Worst-Case Execution Time Based Optimization of Real-Time Java Programs

Stefan Hepp
Institute of Computer Languages
Technical University of Vienna
Email: hepp@complang.tuwien.ac.at

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
Email: masca@imm.dtu.dk

***Abstract*—Standard compilers optimize execution time for the average case. However, in hard real-time systems the worst-case execution time (WCET) is of primary importance. Therefore, a compiler for real-time systems shall include optimizations that aim to minimize the WCET.**

One effective compiler optimization is method inlining. It is especially important for languages, like Java, where small setter and getter methods are considered good programming style. In this paper we present and explore WCET driven inlining of Java methods. We use the WCET analysis tool for the Java processor JOP to guide to optimization along the worst-case path. The tool JCopter is integrated with the WCET analysis tool and is used to explore different inlining strategies. On real-time benchmarks the optimization results in a reduction of the WCET by a few percent up to a factor of about 2.

I. Introduction

Common computer architectures and compilers are designed to optimize the average case performance. However, for real-time systems we are interested in the worst-case execution time (WCET). Some average case optimizations (e.g., higher clock frequency or register allocation in a compiler) will also reduce the WCET. However, some average-case optimizations can actually increase the WCET.

For real-time systems we are interested in building new computer architectures [16] and supporting software (compilers [23], Java virtual machines [15], libraries [7]) that are optimized for the WCET. In this paper we describe the optimization of Java applications for the WCET. The main optimization method employed is method inlining.

Java applications are compiled to class files, which contain Java bytecode for execution by a Java virtual machine (JVM). This bytecode can be interpreted, compiled during run time (just-in-time (JIT) compilation), compiled ahead of time, or executed by a hardware implementation of the JVM (a Java processor). While JIT compilation is a problematic option for real-time systems, the other three execution forms are a valuable option for time-predictable real-time systems.

In this paper we show a tool flow that targets execution of bytecodes on the Java processor JOP [15]. We have chosen JOP as the first target as JOP is optimized for time-predictable execution of Java programs and an open-source WCET analysis tool, WCA [18], is available.

The optimization and transformation of Java applications is performed at the Java bytecode level. The tool JCopter is integrated with the WCET analysis tool for JOP. Although JCopter uses the timing model of JOP to guide the WCET oriented optimization, the output of JCopter are still standard class files. Therefore, this optimization can also be used on other real-time Java platforms.

The paper is organized as follows. The following section presents background on real-time Java and WCET analysis for Java. Section III discusses worst-case driven method inlining. The concrete implementation is presented in Section IV. The results are presented in Section V. Section VI presents related work. The paper is concluded by Section VII.

All tools (JCopter, WCA, JOP) are available in open-source under the GNU GPL. Download instructions can be found at: http://www.jopwiki.com/Download.

II. Background

We base our design and implementation of the WCET driven optimization on real-time Java applications. The described method is general, but for the evaluation we target the execution platform JOP [15]. JOP is a Java processor that is optimized for time-predictable execution of Java bytecode and to simplify WCET analysis. The distribution of JOP also includes a WCET analysis tool [18] that is used to guide the optimizer JCopter.

A. Real-Time Java

Java for real-time systems comes in two flavors: the Real-Time Specification for Java (RTSJ) [2] and Safety-Critical Java (SCJ) [9], [11]. The RTSJ targets soft real-time applications, whereas SCJ is intended to be used in high integrity systems and shall enable certification according to the highest criticality levels. Therefore, SCJ applications need to undergo, besides other certification actions, schedulability and WCET analyses.

With our system we target the SCJ application domain. SCJ is a subset of the RTSJ with a few additional classes to simplify real-time application programming and also the development of a SCJ compliant Java virtual machine (JVM) and runtime libraries. The individual tasks are organized as handlers, where the SCJ infrastructure releases these handlers either periodically or as a result of an (hardware or software) event. Each handler has to implement a single method. In contrast to the RTSJ, this method does not contain the periodic loop with waitForNextRelease(). The single method is *executed* every release and is therefore the entry point for WCET analysis and also the target method for our WCET driven optimization.

