Validating a Worst-Case Execution Time Analysis Method for an Embedded Processor ◇

Jakob Engblom†*
IAR Systems AB
Box 23051, SE-750 23 Uppsala
Sweden
email: jakob.engblom@iar.se

Andreas Ermedahl†
Dept. of Information Technology
Uppsala University
Box 325, SE-751 05 Uppsala
Sweden
email: andreas.ermedahl@docs.uu.se

Friedhelm Stappert‡
C-LAB
Fürstenallee 11
33102 Paderborn
Germany
email: fst@c-lab.de

Abstract

Knowing the Worst-Case Execution Time (WCET) of a program is necessary when designing and verifying real-time systems. When evaluating WCET analysis methods, the common methodology is to compare a WCET estimate with an execution of the same program with known worst-case data on the target hardware. This evaluation method is inadequate, since errors in one part of the analysis might mask errors occuring in other parts of the analysis.

In this paper we present a methodology for systematically testing WCET analysis tools for modern pipelined processors. The methodology is based on a decomposition of WCET analysis into a set of components that should be tested and validated in isolation. Our testing methodology does not require that we have a perfect model of the hardware, thus the validation of the hardware model is considered as a separate problem.

We apply the methodology to our previously published WCET analysis method, and show that the pipeline analysis and the calculation method we use are safe and produce tight results.

We also show that our WCET analysis method can handle programs containing nested loops, functions whose execution times depend on parameters, multiway branches (switch statements) and unstructured code.

Keywords: WCET, pipeline analysis, hard real-time, embedded systems, testing, validation.

1. Introduction

The purpose of *Worst-Case Execution Time* (WCET) analysis is to provide a-priori information about the worst possible execution time of a program before using the program in a system.

Knowing the WCET of a program or piece of a program is necessary when designing and verifying real-time systems. Considering that every day, more and more devices are being controlled by embedded real-time systems (from kitchen appliances, through power grids, to cars and other vehicles), the value of having reliable software cannot be overestimated.

WCET estimates are used in real-time systems development to perform scheduling and schedulability analysis, to determine whether performance goals are met for periodic tasks, and to check that interrupts have sufficiently short reaction times.

To be valid, WCET estimates must be *safe*, i.e. guaranteed not to underestimate the execution time. To be useful, they must be *tight*, i.e. provide low overestimations.

The WCET depends both on the program flow (like loop iterations and function calls), and on architectural factors like caches and pipelines. Thus, both the program flow and the hardware the program runs on must be modelled in a WCET analysis.

When evaluating WCET analysis methods, the com-

*Jakob is an industrial PhD student at IAR Systems (http://www.iar.com) and Uppsala university, sharing his time between research and development work.

†This work is performed within the Advanced Software Technology (ASTEC, http://www.docs.uu.se/astec) competence center, supported by the Swedish National Board for Industrial and Technical Development (NUTEK, http://www.nutek.se).

‡Friedhelm is a PhD student at C-LAB (www.c-lab.de), which is a cooperation of the University of Paderborn and Siemens AG.

◇ Uppsala University, Department of Information Technology Technical Report 2001-030, first release, December 2001 (http://www.it.uu.se/research/reports/). ISSN 1404-3203. Note that this paper describes work performed in the summer of 2000, but only released as a technical report in late 2001.

mon methodology is to compare a WCET estimate with an execution of the same program with known worst-case data on the target hardware.

This evaluation method is problematic, since it mixes the effects of several sources of errors. Errors in program flow analysis, the hardware model used by the method, and the method itself may cancel out each other. Also, if errors are detected, it is very hard to pinpoint the error source.

In this paper, we investigate the safeness (and, to a lesser extent, the tightness) of an implementation of our previously published WCET analysis method [6]. We use a testing methodology designed to isolate the potential errors in each component of our method.

The main contributions of this paper are:

- We present a decomposition of WCET analysis into a set of components that should be tested and validated in isolation.
- We identify the issues involved in systematically testing a WCET analysis method, resulting in a testing methodology.
- We apply the testing methodology to our previously presented WCET analysis method for pipelined processors, and show that our WCET analysis method works as advertised.

The paper is structured as follows: Section 2 presents a division of WCET analysis into components and introduces previous work, and Section 3 presents our WCET analysis tool. Section 4 discusses the issues involved in validating WCET tools. Section 5 presents our test system. Section 6 gives experimental results, and Section 7 our conclusions. Finally, in Section 8 we present our ideas for future work.

