Fast, Interactive Worst-Case Execution Time Analysis With Back-Annotation

Trevor Harmon, Martin Schoeberl *Member, IEEE*, Raimund Kirner *Member, IEEE*, Raymond Klefstad, K.H. (Kane) Kim *Fellow, IEEE*, and Michael R. Lowry

***Abstract*—For hard real-time systems, static code analysis is needed to derive a safe bound on the worst-case execution time (WCET). Virtually all prior work has focused on the accuracy of WCET analysis without regard to the speed of analysis. The resulting algorithms are often too slow to be integrated into the development cycle, requiring WCET analysis to be postponed until a final verification phase.**

In this paper we propose *interactive WCET analysis* as a new method to provide near-instantaneous WCET feedback to the developer during software programming. We show that interactive WCET analysis is feasible using tree-based WCET calculation. The feedback is realized with a plugin for the Java editor jEdit, where the WCET values are back-annotated to the Java source at the statement level. Comparison of this tree-based approach with the implicit path enumeration technique (IPET) shows that tree-based analysis scales better with respect to program size and gives similar WCET values.

***Index Terms*—Real time systems, performance analysis, software performance, software reliability, software algorithms, safety**

I. Introduction

HARD real-time and safety-critical computer systems have to fulfill strict timing guarantees. Missed deadlines may have catastrophic consequences. To verify that all deadlines are met, the worst-case execution time (WCET) of all tasks needs to be known. WCET places an upper bound on the execution time of a given software task. The idea behind WCET is to make timeliness a property that can be formally verified through code analysis, rather than simply measured through experimentation. It yields a provably correct bound rather than an educated guess.

Most WCET related research has been devoted to improving the accuracy of WCET estimates, which by definition are conservative in nature. The literature attacks the problem by modeling the lengthy pipelines [1], multi-level caches [2], memory hierarchies [3], branch target buffers [4], and other sources of unpredictability in modern processors [5], [6].

Despite these efforts, WCET analysis is seldom performed in industry or even ignored entirely [7], [8]. Formal tools and techniques for WCET analysis remain unattractive to many practitioners. There remains a tendency to rely on measurement and experimentation to make claims about a system's predictability. Measurements, however, can only *imply* predictability; they cannot *guarantee* it. Unexpected input data encountered in the field could cause the algorithm to react more slowly than was measured during testing, as illustrated in Fig. 1. While this kind of verification may be suitable for soft real-time systems, measurement is insufficient when developing hard real-time and especially safety-critical systems.

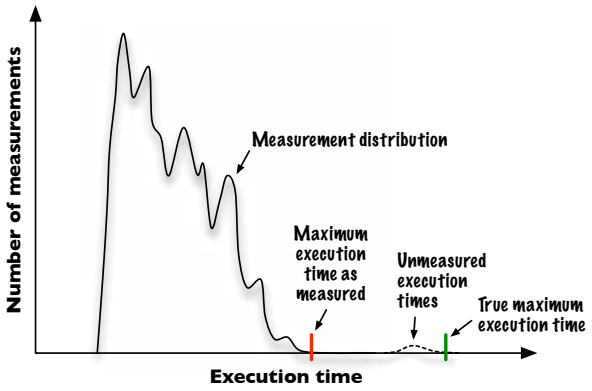

Fig. 1. This histogram of execution time for a fictional real-time task illustrates the weakness of measurement when making claims of predictability. Performance testing may produce a data distribution like the shown one here, leading the developer to underestimate the maximum latency of the task.

The question is why the research community has failed to make WCET analysis techniques more attractive to industry practitioners. This disconnect can be traced, at least in part, to three specific weaknesses in state-of-the-art tools:

Slow performance Virtually all WCET tools focus on obtaining tight bounds; the speed of computing the answer is secondary. As a result, finding the WCET of a large program may require hours or even days of CPU time [6]. Developers are then compelled to postpone the WCET analysis until a final verification step. However, fixing timing errors after the code has been written is expensive, time-consuming, and may necessitate a redesign of the system.

Low-level environments Current tools operate at a very low level. They perform WCET analysis of the assembly code, and there they stop. While the tool may provide a visualization of the results, as shown in Fig. 2, understanding this visualization requires understanding the assembly language of the target processor. Should the processor change, the developer may have to learn yet another assembly language. Furthermore,

Manuscript received January 10, 2011; revised July 19, 2011. Accepted for publication November 3, 2011.

The majority of this work was conducted at the University of California, Irvine, with funding provided by the National Science Foundation's Graduate Research Fellowship Program. The work was also supported by an appointment to the NASA Postdoctoral Program at the Ames Research Center, administered by Oak Ridge Associated Universities through a contract with NASA.

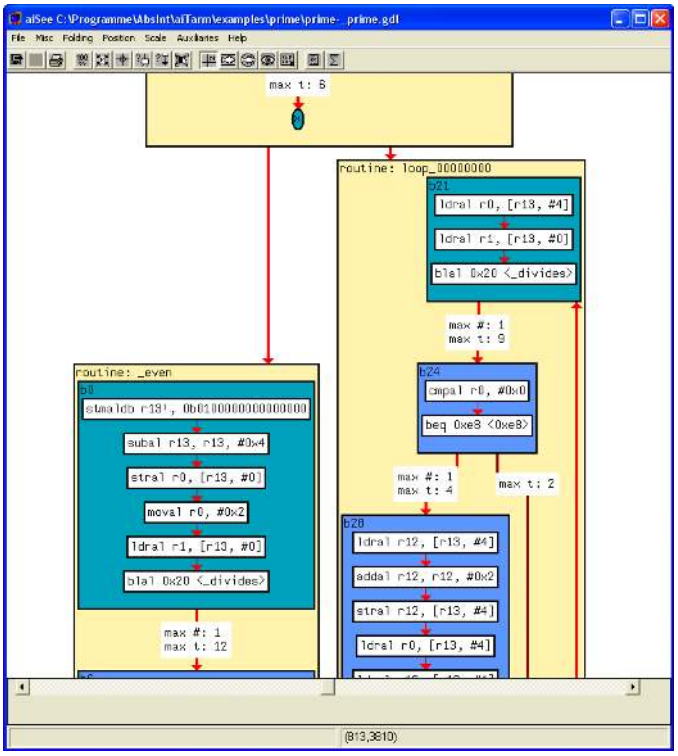

Fig. 2. One of the most advanced tools for automated WCET analysis is aiT. Despite its sophistication, aiT exposes much low-level detail, such as the source code disassembly shown here. (Screenshot by AbsInt Angewandte Informatik GmbH; used with permission.)

presenting analysis results at the assembly level can result in information overload. There is often no need to expose such a deep and complex analysis at the user interface level. In practice, the developer only needs to know whether a program meets its WCET requirements and, if not, where its worst-case paths lie in the source code. The individual machine instructions are irrelevant.

No source code mapping Because virtually all WCET analysis tools operate at such a low level, there is no link between the tool and the original source code. The tools are typically self-contained and exist separately from the source code editor, forcing the developer to mentally map the jumble of mnemonics and hexadecimal numbers back to the original source language. This periodic switch between high-level and low-level thinking not only hampers productivity but also makes identification of the worst-case path overly difficult.

In short, current tools necessitate a slow, uncoupled style of development, as there is no feedback during the implementation phase to indicate whether the code is meeting its timing constraints. Recent studies have criticized this state of affairs, noting that today's tools demand too much manual intervention [8].

In this paper we propose *interactive WCET analysis* as a new method to make exploitation of timing analysis more efficient. The novelty of the method lies in the provision of near-instant WCET feedback to the developer during software programming.

To make the technology of interactive WCET analysis feasible, we focused on selection of the right analysis algorithms and on the properties of the target platform. We show by means of a complete implementation, called Volta, that interactive WCET analysis is feasible using tree-based WCET calculation, even on a modern pipelined processor with caching enabled. Volta performs WCET analysis of Java programs at bytecode level and supports Java processors as target architecture. The feedback is realized with a plugin of the Java editor jEdit,[1] where the WCET values of each statement and compound structure are back-annotated in the Java source. The tool has played a role in the validation of spaceflight software, been adopted for graduate coursework, and served as a platform for other researchers developing new WCET analysis techniques.