B. WCET Analysis

Hard real-time systems need a formal proof that all tasks meet their deadline. Schedulability or response time analysis gives this mathematical proof. For those analyses the WCET of tasks and synchronized code sections needs to be known. The WCET can usually not be measured. Instead, a static WCET analysis has to be used to derive an upper bound of the WCET. An overview of current state-of-the-art WCET tools can be found in [21].

WCET analysis reconstructs the control flow graph from the application binary, needs loop and recursion bounds, and calculates execution times of basic blocks. With this information, the problem of finding the worst case path is formulated as a network flow problem and usually solved as an integer linear programming problem. This is also called implicit path enumeration technology (IPET).

The binary format of Java programs are class files. These class files contain more information than e.g., a binary of a C/C++ compiled program. It is possible to reconstruct the whole class structure from this class file. Therefore, the reconstruction of all needed information for the program analysis is easier than for *normal* binaries.

The harder problem is the analysis of the low-level timing – the execution time of basic blocks. This timing depends on the target execution platform. Therefore, WCET analysis tools contain processor specific versions of the low-level analysis. Modern processors with a lot of hidden state (caches, branch predictors, write buffers, reservation stations, reorder buffers,...) are very hard to model for the WCET analysis. Furthermore, the large state results in long execution times of the pipeline simulation to derive the basic block timings.

We attack this low-level WCET analysis problem by building a processor that is WCET analysis friendly. The Java processor JOP [15] is designed to simplify WCET analysis of Java programs. JOP executes Java bytecodes (the instruction set of the JVM) directly. Therefore, no further translation, which would increases the complexity of the WCET analysis, to machine code is needed. The execution time of most bytecodes is constant. JOP also contains a so called method cache [14], which caches whole methods and is intended to simplify the cache analysis.

The simple timing model of JOP has led to several WCET analysis projects that support JOP as their target [1], [6], [8], [10]. The WCET analysis tool WCA is part of the JOP source distribution [18]. We used WCA to provide WCET bounds and worst-case path information to the optimizer.

C. Cost of Method Invocation

Invoking a virtual method in Java has considerable overhead. On JOP, the invocation of a method costs around 100 cycles. Although this might sound a high number, other Java processors, such as aJile [5], jamuth [19], or SHAP [22], have similar execution times for a method invocation. Java method invocation is more expensive than calling a C function, as the dynamic dispatch, where the receiver method depends on the object type, needs several memory accesses and the stack frame management is slightly more complex in Java than in C.

As method invocation is so costly in Java and good programming practice suggests to hide object fields with short *getter* and *setter* methods, method inlining is a good target for program optimization.

JOP uses a novel instruction cache called method cache [14] that caches whole methods. If the control flow is transferred from one method to another method by a call or a return, the processor loads the method to be executed into the instruction cache if the cache does not already contain the method. A FIFO replacement strategy is used to evict methods from the cache.

The advantage of the method cache is that instruction cache misses only appear at call and return instructions. All other instructions are guaranteed instruction cache hits, which simplifies the WCET analysis. The method cache content is determined only by the sequence of method calls and returns, as well as by the code size of the methods. However, increasing the code size of a method not only increases the instruction cache miss costs associated with that method, it can also increase the number of method cache misses at call sites of other methods. This reduces the gain of optimizations that increase the code size.

III. WCET Driven Method Inlining

Method inlining replaces a call site with the invoked method. Additional code needs to be added to the call site that mimics the behavior of the invoke instruction in the original code, such as storing the parameters into new local variables and performing a null-pointer check for the receiver of the invocation. In the general case, eliminating the invoke instruction reduces the execution time at the call site, but due to the code size increase of the caller the instruction cache costs can increase.

Since we are interested in reducing the WCET, there is no advantage in inlining a call site that is not on the worst-case path. Instead, the increased instruction cache costs may even lead to an increase of the WCET. Therefore we use the WCET analysis tool WCA to find the worst-case path during optimization. Call sites that are not on the worst-case path are not considered for inlining. On the other hand, reducing the execution time of the worst-case path can cause another path to be the new worst-case path. Therefore, to keep the worst-case path information up-to-date during optimization, we perform the WCET analysis after every optimization step again.