2. WCET Analysis Overview and Previous Work

When performing WCET estimation we assume that the program execution is uninterrupted (no preemptions or interrupts) and that there are no interfering background activities, such as direct memory access (DMA) and refresh of DRAM. Timing interference caused by this type of resource contention is assumed to be handled by the subsequent schedulability analysis [2, 14].

To generate a WCET estimate, we consider a program to be processed through the following sequence of steps [7]:

- *Input Data*: determines the possible values of inputs for the program.
- *Program Compilation*: runs the program through a compiler to generate executable code for the target system.
- *Program Flow Analysis*: determines possible program flows, without regard to the time for each "atomic" unit of flow. Also known as high-level analysis.
- *Global Low-Level Analysis*: determines the effects of caches, branch predictors, and other machine-level effects that must be analyzed across the entire program.
- *Local Low-Level Analysis*: determines the effect of the target machine pipeline, memory configuration, bus speeds, etc. Effects that can be handled locally for each "atomic" unit of flow.
- *Calculation Method*: finds the longest executable paths (and their execution time), given the results of the global and local low-level analyses and the flow analysis.

The last four items have been the object of research in the WCET community:

Program Flow Analysis In order to determine the worst-case execution time of the program, we need to analyze the program flow. This provides information about which functions get called, how many times loops iterate, if there are dependencies between different `if`-statements, etc. Flow analysis can either be performed in conjunction with the compilation of the program, e.g. by a special compiler module analyzing the program [11], or by a separate tool [8].

Global Low-Level Analysis The global effects analysis considers the execution time effects of machine features that reach across the *entire program*. Examples of such factors are instruction caches, data caches, branch predictors, and translation lookaside buffers (TLBs). For WCET analysis, instruction caches [9, 10, 16, 26], cache hierarchies [20], data caches [13, 26, 29], and branch predictors [3] have been considered.

Local Low-Level Analysis The local effects analysis handles machine timing effects that depend on a single instruction and its immediate neighbors. Examples of such effects are pipeline overlap and memory access speed. Research have considered simple scalar pipelines [6, 10, 16] and superscalar CPU pipelines [17, 25, 26].

Calculation Method The purpose of the *calculator* is to calculate the final WCET estimate for the program, given the program flow and global and local low-level analysis results. There are three main categories of calculation methods proposed in literature: path-, tree-, or IPET (Implicit Path Enumeration Technique)-based.

In a path-based calculation [10, 26], the final WCET

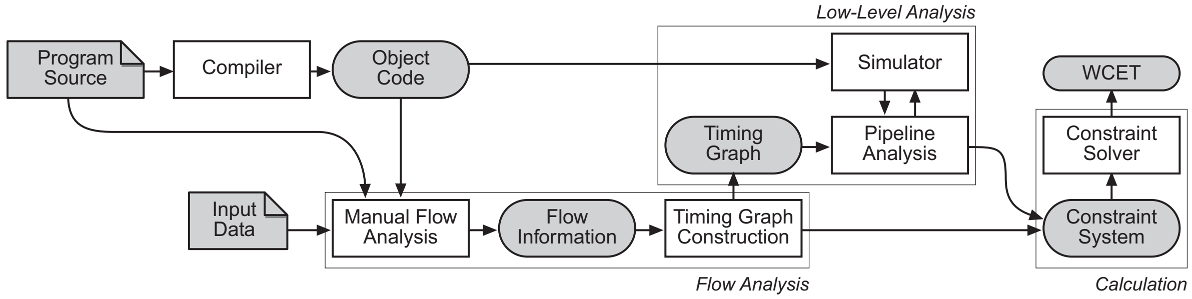

Figure 1. Overview of our current WCET analysis system

estimate is generated by calculating times for different paths in a program, searching for the path with the longest execution time. The defining feature is that possible execution paths are *explicitly* represented.

In tree-based methods [3, 16], the final WCET is generated by a bottom-up traversal of a tree representing the program. The analysis results for smaller parts of the program are used to make timing estimates for larger parts of the program.

IPET-based methods [9, 15, 21, 23] express program flow and atomic execution times using algebraic and/or logical constraints. The WCET estimate is calculated by maximizing an objective function, while satisfying all constraints.

Integrated Approaches

Most WCET tools integrate several of the above components into a single tool, even though the algorithms are kept separate. There are also some methods that completely integrate the WCET analysis, making the above division unusable.

In [22] they perform very sophisticated measurements of programs running on target hardware, aided by static off-line analysis. No attempt is made to perform a static time analysis. In [19] a modified CPU simulator is used to simultaneously perform flow, cache, and pipeline analysis, and calculation.

Validation of WCET Tools

We have found no previous work dealing specifically with the systematic testing of WCET tools.

