N-version Programming in WCET Analysis: Revisiting a Discredited Idea*

Trevor Harmon
NASA Ames Research Center
Mail Stop 269-1
Bldg. N269, Rm. 260, P.O. Box 1
Moffett Field, CA 94035, United States
Trevor.W.Harmon@nasa.gov

Michael R. Lowry
NASA Ames Research Center
Mail Stop 269-2
Bldg. N269, Rm. 236, P.O. Box 1
Moffett Field, CA 94035, United States
Michael.R.Lowry@nasa.gov

ABSTRACT

Worst-case execution time (WCET) analysis is safe in theory, but it may not truly be safe in practice. Even if a particular analysis algorithm is sound, its implementation may contain bugs that result in unsafe WCET estimation. This potential for error is serious, given that the usual purpose of WCET analysis is to verify the correctness of hard real-time systems—software on which entire missions and even human lives may depend.

A possible solution lies in *N-version programming*, where N teams of developers work independently on N unique but equivalent implementations. Although this fault-tolerance technique has been criticized for its statistical assumptions and high cost, it may be perfectly suited to address the inherent risks in implementing WCET analysis tools. This paper argues that N-version programming still has merit and cites an example of how the technique improved the quality of two WCET analysis tools at relatively low cost.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*validation*; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; B.2.2 [**Arithmetic and Logic Structures**]: Performance Analysis and Design Aids—*worst-case analysis*

General Terms

Verification, Reliability

Keywords

N-version programming, worst-case execution time, WCET, real-time systems, safety-critical systems, static analysis, Java, WCET annotations

*This research was 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.

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

1. INTRODUCTION

Reinventing the wheel is not always a bad thing. Iterating over the same problem offers an opportunity for continuous refinement. After all, no design or implementation is ever correct the first time, but perhaps after the third or fourth generation, most of the challenges have been overcome.

In the field of real-time systems, however, the question has not been "Why reinvent the wheel?" but rather "When will the wheel be invented?" Certainly there have been important advances in scheduling theory, but in practice, the tools and techniques for verifying the timing behavior of a program are limited, and even non-existent for some hardware platforms, due to technical difficulties and market realities. As a result, real-time software engineers typically fall back on coarse external observations and empirical evidence to make arguments about predictability. In other words, they perform a sufficient number of tests until they reach some degree of confidence that the system will perform as intended.

This *ad hoc* approach to timing validation is unfortunate, especially considering the role that real-time systems play in our everyday lives. No longer restricted to esoteric space and military applications, they are now responsible for keeping us alive. When we drive, they regulate the engine and brakes. When we fly, they help the pilot take off and land. Yet the status quo is yielding ominous failure statistics. In the automotive industry, for instance, 30% of electronics system breakdowns can be traced back to software timing problems [9]. Both the research community and industry practitioners have a responsibility to make building real-time systems less like an art and more like a science.

Unlike other science-based fields, however, the tools for analyzing the behavior of a real-time system are immature. While doctors have at their disposal remarkable instruments for peering into the inner workings of a patient—X-ray machines, CAT scans, electrocardiograms—software engineers must rely on more rudimentary techniques to diagnose timing symptoms. A common tactic in the safety-critical aerospace industry, for instance, is to over-design systems so that processor utilization is extremely low. The goal is to prevent any unexpected behavior from exceeding the 100% threshold, which would lead to missed deadlines and critical failure.

This tactic is wasteful and potentially unsafe. In spacecraft, for example, CPUs can demand a significant fraction of power resources, and thus an over-provisioned processor may conflict with the limits of the design. The existence

of such a tactic also unmasks a more fundamental problem: Despite decades of research, practitioners still cannot trust modern tools and techniques to produce a real-time system that performs as expected. Even with substantial measurement, there is no certainty in response time and not enough confidence in the predictability of software. For hard real-time systems, a deeper, stronger guarantee is necessary.

2. WHAT IS WCET?

In 1986, while developing a real-time variant of the programming language Euclid, Kligerman and Stoyenko proposed a new way of providing this guarantee [6]. Now known as *worst-case execution time*, or WCET [8, 3, 13], it places an upper bound on the execution time of a given software task, where "execution time" is simply the time a particular processor takes to execute that task. The idea is to make timeliness a property that can be formally analyzed rather than simply measured. It yields a provably correct bound rather than an educated guess. Without it, no guarantee can be made that a system will meet its deadlines.