In order to decide whether to inline a call site or not, we first estimate the gain g of inlining the call site as

$$g = g_l * f - \Delta\text{cache-miss-costs}$$

where g_l is the gain for a single execution of the inlined call site (ignoring cache costs) and f is the estimated execution frequency of that call site per single execution of the target method of the optimization. Δcache-miss-costs is the estimated difference of the instruction cache costs caused by inlining the call site per single execution of the target method. Inlining is only performed if the gain is positive.

To estimate the cache costs, a method cache analysis has been implemented. The analysis first classifies all method cache accesses (invoke instructions and return instructions) as *always hit* (each execution of the instruction is a cache hit), *always miss* (each execution is a cache miss), or *at most one miss* (the accessed method is persistent during the execution of some scope). Based on those classifications and on the estimated execution frequencies of the cache accesses, the analysis then calculates cache miss cost changes for given code size changes.

The cache miss costs of a cache access are calculated as the cache miss cost for a single cache miss, multiplied by the number of cache misses. The number of cache misses is derived from the cache access classification and the estimated execution frequency of that call. The cache miss cost of a single cache miss is proportional to the size of the method to load. In case of a virtual invoke with multiple receiver types, the largest method is used to estimate the cache miss costs.

The cache analysis provides the following three cache approximation modes:

- *always miss*: All cache accesses are classified as *always miss*. This is a very conservative, but comparatively simple over-approximation. The number of cache misses is equal to the execution frequency of that instruction. If the code size of a method m is increased, then the cache miss costs of all call sites of m as well as of all return instructions returning to m are increased. No other cache accesses are affected.
- *always hit*: All cache accesses are classified as *always hit*. If this mode is used, the effects of the method cache are ignored by the optimizer, since all cache costs are assumed to be zero.
- *at most one miss*: This mode uses the fact that if all methods that can be invoked during the execution of a method m fit into the method cache (including m), then every cache access during the execution of m is at most one miss [18]. All cache accesses in m are therefore classified as *at most one miss*. If m and the invoked methods do not fit into the method cache, the cache accesses are classified as *always miss*.

Let *reachable*(m) denote the set of methods that can be reached in the call graph from method m, including m. We call the subgraph of the call graph that contains all methods m where *reachable*(m) fits into the method cache the all-fit region. For every scope in the all-fit region, we can classify all cache accesses in those scopes as *at most one miss*. Figure 1 shows the all-fit region of a simple call graph for a method cache that can hold up to four methods (we assume that initially all methods are of the same size in this example). If we would inline for example node 8 into node 6, thus doubling the size of node 6, then node 4 would no longer be in the all-fit region and be reclassified as *always miss*.

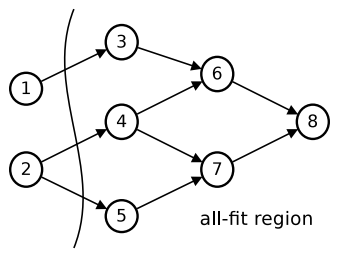

Fig. 1. Call graph with all-fit region for a method cache that can hold up to four methods

If the *at most one miss* analysis mode is used, the number of cache misses of a cache access in the all-fit region depends on the number of times the application can *enter* the all-fit region. Modifying the call graph or changing the size of methods in the all-fit region can increase or shrink the all-fit region. This can affect the number of cache misses and thus the cache miss costs of all nodes that are reachable by the reclassified nodes. If inlining causes other methods to fall out of the all-fit region and be classified as *always miss*, this can lead to large cache cost increases caused by the optimization. It also means that in contrast to the other analysis modes, changing the code size of a method may not only affect the worst-case path in methods that can reach the changed method, but potentially in all methods in the call graph.

Even if we use a very conservative cache cost estimation, the actual gain of inlining a call site can be lower than the estimated gain g if there exists a different path with an execution time greater than $\text{WCET}_{\text{old}} - g$, where WCET_{old} is the WCET prior to optimization. If the WCET of the other path is very close to WCET_{old}, then cache cost increases at call sites on that path can even lead to an *increase* of the overall WCET. It would be possible to use deoptimization to undo the code transformation when the WCET analysis returns a higher WCET bound after an optimization to avoid a WCET increase. However, deoptimization has not yet been implemented in JCopter.

