides new opportunities to address time predictability problem of Java computing without compromising on the performance or even attaining superior single-threaded performance. Specifically, this paper exploits multicore architecture in order to improve the time predictability of Java applications. The multicore architecture has been adopted by major microprocessor companies such as Intel, AMD, IBM, and Sun. In addition to its wide usage in desktop and server markets, the multicore architecture is also increasingly used in embedded micro-controllers. This includes the dual-core Freescale MPC8641D (Freescale Semiconductor Inc., Austin, TX, USA), the dual-core Broadcom BCM1255 (Broadcom, Irvine, CA, USA), the dual-core PMC-Sierra RM9000x2, the quad-core ARM11 MPcore and the quad-core Broadcom BCM1455. Multicore processors are increasingly used in real-time systems due to their superior performance, lower energy consumption and better system density. It is expected that real-time applications will soon use large-scale multicore platforms with tens or even hundreds of cores per chip [14].

This paper exploits multicore processors to enhance time predictability of Java applications without compromising on performance. Our approach divides the Java program and Java virtual machine (JVM) into two parts: one is time predictable, executing real-time tasks and the other deals with all the other tasks, such as dynamic compilation, garbage collection and so on. Two cores are used to separately execute the two parts (This paper focuses on using dual-core microprocessors for the improvement of time predictability of Java applications. While our approach can be applied to general multicore processors with more than 2 cores by dividing cores into two groups: real-time group and non-real-time group, the evaluation of our approach on this general multicore architecture is beyond the scope of this paper). If all the unpredictable tasks are removed from the main core and no real-time task has unbounded dependency on the auxiliary core, then the time predictability of real-time tasks running on the main core can be guaranteed.

Accurately obtaining the worst-case execution time (WCET) of tasks that run on JVMs is a challenging task, if not impossible, especially when dynamic compilation and adaptive optimizations are enabled. In this work, in order to improve the time predictability of dynamic compilation and optimizations (rather than developing a state-of-the-art WCET analyzer for JVMs), we make the following assumptions: 1) the fraction of WCET caused by JVM (by disabling the dynamic compilation and optimizations) can be statically bounded, which are supported by current WCET analysis techniques [15]; 2) there is no cache interference between the different cores in the studied multicore platform; 3) garbage collection is disabled in JVM.

In this paper, we first try to isolate dynamic compilation and move it to the auxiliary core, where baseline compilation and dynamic hot-spot-based optimizations are to be performed. As we find that the auxiliary core may not be fully utilized, we migrate pre-compilation to that core in order to further improve the performance. Experimental results show that our approaches efficiently reduce the number of interruptions. This leads to enhanced time predictability. Using pre-compilation can achieve even better results. Then, we move on to the adaptive optimization system (AOS). This is the key to high performance but it is very sensitive to timing and can cause large execution time variations. We study the impact of AOS sampling intervals and this is the source of its sensitivity to the execution time. After this, we develop the pre-optimization on another core (PoAC) scheme to replace the original AOS. Our experiments show that PoAC greatly reduces time variation that is caused by the AOS. Finally, we compare PoAC with another approach that replaces AOS and it is able to reduce the execution time variations, which is counter based optimization (CBO). Experimental results indicate that PoAC outperforms CBO and PoAC is a better choice for real-time Java applications.

The rest of this paper is organized as follows. Section II describes our multicore approaches to enhance time predictability of Java applications which includes normal compilation/optimization on the auxiliary core and PcAC/PoAC schemes. Section III presents our evaluation methodology and Section IV gives the experimental results. We discuss related work in Section V. Finally, Section VI concludes this paper and provides future research direction.

II. EXPLOITING MULTICORES FOR REAL-TIME JAVA COMPUTING

We first apply multicore techniques on dynamic compilation/optimization, as they are the key for the achievement of high performance on JVMs and also an important source of unpredictability at run-time. Our approach is depicted in Fig. 1. On a single processor, dynamic compilation and optimization have to interrupt the executing tasks and cause unpredictable interference. We attempt to remove dynamic compilation and optimization from the main core and use the auxiliary core for them. As a result, there is no more unpredictable interruption to the main core and the time constraints can be guaranteed.

A. Pre-compilation on Another Core (PcAC)

As shown in Fig. 1 on a single-core system, just-in-time (JIT) compilation is invoked by the application thread whenever it invokes an uncompiled method and the JIT compilation runs on the caller thread as a function call. The application thread has to wait until the JIT compilation is done. These compilation interruptions are generally unpredictable. Thus, the JIT compilation cannot be used in real-time systems. One approach for the elimination of such interruptions is AOT compilation. An AOT compiler compiles all the methods that are used before the Java application is deployed. As a result, the flexibility of Java is limited. Moreover, an AOT compiler misses the opportunity to perform dynamic optimization and it has a great potential to improve the performance based on the run-time information.

In this paper, we propose an alternative approach - PcAC. With an extra core (i.e., the auxiliary core), we can compile Java byte code AOT before its first execution but dynamically at run-time. The objective of the PcAC scheme is to eliminate the compilation interruptions as the AOT compiler does, while at the same time, maintaining the flexibility of the Java language and the possibility of future optimization.

The key issue of the PcAC scheme is the need to pre-compile Java byte code before the deadlines. In the case, where we have the WCETs needed to compile each Java byte code section (usually a Java method) along with the call graph information, we can accurately generate the list of methods that are to be pre-compiled and their worst-case compilation times. However, in general such WCET information is not available. So, we have to use some heuristics in order to reduce the compilation interruptions as much as possible.

1) Call Graph Guided PcAC

Our first heuristic is based on the call graph information. The call graph of Java applications can be generated from static analysis with the Java byte code. Pre-compilation is performed by a thread that is specially designed and pinned on the second core. This is called the guided pre-compilation thread (GPCT). It generates the list of methods that to be pre-compiled based on the call graph of the currently executing Java application. Breadth-first search is used to traverse the call graph to build the method list. Fig. 2 shows a call graph and the list of methods that are to be pre-compiled.

2) Profiling Guided PcAC

Another heuristic is to profile Java applications and get the list of methods in their exact execution order. Profiling has to be used to get the method invocation list. As shown in Fig. 3, GPCT reads profiling data from previous execution and pre-compile methods which follow that order.