Given the importance of WCET, the natural question is how to obtain it. The most dependable and systematic approach involves a *static* analysis, in which the program is never executed, but instead evaluated based on a model of the target processor. This analysis begins with the program's control flow graph, where each node represents a basic block in the original program (that is, a sequence of non-branching CPU instructions). Finding the WCET then becomes a matter of finding the longest path through the graph. The "length" of each path is the execution time of its corresponding basic block.

Although a complete discussion of this computation is beyond the scope of this paper, the key point is that measurement can *imply* predictability, but only WCET analysis *guarantees* it. It ensures that the system will never react more slowly in the field than was measured during testing, as illustrated in Figure 1. Just as a doctor can only verify the true nature of a disease with a microscope, a software engineer can only reveal the true timing properties of a program with the aid of a WCET tool.

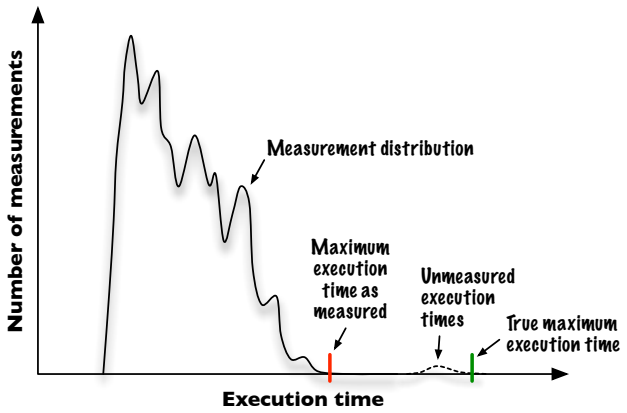

Figure 1: This histogram of execution time for a fictional real-time task illustrates the weakness of measurement when making claims of predictability. Performance testing may produce a data distribution like the shown one here, leading the developer to underestimate the maximum latency of the task.

3. N-VERSION PROGRAMMING FOR WCET

While static WCET analysis is safe in theory, it may not truly be safe in practice. Even if a particular analysis algorithm is sound, the implementation of that algorithm may contain bugs that result in unsafe WCET estimation. The soundness of the CPU model is also a source of error. A modern, out-of-order, heavily pipelined processor, with multi-level caching and complicated shared buses, makes a cycle-accurate model seemingly infeasible. These potentials for error are a serious issue, given that the usual purpose of WCET analysis is to verify the correctness of hard real-time systems—software on which entire missions and even human lives may depend.

One might think to apply some formal verification technique to prove the analyzer's correctness, but this only shifts the problem, as the question then becomes how to verify the analysis verifier, and then perhaps how to verify the verifier that verified the analyzer, and so on, *ad infinitum*. What checks the checker? Who watches the watcher?

This conundrum is, of course, nothing new. Verifying correctness has been a major concern for the aviation and space industries, where a single failure can cause a catastrophe of fire and twisted metal. One approach that has been applied in these fields is known as *N-version programming* [1]. This software engineering practice involves N teams of developers working independently on N unique but equivalent implementations of the same program. In the deployed system, all implementations run concurrently, and when their computations are complete, a separate program examines the results and decides which answer to accept (see Figure 2). For instance, if two implementations of an algorithm agree on a result, but the third one differs, a voting procedure would reject it as incorrect. The approach results in a degree of tolerance to software defects—increasing with the size of N—because each version of the program checks the other N–1 versions.

For some researchers, however, N-version programming is a discredited idea. They argue that the assumption of independent failures among the N versions is statistically invalid [2, 11]. In other words, different programming teams can make similar mistakes. The approach may also be rejected because in many cases it is simply too expensive to be practical. An organization must create a second or third team for the extra implementations, which may double or triple the cost. For these reasons, the widely used DO-178B standard for safety-critical systems discourages N-version programming as a primary tool in the quest for software reliability [11].

Yet in many instances, N-version programming is a sensible solution. Even researchers outside the realm of safety-critical and fault-tolerant systems have found it useful. A book preservation project, for example, relied on a kind of N-version programming to digitize old texts using two separate character recognition programs [12]. If the programs disagreed, the discrepancy was reported and a human took over. The approach was effective in this case because the independent "teams" were commercial off-the-shelf products, so the cost of creating the redundant implementations was, essentially, already paid.