The comparison of the tree-based approach with the *implicit path enumeration technique* (IPET, described in Section IV-A) showed that tree-based analysis scales better and gives similar WCET values. With an artificial benchmark we observed an analysis time of five hours with the IPET based analysis compared to 1.6 seconds with the tree-based analysis.

II. Interactive WCET Analysis

To resolve these shortcomings, we propose a new approach that makes WCET analysis *interactive*. Instead of waiting until the implementation is complete before starting timing analysis, the approach incorporates knowledge of worst-case time into every facet of development. It mandates that WCET analysis be performed *continuously* and *interactively* from the moment the first line of code is written.

This is a sharp contrast to the current model of developing real-time systems, in which WCET analysis is performed only at the verification stage. By integrating automatic WCET analysis into the implementation process, the feedback loop tightens, and verification of timeliness becomes faster and simpler, as shown in Fig. 3. Because it offers the developer knowledge of worst-case time *as the code is written*, the benefits of interactive analysis are twofold. First, by helping to identify and correct timing problems the moment they emerge, deadline violations are no longer revealed late in a system's life cycle. Second, should the implementation later change to fix a bug or add a new behavior, the impact of the change on worst-time time can be seen instantly, without having to perform an additional and separate verification step.

Fig. 4 shows one example of how the vision of interactive WCET analysis works in practice. The text annotating the right-hand side of each statement, as well as each function as a whole, are called *back-annotations*. They indicate how much time the statement consumes in the worst case (for a particular target processor). Because the WCET tool that inserted these back-annotations supports interactive analysis, they are updated in the background automatically and continuously, as the code is written. These mechanics are similar in nature to integrated development environments such as Eclipse [9], in which a compiler runs concurrently in the background to check source code for syntax errors as it is typed. In a similar fashion, a WCET analysis tool, if sufficiently fast and imbued with knowledge of the high-level source language, can run in parallel with an editor to provide immediate feedback on the worst-case behavior of the code.

[1] http://www.jedit.org/

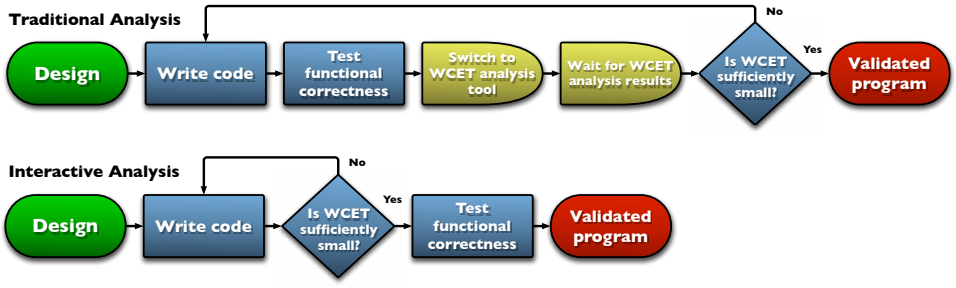

Fig. 3. In the traditional approach to WCET analysis, the developer's feedback loop is large due to the switching to and from a separate tool that may require minutes or even hours to finish its work. Interactive analysis tightens this loop and shortens development time by making the WCET algorithms much faster and integrating them into the development environment.

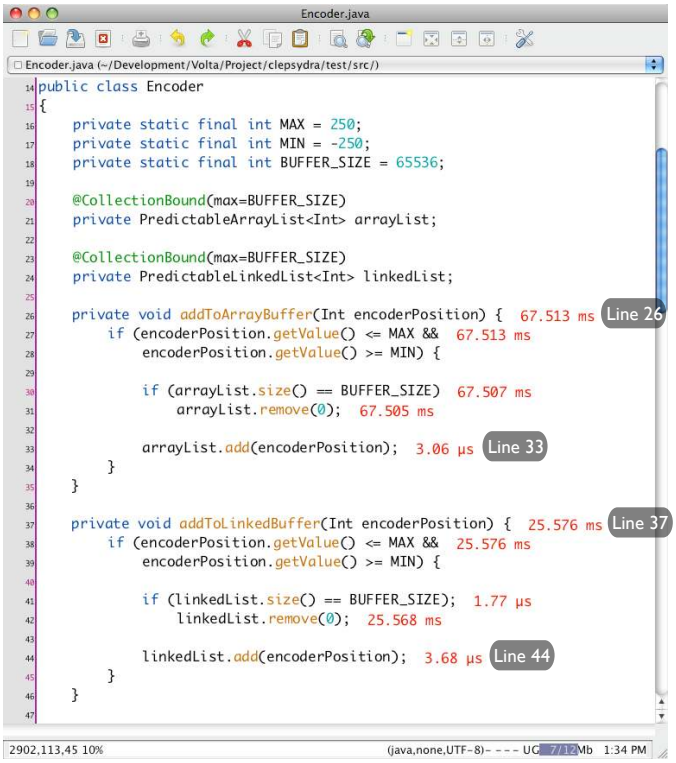

Fig. 4. This screenshot shows an interactive WCET analysis of two alternative implementations of adding an item to a buffer. The analyzer's back-annotations on lines 26 and 37 reveal which version offers better worst-case performance. (For simplicity, the two functions are shown here as part of the same program, but at run-time, only one will actually be invoked.)

This technique not only makes timing verification faster, it can also eliminate mistakes that may arise from false intuition on the part of the developer, thereby improving the worst-case response time of the code. For example, one might expect an append operation to be faster when performed on an array than on a linked list, given that a linked list requires two pointer operations for the insertion, while an array involves only a single pointer copy. (Lines 33 and 44 of Fig. 4 supports this assumption.) However, if an element must first be removed to make room for the new one, the array will require the additional step of shifting up to $N - 1$ elements, while the linked list needs only a few extra pointer operations for the removal. Therefore, two semantically identical functions may have vastly different—and even counter-intuitive—worst-case execution times, as shown on lines 26 and 37 of Fig. 4.

Fig. 5. This control system, consisting of a rotary encoder, pulse counter, and real-time polling software, was assembled as a platform for hardware-in-the-loop experimentation with interactive WCET analysis.

Interactive analysis brings this knowledge into the implementation phase, rather than postponing it until some separate and final verification step, at which point acting upon it may be too difficult or expensive.

As a representative demonstration of interactive analysis in a real-world industrial environment, consider the case of polling a sensor. Determining the maximum frequency at which the sensor can be polled is a common task in real-time systems. For example, a design requirement may state that the sensor must be polled at 20 Hz, but how can one guarantee that the time taken to read the sensor and process the result will be sufficiently small to allow this sampling rate?

To show how interactive WCET analysis can solve this problem, a simple control system was assembled, as illustrated in Fig. 5. It consists of a DC motor that oscillates an axle through a 90°sweep. Attached to this axle is an incremental rotary encoder (quadrature type) that emits pulses as the axle oscillates. A pulse counter then records these pulses to determine the axle's current position. Finally, real-time software running on a Java processor, implemented on an FPGA (Field-Programmable Gate Array), repeatedly polls the pulse counter and stores each sample in an array buffer.

From a real-time systems perspective, the question is now this: Can the software loop sustain a 20 Hz sampling rate in the worst case? Such a question could be answered through interactive WCET analysis without even assembling the hardware. As the screenshot of Fig. 4 has shown, on line 26, the polling task requires 68 milliseconds of execution time, or a maximum frequency of no more than 15 Hz. This is clearly insufficient for the 20 Hz design requirement.

However, consider storing each sample in a linked list instead of an array. Again, interactive analysis provides an answer as soon as the code has been written, with no need for a hardware-in-the-loop test or even a separate verification step using an external tool. Fig. 4 shows (on line 37) that

the loop's WCET reduces to about 25 milliseconds, or a maximum frequency of about 40 Hz. Therefore, the linked list implementation is guaranteed to satisfy the 20 Hz constraint.