We do not consider it feasible to formally prove the correctness of an implementation of a certain method. It might be possible to prove the correctness of a certain algorithm, but in general we must rely on systematic testing to find errors in implementations and algorithms.

3. Our WCET Analysis Tool

Figure 1 gives an overview of our WCET analysis system as implemented today. It is a concrete implementation based on the principles presented in [6]

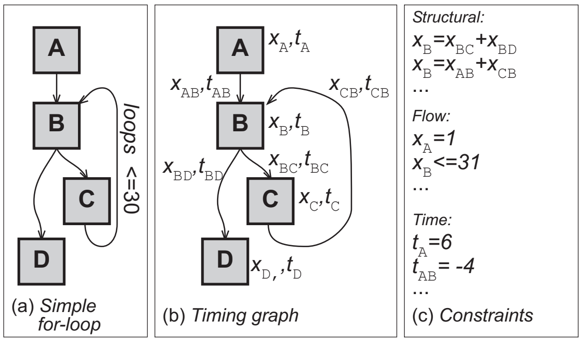

Figure 2. Example of timing graph

(some extensions regarding branch handling had to be made, this is described below). In order to generate a WCET estimate, a program is processed through a number of modules (as described in Section 2 above):

Compiler The compiler is a modified IAR V850/V850E C/Embedded C++ [27] compiler which emits the object code of the program in an accessible format. We only use C code in our prototype tool.

Program Flow analysis At present, we manually inspect the object code, source code, and input data of the test programs, and construct a description of the worst-case program flow to be used by the WCET analyzer. We are working on automating this process.

The program description is based on *contexts* [6, 9]. Each context corresponds to a function or loop in the program, in a certain invocation context (i.e. the sequence of function calls and loop executions leading up to a certain point in the program).

In the case that functions have input-dependent timing behavior, each function invocation is given a separate context.

The program flow information is used to construct a *timing graph*, where the nodes correspond to the basic blocks of the program in the contexts where they execute (a basic block occurring in several calling contexts will be present in several copies). Each such context for a basic block will generate an *execution scenario*.

As shown in Figure 2, the edges and nodes in the timing graph are annotated with *execution time* variables (e.g. $t_{\mathtt{B}}$) and *execution count* variables (e.g. $x_{\mathtt{B}}$). The execution count variables represent the number of times the node or edge is executed in the worst-case execution of the program.

Constraints on the execution counts are used to model program flow. There are constraints to express the finiteness of the program, i.e. the bounding of loops (e.g. $x_{\mathtt{B}} \leq 31$) and the fact that it executes once (e.g. $x_{\mathtt{A}} = 1$), and constraints to make the flow fit together (stating that the flow into and out of each node is the same, e.g. $x_{\mathtt{B}} = x_{\mathtt{AB}} + x_{\mathtt{CB}} = x_{\mathtt{BC}} + x_{\mathtt{BD}}$). For details, we refer to [6]. More constraints can be added manually to cut down the space of paths explored in the calculation.

Global Low-Level Analysis We do not include any cache or other global low-level effect analysis in the current version of the tool, since our target system does not have a cache. Also, we want to ensure that the pipeline modelling is correct before adding caching effects. Cache analysis in the style of [9] is considered for the future.

Note that caches are not very common in the small-but-powerful embedded systems that we are primarily considering as targets for WCET analysis. For instance, most ARM CPUs shipping do not have a cache, and the same goes for most parts from the NEC V850 and Hitachi SH families.

Local Low-Level Analysis We use a *Simulator* to obtain execution times for timing graph nodes and sequences of nodes, instead of a special-purpose pipeline model. The *Pipeline Analysis* feeds the simulator instructions together with information about how the instructions execute. For branches, the simulator needs to know whether they are taken or not, and for memory access instructions the area of memory addressed is needed (to determine the speed of the memory access). The execution information is provided by the execution scenarios.

Times for nodes (e.g. $t_{\mathtt{A}}$) correspond to the execution times of basic blocks in isolation, and times for edges (e.g. $t_{\mathtt{AB}}$) to the effect of the processor pipeline when the basic blocks are executed in sequence (usually an overlap).

Timing effects for sequences of nodes are calculated by first running the individual nodes in the simulator, and then the sequence and comparing the execution times. The process is illustrated in Figure 3. Notice that the edge has a negative time, since the sequence `QR` executes quicker than the sum of `Q` and `R` executed separately.