Algorithm 1 shows the algorithm used to implement the WCET-driven method inliner. The algorithm works by iteratively optimizing the best optimization candidate on the worst-case path and then updating the analyses and the evaluation of the candidates, until no more candidates with a positive gain are found, or until the application code size reaches a predefined limit.

First, the optimizer initializes the method cache analysis and performs a first WCET analysis to find the initial worst-case path. Optimization candidates (i.e., call sites that can be inlined) are searched in all methods that are reachable from the target method.

To achieve a good speedup without increasing the code size too much, the optimizer then sorts all candidates by their estimated gain to application code size increase ratio (line 3). The algorithm then selects the candidate with the highest ratio that is on the current worst-case path for optimization and removes it from the set (line 5). If there is no candidate left

Algorithm 1: WCET-driven method inliner

```
1  initialize WCA, cache analysis;
2  $C :=$ FindInitialCandidates();
3  CalculateRatios($C$);
4  while $C \neq \emptyset \wedge$ application-size $\leq$ max-size do
5      curr := RemoveBestCandidate($C$) or return;
6      Optimize(curr);
7      $C := C \cup$ FindNewCandidates();
8      change := UpdateCacheAnalysis();
9      UpdateWCA(change);
10     UpdateRatios($C$, change);
11 end
```

that is on the current worst-case path and that has a gain $g > 0$, the algorithm terminates.

The selected candidate is then optimized. This can create new optimization candidates, i.e., call sites in the inlined code are added to the candidate set (line 7). The cache analysis must then be updated using the new code size of the optimized method. It returns a set of methods that contain cache accesses for which the cache costs may have changed. For the *always hit* cache analysis mode, this set is always empty, for the *always miss* mode the set contains the optimized method and all direct callers of that method. If the *at most one miss* analysis mode is used, this set can potentially contain the whole all-fit region.

Afterwards, the WCET analysis is rerun to update the current worst-case path. The analysis is performed for all methods that can reach the optimized method or any method in change in the call graph, since a change of the WCET of any of those methods can change the worst-case path in any caller of that method. WCET results of other methods are reused.

Finally, the gain estimations and thus the gain per code-size increase ratios must be recalculated (line 10). Since the gain estimation depends on the cache analysis, all ratios of all candidates in methods in change must be updated. Inlining also decreases the ratios of the callers since the code size increases. Furthermore, candidates that can no longer be optimized are removed from the set. Since the maximum code size and the maximum number of local variables of a method are limited on JOP, inlining a call site can prevent other call sites from being optimized at a later time.

Note that the algorithm is not specific to inlining. It would be possible to use it also for other optimizations that reduce the execution time while increasing the code size, or even to choose between candidates provided by different optimizations, such as inlining and loop transformations. However, this has not yet been explored.

IV. Implementation

We created a tool called JCopter that implements WCET-driven method inlining for Java. It uses the WCET analysis tool WCA to determine the worst-case path as well as worst-case execution frequencies for the optimization target methods.

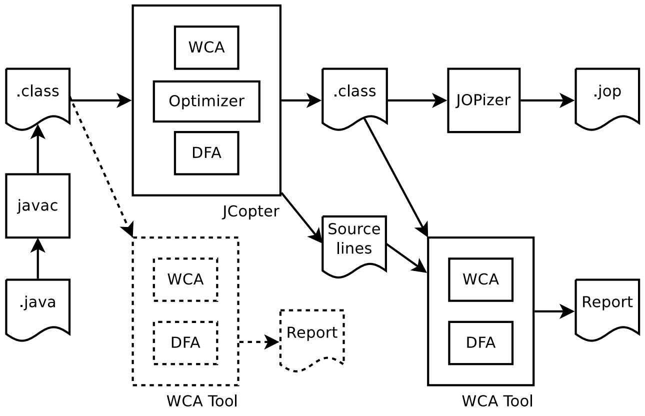

Fig. 2. The JOP tool chain

Before a call site can be inlined, the set of possibly invoked method implementations must be determined. Inlining is only possible if this set contains exactly one non-native method, i.e., when the invoked method is statically known. To find the implementations of a call site, JCopter can use either a class hierarchy analysis, or a receiver type data-flow analysis (DFA) that is also part of the JOP source distribution and that is also used by WCA [18].