3) Other Heuristics

There can also be other heuristics besides the ones that are described above. For example, the size of a method can be considered as a factor to order the pre-compilation. As we can obtain a larger number of methods compiled if small methods are compiled first, which often

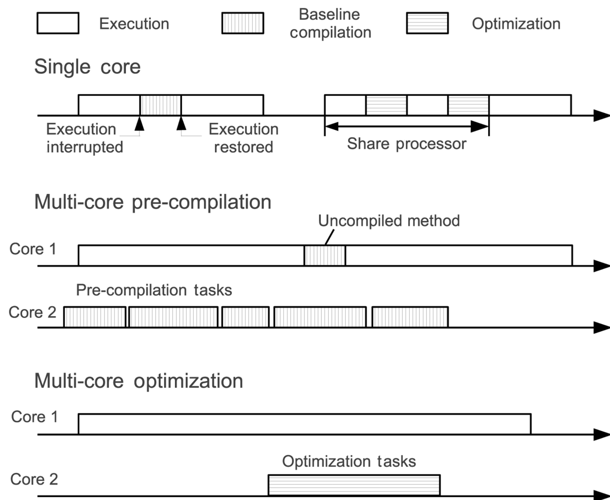

Fig. 1. Dynamic compilation/optimization on single-core and multicore processors.

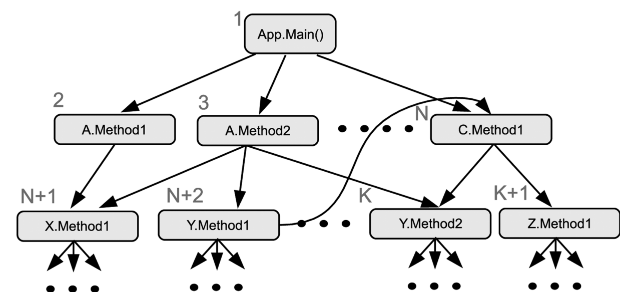

Fig. 2. Call graph guided pre-compliation on another core (PcAC).

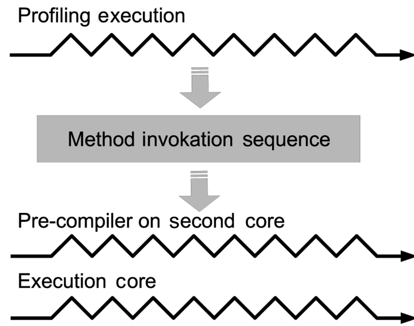

Fig. 3. Profiling guided pre-compliation on another core (PcAC).

take less time. Another possible heuristic is to use the depth-first search (DFS) instead of the breadth-first search. The DFS works better when there are only a few branches in the methods. In this case, the possible execution path in the call graph is basically depth-first. However, these heuristics are not safe in hard real-time systems. The WCETs of JVM and Java programs are needed for the PcAC scheme to satisfy time constraints. Moreover with such information, the PcAC scheme can make even better decision about which method to be pre-compiled and when to perform that.

B. PoAC

Dynamic optimization in JVMs usually runs as a background thread other than the application threads. In single-core systems, this background optimization thread does not pause the application threads, but it shares the processor under a given scheduling scheme. Therefore, the application thread is slowed down while performing optimization. For real-time tasks, deadlines may be missed due to the interruptions that are caused by optimization. By migrating the optimization thread to another core (i.e., the auxiliary core), we can avoid processor sharing with real-time tasks.

However, the unpredictability in JVMs cannot be eliminated by a simple removal of the optimization thread from the main core. The major issue is that the AOS is unpredictable and it is not suitable for real-time systems. As the AOS introduces interruptions into the normal execution to take samples and identify hot-spots of the given Java programs. During the execution time, these interruptions make the AOS very sensitive to even very slight variation. If a method is delayed only slightly during execution then, the samples that are taken by the AOS may report another method as being hot. Even a very small timing interference, usually from the architecture (e.g., cache misses) or operating system, can be magnified by the AOS. Thus it impacts the final execution time. By simply disabling the AOS is one way for maintaining time predictability, but this result in a significant performance loss. In the remainder of this section, we address this problem by adopting multicore techniques.

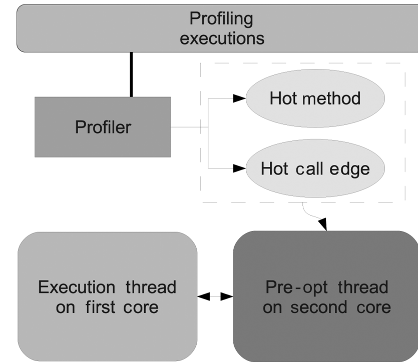

Fig. 4. Implementation of the pre-optimization on another core (PoAC) scheme.

We first evaluated the impact of AOS interrupt intervals on the execution time. Usually, the interval is chosen based on the balance between interrupt overheads and the accuracy of identifying hot-spots. But in real-time systems, the impact on time predictability also needs to be considered. We performed sensitivity study while varying the interval from 20 ms to 40 ms and 80 ms. The experimental results show that all the intervals cause a large execution time variation.

Then we implement the PoAC scheme as shown in Fig. 4. The profiler collects two kinds of run-time data during profiling executions: hot methods and hot call edges. Hot methods, together with the advices indicating which optimization level should be used, are stored for the PoAC thread. Moreover, hot call edge information is collected for the inliner of the pre-optimization thread. With PoAC, the adaptive optimization system is now disabled on the main core, as well as all adaptive controller tasks, such as the sampling thread, the call edge listener and so on. This brings out two good side effects: First, the overheads of these adaptive controllers can now be avoided. Second, time predictability on the main core is improved by the reduction of thread switches that are caused by them. Nevertheless, there are also other overheads that are involved. First of all, profiling takes some extra time but it is a one-time overhead. Another overhead is the time to set up and start an auxiliary core. It slows down the booting stage of the Java Virtual Machine. However, this overhead is predictable as it is constant for all Java applications.

III. EVALUATION METHODOLOGY

We implement and evaluate our multicore approaches based on Jikes RVM 2.9.3 [16]. This is a fully functional JVM with open source code. We extend it in order to run and evaluate our approaches. The main extensions include the following: 1) binding Java threads to a fixed core; 2)

isolating the compiler, optimizer and pre-compiler from the application threads; 3) providing a prioritized fully preemptive scheduler; 4) building communication channels between the two cores involved.

We also implement a CBO mechanism on the Jikes RVM. This is a replacement of the AOS. Our CBO uses invocation counters for Java methods and injects counter update code sections into method prologues. We expect the CBO to remove the sensitivity of AOS and thus reduce the execution time variation. Experiments are conducted in order to compare the PoAC and the CBO and also to analyze their advantages and disadvantages.

We use the SPEC JVM98 benchmark suite [17] to evaluate our implementation. For our baseline compilation and pre-compilation, both the input size and the iteration number of benchmarks are set to the minimal, as compilation occurs only during the first time a method is invoked. Larger input sizes and iteration numbers are used to evaluate optimization on the auxiliary core as dynamic optimizer needs time to detect program hot spots. All the benchmarks run on a dual-core Intel x86 PC with 2 GB memory, with two Pentium 4 processors running at 3.4 GHz. The operating system used is Linux with kernel 2.6.24-SMP.