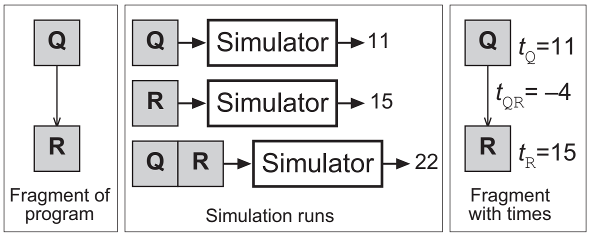

Figure 3. Timing effect calculation

Since pipeline effects can potentially appear across sequences of nodes longer than two, we run progressively longer sequences of nodes until a *termination condition* is satisfied (if timing effects are detected for longer sequences, new time and execution time variables would be generated). The termination condition depends on the CPU used, and should only be true when there is no possibility for a longer sequence to have any effect on the execution time of the program [6].

The simulator is assumed to be trace-driven, which means that it does not have a semantic model of the CPU. It only models the pipeline behavior, given a stream of instructions.

Calculation Method Our calculation technique is based on the *Implicit Path Enumeration Technique* (IPET) [15, 21, 23].

The WCET estimate is generated by maximizing the sum of the products of the execution counts and execution times (subject to the flow constraints):

$$WCET = maximize(\sum_{\forall entity} x_{entity} \cdot t_{entity})$$

This maximization problem is then solved using a constraint solver or integer linear programming (ILP) system. At present, we use the constraint solver of the SICStus Prolog system [12].

4. Validating WCET Tools

According to Section 2 above, we consider WCET analysis to be divided into several independent components. It is necessary to consider the correctness (and effectiveness) of each component in isolation, since otherwise errors in one component may mask errors in other components.

For example, a pessimistic hardware model might mask errors in a flow analysis that generates too short program paths – the resulting estimates might appear to be safe for any given set of test cases, but there could be cases where the analysis would be unsafe.

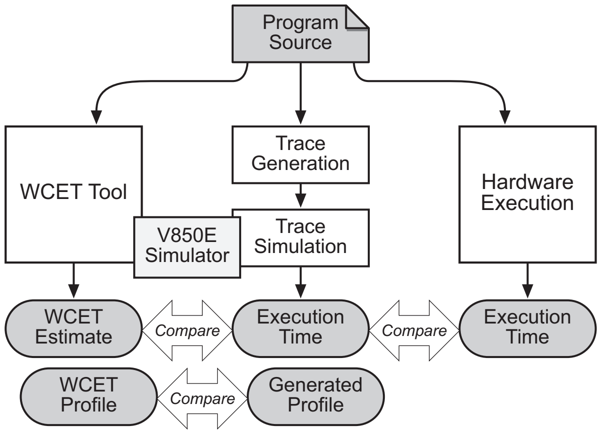

Figure 4. Overview of our testing method

It is thus necessary to consider an incremental testing methodology where each component is tested before the whole system is put to test. Each component must be safe and tight in its own right in order for the complete analysis system to be safe and tight.

In the following, we show how we apply the idea of component-wise isolation to our tool in order to show that the *pipeline analysis* (but not the simulator) and *calculation method* are safe.

4.1. Obtaining a Known WCET

In order to check the correctness of our tool, we need to have a known worst-case execution to compare to. This is obtained by executing instrumented test programs with known worst-case data on a workstation, generating an *execution trace* showing how the programs execute. The trace is a long list of basic block names, corresponding to the order in which the blocks are executed.

The trace is used to drive the simulator we use for our local low-level analysis. The result is an execution time corresponding to an execution of the program on the simulator. Thus, discrepancies between the simulator and the real hardware do not affect the validation.

For other WCET analysis tools, different methods might be required to obtain a known WCET that depends on the hardware model used and not on the real hardware.

The trace is also used to obtain an *execution profile* for the worst-case execution. The profile is a count of how many times each basic block in the program is executed (with no sequence information).

Figure 4 gives an overview of our trace-based testing method. The comparison between the output of the WCET tool and the trace simulation will be used to determine whether our WCET tool is correct or not, and the comparison to the real hardware will indicate the quality of the hardware model.

4.2. Freezing Other Components

The calculation method and pipeline analysis are isolated by making the other components of the WCET analysis method constant.

Input Data In order to avoid errors due to differences in input data between different runs, the input data has to be fixed and produce a WCET execution for all our experiments. This is achieved by manual code inspection and systematically doing testruns.

Program Flow Analysis The program flow analysis is a potentially large source of errors. Our solution here is to use simple programs where the flow is easy to deduce and known (since we know the input data, that source of uncertainty is removed). We check the correctness of the program flow model and that it corresponds to the WCET execution by inspection.

Since no heavy loop optimizations were performed by the compiler, the source code and object code have the same structure, avoiding the problems with structural correspondence reported in [5, 18].