The key observations of this experiment are threefold. First, knowledge of whether an implementation exceeds a real-time deadline constraint can be obtained without performing any live tests. Second, interactive WCET analysis enables a kind of "what-if" style of programming. A developer can experiment with different implementations of a function and quickly compare the impact on the WCET. And third, because interactive analysis adjusts for changes to the code as they are made, it can pinpoint exactly when an addition, no matter how small, causes a deadline to be exceeded.

III. A Platform for Interactive Analysis

Achieving this kind of interactivity requires a rethinking of the established WCET analysis techniques. Instead of emphasizing accuracy above all else, factors such as the speed of the analysis and the ability to map the results to high-level source code should be emphasized.

To that end, we developed a complete stack of tools and libraries with which to explore this theme of interactivity. Fig. 4, for example, is not a mock-up but an actual screenshot of our WCET tool Volta. All of the techniques demonstrated in this work have been implemented as a tool suite, complete with unit tests and documentation, in a project called Volta.[2] The entire Volta suite is published under an open-source license,[3] inviting critical comparison and making the reported experiments repeatable. It is also built in a modular, extensible fashion to facilitate future research. Volta currently consists of the following components:

Cascade A hybrid of a decompiler and a control flow analyzer, Cascade's purpose is to generate a data structure—either a control flow graph or a control flow tree—that is suitable as the input to a WCET analyzer. It supports the concept of source-annotated control flow information, which is a prerequisite for mapping WCET analysis results to a high-level source language.

Clepsydra Built on top of Cascade, the Clepsydra[4] tool performs the actual WCET analysis. It supports a plug-in architecture (via the Strategy pattern) that allows analysis algorithms, CPU models, and loop bound detectors to be swapped cleanly at runtime. More importantly, Clepsydra includes a novel WCET analysis algorithm that is sufficiently fast for interactive analysis.

Canteen In hard real-time systems, general-purpose libraries pose a special problem, as they are designed with average-case performance in mind. Canteen is a class library that demonstrates how to write general-purpose, reusable code that can still be analyzed by a WCET tool such as Clepsydra. For the sake of brevity, however, Canteen will not be discussed further; more information about it can be found in prior work [10].

[2]Named after the lake in Ghana, not the physicist from Italy. The nomenclature of Volta and its components—Cascade, Clepsydra, Canteen—is meant to invoke the theme of a controlled flow of water.

[3]Source code is available for download at http://volta.sourceforge.net/

[4]A clepsydra is a clock that measures time by the escape of water.

To be sure, the goal of these tools and libraries to support interactive analysis is both challenging and ambitious. Some researchers have even argued that industrial-strength WCET analyzers are simply too difficult to implement, largely due to the increasing complexity of modern processors [11]. To make the problem tractable, Volta operates under some simplifying assumptions about the hardware and software under analysis.

A. Hardware Assumptions

From the hardware perspective, the modern microprocessor alone is a major obstacle in WCET analysis. Architectural advancements in processor design—long pipelines, branch prediction, and complex multi-level caches—have focused on making the average case as fast as possible. Unfortunately, the shrinking of this average has not come without cost. While average execution time may be low, its standard deviation has grown, resulting in large worst-case times. Theoretically, a highly sophisticated WCET analyzer could model the flow of data through this CPU to predict a tighter worst-case time, but this is a formidable task that is still an open area of research [12]–[14].

Rather than fight the increasingly hard-to-predict behavior of modern general-purpose processors, a new strategy is to build WCET analysis-friendly processors [15]. The approach is exemplified by specialized processors that understand Java bytecode as their native instruction set [16]–[18]. These processors offer a clear advantage for WCET analysis: Running bytecode directly on the processor eliminates the need for virtual machines and just-in-time compilation, making execution time far less variable.

These characteristics make WCET analysis much less complicated while yielding a tighter bound at the same time. For this reason, Volta assumes that the program under analysis will execute on a Java-specific processor. It includes, by default, a model of the Java Optimized Processor (JOP) [17], a fully functional VHDL soft core that runs on several varieties of off-the-shelf FPGAs. JOP has been used successfully in commercial applications [19], [20]. Although Volta's plug-in capability allows future support for similar architectures, all of the empirical results presented in this work were obtained with JOP as the target.

Although moving to such a new and unusual architecture may seem drastic, a questionnaire distributed to WCET tool users in 2003 revealed that 75% of respondents would adopt a processor with more predictability even if it meant a loss in average performance [21]. Certainly, the most important feature of a processor for hard real-time systems is not how fast it can go but how much it can be slowed down by a series of unfortunate events.

Furthermore, for the design phase, the main application area of Volta, it might be appropriate to use conservative approximations for the execution time of instructions to find the WCET hotspots. Therefore, also more complex processors can be used with fast WCET feedback. The final WCET verification, where analysis time is not the main issue, then has to be done with the more complex processor model. We are considering this approach to integrate other Java processors,

such as picoJava [16] and jamuth [22] into Volta. The idea is to build (automatically) test programs, measure their execution time, and then derive a timing model on the level of bytecode instructions.

B. Software Assumptions

The tools and libraries in Volta analyze code written in Java. This language has already been adopted for real-time projects in industry and the military [23], [24], and it is becoming a more common topic in real-time systems research [25]–[29]. However, certain run-time characteristics of Java, such as dynamic class loading, make Java more difficult than other languages for conducting WCET analysis. Therefore, Volta places the following constraints on the code under analysis:

- All loop statements must have bounded iterations that are supplied as source code annotations [30].
- Recursive function calls, exception handling, dynamic heap allocations, dynamic class loading, and dynamic dispatch are prohibited. (A simple and safe, though pessimistic, workaround is to select the maximum WCET of all possible receivers.)
- Multithreading and interrupt handling are assumed to be absent. However, concurrency could be achieved through alternative techniques that are more WCET-friendly, such as time-triggered scheduling [31].
- The control flow graphs produced by the compiler are assumed to be reducible, which should always be the case given that there is no goto statement in the Java source language.

These requirements may seem exceedingly strict, but they are not uncommon in hard real-time systems, where timing guarantees are more important than speed and flexibility. For instance, industry guidelines for safety-critical software impose similar limitations on developers [32], and modern WCET tools typically do not support dynamic memory management functions and require manual annotations when a loop bound cannot be automatically determined.

Furthermore, the limitations have not prevented Volta's use in various settings. For example, it has been applied in NASA's Integrated Vehicle Health Management (IVHM) program for WCET analysis of Bayesian verification algorithms. It was a topic of study in a master's course at the Delft University of Technology. And other researchers in the field, working independently, have extended Volta to support new abilities, such as incremental analysis and recursion.[5] Overall, Volta's restrictions provide a reasonable tradeoff between analysis results and analysis efforts.

IV. Achieving Fast, Interactive WCET Analysis

With these assumptions, the foundation for constructing an interactive WCET analysis tool is in place. The next step is to produce a data structure expressing the order of execution of individual statements—that is, the control flow. The Volta tool suite offers an alternative to the (control flow graph) CFG: a control flow *tree* [33]. This data structure is similar to an abstract syntax tree (AST), except it is "concrete" in that it preserves all machine code, and allows for fast identification of loop structures.

A. Longest Path Computation in WCET Analysis

Finding an upper bound on the execution time of a task is normally broken down into three sub-problems: control flow reconstruction, low-level analysis, and longest path computation. Cascade already handles the first problem.

The second problem, also known as execution time modeling, assigns to each basic block in the control flow an execution time. The amount must be derived from a model of the target processor. With the assumption of Java processors, the problem reduces to summing the cycle count for each bytecode instruction in isolation. This is possible because most bytecodes have a best-case execution time that is identical to their worst-case execution time, without any pipeline dependencies between them. On the JOP, for instance, the cycle count of the GETSTATIC bytecode instruction is always $12+2r_{ws}$, where r_{ws} is the number of wait states for a memory read. Almost all bytecode timings can be computed with a simple formula such as this, and therefore execution time modeling is not an impediment to fast, interactive analysis.