Although both the Jikes RVM and the SPEC JVM98 are not designed for real-time purposes, we can still get time predictability results by counting the compilation/optimization interruption numbers, as well as measuring time variation of different executions. Execution time variation is defined as the standard deviation of the execution times for each benchmark with different inputs. Even for the same input, the execution time still varies due to the factors such as different adaptive optimization strategies, cache hits/misses, external interferences from the operating system or other threads, etc. Thus, we also measure this variation of the execution time by running the benchmark programs multiple times and then, by calculating their standard deviations of execution times.

IV. EXPERIMENTAL RESULTS

A. Impact of Dynamic Compilation and Optimization

We first run the benchmarks on a single core to see how dynamic compilation affects the performance of Java applications. Fig. 5 shows the percentage of the total execution time that the baseline compilation takes. The results are obtained with the smallest input and the benchmarks are executed only once. This is because baseline compilation is only performed at the first occurrences of the methods. We observe that the compilation time varies from 2% to 16% of the total execution time for different benchmarks. This amount of time is usually unpredictable and this is unacceptable in real-time systems.

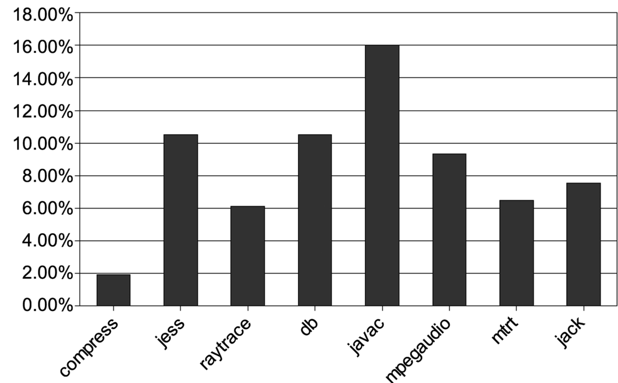

Fig. 5. Percentage of baseline compilation in the total execution time.

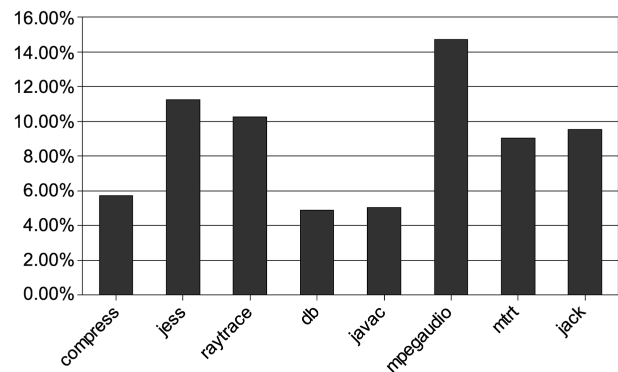

Fig. 6. Percentage of time used by optimization in the total execution time.

Fig. 6 shows the percentage of time that is used by the optimization in the total execution time for all the 8 benchmarks. The results shown are the maximum values among 9 groups of experiments and they are performed with three input sizes (1, 10, 100) and three iteration numbers (1, 10, 100). We can see that 5% to 15% of the total execution time is used to optimize the hot methods, during which all the real-time tasks that share the processor have to wait. This leads to greater time variation and less predictability.

B. Reducing Interruptions with PcAC

The objective of applying pre-compilation on another core is to avoid interruptions on the main core. They are caused by dynamic compilation. However, it is very difficult to eliminate all these interruptions due to the complexity of Java programs. With the different inputs, some pre-compiled methods may not be always executed. In some other cases, methods are called before the completion of compiling by the pre-compiler. As a result, some interruptions still occur on the main core. In this case, what the pre-compilation can do is to reduce the number

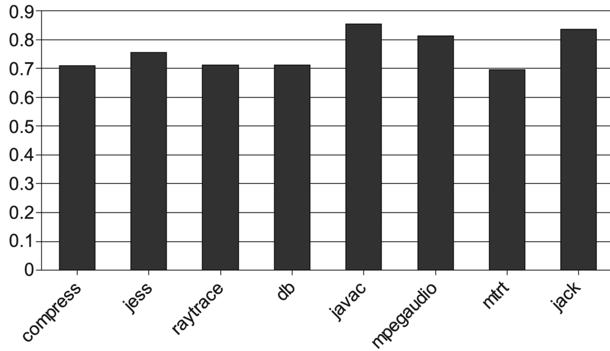

Fig. 7. Normalized dynamic compilation interruptions by the call graph based pre-compliation on another core (PcAC), with respect to the single-core scheme.

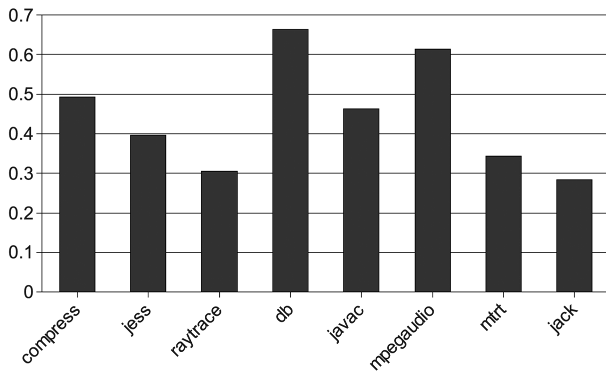

Fig. 8. Normalized dynamic compilation interruptions of the profiling pre-compliation on another core (PcAC), with respect to the single-core scheme.

of interruptions to the maximum. Fig. 7 shows the normalized number of interruption under our call graph guided PcAC scheme and they are normalized to the number of interruptions in the single core case. We find out that the pre-compilation can reduce interruptions by about 20% on an average. The result in Fig. 7 is not quite satisfying. In order to study the best results that can be achieved, the profile-guided pre-compilation is performed. The result is listed in Fig. 8. As much as 70% of compilation interrupts are eliminated.

C. Timing Sensitivity of AOS

We vary the interrupt interval from 20 ms (default value of Jikes RVM) to 40 ms and 80 ms. Fig. 9 shows the execution time when the interval is 40 ms, normalized to the case when the interval is 20 ms Where, *x* axis shows the input size and the iteration number of benchmarks. Fig. 10 shows the results when the interval is 80 ms. As we can see in these two figures, performance gets worse when the total execution time is relatively short (i.e., with a small input size and/or a small iteration number). This is due to the AOS which cannot correctly identify the hot-

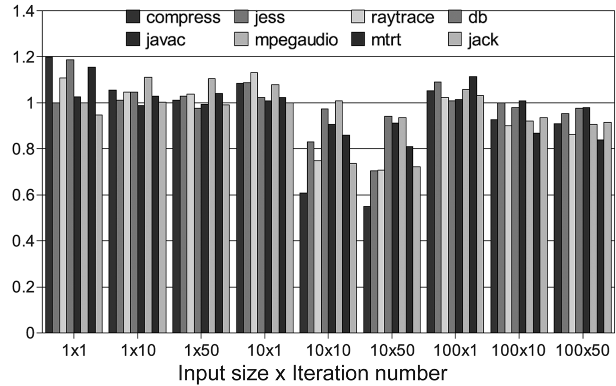

Fig. 9. Execution time when the interval is 40 ms, normalized to the case when the interval is 20 ms.

