An Abstract Interpretation-Based Timing Validation of Hard Real-Time
Avionics Software*

Stephan Thesing
Universität des Saarlandes
FR Informatik
Postfach 15 11 50
D-66041 Saarbrücken, Germany
thesing@cs.uni-sb.de

Jean Souyris
Airbus France
316, route de Bayonne
Saint-Martin-du-Touch
F-31063 Toulouse cedex 03, France
jean.souyris@airbus.com

Reinhold Heckmann
AbsInt GmbH
Stuhlsatzenhausweg 69
D-66123 Saarbrücken, Germany
heckmann@absint.com

Famantanantsoa Randimbivololona
Airbus France
316, route de Bayonne, Saint-Martin-du-Touch
F-31063 Toulouse cedex 03, France
famanta.randim@airbus.com

Marc Langenbach
Universität des Saarlandes, FR Informatik
Postfach 15 11 50, D-66041 Saarbrücken, Germany
mlangen@cs.uni-sb.de

Christian Ferdinand
AbsInt GmbH
Stuhlsatzenhausweg 69
D-66123 Saarbrücken, Germany
ferdinand@absint.com

Abstract

Hard real-time avionics systems like flight control software are expected to always react in time. Consequently, it is essential for the timing validation of the software that the worst-case execution time (WCET) of all tasks on a given hardware configuration be known. Modern processor components like caches, pipelines, and branch prediction complicate the determination of the WCET considerably since the execution time of a single instruction may depend on the execution history. The safe, yet overly pessimistic assumption of no cache hits, no overlapping executions in the processor pipeline, and constantly mispredicted branches results in a serious overestimation of the WCET. Our approach to WCET prediction was implemented for the Motorola ColdFire 5307. It includes a static prediction of cache and pipeline behavior, producing much tighter upper bounds for the execution times. The WCET analysis tool works on real applications. It is safe in the sense that the computed WCET is always an upper bound of the real WCET. It requires much less effort, while producing more precise results than conventional measurement-based methods.

1. Introduction

Airbus’ flight control software consists of tasks with specified deadlines. It is the duty of the developer to guarantee that the tasks making up the system will always meet these specified deadlines. For this, the worst-case execution times (WCET) of the tasks must be determined. Since the exact WCET is hard to obtain, an upper bound for the WCET must be known (from now, we follow common ter-

*This work was partly supported by the RTD project IST-1999-20527 “DAEDALUS” of the European FP5 program.
minology in taking WCET to mean upper bound to WCET). It is crucial for the safety of the system that any computed WCET is safe, i.e., no execution of the task takes longer. But, the WCET should also be tight, i.e., a possible overestimation should be as low as possible to increase the utilization of the processor and thus the performance of the system.

The presence of optimizations in compilers makes it very difficult to argue about the timing behavior of high-level source programs or specifications. In addition, validation requirements imposed by the certification authorities, e.g., DO 178B for air traffic software, require that validation be performed on the code that actually runs in the real-time system, i.e., on the machine code itself. It is for these two reasons that timing validation must be performed on the object code of the program.

There is a tremendous gap between the cycle times of modern microprocessors and the access times of main memory. Caches are used to overcome this gap in virtually all performance-oriented processors (including high-performance microcontrollers and DSPs). Pipelines enable acceleration by overlapping the executions of different instructions. The consequence is that the execution behavior of the instructions cannot be analyzed separately since this depends on the execution history.

Cache memories and pipelines usually work very well, but under some circumstances minimal changes in the program code or program input may lead to dramatic changes in the execution time. For (hard) real-time systems like a flight control computer, this is undesirable and possibly even hazardous. The widely used classical methods of predicting execution times are not generally applicable. Software monitoring or the dual loop benchmark change the code, which in turn impacts the cache behavior. Hardware simulation, emulation, or direct measurement with logic analyzers can only determine the execution times for some inputs and cannot be used to infer the execution times for all possible inputs in general.

Consequently, Airbus uses its own method for its most time-critical avionics software. This method is based on strict design and coding rules, the most deterministic usage of the internal speed-up mechanisms of the microprocessor, and measurements of pieces of code whose limited size makes it possible to obtain a WCET for all their possible inputs. This method allows Airbus to compute a safe WCET for the whole program by combining the WCETs of the small pieces. An appropriate combination formula exists thanks to the design and coding rules. However, this method poses the following drawbacks: it limits the effective power of the CPU, requires some effort for the measurements and related intellectual analysis, and cannot be performed too early during software development because the hardware has to be available for measurement. To be sure, complex extensive verification and justification of the measurement process is required in order to ensure that an upper bound of the WCET is really being observed.