Global Low-Level Analysis Our target hardware does not have caches or branch predictors, thus, no global low-level effects will be present and no global low-level analysis is needed.

4.3. Validating the Calculation Method

For the calculation method we want to show that the use of a constrained maximation problem to model program flow finds the longest executable path through the program.

We use the fact that the maximization process generates a program execution profile: the resulting execution counts for the timing graph nodes are mapped back to the corresponding basic blocks to get a profile that is comparable to that of the trace generation.

If the execution profiles from the WCET analysis and the trace generation do not agree, there are errors in our method.

4.4. Validating the Pipeline Analysis

For the pipeline analysis we want to show that it is correct to generate a program execution time by adding execution times for timing graph nodes and short sequences of nodes.

Errors might be hidden by the calculation method, but if the execution profiles of the WCET tool and the worst-case execution agree, then any error in the execution time will be due to errors in the pipeline analysis.

4.5. Validating the Simulator

In order to get some indication of the overall quality of our approach, we still need to compare our results to execution on the real hardware. However, this is not considered a relevant test for the correctness of our algorithms due to the potential presence of errors in the simulator's hardware model.

4.6. Validating Global Low-Level Analysis

In some designs, caches are used and thus cache analysis needs to be used and validated. Cache analysis must be tested against the real hardware, and provided that the target CPU has performance counters[1], it should be possible to compare the number of cache hits and misses in a real execution with those predicted by the cache analysis.

A potential problem is that for out-of-order and speculative processors, the pipeline flow and cache effects cannot be effectively separated. However, such architectures are not very likely to appear in safety-critical embedded systems due to their inherent unpredictability.

5. Test System and Prototype Implementation

For our first implementation of a WCET tool, we have chosen to use the NEC V850E CPU [4] as our hardware platform. The V850E is a typical modern embedded 32-bit RISC CPU.

5.1. Compiler

We compile our programs using the IAR C compiler for the NEC V850E. The compiler is run with medium size optimizations, generating clean and simple code. There are two versions of the compiler, one that dumps a format suitable for our tool and one that outputs code suitable for hardware execution.

5.2. Hardware

Execution times on real hardware are obtained by running the programs on a V850E emulator. The emulator was set up to emulate a single-chip configuration with 60kB of internal RAM and 64kB of internal ROM. No I/O was used. The IAR C-Spy debugger was used to run programs on the emulator.

Note that the execution time for a program is a little "vague". The emulator reports only instruction fetches, while the simulator reports the time from first fetch to the last instruction leaving the pipeline. However, this vagueness at the edges of the programs is limited to a few cycles, and it is only for very small programs (like `fibcall`) that the error might be significant.

The times used were the time from the fetch of the first instruction in `main()` until the fetch of the first instruction after the termination of `main()`[2].

5.3. Branch Handling

Branches on the V850E have varying execution time depending on whether they are taken or not.[3]

When a sequence of nodes is executed, it is easy to determine whether a branch is taken or not – just determine whether the next node in the sequence is on the taken path of the branch or not.

In the case that an execution scenario ends with a branch and is the *last* in a sequence, we need to define a static worst case of the branch. Static inspection of instruction properties made us set the static worst case to "branch taken", since this involves more processing.

5.4. Termination Condition for the V850E

Our algorithm for pipeline analysis (see Section 3 above), requires a *termination condition* that determines when to stop generating times for longer sequences of timing graph nodes. According to [6], for CPUs like the V850E, we can stop generating longer sequences when there is no possibility for the first node in the sequence to affect the last.

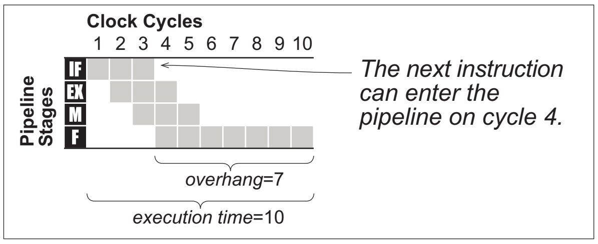

Figure 5. Pipeline overhang

The condition we use in the present tool is to stop when the number of instructions executed after the end of the first node of a sequence of nodes is greater than the *overhang* of the first node. As illustrated in Figure 5, the overhang is the number of instruction fetch slots that occur after the fetch of the last instruction in the node until all instructions from the node finished (when the node is executed in isolation).

[1] Like those present on desktop CPUs like Pentium and PowerPC.

[2] Note that before `main()` can be started, the C-startup code has to be run, and that the `exit()` function is run after the program ends to stop the execution.