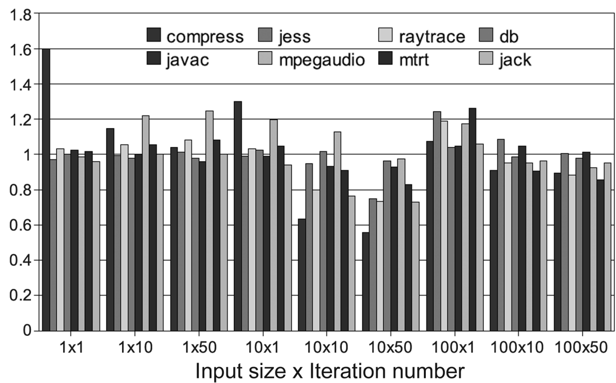

Fig. 10. Execution time when the interval is 80 ms, normalized to the case when the interval is 20 ms.

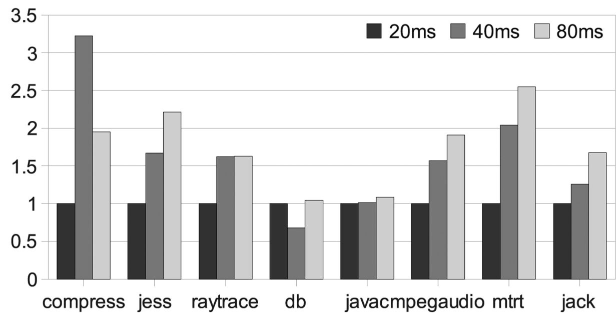

Fig. 11. Standard deviation of the execution times, normalized to the case when the interval is 20 ms.

spots with a small number of interruptions. But for a large input and a large iteration number, even long intervals can get enough samples to identify hot-spots and then perform the needed proper optimizations. In these cases, performance is better due to the reduction in the interrupt overheads.

Unlike performance, time predictability depends more on the different benchmark behaviors. As shown in Fig. 11, only db, which is a small benchmark, repeatedly executes

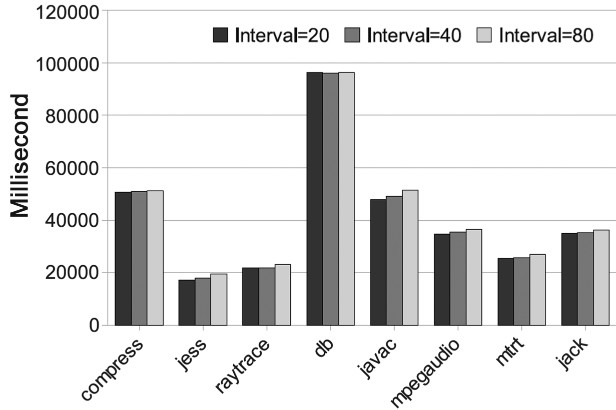

Fig. 12. Average execution time of different intervals with 5 inputs.

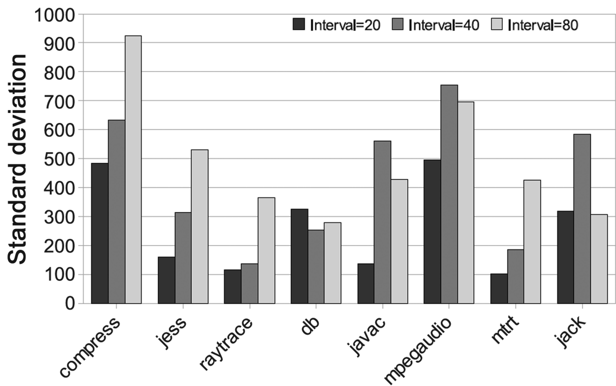

Fig. 13. Standard deviation of execution times of different intervals with 5 inputs.

several methods. It has a smaller time variation when the interval increases. It is obvious that the inaccurate hot-spot detection makes the AOS even more sensitive. As a result, the time predictability gets worse.

We also test the timing sensitivity of the AOS with different inputs. In order to obtain various inputs for the SPEC JVM98 benchmarks, we change the order of the three input files for each benchmark. With this design, we get 5 groups of results, each with a unique input, but all of them take very similar execution times. The results are shown in Figs. 12 and 13. They are similar to Figs. 9-11. Therefore, it is hard to simply tune the AOS to obtain better time predictability and other approaches are also needed.

D. Results of the PoAC Scheme

Due to great timing sensitivity of the AOS to external interferences and different inputs, as shown in the previous section, a predictable JVM should not use the AOS to detect and optimize hot-spots in Java applications. Therefore, we propose the PoAC scheme the aim at better pre-

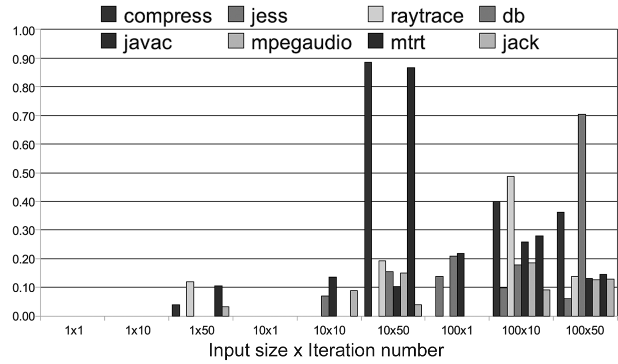

Fig. 14. Standard deviation of execution times by the pre-optimization on another core (PoAC) normalized to the BASE scheme.

dictability without the degradation of the performance.

1) Time Variation of PoAC

We measure the execution time variation by running benchmarks multiple times and by calculating the standard deviation of their execution times. The smaller the variation is better is the time predictability. We test the original JVM with the AOS as our baseline scheme (denoted by BASE) and also our PoAC scheme. For both BASE and PoAC schemes, we run the benchmarks five times with the same inputs. Execution time variation results are shown in Fig. 14, where *x* axis denotes the input size and the iteration number of each group of benchmarks. We can see a significant reduction in the time variation in Fig. 14, especially for short executions. The execution time variation from 1×1 to 10×10 is quite small or even zero. As the PoAC scheme eliminates the unpredictability of the adaptive sampling mechanism, now all the optimizations take places at nearly the same time and also complete at about the same time. However, execution time variation still exists when the execution time gets larger. This is due to the interferences of other background processes on the experimental platform and they cannot be completely removed. In our future work, we plan to perform the experiments on a simulator where these interferences can be avoided. Then, the time predictability of the PoAC scheme can be accurately evaluated.