Starting from Airbus’ assessment that its current measurement-based method might not scale up to future projects, a new approach based on static program analysis has been investigated and evaluated in the DAEDALUS project. In static program analysis or—to be more exact—within the framework of abstract interpretation [1, 12], abstract versions of the program semantics are defined in order to obtain a safe approximation of the real behavior of the program so that results are produced that are valid for all executions. These abstract versions should be designed so that they can be computed efficiently without giving up too much precision. In addition, since this process does not require the real hardware to execute the code on, the analysis can also be performed in an early stage of the development of a new system. Our implementation additionally provides detailed information on the pipeline and cache states at arbitrary program points. This information can be used in code optimization.

In the following sections, our static analysis method is presented as applied to a system based on the Motorola ColdFire 5307. Then a more detailed description of the tool implementing the WCET analysis is given along with some results from its evaluation on real-life benchmarks by Airbus France. This paper is concluded by a summary of our results, a comparison with other research in this area, and a presentation of future work.

2. WCET Computation

In our approach [4] the determination of the WCET of a program task is composed of several phases (see Figure 1): First, the control flow has to be reconstructed from the given object code. Then a value analysis computes address ranges for instructions accessing memory. This information is used in cache analysis, which classifies memory references as cache misses or hits [3]. To increase precision, this is performed as part of the pipeline analysis, which predicts the behavior of the program on the processor pipeline [7] and computes execution times for all basic blocks. Then path analysis determines a worst-case execution path of the program from this timing information [18].

The separation of WCET determination into several phases has the additional benefit that different methods tailored to the subtasks can be used. In our case, value analysis and cache/pipeline analysis are done by abstract interpretation [1], a semantics-based method for static program analysis. Path analysis is done by integer linear programming.

In order to obtain as much precision as possible, the value and cache/pipeline analyses are implemented as
whole program analyses, i.e., they assume a linked executable to be available.

2.1. Reconstruction of the Control Flow from Binary Programs

The starting point of our analysis framework (see Figure 1) is a binary program and additional user-provided information about numbers of loop iterations, upper bounds for recursion, etc.

In the first step a parser reads the executable and reconstructs the control flow [15, 16]. This requires some knowledge about the underlying hardware, e.g., which instructions represent branches or calls. The reconstructed control flow is annotated with the information needed by subsequent analyses and then translated into CRL (Control Flow Representation Language—a human-readable intermediate format designed to simplify analysis and optimization at the executable/assembly level). This annotated control-flow graph serves as the input for microarchitecture analysis.

2.2. Value Analysis

Value analysis determines ranges for values in registers, thus resolving indirect accesses to memory. The results of the analysis are so good that only a few indirect accesses cannot be determined exactly.

2.3. Cache Analysis

Cache analysis classifies the accesses to main memory. The analysis in our tool is based upon [3], however it had to be modified to reflect the pseudo-round-robin replacement policy of the ColdFire 5307 cache. The modified algorithm distinguishes between sure cache hits and unclassified accesses.

2.4. Pipeline Analysis (for MCF 5307)

Pipeline analysis models the pipeline behavior to determine execution times for a sequential flow (basic block) of instructions, as done in [13, 14]. It takes into account the current pipeline state(s), in particular resource occupancies, contents of prefetch queues, grouping of instructions, and classification of memory references by cache analysis. The result is an execution time for each basic block in each distinguished execution context.

The ColdFire family of microcontrollers is the successor to Motorola’s M68k architecture. The ColdFire 5307 [6] is an implementation of the version 3 ColdFire architecture. It contains an on-chip 4K SRAM and a unified 8K data/instruction cache.

The ColdFire implements a subset of the M68K opcodes, restricting instruction lengths to two, four, or six bytes, thereby simplifying the decoding hardware. The CPU core and the external memory bus can be clocked with different speeds (e.g., 20MHz bus clock and 60MHz internal core clock).

The MCF5307 has two pipelines decoupled by an instruction buffer (see Figure 2): a fetch pipeline fetches instructions from memory, partially decodes them, performs
branch prediction, and places the instructions in an instruction buffer, consisting of a FIFO with eight entries (complete instructions). The execution pipeline consists of two stages. It receives complete instructions from the instruction buffer, decodes and executes them.

The implementation of the pipeline analysis is based on a formal model of the pipeline (see Figure 3). This formal model consists of several units with inner states that communicate with one another and the memory via signals, and evolve (processor) cycle-wise according to their inner state and the signals received.