[3] Note that the problem with different timing for taken and not taken branches has been avoided by most previous research in WCET analysis. In [10] and [16], processors (SPARC and MIPS R3000) that employ delay slots to mask the timing effect of taken branches are used, giving branches constant timing behavior. In [28], no attempt to model a pipeline is made and constant instruction timing (except for cache effects) is assumed.

Program	Description	Properties	Lines of Code	Executable Code Size (bytes)
`fibcall`	Simple iterative fibonacci calculation, ran to calculate fib(30).	Parameter-dependent function, single-nested loop.	22	48
`matmult`	Matrix multiplication of two 20x20 matrices.	Multiple calls to the same function, nested function calls, triple-nested loops.	81	260
`jfdctint`	Discrete-cosine transformation on a 8x8 pixel block.	Long calculation sequences, single-nested loops.	169	736
`insertsort`	Insertion sort on a reversed array	Input-data dependent nested loop with worst-case $n^2/2$ iterations	37	136
`duff`	Using "Duff's device" [24] to copy a 43 byte array	Unstructured loop with known bound, jump table for `switch` statement	32	186

Figure 6. Benchmark programs

Since we can only see that instructions get accepted in our current simulator, we cannot check the overhang directly. A safe estimation is produced by taking the time at which the last instruction was fed into the pipeline (instead of the time at which a potential instruction after the last would enter the pipeline). The resulting estimate is obviously greater than the definition of the overhang, and thus safe (but slightly pessimistic).

6. Experiments

Since any reasonable set of test programs could be used to check the correctness of our method, we let some secondary objectives affect the choice of test programs.

We want to ensure that our WCET analysis tool can handle various types of structures and flows appearing in real programs, like function calls, calling-context dependent execution times, loops (with various nesting depths), and unstructured loops. We also want to vary the code between computation-intensive and decision-intensive.

The benchmarks we have selected are listed in Figure 6. All the programs have been used by other groups [17, 19], and their worst-case behavior is known.[4]

Since we are targeting embedded systems, we had to make slight modifications to the benchmark programs: no operating system calls are allowed, which means that input data for input-dependent programs was integrated into the program, and that calls to library functions (like `printf()`) were removed.

[4]We are aware that the benchmarks program do not represent "typical embedded programs", but that is not relevant for the present purpose of proving the pipeline analysis. It would be relevant if we wanted to demonstrate the precision achievable on real programs for an end-to-end WCET method.

Program	Analysis Default	Analysis Opt	Sim	Real
`fibcall`	287	286	286	312
`matmult`	239528	239528	239528	222236
`jfdctint`	5550	5550	5550	4843
`insertsort`	2077	1249	1249	1080
`duff`	-	1226	1226	1081

Figure 7. Measured Execution Times

We have run our five test programs through our WCET tool, through trace generation and simulation, and on real hardware. The results of the timing measurements are shown in Figure 7. All times are measured in clock cycles. For the WCET tool analysis results, we have given two values: one obtained using simple loop bounds for all loops (i.e. just the maximum number of loop iterations), and an "optimized" estimate where we manually added information about infeasible paths to the program-describing constraint system. For `duff` we had to do optimized modelling only, since simple loop bounds are inapplicable to unstructured code.

We use the "optimized" estimates for the validation, since they are the results that correspond to the real power of our model. Using just the simple loop bounds would have been an artificial restriction that does not take full advantage of the power of our program flow modeling.

6.1. Results for the Calculation Method

For the optimized program flow models, the profiles generated by the WCET tool and the real execution match exactly (not shown in any table, since that would be a rather boring long list of numbers).

This indicates that the constrained maximization does find the worst-case path, and that it does not push the execution counts beyond the actual worst case. Thus, we consider the IPET-based modeling to be valid.

6.2. Results for the Pipeline Analysis

Since the execution profiles match, the agreement in execution time between the WCET tool and the trace-driven simulation indicate that the pipeline analysis method is correct.

It should be noted that we do not expect to need to model very long sequences of nodes in order to correctly handle pipeline overlap. For the V850E, we never needed sequences longer than two nodes.

There is also no overestimation, which indicates the ability of our method to obtain tight execution time analysis.

6.3. Simulation vs. Hardware

Compared to the hardware, the simulated execution times for four of the programs are between 5 and 15% greater. Thus, the simulator usually overestimates the execution time. Interestingly, for `fibcall` the simulated time is *less* than the hardware execution time.

Our preliminary analysis is that most of the overestimating timing errors are due to a dual-issue optimization in the V850E that we have not modelled. The underestimation in `fibcall` is probably related to branch handling.