Our experiments also show that for a given input, the PoAC scheme works well on the reduction of the execution time variation. We apply 5 different inputs to the benchmarks and run them under both BASE and PoAC schemes for 30 iterations and the results of execution time variation are shown in Fig. 15. This indicates that the standard deviation of PoAC scheme is just 40-50% on average as compared to the BASE scheme. However, the time variation cannot be eliminated completely, as different inputs are given to the programs. The reason that the PoAC still works well with different inputs is that it disables the AOS in Jikes RVM and this is very sensitive to

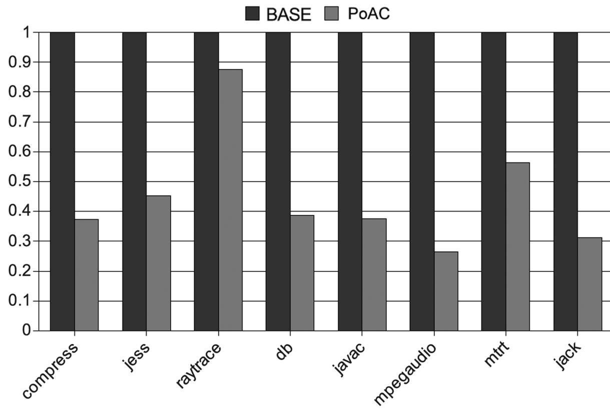

Fig. 15. Standard deviation of execution times by the pre-optimization on another core (PoAC) with 5 different inputs normalized to the BASE scheme.

the variation of the execution time. AOS brings some non-linear dynamics into the whole system. Thus, even a very tiny interference from the architecture, hardware interrupts or OS processes will be magnified by the AOS and it can affect the total execution time of the program. With the PoAC, the sensitive of the AOS is avoided. So, the JVM can operate in a more predictable manner.

2) Performance Impacts of the PoAC Scheme

With the PoAC, unpredictability caused by the AOS can be avoided. But it is important not to compromise on the performance. In order to evaluate the performance impact of the PoAC, we set up 9 groups of experiments with 3 input sizes and 3 iteration numbers. With this setting, we can observe the trend of the performance impact with different execution times. The baseline scheme is a single core JVM with adaptive optimization. The experimental results are varied for different benchmarks. One observation is that 8 benchmarks in SPEC JVM98 can be classified in to two groups, with similar results in each group, as shown in Figs. 16 and 17, respectively.

The first group of benchmarks shows good performance improvement for most of the configurations. As depicted in Fig. 16, only the left end of the *x* axis show worse performance compared to the baseline scheme. From 1 × 50 to 100 × 10, these 4 benchmarks have better performance with PoAC and this indicates that the profiling information is beneficial for these benchmarks. The reason for decreased performance on the left end is that execution time is too short to benefit from PoAC and the constant overheads mentioned in Section I are relatively significant. On the other hand, at the right end (i.e., with the largest number of inputs and iterations), the performance degrades. Particularly, for mtrt, we find out that its performance actually becomes worse than that of the baseline scheme. One possible reason is that with very long execution time, information from the profiling is not accurate as compared to the online adaptive optimizer

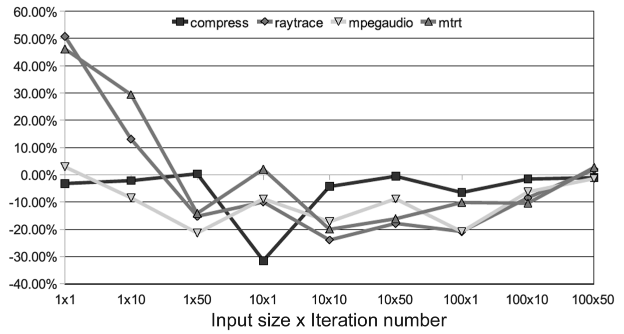

Fig. 16. Performance impact of execution times for compress, raytrace, mpegaudio, and mtrt.

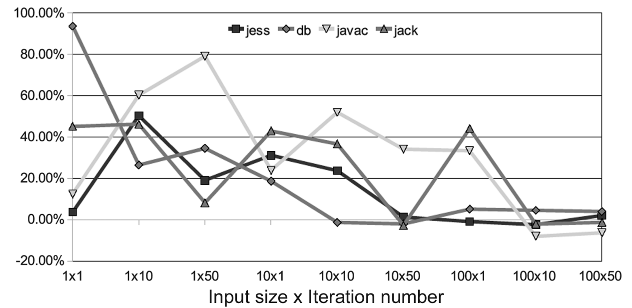

Fig. 17. Performance impact execution times for jess, db, javac, and jack.

used in the baseline scheme. This leads to worse performance. However, in between the two extremes our PoAC properly optimizes the needed methods, benefits from removing the adaptive controller and it avoids the warm-up time to identify hot methods. As a result, the performance is improved with the configurations in the middle.

The other group does not work well with the PoAC, as shown in Fig. 17. This is because profiling information is not accurate enough to provide proper advices for these benchmarks and this exhibits different behaviors with different inputs. The profiling execution under one con-

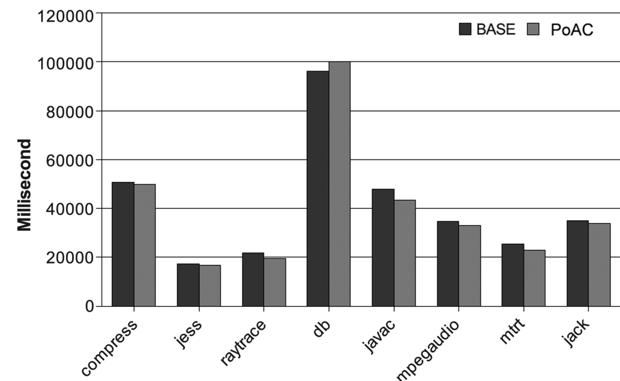

Fig. 18. Performance impacts for 5 inputs.

figuration (100 × 10) cannot be used to accurately predict the behaviors under other configurations. As a result, PoAC cannot optimize the hottest methods or optimized them on a wrong level. It is also possible that the optimization itself takes too much time and consumes too much resource and this result in negative impacts to the application thread on the main core. However, we cannot conclude that the PoAC does not work for these benchmarks. What it requires is the proper profiling data that is specially tuned for each of these benchmarks.