The decomposition into units accounts for reduced complexity and easier validation of the model. Units often map directly to pipeline stages, but also may represent more than one stage or introduce virtual pipeline stages that are not present in the hardware but facilitate the design of the pipeline model (e.g., SST).

Signals may be instantaneous, meaning that they are received in the same cycle as they are sent, or delayed, meaning that they are received one cycle after they have been sent. Signals may carry data with them, e.g., a fetch address. Note that these signals are only part of the formal pipeline model. They may or may not correspond to real hardware signals.

The inner states and emitted signals of the units evolve in each cycle. The complexity of this state update varies from unit to unit. It can be as simple as a small table, mapping pending signals and the inner state to a new state and signals to be emitted, e.g., for the IAG unit. This can be much more complicated if multiple dependencies have to be considered, e.g., instruction reconstruction and branch prediction in the IED stage. In this case, the evolution is formulated in pseudo code.

Full details of the model can be found in [5] and [7].

The pipeline analysis of an executable program should be understood as working on three levels: A program-level analysis computes abstract execution states for all the basic blocks of the program. For this it iterates the block-level analysis over the basic-block graph of the program, until stability, i.e., a fixed point, is reached (none of the abstract execution states at a basic block changes any more).

A block-level analysis goes through the sequence of instructions of each basic block, and an instruction-level analysis performs a cycle-wise evolution of each instruction. The pipeline analysis on the block level works as follows: A set of abstract pipeline states at a program point contains states possibly reachable by some execution. Pipeline analysis of a basic block starts with such a set of abstract states determined by the context, by which the block was reached. Instruction-level analysis of each instruction starts with the set of abstract pipeline states determined at its entry and computes a set of successor states at its exit, which is propagated to other basic blocks. This is done by the cycle-wise evolution of pipeline states reminding one of the simulation of the instruction. However, the abstract execution on the instruction level is in general non-deterministic, since information determining the evolution of the execution state are missing, e.g., non-predictable cache contents. Therefore, the evolution of the instruction execution may have to branch into several successor states. All the states computed in such tree-like structures form the set of entry states for the successor instruction(s).

Together the analyses on these three levels form an abstract execution of the block’s instruction sequence, since they abstract from certain parts of the execution state that are irrelevant for the determination of execution times, e.g., data values in registers.

The output of the pipeline analysis is the number of cycles a basic block takes to execute, for each context, obtained by taking the upper bound of the number of simulation cycles for the sequence of instructions for this basic block. These results are then fed into the path analysis to obtain the WCET for the whole program.

The result of each cycle update, i.e., the set of pipeline states at each instruction (or basic block), is visualized by the analysis and can be examined with the WCET tool, cf. section 3. This makes it possible to follow very precisely the evolution of the pipeline during the execution of a program part, and discover and explain the causes of unexpected timing behavior.

Figure 3. Map of the formal pipeline model
2.5. Path Analysis

Using the results of the microarchitecture analyses, path analysis determines a safe estimate of the WCET. The program’s control flow is modeled by an integer linear program (ILP) [17, 18] so that the solution to the objective function is the predicted worst-case execution time for the input program.

The generated ILP describes the control-flow restrictions of the basic blocks in the program, e.g., for the blocks of an if-then-else statement. The minimal and maximal iteration counts for loops (provided by the user) are integrated as constraints. Additional control-flow constraints from the user, representing knowledge about dependencies of program parts, can be integrated as further constraints. The objective function to be maximized under all these constraints is the sum over the weighted execution counts of basic blocks on the program’s paths, whereby the execution count is multiplied by the WCET of the basic block determined for the appropriate context by the cache/pipeline analysis.

2.6. Analysis of Loops and Recursive Procedures

Loops and recursive procedures are of special interest since programs spend most of their runtime there. Treating them naively when analyzing programs for their cache and pipeline behavior results in a high loss of precision.

The user has to provide lower and upper bounds for the iteration counts of all loops. These bounds are later used by the path analysis to compute the global WCET from an ILP. Furthermore, these bounds can also be used to increase the precision of the value and cache/pipeline analyses.

Frequently the first execution of the loop body loads the cache, and subsequent executions find most of their referenced memory blocks in the cache. Hence, the first iteration of the loop often encounters cache contents quite different from that of later iterations. Therefore, it is useful to distinguish the first iteration of loops from the others. This is done in the VIVU approach (virtual inlining, virtual unrolling) so that memory references are considered in different execution contexts, essentially nestings of first and non-first iterations of loops [11].