A similar approach could be applied to WCET analysis.

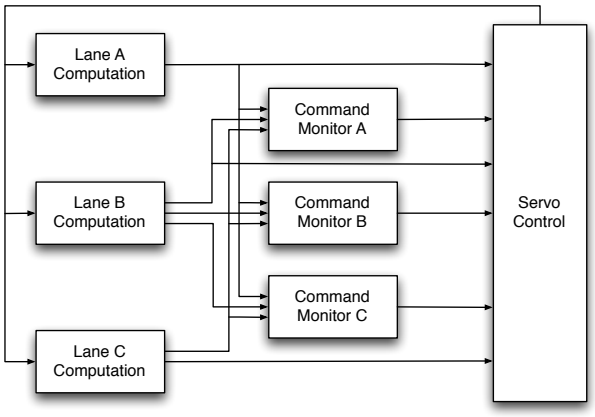

Figure 2: This simplified sketch of a three-lane flight simulator [7] illustrates a real-world example of N-version programming. The three control signals from the autopilot lane computations—each of which was developed by independent programming teams—are fed into a separate servo logic function that decides which, if any, are faulty.

Instead of voting, however, as in a conventional N-version programming scenario, the single "worst" result must be taken. For instance, if one tool indicates that the WCET of a task is 10 milliseconds, while another says 12 milliseconds, and a third reports 15 milliseconds, the user should reject the first two results.

In addition to providing more reliable WCET analysis, this tactic would also help detect bugs in new analysis tools that are still undergoing development. This very approach was applied successfully to the development of Clepsydra, a WCET analyzer for Java [4]. Every line of code in this tool was new and untested, and the sheer difficulty of computing WCET by hand meant that verification of the tool's correctness was limited. Fortunately, a similar WCET analysis tool for Java had recently been published [10], allowing direct comparison of the two tools across a variety of test cases. In some instances, discrepancies between the results were traced to bugs in Clepsydra (usually related to cache analysis), while other discrepancies were caused by the second tool's handling of array variables. In both cases, the bugs were logged and eventually fixed, leading to an improvement in quality of both tools.

Even when there is no second or third tool for comparison, N-version programming can still provide benefits. This counter-intuitive observation comes from the fact that a single tool may incorporate different versions of the same algorithm. The Clepsydra tool, for example, initially provided a longest-path search algorithm based on integer linear programming (ILP), but it was found to be impractically slow in some cases due to the NP-hard complexity of ILP. A simpler, faster alternative based on recursion of the control flow tree was thought to produce identical results, at least in theory [4]. Testing this theory was simply a matter of applying N-version programming: A battery of test cases was run against the faster but untested tree-based algorithm and the slower but known-good ILP implementation, and the results were compared. Discrepancies were indeed found and traced back to implementation mistakes, but eventually both algorithms yielded the same answers for all tests.

4. TOWARD A WCET ECOSYSTEM

Eventually, the proliferation of WCET analysis tools could become a large-scale instance of N-version programming. Teams of programmers around the globe, working independently to create the same type of tool, would implicitly provide a check for the other, ensuring that WCET analysis is safe. Importantly, the work of creating these redundant implementations would be distributed across all customers of the tools, making the cost of N-version programming negligible.

At the moment, this scenario is only a dream. It assumes that a sufficient number of WCET analysis tools is readily available, but for now there exist only a handful of such tools from academia and industry. What is needed is an ecosystem: a healthy population of a variety of interchangeable tools from a variety of independent vendors.

The advent of such an ecosystem is currently blocked, however, largely due to technical limitations. For example, some tools target just one or two specific processors, and all of them place certain restrictions on the kind of source code that they can analyze. Some allow recursion, while others do not; some support C++, while others do not. Because each tool handles a slightly different set of processor models and source code features, they cannot act as drop-in replacements for each other. For the goal of N-version programming, which demands interoperability, these are major obstacles.

Yet even small obstacles, put in place by the arbitrary decisions of WCET tool implementors, remain unsolved. Consider loop bound annotations, which allow the tool's user to specify the maximum number of iterations of a loop. Such annotations are necessary in cases where the tool cannot automatically determine the loop bound. Inexplicably, every tool insists on its own proprietary syntax for expressing this bound, as shown in Table 1. Each row of the table provides an example of how a loop bound of ten would be annotated for a particular tool. Observe that not one of these fourteen tools shares a common syntax.