The optimizer JCopter, the WCET analysis WCA, and the data-flow analysis are implemented on top of a common framework. This allows for a simple and efficient interaction between the tools. After an optimization, the WCET analysis is restarted only for methods where the worst-case path or the WCET may have changed, i.e., for methods where the cache analysis results change and for methods that can reach the optimized method in the call graph. Analysis results for other methods are reused.

The tool chain for compiling applications for JOP is shown in Figure 2. JCopter takes Java class files as input and performs method inlining. The optimized program is then stored as Java class files again. The JOPizer tool creates the binary that can be downloaded to the processor. To support executing WCA on the optimized application to calculate the WCET bound of the optimized application, we also store references to the source code annotations for the inlined code in a database. This is required to supply user-supplied loop bound annotations to the WCET analysis. Optionally, WCA can also be used to analyze the unoptimized application in case the optimizer is not used, or to compare the WCET bounds of the optimized and the unoptimized application.

A second inliner, called SimpleInliner, has been implemented in JCopter, which exploits the fact that on JOP inlining has no adverse side effects on the WCET as long as the code size of a method is not increased. This optimizer is used to inline getter, setter, stubs, and some wrapper methods, as well as methods containing small arithmetic expressions. In contrast to the generic method inliner presented above, SimpleInliner uses a different method to modify the prologue at the call site that does not copy the parameters into new local variables. Instead, the inlined code only uses the stack.

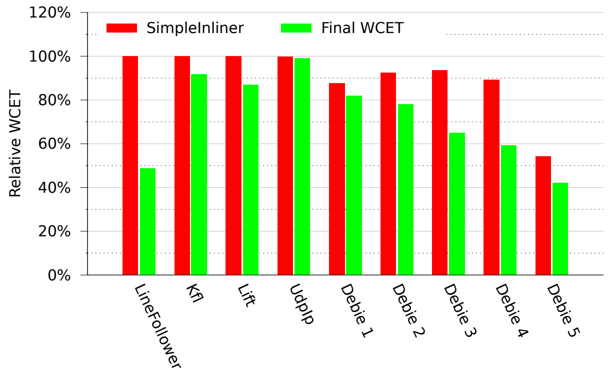

Fig. 3. WCET bound reduction due to SimpleInliner and final WCET bound reduction, compared to unoptimized code

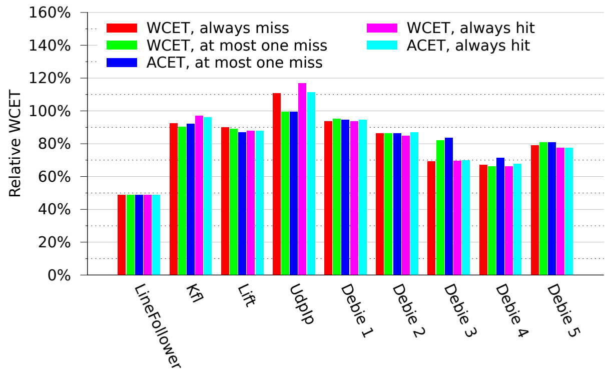

Fig. 4. WCET bound reduction for various inliner configurations

While this restricts the set of methods that can be inlined, it generates a more efficient code, both in terms of code size and speed. Interaction with the WCET analysis is not required. SimpleInliner is executed prior to the WCET-driven inliner, so that small methods are inlined using the fast optimizer, while the slower WCET-driven inliner is only used for larger methods.

JCopter also implements optimization passes that remove unused methods, fields and classes, as well as unused constants to reduce the size of the application binary and to remove methods that are no longer called after inlining. They have no negative impact on the execution time.