Using the loop iteration bound information, the analyses can virtually unroll not only the first iteration, but all iterations. With this the analyses can distinguish more contexts and the precision of the results is increased.

2.7. Safety of the analyses

Reliability of the results obtained by the analyses is a crucial property. The value and cache/pipeline analyses are based on the framework of abstract interpretation, which gives a guarantee that the results obtained are a safe approximation of any real execution of a program. Thus, the value analysis computes intervals for the values in registers guaranteed to contain all values possible during an execution. These intervals are used to compute address ranges for accesses to memory in a safe way. In the same way, all memory blocks predicted to be in the cache by the cache analysis are guaranteed to be in the cache during any concrete execution.

The pipeline analysis is based on an abstract model of the processor. Since the analysis follows every possible evolution that may be feasible under this model, it is guaranteed to process every possible pipeline state that can occur during executions of a program. The only thing to be done is to validate the pipeline model against the real processor hardware, which is currently done by comparing real against predicted execution traces. In the future, formal methods will be employed.

Finally, the path analysis is guaranteed to find a maximal solution to the global WCET, which is constrained by the ILP.

Thus, the results of the whole analysis are based on safe theoretical frameworks that guarantee the necessary safety property.

3. aiT – The ColdFire WCET Tool

The user interface to the WCET tool aiT is depicted in Figure 4. The user has to select the executable to be analyzed and a starting point for the analysis, usually a procedure entry.

There are three possible actions:

- The **CFG** button produces a visualization without WCET analysis. The result is the combined call and control-flow graph without any WCET annotations (see Figure 5).

- The **Analyze** button starts a full WCET analysis. Its results are merged into the call and control-flow graph without any WCET annotations (see Figure 6). The overall WCET can be read off as well as the contribution of every basic block along a critical path.

- The **Visualize** button starts WCET analysis without path analysis. For each program point and execution context, the resulting cache and pipeline states are displayed on demand. This way, the user can observe the cycle-wise evolution of pipeline and cache states and gain a better understanding of the interaction between the software and hardware.
3.1. Practical Experiences

In the course of the DAEDALUS project, the WCET analyzer was evaluated by Airbus France in order to assess its usefulness. This procedure involved an evaluation of the precision of the predicted WCET. Another aspect was usability, i.e., whether a “normal” developer can use the tool on common computer hardware. Other aspects included scalability and performance on programs of real-life size and the overall reliability of the tool (i.e., no program crashes, etc).

The evaluation was done on a representative benchmark for the most time-critical avionics software. This benchmark consists of twelve tasks activated in round-robin fashion by a non-preemptive scheduler. Each task is made up of hundreds of smaller components called nodes and functions. The WCET tool was run on the entire tasks and each of the nodes and functions in the tasks. The results obtained were then compared to the WCETs obtained by a different, well-established method.

3.2. Results

The results of Airbus France’s evaluation were very positive. The tool was run several thousand times with very high reliability. It was able to handle real-life sized programs on common hardware (1GHz Athlon). The evaluation process was performed by Airbus France personnel and did not require an operator highly specialized in program analysis.

The graphical visualization features of the tool with regard to WCET distribution over all called functions and cycle-wise pipeline evolution were very helpful. The analyzer was able to deliver a safe and tight WCET prediction. The results were typically better than those obtained with the legacy method used for comparison. The application is comprised of 12 tasks whose individual WCET is required. For a given version of the application, Table 1 compares the WCETs computed by Airbus’ method with the results of AbsInt’s aiT. The pronounced improvements in precision are due to the fact that the legacy method had to be overly conservative in its WCET computations. aiT can be more precise due to its detailed modeling of the hardware features, but its results are still correct upper bounds of the (unknown) true worst-case execution times.

Since the true WCETs cannot be known in general, the only way to check results from WCET computation methods is to measure real executions (which give times probably below the real WCET) and compare the measurements against the analyses results. Doing this, it was verified that the predicted WCETs are greater than, but not far away from, observed running times.