There remains, however, the final step of WCET analysis: the actual search for the longest path. It can be mapped to a well-known problem from graph theory: Finding the maximum flow through a single-source, single-sink directed acyclic graph. The graph in this case is the control flow graph of a computer program, and the flow capacity of an edge is the instruction time of a basic block.

Implicit Path Enumeration Technique: The common approach to search the longest path is the *implicit path enumeration technique*, or simply IPET [34]. In IPET the maximum flow is defined with *constraints* on the legal flow through the graph [35]. The paths through the graph are never actually explored; they are derived implicitly from the constraints, thus giving the method its name. Accounting for a loop is simply a matter of adding an additional constraint to bound the amount of flow—that is, the number of iterations—through the loop. IPET based WCET analysis tools commonly create linear flow constraints, which allows integer linear programming (ILP) to solve the constraint system. IPET is today the most commonly employed longest path search algorithm in WCET tools, both commercial and academic, including Bound-T [36], aiT [37], and Chronos [38].

B. Rethinking the Longest Path Problem

Despite the sophistication and variety of these tools, they all suffer from a common weakness: Little or no attention is given to improving the *speed* of analysis, only its accuracy. In fact, IPET can be quite slow; it is an NP-hard problem whose running time grows exponentially with the input size. This slothful performance makes IPET impractical for the kind of interactive analysis we envision.

[5] Kevin Buell and James Collofello. "Worst-Case Execution Time Analysis of Incremental Changes to Recursive Methods," in the 3^{rd} NASA Formal Methods Symposium (in submission).

IPET is not the only means of finding the longest path through a program, however. A *tree-based* approach, also known as the *structural* approach, was the basis of the very first implementations of WCET analysis [33], [39]. It operates by recursively descending the nodes of a program's control flow tree, returning the execution time for each node. When the algorithm encounters a new node, it decides how to compute the WCET based on the node's type. For straight-line code, the time to execute each instruction is simply summed. For branches (if and switch statements), the path whose execution time is highest—the "worst" path—is taken as the total time. For loops, the maximum number of iterations is simply multiplied by the WCET of the loop's body. The value returned for the root node is then the total WCET.

Despite being relatively easy to implement and understand, the tree-based algorithm has fallen out of favor in WCET research. It suffers from certain drawbacks, such as a susceptibility to the "false path" (or "infeasible path") problem, in which data dependencies between two if statements can fool the algorithm into computing an overly pessimistic WCET [40]. In contrast, IPET can eliminate a false path by adding a single constraint to its ILP formulation.

But tree-based algorithms have a benefit that is not so commonly recognized: raw speed. The complexity of a recursive descent to determine WCET is $\theta(n)$, where n is the number of nodes in the control flow tree. This processing cost of $\theta(n)$ does not include the false-path analysis, i.e., flow information like loop iteration bounds have to be given as code annotations. However, this linear running time is clearly superior to the exponential complexity of IPET. It has the potential to reduce the longest path search to just a few seconds, making analysis of real-time software happen *in real time*.

The challenge, then, is to modify the tree-based technique so that its accuracy is on a par with comparable ILP-based techniques. For example, the extension of the tree-based technique for nested loops has been proposed by Colin and Puaut [41]. In future work we plan to extend our simple tree-based calculation to detect and eliminate false paths, which will provide a user-selectable tradeoff between speed and accuracy, still faster than ILP-based calculation. Furthermore, we think of combining tree-based calculation with clustered IPET-based calculation [42], which is applicable to the local scopes of flow information. As Volta supports both analysis methods, this extension shall be straightforward.

C. An Example from JOP's Dual-Method Instruction Cache

Regrettably, improving the accuracy of tree-based techniques is far from trivial. Handling any kind of dependency across siblings in the control flow tree is difficult due to the nature of tree traversal, which restricts the scope of each step to a locality of nodes. With sufficient effort, though, solving this tricky situation is still possible. Colin [43], for instance, describes how a tree-based algorithm can resolve the false path problem.

As further evidence that a tree-based technique can be as accurate as its IPET counterpart, at least for certain code structures, consider the instruction cache implemented in the JOP. Instead of a traditional block-based cache, JOP's cache is *method-based* [44]; it always stores complete Java methods. It fills only on an invoke or a return instruction, meaning that all other instructions are a guaranteed cache hit. JOP can be configured to store more than one method at a time. For example, a *dual method cache* stores two methods at once using a least-recently used replacement strategy.

While this dual method cache improves performance, it also complicates WCET analysis. Whether a method invocation is a hit or miss depends not only on the structure of the program but on the input data, as well. For certain code structures, however, the hit ratio can be determined without regard to input. For example, an invocation of a single method within a loop is always a hit if the cache can hold two methods simultaneously.

Schoeberl and Pedersen used this observation to modify IPET to support dual method cache architectures [45]. They simply search the CFG for method invocations that are a) leaf methods—that is, methods that do not invoke any other method—and b) the only invocation within a loop. For the dual-method cache, this particular code structure ensures that all invocations and returns of the leaf method are cache hits (except of course for the first loop iteration). Accounting for this special case of an improved hit frequency only requires adding a small set of new constraints—one set for each conforming leaf node—to the original ILP formulation.

D. Improving the Accuracy of Tree-based Cache Analysis

The intent of this section is to show an example of how the tree-based approach can be refined and extended so that it can produce WCET estimates that are not only as accurate as IPET but far faster as well, at least for certain code structures.

One way to achieve this result is with a novel *two-pass variation* of the standard tree-based approach. When the algorithm first encounters a loop, it treats any method invocation in this first pass as a cache miss. Next, the algorithm must *correct* this initial WCET estimate to account for method invocations that are known to be cache hits (e.g., leaf methods). To do so, it merely performs a second pass over the loop body to search for these particular invocations and, when one is found, it adds the incorrect miss penalty—that is, the number of cycles that would be spent servicing the cache miss—to a running total. (The algorithm refers to these penalties as "gain times" rather than penalties because in this context they have a beneficial impact: They restore the time that was lost due to the overly conservative cache miss assumption of the first pass.) Finally, this sum is subtracted from the original estimate to arrive at the final WCET value for the loop.

Fig. 6 provides a pseudocode listing of this approach. Note that the second pass occurs in the getGainTime method, where the search for known cache hits occurs. The actual determination of cache hits is implicit in the "number of cache hits of the invocation" statement. Determining this number requires checking whether the invocation is a leaf method. If it is, the number of cache hits will equal the loop bound minus one.

Even with its additional support for cache analysis, the two-pass variation retains the $\theta(n)$ complexity of the original,

```
1  getTotalWCET()
2    return getWCET(root node)
3
4  getWCET(node)
5
6    if node is null return 0
7
8    if node is of type IF_THEN_ELSE
9      cycles = getExprWCET(node.conditional_expression) +
10             max(getWCET(node.then_branch),
11                 getWCET(node.else_branch))
12
13   else if node is of type LOOP
14     expression_cycles =
15            getExpressionWCET(node.conditional_expression)
16     cycles = expression_cycles +
17              (getWCET(node.loop_body) + expression_cycles)
18                  * node.loop_bound
19     if every parent node is not of type LOOP
20       cycles -= getGainTime(node.loop_body)
21
22   else if node is of type STATEMENT
23     cycles = getExpressionWCET(node.statement_expression)
24
25   return cycles + getWCET(node.next)
26
27 getExpressionWCET(expression)
28   return sum of CPU cycles of all instructions in the
29     basic block (assumes all invocations are cache misses)
30
31 getGainTime(node)
32   gainTime = 0
33   for all children of node
34     if the child contains a method invocation
35       gainTime += number of cache hits of the invocation *
36                   cache miss penalty
37   return gainTime
```

Fig. 6. The standard recursive descent algorithm in tree-based analysis cannot account for method invocations because method cache hits are not constant across loop iterations. A two-pass variation of the algorithm, shown here in pseudocode, solves this problem.

where n is the number of nodes in the control flow tree. An informal proof is as follows: First, assume that the getGainTime method is not present. Without it, the getTotalWCET method is a simple recursive descent of the tree, touching every node exactly once for a complexity of $\theta(n)$. Now consider the presence of getGainTime. This method is invoked on every loop and visits all of the loop's child nodes a second time. (The conditional on line 17 ensures that, in the case of nested loops, the nodes are revisited no more than once.) Even if every node in the control flow tree were a loop, the additional complexity of getGainTime would be $\theta(n)$. The total complexity of getTotalWCET for the two passes is then $\theta(n + n)$, or $\theta(n)$. The algorithm therefore exhibits linear running time, as empirical evidence in Fig. 9 will corroborate.