There are many reasons why time measured on a machine model could deviate from times measured on the real machine:

- The hardware manuals may contain errors or be vague on certain issues. This could be because of intentional simplification, unintentional errors, or intentional errors or vagueness (in order to protect smart designs from being copied by competitors).
- The implementation of the machine model might contain bugs.
- The hardware implementation might deviate from the design. There can be bugs in the hardware that cause the timing to deviate from the manuals, even if the manuals are "correct" visavi the design of the hardware.

Given these problems, Petters and Färber [22] conclude that a hardware model is inherently impossible to construct, at least for complex CPUs of the Pentium II class. However, we see several reasons why hardware models (simulators) are unavoidable and desirable for embedded real-time systems work:

- Simulators are a mandatory part of embedded development environments. There will be hardware models available, and the trend is towards cycle-accurate simulators, which can be used for WCET analysis.
- Timing small pieces of a program is very difficult on hardware, but easy on a simulator.
- Hardware measurement requires specialized hardware and complex system setups.
- The hardware might not be available until late in the development process. Working with a model of the hardware, verification work can start early (assuming that the model can be built from preverified components).

We consider the development and validation of the simulator to be a separate problem, since the simulator is a separate component in the WCET tool.

6.4. Complexity of Our Approach

There have been some concern that the IPET approach in general and our approach in particular has the potential to be computationally very expensive. Specific worrisome points have been:

- The constraint solving computation time required to calculate the execution time of a program is potentially exponential.
- The constraint system could get unmanageably large, even if the solution method as such is efficient.
- The simulation of long sequences might only terminate after a large number of very long sequences have been run.

Figure 8 shows some complexity measures for our benchmark programs. The columns give the number of basic blocks in the program code, the number of nodes in the timing graph, the number of execution count variables in the constraint system, and the number of sequences run in the pipeline analysis.

The number of sequences executed is usually quite low, but explodes for the `duff` program[5]. We should define a more efficient termination condition for the V850E. However, the run time is still not very long, at most a minute on a Pentium III/700 Mhz.

The number of variables does not seem to explode with the program size, but is rather linear in the num-

[5] This is due to one node having a very long pipeline overhang: a multiple-pop instruction that is fetched and then executes for 15 more cycles.

Program	BBs	Nodes	Vars	Seq Runs
fibcall	7	7	15	42
matmult	27	36	79	310
jfdctint	14	14	29	67
insertsort	7	7	15	36
duff	18	18	45	580

Figure 8. Complexities

ber of timing graph nodes.

The run time for the constraint solver was never a problem – a matter of seconds even on a rather slow UltraSparc-I machine. This is consistent with the results of other groups that only use linear constraints to model programs [23, 28]. The groups that have reported unmanageable execution times for IPET analysis have used more complex non-linear constraints [15, 21].

7. Conclusions

In this paper, we have dealt with the issues involved in validating a WCET analysis method. For safety-critical systems development, it is necessary to validate WCET analysis methods.

We have demonstrated how WCET analysis can be decomposed into a set of components, each of which should be validated in isolation. Only by composing well-tested and safe components is it possible to build a reliable WCET tool.

We have presented a testing methodology for isolating the pipeline analysis and calculation method from the program flow analysis, cache analysis, and hardware modelling (CPU simulator). We have applied this methodology to our previously published WCET algorithm [6], and given evidence that the pipeline analysis and calculation method are safe and tight.

In order to achieve the isolation, we carefully controlled the test programs and the associated flow analysis. By comparing a trace-driven simulation of the actual worst-case of a program with the output of the WCET tool, we removed the effect of any errors in the machine model. It is not necessary to use real hardware for the algorithm validation.

Given that we have a validated pipeline analysis and calculation method, we can add cache analysis and other global low-level analyses to our tool. Each such analysis will have to be tested in isolation, before incorporation into the tool (as discussed above).

Since our pipeline analysis method is shown to be correct in isolation, we can change the simulator used to retarget the system to another CPU. This does not affect the correctness results for the pipeline analysis as such. If the new simulator uses a correct hardware model, the resulting new analysis tool will also be correct.

Regarding program features, we have demonstrated that our modelling and calculation method handles function calls, loop bounds dependent on function call argument, nested loops, multi-way branches and unstructured code. Thus, our approach is capable of handling real-world programs.

8. Future Work

In this paper, we have dealt with a simple non-cached pipelined CPU. We plan to extend the WCET analysis method to include cache- and branch prediction analysis.

On the flow analysis side, we plan deeper investigations on how to express path constraints to keep the WCET analysis tight even for programs with complex program flow. This is a necessary step on the way to fully automated program flow analysis.