The code analyzed didn’t have to be instrumented in order to apply the tool. No changes in the development pro-
Airbus’ method | aiT’s results | precision improvement
--- | --- | ---
1 | 6.11 ms | 5.50 ms | 10.0 %
2 | 6.29 ms | 5.53 ms | 12.0 %
3 | 6.07 ms | 5.48 ms | 9.7 %
4 | 5.98 ms | 5.61 ms | 6.2 %
5 | 6.05 ms | 5.54 ms | 8.4 %
6 | 6.29 ms | 5.49 ms | 12.7 %
7 | 6.10 ms | 5.35 ms | 12.3 %
8 | 5.99 ms | 5.49 ms | 8.3 %
9 | 6.09 ms | 5.45 ms | 10.5 %
10 | 6.12 ms | 5.39 ms | 11.9 %
11 | 6.00 ms | 5.19 ms | 13.5 %
12 | 5.97 ms | 5.40 ms | 9.5 %

Table 1. Comparison between Airbus’ legacy method and aiT

process of the programs to be analyzed were necessary.

The results of the evaluation are very encouraging. We believe that the WCET tool not only can be used in verifying that WCET constraints are met but also in earlier stages of the development process as well. At a stage when the software is already available, but working hardware is not, the tool can be used for a performance evaluation. Based on the contributions of the program parts to the WCET one can make design decisions, e.g., with respect to static scheduling or code/data placement. The effects on the cache and pipeline can be viewed using the visualization options of the tool and causes for unexpected local timing behavior identified.

4. Conclusions

We have presented a tool to obtain WCETs for time-critical real-time systems. The tool is based on static analysis techniques and is thus applicable without requiring the real hardware of the system. The tool was evaluated in a real-life setting by Airbus France and the results are very promising. We have shown that our tool not only can provide safe and tight WCET bounds but that it can also be applied to realistically sized programs and environments as well.

4.1. Future Work

The methodology underlying the WCET analyzer for the ColdFire 5307 is also being used to implement a WCET analyzer for the PowerPC 755. This processor is much more sophisticated, featuring super-scalarity, out-of-order execution, branch prediction and folding, and truly parallel execution units. A prototype of the WCET analyzer for PowerPC 755 is currently undergoing an extensive verification process.

We plan to model more processors for further WCET analyzers in the same way. Also, a more formal way to specify the pipeline model and derive its implementation is being investigated.

4.2. Related Work

A vast body of literature on WCET determination exists. We have only listed references dealing with complex processors containing all features considered in combination, not architectural features in isolation.

Li et al. suggest a solution using integer linear programming [8]. Both cache and pipeline behavior prediction are formulated as a single linear program. The i960KB is investigated, a 32-bit microprocessor with a 512 byte direct mapped instruction cache and a fairly simple pipeline. Only structural hazards need to be modeled, thus keeping the complexity of the integer linear program moderate. Variable execution times, branch prediction, and instruction prefetching are not considered at all. Using this approach for super-scalar pipelines does not seem very promising, considering the analysis times reported in the article. Nonetheless, the description of the worst-case path through the program via ILP is an elegant method and can be efficient if the size of the ILP is kept small. This is the case in our tool.

Lundqvist and Stenström present an integrated approach for obtaining WCET bounds through the simulation of the pipeline in [9, 10]. They extend a pipeline simulator to handle unknown values in inputs. We share conceptual similarities with this approach in that we perform a cycle-wise evolution of a pipeline (model). In contrast to our approach, Lundqvist and Stenström use an integrated method in which value analysis for register/memory contents and execution time computation are parts of the same simulation. If the simulation cannot determine a branch condition exactly due to dependencies on unknown (input) values, both branches have to be simulated. This method does not guarantee termination of the analysis, but offers the advantage of sometimes determining loop bounds and/or recursion bounds “for free” (providing they don’t depend on unknown input values in a non-trivial way). However, we feel that this analysis is very costly due to the huge amount of data that has to be kept for each branch followed.

In contrast, our method does not retain information like register or memory contents in the pipeline analysis phase, contents that have already been determined in the value analysis to predict conditional and computed branches, for example. In [10] experiments with a PowerPC-like architecture are conducted for small example programs using an
extended PSIM simulator with simple reservation tables for instructions. All in all, it is not clear how well this method scales up to programs of realistic size.

In contrast to Lundqvist and Stenström’s integrated approach, Engblom presents a WCET tool in [2] with a clear separation of all the analysis modules. The modules communicate using interface data structures. One main component is a simulator that estimates the execution time for a given sequence of instructions. These timing estimates are composed to form the execution time of the entire program. The quality of the WCET obtained is greatly influenced by the quality of the simulator used. Cache behavior prediction is not incorporated in the tool as the addressed targets do not have any caches. This eliminates the problem of cache and pipeline interaction, which becomes more difficult with increasingly complex pipelines, prefetching, and branch prediction. The author comes to the conclusion that “…out-of-order processors are definitely too complex to model with current techniques.”

References