E. Qualitative Analysis

To clarify, this tree-based alternative is no panacea. The algorithm presented here improves WCET accuracy only for the dual-method cache architecture and only for certain code patterns. Other kinds of architectures and patterns would require additional work to devise new algorithms that enhance the accuracy of the canonical tree-based approach without sacrificing its speed. This requires more development effort than IPET on the part of WCET tool authors, but such is the trade-off: Tree-based approaches are fast but make high accuracy difficult to achieve, while IPET is slower but increasing accuracy is comparatively simpler. For the purpose of interactive analysis, however, the extra effort of the tree-based approach is justified.

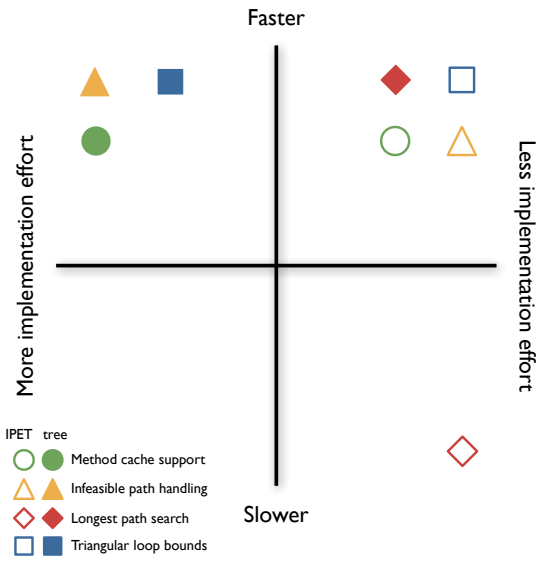

Fig. 7. This matrix offers a qualitative comparison of certain aspects of WCET analysis, showing approximately where they lie on the spectra of runtime complexity (faster vs. slower) and difficulty of implementation.

Fig. 7 provides a visual perspective on these trade-offs. The matrix shows the location of various aspects of WCET analysis on the orthogonal spectra of runtime complexity and difficulty of implementation. For example, the longest path search of a control flow tree is relatively easy to implement and executes very quickly, and therefore it lies in the top-right quadrant of the matrix. By comparison, the longest path search of a control flow graph is approximately as easy to implement but requires far more time to execute, and therefore it lies in the bottom-right quadrant. (Note that this is merely a qualitative comparison meant to illustrate the relationships between certain aspects of WCET analysis.)

An ideal WCET tool would have all of its aspects in the top quadrants (making it suitably fast for interactivity), so we are faced with two choices: 1) Devote the effort needed to implement infeasible path handling, cache analysis, etc. for the tree technique, or 2) make the longest path search of IPET fast. The latter option means fighting an NP-hard problem, however. Therefore, we conclude that the former option is more tractable and more promising.

Here we also have to say that we do not consider the problem of calculating flow information automatically. While there are approaches to find many false-path constraints automatically [46], [47], they have a quite considerable running time. For the smooth interactivity of WCET analysis we assume that program updates since the last derivation of flow information have not invalidated them. Of course, whenever flow information is invalidated, flow information has to be updated (manually or automatically by a tool).

V. Clepsydra: An Interactive Analysis Tool

To test the performance and accuracy of the two-pass tree technique, a worst-case execution time analyzer called Clep-

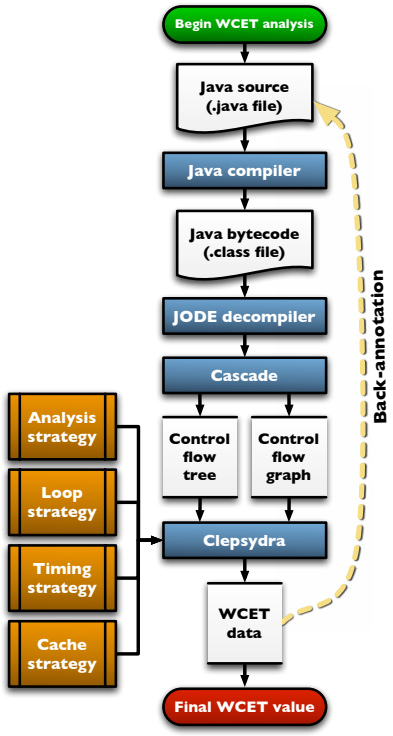

Fig. 8. This flow chart illustrates how the Cascade and Clepsydra tools work together to provide interactive WCET analysis with back-annotation, represented in the figure by a dotted line.

sydra is part of the Volta tool suite. Built on top of Cascade, it was designed with the goals of interactive analysis in mind: fast response times, tight integration with development tools, and a mapping of analysis results to source code.

Fig. 8 provides an overview of how Clepsydra combines all of these goals into a single coherent process. The analysis begins when the developer supplies a Java source file, which is immediately fed into a custom Java compiler that supports WCET annotations [48]. Cascade then reconstructs the bytecode output of this compiler into an analysis-friendly control flow graph or tree. Finally, Clepsydra performs the actual analysis and produces worst-case timing values, or back-annotations, for every statement and compound structure in the decompiled source code. The dotted line in the figure represents the data flow of back-annotations from Clepsydra's analysis results to the source code.

To present the back-annotations in a suitable user interface, Clepsydra includes an example of an editor plugin, a screenshot of which appeared in Fig. 4. This particular example is based on the programmer's editor jEdit, but the approach can be adapted for any editor that allows the contents of its window to be decorated by a plugin. A ready-to-use implementation can be found in the Volta distribution. It is the first instance of interactive integration of WCET back-annotations in a programming environment.

A. Performance Benchmarks

Given that raw speed is a fundamental requirement of interactive analysis, the performance of Clepsydra is key. Constructing control flow data structures and analyzing them

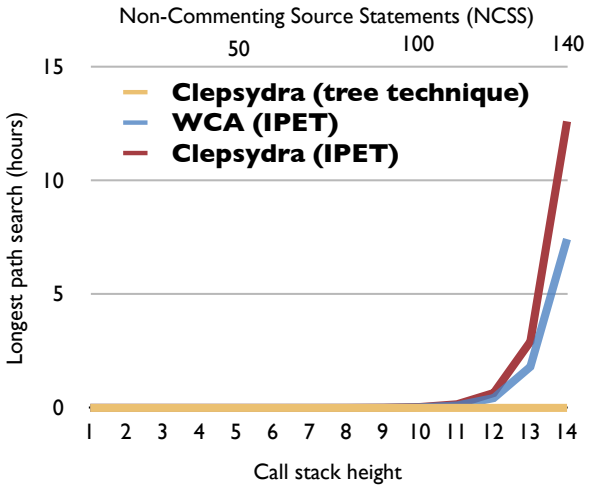

Fig. 9. This graph compares the running time of the longest path search for the two-pass tree technique and IPET (Clepsydra and WCA implementations).

quickly has a direct impact on the user's perceived quality of the interaction. Therefore, these tasks must be fast enough to analyze the program *as it is written*.

To show that tree-based techniques offer superior performance over IPET, making them ideal for interactive analysis, a deliberately simple test program was devised. It consists of a series of identically structured functions, where each function has a cyclomatic complexity of three. (For the complete source code, refer to the Volta distribution.) This code pattern yields a steadily increasing call stack that mimics the large call stacks that commonly occur in object-oriented programs.