V. Evaluation

We evaluate JCopter with several real-time Java benchmarks: the embedded Java benchmark suit JemBench [17], a small line following robot application, and the WCET benchmark DEBIE.[1]

We use the application benchmarks Kfl, Lift, and UdpIp from the JemBench suite. The benchmarks have been created from real-world controller applications. The UdpIp and Lift benchmarks use virtual invocations, while the Kfl application is implemented only with static methods. New objects are only created during initialization. The JemBench applications have been written with the high invocation costs in mind, therefore getter and setter and similar small methods are not used in those benchmarks. Benchmark WCET results for the unoptimized JemBench benchmarks and the line-follower benchmark have been published in [18].

The Java port of the DEBIE benchmark has been used to test the optimizer with a larger application. The benchmark is based on the on-board software of the DEBIE space debris impact monitoring instrument that is written in C. However, the Java port of the benchmark has been written using a more object oriented approach, giving the inliner far more optimization opportunities than the JemBench benchmarks. Three interrupt handlers and two periodic tasks of the DEBIE benchmark have been used for evaluation. WCET results of the unoptimized Java port of the benchmark have been published in [20].

The context-sensitive data-flow analysis was used for all benchmarks to analyze receiver types and to find loop bounds that are used by the WCET analysis for the execution frequency estimation.

The benchmarks were first optimized using SimpleInliner only, i.e., only very small methods were inlined, without feedback from the WCET analysis or the cache analysis. Figure 3 shows the reduction of the WCET bound that has been achieved with this simple optimization, relative to WCET bounds of the unoptimized applications. While the speedup of the JemBench applications is negligible, SimpleInliner managed to reduce the WCET bounds of the Debie 1 to Debie 4 benchmarks by about 9% on average, while the WCET bound of Debie 5 was even reduced by 45%.

The benchmarks were then further optimized using the WCET-driven inliner. Figure 4 shows the WCET bound reduction that has been achieved using the WCET-driven inliner after small methods have been inlined (i.e., not including the reduction achieved by SimpleInliner). The following optimizer configurations are shown in this figure:

- **WCET-driven, always miss**: Only call sites on the worst-case path are optimized. All cache accesses are classified as *always miss*, leading to high cache costs and low gain estimations. This is the most conservative optimizer configuration, using a comparatively simple method cache costs analysis.
- **WCET-driven, at most one miss**: Again, the optimizer only inlines call sites on the current worst-case path. Cache accesses are classified as *at most one miss* if they are in the all-fit region, else they are classified as *always miss*. This configuration provides a more precise cache analysis, but the complexity of the cache analysis increases. The cache analysis uses execution frequencies provided by WCA for the worst-case path. Cache costs outside the worst-case path are therefore ignored. This results in an overestimation of the gain of inlining a call site in case the worst-case path changes. However, accounting for cache costs on paths other than the worst-case path lead to an overly pessimistic optimization

[1]https://gate.etamax.de/edid/publicaccess/debie1.php

behavior in our benchmarks.

- **ACET-driven, at most one miss**: This configuration is similar to the above configuration, except that the WCET analysis is not used to find the worst-case path. Instead, all call sites are potential optimization candidates. The optimizer uses over-estimations for the execution frequencies of all call sites.
- **WCET-driven, always hit**: Only call sites on the worst-case path are optimized. However, the cache analysis is effectively disabled. All cache costs are assumed to be zero, leading to more aggressive optimization.
- **ACET-driven, always hit**: Both the cache analysis and the WCET analysis are disabled. All call sites are optimization candidates, cache accesses are always classified as *always hit*. This is the most aggressive optimizer configuration.

The best total WCET bound reduction after both SimpleInliner and the WCET-driven inliner have been applied, compared to the unoptimized applications, is shown in Figure 3. The line following application completely fits in into the method cache, and the optimizer was able to inline nearly all call sites without large cache miss penalties. For all inliner configurations, the WCET bound was reduced by 51%.

For the JemBench applications, the WCET-driven inliner could only reduce the WCET bound by between 1% (UdpIp) and 13% (Lift) in the best case. SimpleInliner did not contribute to the speedup. However, some configurations of the inliner even lead to an increase of the WCET bound of UdpIp. In this benchmark, some call sites caused significant WCET bound increases when they were inlined due to an incorrect gain estimation, rendering the WCET speedup that has been achieved by inlining other call sites void.