We have also evaluated the PoAC with 5 different inputs. The experimental results are shown in Fig. 18. As the execution time for this experiment is long enough, most of the benchmarks get better performance compared to the baseline AOS on a single core.

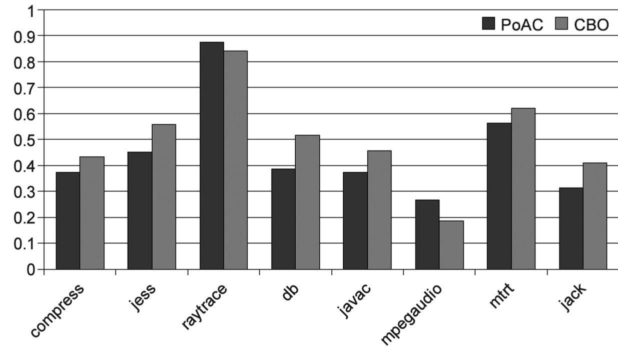

Fig. 19. Standard deviations of pre-optimization on another core (PoAC) and counter based optimization (CBO) with 5 inputs, normalized to the BASE scheme.

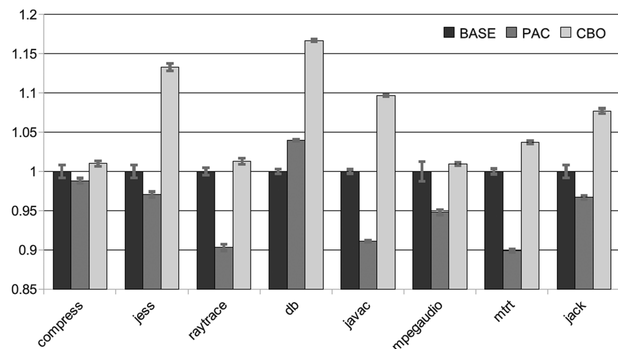

Fig. 20. Execution times of the pre-optimization on another core (PoAC) and the counter based optimization (CBO) schemes with 5 inputs, normalized to the BASE scheme.

E. Comparison of PoAC and CBO

There is another replacement of AOS that may improve time predictability. This is the CBO scheme. The CBO injects code that updates counters into Java programs and then identifies hot methods by using counter values. It does not rely on the sensitive timer based sampling mechanism as the AOS does. Hence, better time predictability can be expected with the CBO.

We implemented a simple CBO scheme that updates an invocation counter for each method. The counters are updated in the prologues of methods, and the optimization is triggered when a counter reaches a given threshold. The threshold value of each benchmark is manually selected. All the hot methods are optimized at level 1, as there is neither profiling information nor cost/benefit model that can be used to decide the optimization level. Like PoAC, CBO performs execution and optimization on the auxiliary core. However, the injected code that updates the counters has to run on the main core. The counters are implemented by using arrays that are mapped to the method IDs. They are updated in each method's prologue.

We evaluated the counter based scheme with 5 different inputs and compared its results with those of the PoAC scheme. As shown in Fig. 19, the variation of the execution times with the CBO reduces significantly compared to that of AOS, and it is close to the results of PoAC. This experiment result indicates that CBO is almost as good as PoAC in terms of raytrace and mpegaudio. However, he reduction in the execution time variation: PoAC works better with 6 benchmarks and CBO outperforms PoAC for CBO shows the worst performance as shown in Fig. 20 for all 8 benchmarks compared to both PoAC and AOS.

The performance problem of CBO is due to the way it selects hot methods. Those methods that are invoked the most are not necessarily the ones with the longest execution times. CBO cannot identify the hot methods as accurately as the sample based on the AOS or profile based PoAC. Moreover, lack of optimization level estimation prevents CBO from more aggressive optimizations and this also wastes time on optimizations for unimportant methods. Lastly, it is hard to set the optimal threshold values of counters, as the threshold values changes while running different Java programs.

Besides the performance issue, the code injection performed by the CBO has also a potential risk that may worsen time predictability in real-time Java computing. Compared to PoAC that leaves original Java code unchanged, CBO inserts counter update code into every method of a Java program. The access to a counter which is stored in an array in the heap may interfere with other data accesses in the original Java program. This may introduce unexpected cache stalls on the main core. In other words, injected counter update code makes it more complicated to predict the behavior of the original Java program. To summarize, CBO is able to reduce the execution time variation like PoAC by eliminating unpredictability that is caused by the AOS in JVMs but it is not suitable for real-time Java computing. This is due to its performance problem and increased complexity of timing analysis.

V. RELATED WORK

Most of the researches on real-time Java computing have focused on the improvement in the time predictability of garbage collection. The first idea is to remove the unpredictable garbage collection (GC) from real-time Java system. That is why scoped memory and immortal memory section are defined in the real-time specification for Java (RTSJ) [1]. In this case, the objects in real-time threads are managed by programmers instead of JVMs. The developers take care of the memory areas that are used for real-time tasks and they leave the other part to GC. The GC thread holds lower priority than real-time threads. Thus it cannot interfere with them. Time predictability is guaranteed but flexibility is lost in development. Most of the implementations of RTSJ support this kind of technique bypassing GC. Besides, researchers made efforts to improve the efficiency of this scoped memory [4]. Corsaro and Schmidt [18] presented another implementation called as jRate [2]. They presented an informal introduction to the semantics of the scoped management rules of the RTSJ. It also provided a group of design patterns which can be applied to achieve higher efficiency of the scoped memory. [3] provided a complete programming model of scoped memory model in order to simplify the memory analysis that has to be done by developers. Bypassing GC is proved easy to apply to the existing JVMs and it brings a small overhead to the runtime memory management. But it is of limited use due to the great difficulty it has brought to the developers. Without the GC feature, programmers have to pay much more attention to the objects in real-time tasks in order to avoid memory error. Although the scoped memory idea may be a good solution to small short-term projects, real-time GC is definitely necessary for future real-time Java systems.

Researchers have also studied how to design time-predictable garbage collectors. The incremental copying collector [19] is the first attempt for such real-time GCs. In this work, the memory mutator operation leads to GC operation. Hence, GC is predictable and the worst case is that every read or allocation invokes a certain mount of collection operations. It is not very efficient but it provides a way to implement real-time GC. Network-basedGC was then developed by many other researchers, such as [12] and [13] in the Jamaica VM. The basic idea is the same but the overhead of allocation detection and unnecessary collection is significantly reduced. Even hardware can be used to assist GC efficiency and reduce WCET bounds, as it is presented in [20].