Fig. 9 shows the time needed for the actual WCET for each of these functions in turn. For example, the fifth measurement along the horizontal axis has as its entry point the fifth function in the program (which calls the fourth function, and so on). All tests were conducted on a 2.66 GHz Intel Core 2 Duo machine running Java 1.6.0. In the graph, "tree technique" refers to the two-pass variation described in Section IV-D, while "IPET" refers to an ILP-based implementation that relies on lp_solve[6] to perform the actual equation solving.

To eliminate the *ILP solver implementation* as a variable in this experiment, the same measurements were also taken using an alternative implementation called GLPK,[7] but the performance differences were found to be negligible. To eliminate the *IPET implementation* as a variable in this experiment, the same measurements were also taken using WCA [49], the WCET analysis tool that ships with the JOP distribution. WCA supports IPET and model checking based WCET analysis, not the tree technique. The WCA was developed independently of Volta and acts as an external "control group" to verify that any sluggishness measured in IPET is not merely the result of a poor implementation in Clepsydra.

Note that WCA is somewhat faster than Clepsydra's IPET due to differences in the control flow graph structure of the two implementations. Cascade splits each basic block according to its representative source code statements. (This is necessary for the back-annotation shown in Fig. 4.) For example, two

[6]http://lpsolve.sourceforge.net/
[7]http://www.gnu.org/software/glpk/

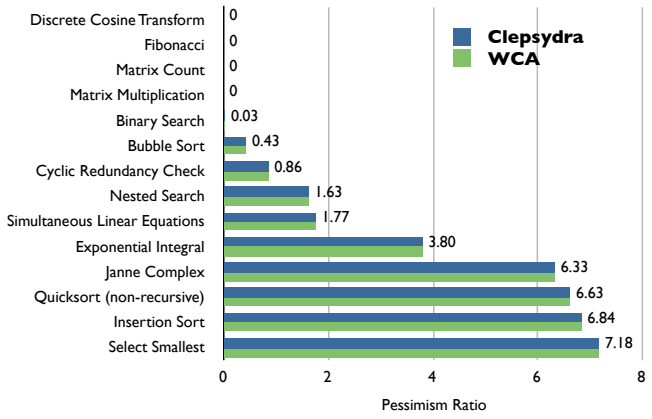

Fig. 10. These numbers represent Clepsydra's pessimism ratio for a variety of WCET benchmarks. The ratio in each case compares the WCET predicted by Clepsydra to the true WCET measured in hardware. For comparison, the figure also shows the benchmarks of the WCA tool, which were identical to Clepsydra's in every instance.

successive variable increment statements, such as i++, would appear in WCA's CFG as a single node and in Cascade's as two nodes. While this does not affect the correctness of the algorithm, it does slow down the ILP computation due to the extra constraint variables.

As shown in the graph, all techniques perform well early on, but starting with function #11 IPET begins to falter. When the call stack height reaches 14, IPET requires over five hours to compute its result, while the tree-based alternative completes in 1.6 seconds. The tree method is thus the best choice for interactive analysis, even with the overhead of method cache support.

B. Accuracy Benchmarks

Of course, speed is not the only factor. The tightness, or accuracy, of the WCET estimation is still a concern. To evaluate the accuracy of Clepsydra, a set of fifteen WCET benchmark programs was created. It is based on a similar suite of benchmarks from the Mälardalen Real-Time Research Center [50]. Provided in the Volta distribution under a public domain license, it is intended to serve as a *de facto* standard for evaluating any Java-based WCET analysis tool.

Fig. 10 shows the results of running these benchmarks on Clepsydra. The algorithm was then executed on a physical JOP (implemented in hardware on an Altera Cyclone FPGA) with input data that triggers worst-case performance, and the running time was measured with clock-cycle accuracy. Finally, this running time was recorded as the true WCET[8] and compared to Clepsydra's prediction as a simple *pessimism ratio*: $pessimism = \frac{predicted - true}{true}$

The results vary widely. Discrete Cosine Transform, Fibonacci, Matrix Count, and Matrix Multiplication exhibit the ideal behavior of 0% pessimism because they are simple loops. The Select Smallest benchmark, a complex piece of code with many nested conditionals and loops, fared the worst at more than 700% pessimism. (That is, the time predicted by Clepsydra was about eight times larger than the actual worst-case time.)

The poor pessimism for some of these benchmarks is in part by design. They are intended to stress typical weaknesses that often afflict WCET analyzers. The Janne Complex benchmark, for example, has an inner loop whose maximum number of iterations depends heavily on the outer loop's current iteration number. Structural analyzers that ignore data flow, such as Clepsydra, suffer from this behavior.

Overall, the largest increase in pessimism was often observed to be a result of ineffective loop bound annotations. The Insertion Sort and Quicksort benchmarks in particular expose this problem; they also contain inner loops whose bounds depend on the outer loop's state. The loop bound annotation mechanism currently supplied with Clepsydra can only specify constant bounds, leading to overly conservative estimates. Future work will focus on adding a more expressive annotation language. Nevertheless, for the benchmarks that represent numerical computations typically found in real-time systems, such as the matrix and DCT benchmarks, the bounds are quite tight.

It should also be noted that for every one of these benchmarks, the accuracy of the two-pass algorithm was identical, as intended, to Schoeberl's and Pedersen's IPET implementation. This result reinforces the claim of Section IV-B that tree-based analysis algorithms can offer extraordinary increases in performance while maintaining reasonable accuracy.

VI. Related Work

One of the earliest attempts at interactive analysis came in 1996 when Ko et al. developed a graphical interface for a WCET tool [51]. The interface allowed the user to select a specific portion of source code for analysis, and the tool would then return the WCET of the selection. The primary innovation in this work was to allow specification and presentation of timing predictions at the source code level while retaining the accuracy of low-level analysis. Back-annotation was not provided, however, and there was no investigation into the speed of analysis. Another prototype for an integrated development environment came from Ribeiro et al. [52]. The aim was to provide continuity in a real-time software project through the phases of implementation, debugging, and testing. A novel feature in the environment was a graphical display of control flow showing each source code element's contribution to the total WCET. This was one of the very first realizations of back-annotation, although the data was displayed in a separate window and was not integrated into an interactive source code editor. There was also no mention of the speed at which the graphic could be generated.

A step closer toward true back-annotation arrived as a side-effect of Kirner's study of optimizing compilers in the context of WCET analysis [53]. He created a prototype tool chain to display WCET calculations as static back-annotations into source code. Kirner et al. also developed true back-annotation of WCET results into the modeling environments of MATLAB and Simulink [54], but the WCET calculation method relied on IPET and was too slow for interactive analysis.

[8]Strictly speaking, the true WCET of an arbitrary program cannot be known. However, the algorithms of Fig. 10 are either simple enough or well-enough understood that direct measurement can find the true WCET by supplying the appropriate input data.

From the Java domain, the popular Real-time Specification for Java (RTSJ) [55] has no provisions for WCET analysis. Cost measurement and enforcement (defined in the ReleaseParameters class) is strictly an optional facility for RTSJ implementations. However, the year before RTSJ's release, a research prototype called Skånerost explored interactive analysis in a Java environment [56]. The tool combined WCET analysis and compilation to provide frequent feedback to the programmer, updating continuously as the source code changes. Skånerost made no attempt at back-annotation and presented analysis results as raw bytecode instructions.

Among commercial WCET analysis tools, RapiTime [57] is the only product to offer a feature that comes close to back-annotation. The tool is able to color-code the worst-case path in the source code. The information does not include the actual numeric worst-case times, however, and the color-coding is displayed as part of a static, read-only report.

It is also worth mentioning that our approach of back-annotated timing analysis results may not be only useful for the development of real-time system, but helps for performance optimization as well. There are many other applications of program analysis back-annotation via an editor plugin to give the developer feedback about the program. For example, Fauster et al. has developed a plugin to highlight input-data dependent program code in order to show variable control flow [58].