Furthermore, we would like to compare the IPET-based calculation method with a path-based approach like e.g. [26].

We have a Master's Thesis project underway that aims to remove the errors in our V850E simulator by systematic comparison between the simulator and the hardware. Since the CPU Simulator is designed to be generic, it should be easy to adapt it to different processors.

We plan to integrate the WCET tool into the software synthesis environment CHaRy [1], and specialize the WCET tool for other embedded CPUs.

Since our pipeline analysis is designed to be easy to port to new simulators, we can easily port our WCET analysis tool to new hardware architectures, as long as an there exists cycle-correct simulators. The WCET analysis and the simulator can then be tested and verified in isolation as outlined in this article.

The long term goal is to integrate a WCET analysis tool into the IAR Embedded Workbench integrated development environment, and provide WCET analysis as a standard and accessible tool for embedded systems developers.

Acknowledgements: The authors would like to thank NEC Europe for providing us with the hardware we needed to perform our experiments, and we would like to extend a special "thank you" to Alfred Hoeltermann at NEC for his helpful answers to our questions about the behavior of the hardware.

IAR Systems has provided the opportunity to work with the internals of their compilers and other development tools.

We would also like to thank Peter Altenbernd, Hans Hansson and Mikael Sjödin for their fruitful comments on drafts of this article.

References

[1] P. Altenbernd. CHaRy: The C-Lab Hard Real-Time System to Support Mechatronical Design. In *Proceedings IEEE International Symposium and Workshop on Systems Engineering of Computer Based Systems*. IEEE Computer Society Press, 1997.

[2] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding Instruction Cache Effects to Schedulability Analysis of Preemptive Real-Time Systems. In *Proc. 2^{nd} IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pages 204–212. IEEE Computer Society Press, June 1996.

[3] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Journal of Real-Time Systems*, May 2000.

[4] NEC Corporation. *V850E/MS1 32/16-bit Single Chip Microcontroller: Architecture*, 3^{rd} edition, January 1999. Document no. U12197EJ3V0UM00.

[5] J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution times analysis for optimized code. In *Proc. of the 10^{th} Euromicro Workshop of Real-Time Systems*, pages 146–153, June 1998.

[6] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proc. 6^{th} International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, December 1999.

[7] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Towards industry-strength worst case execution time analysis. Technical Report ASTEC 99/02, Advanced Software Technology Center (ASTEC), April 1999.

[8] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Proc. Euro-Par'97 Parallel Processing, LNCS 1300*, pages 1298–1307. Springer Verlag, August 1997.

[9] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[10] C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), January 1999.

[11] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, May 2000.

[12] Intelligent Systems Laboratory. SICStus Prolog user's manual. ISBN 91-630-3648-7, Swedish Institute of Computer Science, 1995.

[13] S.-K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Proc. of RTAS'96*, pages 230–240. IEEE, 1996.

[14] C. Lee, J. Han, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling. In *Proc. 17^{th} IEEE Real-Time Systems Symposium (RTSS'96)*, December 1996.

[15] Y-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. of the 32:nd Design Automation Conference*, pages 456–461, 1995.

[16] S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An accurate worst-case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[17] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A worst case timing analysis technique for multiple-issue machines. In *Proc. 19^{th} IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998.

[18] S-S. Lim, J. Kim, and S. L. Min. A worst case timing analysis technique for optimized programs. In *Proc. of the fifth International Conference on Real-Time Computing Systems and Applications (RTCSA); Hiroshima, Japan*, pages 151–157, Oct 1998.

[19] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Real-Time Systems*, May 2000.

[20] F. Müller. Timing predictions for multi-level caches. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, pages 29–36, Jun 1997.

[21] G. Ottosson and M. Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, June 1997.

[22] S. Petters and G. Färber. Making worst-case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Proc. 6^{th} International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, December 1999.

[23] P. Puschner and A. Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technische Universität, Institut für Technische Informatik, Wien, April 1995.

[24] E. Raymond. The jargon file, version 4.2.0. `http://www.tuxedo.org/~esr/jargon/html/index.html`, February 2000.

[25] J. Schneider and C. Ferdinand. Pipeline behaviour prediction for superscalar processors by abstract interpretation. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*. ACM, May 1999.

[26] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[27] IAR Systems. *V850 C/EC++ Compiler Programming Guide*, 1^{st} edition, January 1999.

[28] H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proc. 19^{th} IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998.

[29] R. White, F. Müller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *Proc. 3^{rd} IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, pages 192–202, June 1997.