A number of studies have also tried to exploit the multiprocessor in order to improve the time predictability of Java garbage collection. The first concurrent GC implementation for the multiprocessors was presented by [21]. This applied the algorithm in [22] on a 64-processor machine. It greatly reduced the pause time of GC to a millisecond level. After this, various kinds of concurrent GC algorithms and implementations were presented by researchers to further improve the efficiency. Xian and Xiong [23] showed that their technique can effectively reduce the memory amount that was used by concurrent real-time GC. Sapphire [24] implemented a copying collector for Java with low overheads and a short pause time. Pizlo et al. [25] proposed two lock-free concurrent GC algorithms CHICKEN and CLOVER that have the pause time in the order of microseconds and compared them with another algorithm STOPLESS [26]. Although these algorithms are designed for and implemented by C\#, it is easy to adapt them to Java as they are similar in many aspects.

Besides all the above software approaches a hardware implementation of JVM called as a Java processor is also presented as a solution for real-time systems. Basically, a Java processor is a stack based processor and it directly executes Java bytecode (JBC). In the Java processors, method and stack caches take the places of instruction and data caches respectively. It is possible that Java processors are designed to be deterministic in terms of the execution time. Komodo [27] is an early implementation of a Java processor that provides main Java features and supports real-time tasks. [28] continued working on the Komodo processor with advanced scheduling and event-handling algorithms. SHAP [29] is another Java processor that is specifically designed for real-time systems. It implements fast context switching and concurrent GC. JOP [30, 31] is a well engineered Java processor where WCET can be analyzed. Method caches in JOP simplify the analysis of WCET as only thread switching can introduce cache misses. Tools for performing WCET analysis on JOP is provided in [32]. The high-level WCET analysis is based on integer linear programming (ILP) and the low-level timing model is provided by JOP properties. Similar works have also been done in [33, 34]. Besides, Harmon and Klefstad [35] adapted their work of WCET annotations to Java processors. They made the WCET analysis interactive to developers in order to encourage feedbacks.

In addition to software and hardware based techniques to improve the time predictability of Java computing, several efforts have been made to accurately estimate the WCET of Java programs. [36] described a general method based on the ILP technique that can be applied to Java programs. [37] provided an implementation of the WCET analyzer which is based on Java annotations. [38] modified the Kaffe [39] and Komodo [27] to facilitate WCET analysis of Java applications that run on the two JVMs. In these studies, both control flow and data flow are considered. The works in [40, 41] gave various extensions such as loop bounds, timing modes and dynamic dispatch semantics into WCET analysis. [42] then tried to provide a standard of Java annotations for WCET analysis and this was based on all the previous works.

Our research in this paper is orthogonal to the above

mentioned related work. To the best of our knowledge, our work is first to exploit multicore processors in order to reduce the unpredictability of dynamic compilation and adaptive optimization of Java computing while achieving comparable performance. As multicores are made available, there is a trend of having more cores in a chip. We believe that exploiting the concurrent execution on more than one core can be a promising method to support high-performance in real-time Java computing.

VI. CONCLUSION

Recently there have been growing interests in using Java for a wide variety of both soft- and hard-real-time systems. This is primarily due to Java's attractive features such as platform independence, scalability and safety. Dynamic compilation and adaptive optimization are crucial techniques for the improvement of performance in Java programs. However, these two features are detrimental to time predictability of Java computing. Most of the real-time Java researches are focused on addressing the time predictability of garbage collection, while either ignoring the time unpredictability of dynamic compilation and adaptive optimization or simply disabling them even if the performance is much degraded. Therefore, it is important to improve the time predictability or reduce the time variation of dynamic compilation and adaptive optimization without significantly affecting the performance.

With the widespread usage of multicore computing, this paper proposes several methods to exploit multicores for improving time predictability of Java computing while attaining performance comparable to that with traditional dynamic compilation and adaptive optimization. Specifically, the PoAC approach can efficiently eliminate interruptions due to dynamic compilation. Hence, it can improve time predictability. With good heuristics and tight WCET information, we believe most of the run-time (i.e., after the boot-up stage of JVM) interruptions can be avoided by PcAC scheme. We also observe that both smaller execution time variation and higher performance by migrating optimizations to an auxiliary core. Furthermore, we propose PoAC to replace the AOS, which is very timing sensitive. This in turn brings a lot of unpredictability to Java computing. Our experimental results indicate that PoAC is able to greatly reduce the variation in the execution time, even when subject to different inputs. At the same time, the performance of PoAC is competitive with AOS. Moreover, we implement CBO which also reduces execution time variation, and we compare it with PoAC. Despite the fact that CBO achieves a reduction of execution time variation compared to that of the PoAC, it shows worse performance. It increases the complexity of timing analysis by injecting code into programs. To summarize, the PoAC appears to be the best for real-time Java computing scenarios.

All these studies on time-predictable dynamic compilation/optimization are just the first step towards our final goal of designing a fully time predictable real-time JVM that has high-performance and is platform independent. In our future work, we like to study on the architectural impacts (such as the number of cores and cache configurations) on the proposed techniques by using an extensive simulation. While this paper focuses on compilation, it will be interesting to study on the operating system scheduling to schedule the application threads and compilation/optimization threads on multicores to ensure time predictability while improving the utilization of the multiple cores. Finally, we are interested in using real-time Java benchmarks and in the integration of our approach with formal WCET analysis to accurately estimate the worst-case performance of our approaches.

REFERENCES

1. J. Gosling, G. Bollella, P. Dibble, S. Furr, and M. Turnbull, The Real-time specication for Java, Boston, MA: Addison-Wesley, 2000.
2. F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek, "Real-time Java scoped memory: design patterns and semantics," *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Vienna, 2004, pp. 101-110.
3. C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao, "Scoped types and aspects for real-time Java memory management," *Real-Time Systems*, vol. 37, no. 1, pp. 1-44, 2007.
4. W. S. Beebee and M. C. Rinard, "An implementation of scoped memory for real-time Java," *Proceedings of the First International Workshop on Embedded Software*, Tahoe City, CA, 2001, pp. 289-305.
5. IBM, "WebSphere real time," http://www-01.ibm.com/software/webservers/realtime/#.
6. Oracle, "Sun Java real-time system," http://java.sun.com/javase/technologies/realtime/index.jsp.
7. Purdue University College of Science, "Open virtual machine (OVM)," http://www.cs.purdue.edu/homes/jv/soft/ovm/.
8. D. F. Bacon, P. Cheng, and V. T. Rajan, "The metronome: a simpler approach to garbage collection in real-time systems," *OTM Workshops*, 2003, pp. 466-478.
9. D. F. Bacon, P. Cheng, and V. T. Rajan, "A real-time garbage collector with low overhead and consistent utilization," *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New Orleans, LA, 2003, pp. 285-298.
10. H. Cho, C. Na, B. Ravindran, and E. D. Jensen, "On scheduling garbage collector in dynamic real-time systems with statistical timing assurances," *Real-Time Systems*, vol. 36, no. 1-2, pp. 23-46, 2007.
11. S. G. Robertz and R. Henriksson, "Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems," *ACM SIGPLAN Notices*, vol. 38, no. 7, pp. 93-102, 2003.
12. M. Kero, J. Nordlander, and P. Lindgren, "A correct and

useful incremental copying garbage collector," *Proceedings of the 6th International Symposium on Memory Management*, New York, NY, 2007, pp. 129-140.