The DEBIE benchmarks on the other hand benefited significantly from inlining. WCET bound reductions between 18% (Debie 1) and 58% (Debie 5) were observed, partially due to the performance of SimpleInliner.

Inlining and removal of unused methods (as well as fields and constants) leads also to a considerable reduction of the code size. The code size of the JemBench has been reduced by about 58%, while the code size of the DEBIE benchmark has been reduced by about 32% after optimization. Inlining did not increase the code size, since the inliner prefers to inline small methods or methods that are only called once, due to the instruction cache costs. The code size even decreased slightly after inlining, since the optimizer was able to remove more classes and constants after inlining.

The runtime of the optimizer primarily depends on the number of call sites in the code. The WCET-driven inliner takes about 10 seconds for Kfl and about 34 seconds for the Debie 5 benchmark on a 2.8 GHz desktop PC, while all other benchmarks have been optimized within a few seconds. Most of the time is spent in WCA for updating the worst-case path. Disabling the WCA therefore reduces the time required for optimization to less than a second for all benchmarks, while still achieving good WCET bound improvements.

VI. Related Work

Zhou et al. present a fast inlining algorithm for embedded systems in [24] that also selects call sites based on a heuristic *rebate_ratio* to achieve a maximum gain without increasing the application code size beyond a predefined limit. The call sites with the highest calling frequency per estimated code size increase are selected first for inlining. However, the algorithm is designed to target the average case execution time and does not take cache costs into account.

Zhao et al. present path optimizations that are applied on the worst-case path [23], such as superblock formation (i.e., merging basic blocks into a region that has one entry but multiple exits) and duplication of the worst-case path within loops. The optimizations are used to create new optimization opportunities on the worst-case path, but they also increase the code size of the functions. However, the optimizations were evaluated on a target architecture that does not have an instruction cache.

The WCET-aware C Compiler (WCC) provides a framework for several WCET analysis driven optimizations [4]. WCC uses the AbsInt aiT WCET analyzer[2] to calculate WCET bounds for basic blocks and tasks and to find the worst-case path. Code optimizations are performed both on a high-level intermediate representation and on a low-level intermediate representation (LLIR). WCET analysis is performed on the LLIR, and back-annotation is used to map WCET analysis results back to the high-level representation.

WCC includes a WCET-driven function inliner that uses decision trees to decide on whether to inline a call site or not [13]. The decisions are based on call site features such as the code size and the WCET of the involved methods, the number of call sites in the methods, or the register pressure at the call site. Supervised machine learning is used to create the decision trees. The gain of inlining the call site is assessed for several call sites in benchmark applications using the WCET analysis. This information is used as training set for the machine learner. The inlining heuristic can be automatically tuned to new architectures simply by using a different hardware model for the WCET analysis and retraining the heuristics. The architecture used for evaluation is a Infineon TriCore TC1796 that includes both a scratchpad instruction cache and a set associative instruction cache.

To reduce the instruction cache costs, Falk et al. perform a static WCET-aware scratchpad allocation in WCC [3]. The WCET of the program is modeled using integer linear programming (ILP), which contains decision variables for each basic block that decide if a basic block should be left in the main memory or if it should be placed into the faster but smaller scratchpad memory. An ILP solver can then be used to find an optimal allocation of basic blocks to the instruction scratchpad memory.

The WCC compiler also uses the Invariant Path paradigm [12] to find paths that are guaranteed to be part of the worst-case path. The authors demonstrate that this information

[2]http://www.absint.com/ait/

can be used to reduce the number of times the WCET analysis needs to be restarted during compilation, which drastically reduces the time required for WCET optimization.

To our knowledge, this is the first work that explores compiler optimizations for architectures that employ a method cache.