VII. Conclusion

The WCET research community has largely failed to address certain needs of real-time practitioners. The vast majority of analysis techniques are concerned only with obtaining a tight and accurate bound. There has been little attention on other features that industry desires, such as finding a rough but adequate WCET estimate very quickly, then back-annotating those results into the source code. As a result, it is easier to find reports of *unsuccessful* attempts at moving real-time systems theory outside of the academic environment [59].

We have therefore developed an interactive approach to WCET analysis that provides the developer with nearly instantaneous WCET feedback, starting when the first line of code is written. The interesting question addressed in this paper is how to make this approach work in practice, given that current techniques for WCET calculation, such as IPET, are too slow for instantaneous feedback. Our strategy was to resurrect the tree-based approach, which had been superseded due to its inflexibility when specifying flow information or modeling context-specific instruction timing. However, it remains significantly faster than any IPET-based approach and is thus suitable for interactive analysis. We have further shown that it is flexible enough to accurately model complex hardware features such as a method-based cache.

In this paper we also presented a practical WCET analysis framework for the research processor JOP. Based on our experience with this tool suite, we conclude that choosing the right trade-offs at the software level allows fast and interactive WCET analysis with reasonable accuracy, even on a pipelined caching processor. Inherently connected with this approach, however, are the relatively high overestimations in the event of complex control flow. Despite this drawback, bringing interactive feedback of WCET information to the early implementation stages is a necessary improvement for making WCET analysis more practical to system developers.

Acknowledgment

We would like to thank the anonymous reviewers for their detailed comments that helped to improve the paper.

References

[1] J. Engblom, A. Ermedahl, M. Sjoedin, J. Gustafsson, and H. Hansson, "Worst-case execution-time analysis for embedded real-time systems," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 4, pp. 437–455, August 2003.

[2] D. Hardy and I. Puaut, "WCET analysis of instruction cache hierarchies," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 677–694, 2011.

[3] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 966–978, 2009.

[4] D. Grund, J. Reineke, and G. Gebhard, "Branch target buffers: WCET analysis framework and timing predictability," *Journal of Systems Architecture*, vol. 57, no. 6, pp. 625–637, 2011.

[5] P. Puschner and A. Burns, "Guest editorial: A review of worst-case execution-time analysis," *Real-Time Systems*, vol. 18, no. 2-3, pp. 115–128, May 2000.

[6] 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," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, April 2008.

[7] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper, "Applying static WCET analysis to automotive communication software," in *Proceedings of the Seventeenth Euromicro Conference on Real-Time Systems (ECRTS 2005)*. Washington, DC, USA: IEEE Computer Society, July 2005, pp. 249–258.

[8] J. Gustafsson and A. Ermedahl, "Experiences from applying WCET analysis in industrial settings," in *Proceedings of the Tenth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2007)*. Washington, DC, USA: IEEE Computer Society, May 2007, pp. 382–392.

[9] E. Gamma and K. Beck, *Contributing to Eclipse: Principles, Patterns, and Plugins*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2003.

[10] T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad, "Toward libraries for real-time Java," in *Proceedings of the Eleventh IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC 2008)*, May 2008, pp. 458–462.

[11] R. Kirner and P. Puschner, "Discussion of misconceptions about WCET analysis," in *Proceedings of the Third International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*, July 2003, pp. 61–64.

[12] S. Altmeyer and C. M. Burguiere, "Cache-related preemption delay via useful cache blocks: Survey and redefinition," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 707–719, 2011.

[13] A. Ermedahl, F. Stappert, and J. Engblom, "Clustered worst-case execution-time calculation," *IEEE Transactions on Computers*, vol. 54, no. 9, pp. 1104–1122, September 2005.

[14] J. Gustafsson and A. Ermedahl, "Merging techniques for faster derivation of WCET flow information using abstract execution," in *Proceedings of the Eighth International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, Prague, Czech Republic, July 2008.

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

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

[17] M. Schoeberl, "A Java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. 54, no. 1–2, pp. 265–286, 2008.

[18] L. Yan and Z. Liang, "An accelerator design for speedup of Java execution in consumer mobile devices," *Computers & Electrical Engineering*, vol. 35, no. 6, pp. 904 – 919, 2009.

[19] M. Schoeberl, "Application experiences with a real-time Java processor," in *Proceedings of the 17th IFAC World Congress*, Seoul, Korea, July 2008, pp. 9320–9325.

[20] G. Michel and J. Sachtleben, "An integrated gyrotron controller," *Fusion Engineering and Design*, vol. In Press, Corrected Proof, pp. –, 2011.

[21] R. Wilhelm, J. Engblom, S. Thesing, and D. Whalley, "Industrial requirements for WCET tools: Answers to the ARTIST questionnaire," in *Proceedings of the Third International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*, July 2003, pp. 39–43.

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

[23] J. Auerbach, D. F. Bacon, B. Blainey, P. Cheng, M. Dawson, M. Fulton, D. Grove, D. Hart, and M. Stoodley, "Design and implementation of a comprehensive real-time Java virtual machine," in *Proceedings of the Seventh ACM and IEEE International Conference on Embedded Software (EMSOFT 2007)*. New York, NY, USA: ACM, September 2007, pp. 249–258.

[24] A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek, "A real-time Java virtual machine with applications in avionics," *Trans. on Embedded Computing Sys.*, vol. 7, no. 1, pp. 1–49, 2007.

[25] 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.

[26] C. Pitter and M. Schoeberl, "A real-time Java chip-multiprocessor," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 1, pp. 9:1–34, 2010.

[27] J. J. Hunt, I. Tonin, and F. B. Siebert, "Using global data flow analysis on bytecode to aid worst case execution time analysis for realtime Java programs," in *Proceedings of the Sixth International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2008)*. New York, NY, USA: ACM, September 2008, pp. 97–105.

[28] J. H. Spring, F. Pizlo, J. Privat, R. Guerraoui, and J. Vitek, "Reflexes: Abstractions for integrating highly responsive tasks into Java applications," *ACM Trans. Embedded Comput. Syst*, vol. 10, no. 1, 2010.

[29] M. Kim and A. Wellings, "Efficient asynchronous event handling in the real-time specification for java," *ACM Trans. Embed. Comput. Syst.*, vol. 10, pp. 5:1–5:34, August 2010.

[30] T. Harmon and R. Klefstad, "Toward a unified standard for worst-case execution time annotations in real-time Java," in *Proceedings of the Fifteenth International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2007)*. IEEE Computer Society, March 2007.

[31] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, January 2003.

[32] MISRA, *MISRA-C: Guidelines For The Use Of The C Language In Critical Systems*. The Motor Industry Software Reliability Association (MISRA), October 2004.

[33] A. Colin and I. Puaut, "A modular and retargetable framework for tree-based WCET analysis," in *Proceedings of the Thirteenth Euromicro Conference on Real-Time Systems (ECRTS 2001)*. Washington, DC, USA: IEEE Computer Society, June 2001, pp. 37–44.

[34] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 12, pp. 1477–1487, December 1997.

[35] S. Dasgupta, C. H. Papadimitriou, and U. Vazirani, *Algorithms*. McGraw-Hill Higher Education, 2008, ch. 7.

[36] N. Holsti and S. Saarinen, "Status of the Bound-T WCET tool," in *Proceedings of the Second International Workshop on Worst-Case Execution Time Analysis (WCET 2002)*, June 2002.

[37] C. Ferdinand and R. Heckmann, "ait: Worst-case execution time prediction by static program analysis," in *Building the Information Society*, ser. IFIP International Federation for Information Processing. Springer Boston, 2004, vol. 156, pp. 377–383.

[38] X. Li, Y. Liang, T. Mitra, and A. Roychoudhur, "Chronos: A timing analyzer for embedded software," *Science of Computer Programming*, vol. 69, pp. 56–67, December 2007.

[39] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Real-Time Systems*, vol. 1, no. 2, pp. 159–176, September 1989.