13. F. Siebert, "Hard real-time garbage-collection in the Jamaica virtual machine," *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, Hong Kong, 1999, pp. 96-102.
14. J. M. Calandrino, J. H. Anderson, and D. P. Baumberger, "A hybrid real-time scheduling approach for large-scale multicore platforms," *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, Washington, DC, 2007, pp. 247-258.
15. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom, "The worst-case execution-time problem: overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1-53, 2008.
16. The Jikes RVM project, "Jikes RVM," http://www.jikesrvm.org/.
17. Standard Performance Evaluation Corporation (SPEC), "SPEC JVM98 benchmarks," http://www.spec.org/jvm98/.
18. A. Corsaro and D. C. Schmidt, "The design and performance of the jrate real-time Java implementation," *On the Move to Meaningful Internet Systems, 2002: DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, Irvine, CA, 2002, pp. 900-921.
19. H. G. Baker Jr, "List processing in real time on a serial computer," *Communications of the ACM*, vol. 21, no. 4, pp. 280-294, 1978.
20. W. J. Schmidt and K. D. Nilsen, "Performance of a hardware-assisted real-time garbage collector," *Proceedings of the 6th International Conference on Architectural support for Programming Languages and Operating Systems*, New York, NY, 1994, pp. 76-85.
21. P. Cheng and G. E. Blelloch, "A parallel, real-time garbage collector," *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, New York, NY, 2001, pp. 125-136.
22. G. E. Blelloch and P. Cheng, "On bounding time and space for multiprocessor garbage collection," *ACM SIGPLAN Notices,* vol. 34, no. 5, pp. 104-117, 1999.
23. Y. Xian and G. Xiong, "Minimizing memory requirement of real-time systems with concurrent garbage collector," *ACM SIGPLAN Notices*, vol. 40, no. 3, pp. 40-48, 2005.
24. R. L. Hudson and J. E. B. Moss, "Sapphire: copying GC without stoppingthe world," *Proceedings of the 2001 joint ACM-ISCOPE Conference on Java Grande*, New York, NY, 2001, pp. 48-57.
25. F. Pizlo, E. Petrank, and B. Steensgaard, "A study of concurrent real-time garbage collectors," *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, 2008, pp. 33-44.
26. F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard, "Stopless: a real-time garbage collector for multiprocessors," *Proceedings of the 6th International Symposium on Memory Management*, New York, NY, 2007, pp. 159-172.
27. U. Brinkschulte, C. Krakowski, J. Kreuzinger, and T. Ungerer, "A multithreaded Java microcontroller for thread-oriented real-time event handling," *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, 1999, p. 34.
28. J. Kreuzinger, U. Brinkschulte, M. Pfeer, S. Uhrig, and T. Ungerer, "Real-time event-handling and scheduling on a multithreaded java microcontroller," *Microprocessors and Microsystems*, vol. 27, no. 1, pp. 19-31, 2003.
29. M. Zabel, T. B. Preuber, P. Reichel, and R. G. Spallek, "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*, Washington, DC, 2007, pp. 59-62.
30. M. Schoeberl, "JOP: a Java optimized processor for embedded real-time systems," Ph.D. dissertation, Vienna University of Technology, Vienna, 2005.
31. M. Schoeberl, "A Java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. 54, no. 1-2, pp. 265-286, 2008.
32. M. Schoeberl and R. Pedersen, "WCET analysis for a Java processor," *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems*, New York, NY, 2006, pp. 202-211.
33. Z. Chai, Z. Q. Tang, L. M. Wang, and S. L. Tu, "An effective instruction optimization method for embedded real-time Java processor," *Proceedings of the 2005 International Conference on Parallel Processing Workshops*, Washington, DC, 2005, pp. 225-231.
34. Z. Chai, W. Zhao, and W. Xu, "Real-time Java processor optimized for RTSJ," *Proceedings of the 2007 ACM Symposium on Applied Computing*, New York, NY, 2007, pp. 1540-1544.
35. T. Harmon and R. Klefstad, "Interactive back-annotation of worst-case execution time analysis for java microprocessors," *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Washington, DC, 2007, pp. 209-216.
36. P. Puschner and G. Bernat, "WCET analysis of reusable portable code," *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, Washington, DC, 2001, p. 45.
37. G. Bernat, A. Burns, and A. Wellings, "Portable worst-case execution time analysis using Java byte code," *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stockholm, 2000, pp. 81-88.
38. I. Bate, G. Bernat, and P. Puschner, "Java virtual-machine support for portable worst-case execution-time analysis," *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Washington, DC, 2002, pp. 83-90.
39. "The Kaffe virtual machine," http://www.kaffe.org/.
40. E. Y. Hu, A. Wellings, and G. Bernat, "XRTJ: an extensible distributed high-integrity real-time Java environment," *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, Tainan, Taiwan, 2003, pp. 208-228.
41. E. Y. Hu, G. Bernat, and A. Wellings, "Addressing dynamic dispatching issues in WCET analysis for object-oriented hard real-time systems," *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Washington, DC, 2002, pp. 109-116.
42. T. Harmon and R. Klefstad, "Toward a unied standard for worst-case execution time annotations in real-time Java," *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, 2007, pp. 1-8.

Yu Sun

Yu Sun received his Ph.D. degree in Electrical and Computer Engineering in 2010 at Southern Illinois University Carbondale, Carbondale, IL, USA, and B.E. degree in Computer Science at Tsinghua University, Beijing, China in 2003. His research interests include compiler optimization, parallelization, embedded system and real-time system. He is currently working as a senior software engineer at MathWorks, Inc.

Wei Zhang

Dr. Wei Zhang is an associate professor in Electrical and Computer Engineering of Virginia Commonwealth University. Dr. Wei Zhang received his Ph.D. from the Pennsylvania State University in 2003. From August 2003 to July 2010, Dr. Zhang worked as an assistant professor and then as an associate professor at Southern Illinois University Carbondale. His research interests are in embedded and real-time computing systems, computer architecture, compiler, and low-power systems. Dr. Zhang has received the 2009 SIUC Excellence through Commitment Outstanding Scholar Award for the College of Engineering, and 2007 IBM Real-time Innovation Award. His research has been supported by NSF, IBM, Intel, Motorola and Altera. He is a senior member of the IEEE. He has served as a member of the organizing or program committees for several IEEE/ACM international conferences and workshops.