VII. Conclusion and Future Work

Standard compilers optimize code to minimize average-case execution time. Such optimization might reduce the worst-case execution time (WCET), but it can also increase it. In real-time systems we are mainly concerned about the WCET. Therefore, a compiler shall minimize the WCET. In this paper we presented the optimization tool JCopter, which is integrated with a WCET analysis tool, and performs program optimization along the worst-case path to reduce the WCET. The main optimization method evaluated is method inlining. On real-time Java applications JCopter achieves a reduction of the WCET of up to a factor of 2.

We will extend the presented work in the FP7 T-CREST (Time-predictable Multi-Core Architecture for Embedded Systems) project. We plan to integrate an improved WCET-driven function inliner into a C compiler that is based on LLVM. WCET information will be supplied by the AbsInt WCET analysis tool aiT. We also plan to explore other optimizations such as WCET-driven function splitting that will help to reduce the instruction cache costs.

Acknowledgment

Part of this work was supported by the FP7-ICT Project 288008 Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST). We also would like to thank Benedikt Huber for his help on integrating JCopter with the WCET analysis tool WCA.

References

[1] 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)*, pages 106–114, New York, NY, USA, 2008. ACM.

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

[3] H. Falk and J. C. Kleinsorge. Optimal static WCET-aware scratchpad allocation of program code. In *The 46th Design Automation Conference (DAC)*, pages 732–737, San Francisco / USA, jul 2009.

[4] H. Falk and P. Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Journal on Real-Time Systems*, 46(2):251–300, oct 2010. DOI 10.1007/s11241-010-9101-x.

[5] 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*, page 53. IEEE Computer Society, 2001.

[6] T. Harmon. *Interactive Worst-case Execution Time Analysis of Hard Real-time Systems*. PhD thesis, University of California, Irvine, 2009.

[7] T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad. Toward libraries for real-time Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, pages 458–462, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[8] T. Harmon, M. Schoeberl, R. Kirner, R. Klefstad, K. K. Kim, and M. R. Lowry. Fast, interactive worst-case execution time analysis with back-annotation. *IEEE Transactions on Industrial Informatics*, accepted for publication, 2012.

[9] 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)*, York, United Kingdom, Mar. 2009.

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

[11] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Wellings. Safety-critical Java technology specification, public draft, 2011.

[12] P. Lokuciejewski, F. Gedikli, and P. Marwedel. Accelerating WCET-driven optimizations by the invariant path paradigm - a case study of loop unswitching. In *The 12th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 11–20, Nice / France, apr 2009.

[13] P. Lokuciejewski, F. Gedikli, P. Marwedel, and K. Morik. Automatic WCET reduction by machine learning based heuristics for function inlining. In *Proceedings of the 3rd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART)*, pages 1–15, Paphos / Cyprus, jan 2009.

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

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

[16] M. Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.

[17] M. Schoeberl, T. B. Preusser, and S. Uhrig. The embedded Java benchmark suite JemBench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, pages 120–127, New York, NY, USA, August 2010. ACM.

[18] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[19] 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)*, pages 230–237, New York, NY, USA, 2007. ACM Press.

[20] R. von Hanxleden, N. Holsti, B. Lisper, E. Ploedereder, R. Wilhelm, A. Bonenfant, H. Casse, S. Bnte, W. Fellger, S. Gepperth, J. Gustafsson, B. Huber, N. M. Islam, D. Kstner, R. Kirner, L. Kovacs, F. Krause, M. de Michiel, M. C. Olesen, A. Prantl, W. Puffitsch, C. Rochange, M. Schoeberl, S. Wegener, M. Zolda, and J. Zwirchmayr. Wcet tool challenge 2011: Report. In *Proceedings of the 11th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2011.

[21] 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. Stenström. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[22] 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)*, pages 59–62, Lübeck, Germany, Aug. 2007.

[23] W. Zhao, W. Kreahling, D. Whalley, C. Healy, and F. Mueller. Improving WCET by applying worst-case path optimizations. *Real-Time Syst.*, 34:129–152, October 2006.

[24] X. Zhou, L. Yan, and J. Lilius. Function inlining in embedded systems with code size limitation. In *Proceedings of the 3rd international conference on Embedded Software and Systems*, ICESS '07, pages 154–161, Berlin, Heidelberg, 2007. Springer-Verlag.