[40] P. Altenbernd, "On the false path problem in hard real-time programs," in *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems (EURWRTS 2006)*. Los Alamitos, CA, USA: IEEE Computer Society, June 1996, pp. 102–107.

[41] A. Colin and I. Puaut, "Worst case execution time analysis for a processor with branch prediction," *RTS*, vol. 18, no. 2, pp. 249–274, May 2000.

[42] A. Ermedahl, F. Stappert, and J. Engblom, "Clustered worst-case execution time calculation," *IEEE Transactions on Computers*, vol. 54, no. 9, pp. 1104–1122, Sep. 2005.

[43] A. Colin and G. Bernat, "Scope-tree: A program representation for symbolic worst-case execution time analysis," in *Proceedings of the Fourteenth Euromicro Conference on Real-Time Systems (ECRTS 2002)*. Washington, DC, USA: IEEE Computer Society, June 2002, pp. 50–59.

[44] M. Schoeberl, "A time predictable instruction cache for a Java processor," in *On the Move to Meaningful Internet Systems 2004*, ser. Lecture Notes in Computer Science, R. Meersman and Z. Tari, Eds., vol. 3292, January 2004, pp. 371–382.

[45] M. Schoeberl and R. Pedersen, "WCET analysis for a Java processor," in *Proceedings of the Fourth International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, October 2006.

[46] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution," in *Proc. 27th IEEE Real-Time Systems Symposium (RTSS 2006)*, Rio de Janeiro, Brazil, Dec. 2006.

[47] T. Chen, T. Mitra, A. Roychoudhury, and V. Suhendra, "Exploiting branch constraints without exhaustive path enumeration," in *Proc. 5^{th} International Workshop on Worst-Case Execution Time Analysis*, R. Wilhelm, Ed., Palma de Mallorca, July 2005, pp. 40–43.

[48] T. Harmon, "Interactive worst-case execution time analysis of hard real-time systems," Ph.D. dissertation, University of California, Irvine, 2009.

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

[50] J. Gustafsson, "The worst case execution time tool challenge 2006," in *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006)*, Paphos, Cyprus, November 2006, pp. 233–240.

[51] L. Ko, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M. Harmon, "Supporting the specification and analysis of timing constraints," in *Proceedings of the Second IEEE Real-Time Technology and Applications Symposium (RTAS 1996)*, Boston, MA, USA, June 1996, pp. 170–178.

[52] J. R. P. Ribeiro, N. C. da Silva, and C. E. Morón, "A visual environment for the development of parallel real-time programs," in *Proceedings of the 12th International Parallel Processing Symposium / Ninth Symposium on Parallel and Distributed Processing (IPPS/SPDP 1998)*, ser. Lecture Notes in Computer Science, vol. 1388. Springer Berlin, March 1998, pp. 994–1014.

[53] R. Kirner, "Consideration of optimizing compilers in the context of WCET analysis," in *Proceedings of the Deutsche Informatiktage 2000*, October 2000, pp. 123–126.

[54] R. Kirner, R. Lang, G. Freiberger, and P. Puschner, "Fully automatic worst-case execution time analysis for Matlab/Simulink models," in *Proc. 14th Euromicro Conference on Real-Time Systems*, Vienna University of Technology. Vienna, Austria: IEEE, June 2002, pp. 31–40.

[55] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull, *The Real-Time Specification for Java*, G. Bollella, Ed. Addison Wesley Longman, January 2000.

[56] P. Persson and G. Hedin, "An interactive environment for real-time software development," in *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 2000)*. Washington, DC, USA: IEEE Computer Society, June 2000, pp. 57–68.

[57] G. Bernat and M. Bennett, "Identifying opportunities for worst-case execution time reduction in an avionics system," in *Proceedings of the Twelveth International Conference on Reliable Software Technologies (Ada-Europe 2007)*, June 2007.

[58] J. Fauster, R. Kirner, and P. Puschner, "Intelligent editor for writing WCET-oriented programs," in *Proc. 3rd International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, Oct. 2003.

[59] M. Nolin, J. Mäki-Turja, and K. Hänninen, "Achieving industrial strength timing predictions of embedded system behavior," in *The 2008 International Conference on Embedded Systems and Applications (ESA 2008)*, Las Vegas, Nevada, USA, July 2008.

Trevor Harmon completed his Ph.D. in 2009 at the University of California, Irvine, where he received a Graduate Research Fellowship award from the National Science Foundation. The following year he was accepted into the NASA Postdoctoral Program to expand his research in worst-case execution time. He also contributed to the NHTSA-NASA Study of Unintended Acceleration in Toyota Vehicles. Dr. Harmon is currently a software engineer for Intel Corporation in Santa Clara, California.

Martin Schoeberl is Associate Professor at the Department of Informatics and Mathematical Modelling of the Technical University of Denmark. Before joining DTU, he was Assistant Professor at the Institute of Computer Engineering of the Vienna University of Technology.

His research interest is in time-predictable computer architecture and real-time Java. He is a member of the expert group for the Safety-Critical Java Specification. Martin Schoeberl has published more than 80 refereed conference and journal papers.

Raimund Kirner received his M.Sc. and Ph.D. degrees from the Vienna University of Technology, Austria, in 2000 and 2003, respectively. In 2011 he received his habilitation (*Privat-Dozent* degree) from the same institution.

From 2000 to 2010 he worked as a research assistant and assistant professor at the Vienna University of Technology. There he was principal investigator of three research projects, working on reliability of real-time systems, especially worst-case execution time analysis. In 2010 he became a Principal Lecturer at the University of Hertfordshire, United Kingdom, working on embedded and parallel computing. He is the local coordinator there of the IST-FP7 project Advance. He chaired the program committee of WDES 2006 and WCET 2008 and has been a member of numerous technical program committees. He is a member of the IEEE Computer Society, the ACM, the IFIP WG 10.2 (Embedded Systems) and the Austrian Computer Society (OCG).

Raymond Klefstad received his Ph.D. in 1988 from the Department of Information and Computer Science at the University of California, Irvine (UCI). He did research as an Adjunct Professor in the Department of Electrical Engineering and Computer Science in the Henry Samueli School of Engineering at UCI, and he is currently serving as a lecturer at the University of California, Riverside. His research focuses on computer architecture, middleware frameworks, and component systems for distributed, real-time, and embedded computing.

Kwang H. (Kane) Kim passed away on June 2, 2011, after a long illness. He was a professor at the University of California, Irvine, in the Department of Electrical Engineering and Computer Science. Dr. Kim contributed immensely to UC Irvine, to his field, and to the profession as a whole. He established the Computer Engineering program at UCI in 1986 and over the years contributed actively to its growth. With tremendous passion he planted and nurtured fruitful pioneering areas of research and education in real-time computing, fault-tolerant computing, distributed computing, embedded systems, and related areas. Professor Kim was greatly respected by his peers for his brilliance, foresight, persistence, and outstanding accomplishments. He received many awards, including the prestigious IEEE Technical Achievement Award, the IEEE Tsutomu Kanai Award, and the SDPS Transformational Award, among others.

Michael R. Lowry serves as NASA's Chief Scientist for Reliable Software Engineering. Dr. Lowry has been principle investigator on advanced software engineering technologies through NASA Aeronautics and Space R&D programs since 1996. The major theme of his research has been the automation of mathematically based methods for software engineering including generation and verification of NASA aerospace software systems. He leads human space research for development of advanced software verification tools.

Dr. Lowry received his B.S. and M.S. from MIT and his Ph.D. in 1989 from Stanford University in Computer Science. From 1989 to 1992 he was a computer scientist at the Kestrel Institute, and was principal investigator on several projects related to AI and Software Engineering.

Dr. Lowry is an IEEE Automated Software Engineering Fellow with numerous publications. He serves on the following journal editorial boards: Springer Journal of Automated Software Engineering, ACM Transactions on Intelligent Systems and Technology, and Innovations in Systems and Software Engineering Journal.