This compatibility issue—merely an artificial boundary—is therefore a sign of immaturity in the field. There is no technical reason why WCET tools cannot share the same syntax for loop annotations. Tool implementors simply have not agreed on a common standard [5]. There are clearly opportunities for practitioners and researchers to work together more closely and more often, for the benefit of the field as a whole. Until this happens, source code for hard real-time systems will remain locked to the flavor of a particular analysis tool, and the potential of N-version programming cannot be realized.

5. CONCLUSION

Software faults may not be strictly independent, as required by N-version programming. If, for example, software derives from the same specification or requirements document, different teams of developers creating that software may make the same mistakes when interpreting the document. WCET analysis tools, however, are not built from a spec. Approaches to cache analysis, in particular, rely on creative strategies and innovative algorithms, rather than

Table 1: A sample of WCET loop annotation styles

Tool	Example Syntax
aiT	/* ai: loop here max 10; */
Bound-T	loop repeats 10 times;
calc_wcet_167	WCET_LOOP_BOUND (10)
Chronos	line45 <= 10
Clepsydra	@LoopBound(max=10)
Heptane	for (i = 0; i < N; i++) [10] {...}
OTAWA	loop 0x0005013c 10;
RapiTime	#pragma RPT loop_max_iter (10);
Skånerost	/*$ loop-bound 10 */
TuBound	#pragma wcet_trusted_loopbound(10)
WCA	//@WCA loop=10
WCC	_Pragma("loopbound min 10 max 10")
WCETAn	WCETAn.Loopcount(10);
XRTJ	//@ Loopcount(10)

simply implementing a set of use cases. The inherent complexity of the analysis makes independent faults more likely.

Whatever the explanation, it is clear that N-version programming, given the proper context, can lead to faster bug discovery and correction. In fact, some of Clepsydra's bugs may never have been found otherwise. This anecdotal evidence shows that N-version programming has merit and, though not the only means of ensuring software correctness, should never be rejected outright.

Of course, more research is needed. Experiments like Brilliant et al.'s large-scale study [2], which would involve multiple WCET analysis tools tested against dozens of independent real-time software programs, could show the viability of N-version programming. If successful, the results would apply to other situations, as well. Defect finders and model checkers, for instance, are likely to exhibit the same independence of faults, allowing them to check each other for correctness.

Even if such an experiment sparks a renewed interest in N-version programming, it cannot be applied to WCET analysis tools on a larger scale until certain compatibility problems, such as annotation syntax, are overcome. With sufficient effort and cooperation, however, a healthy ecosystem of interoperable tools will one day thrive.

6. REFERENCES

[1] A. A. Avižienis. *The Methodology of N-Version Programming*, chapter 2. John Wiley and Sons, 1995.

[2] S. S. Brilliant, J. C. Knight, and N. G. Leveson. Analysis of faults in an N-version software experiment. *IEEE Transactions on Software Engineering*, 16(2):238–247, February 1990.

[3] 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*, 4(4):437–455, August 2003.

[4] T. Harmon, R. Kirner, M. Schoeberl, and R. Klefstad. A modular worst-case execution time analysis tool for Java processors. In *Proceedings of the Fourteenth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, pages 47–57, April 2008.

[5] 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.

[6] E. Kligerman and A. D. Stoyenko. Real-time Euclid: a language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12(9):941–949, September 1986.

[7] M. R. Lyu and Y.-T. He. Improving the N-version programming process through the evolution of a design paradigm. *IEEE Transactions on Reliability*, 42(2):179–189, June 1993.

[8] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2-3):115–128, May 2000.

[9] B. Rieder, I. Wenzel, K. Steinhammer, and P. Puschner. Using a runtime measurement device with measurement-based WCET analysis. In *Proceedings of the 2007 International Embedded Systems Symposium (IESS 2007)*, pages 15–26, June 2007.

[10] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 2010.

[11] L. Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, July/August 2001.

[12] L. von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum. reCAPTCHA: Human-based character recognition via web security measures. *Science*, August 2008.

[13] 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*, 7(3):1–53, April 2008.