Static Memory and Timing Analysis of Embedded Systems Code

Christian Ferdinand *Reinhold Heckmann* *Bärbel Franzen*

AbsInt Angewandte Informatik GmbH

Science Park 1, D-66123 Saarbrücken, Germany

Phone: +49-681-38360-0 e-mail: info@absint.com

Fax: +49-681-38360-20 Web page: http://www.absint.com

Abstract

Failure of a safety-critical application on an embedded processor can lead to severe damage or even loss of life. Here we are concerned with two kinds of failure: stack overflow, which usually leads to run-time errors that are difficult to diagnose, and failure to meet deadlines, which is catastrophic for systems with hard real-time characteristics. Classical validation methods like code review and testing with repeated measurements require a lot of effort, are expensive, and do not really help in proving the absence of such errors. **AbsInt**'s tools **StackAnalyzer** and **aiT** (timing analyzer) provide a solution to this problem. They use abstract interpretation as a formal method that allows to obtain statements valid for all program runs with all inputs.

1 Introduction

The use of safety-critical embedded software in the automotive and avionics industries is increasing rapidly. Failure of such a safety-critical embedded system may result in the loss of life or in large damages. Also for non-safety-critical applications, software failure may necessitate expensive updates. Therefore, utmost carefulness and state-of-the-art machinery have to be applied to make sure that an application meets all requirements. To do so lies in the responsibility of the system designer(s).

Traditional certification standards evaluate the quality of a software system by assessing the quality of the development process that produced it. Yet the benefit of such a process-based assurance is limited. Therefore, switching to a product-based assurance process is advised, which judges the quality of a software product by examining its properties.

Classical software validation methods like code review and testing with debugging are very expensive and cannot really guarantee the absence of errors. Formal verification methods provide an alternative, in particular for safety-critical applications. One such method is *abstract interpretation* [2], which allows to obtain statements that are valid for all program runs with all inputs. Such statements may be absence of violations of timing or space constraints, or absence of runtime errors. For example, stack overflow can be detected by **AbsInt**'s **StackAnalyzer**, and violations of timing constraints are found by **AbsInt**'s **aiT** tool [5] that determines upper bounds for the worst-case execution times of the tasks of an application. Among other things, these tools perform a *value analysis* as the principal source of information about the values manipulated by the analyzed program.

2 Value Analysis

Value analysis tries to determine the values stored in the processor's memory for every program point and execution context. Often, it is sufficient to restrict value analysis to the processor registers, but sometimes, it is useful to get information about main memory as well.

Value analysis is a static analysis method producing results valid for every program run and all inputs to the program. Therefore, it cannot always predict an exact value for a memory location, but determines *abstract values* instead that stand for sets of concrete values. More precisely, it computes for each program point and execution context an *abstract state* that maps memory locations to abstract values. Each machine instruction is modeled by a transfer function mapping input states to output states in a way that is compatible with the semantics of the instruction. At control-flow joins, the incoming abstract states are combined into a single outgoing state using a combination function. Because of the presence of loops, transfer and combination functions must be applied repeatedly until the system of abstract states stabilizes. Termination of this fixed-point iteration is ensured on a theoretical level by the monotonicity of transfer and combination functions and the fact that a memory location can only hold finitely many different values. Practically, value analysis becomes only efficient by applying suitable widening and narrowing operators as proposed in [2].

There are several variants of value analysis depending on what kinds of abstract values are used. The simplest form of value analysis is *constant propagation*: an abstract value is either a single concrete value or the statement that no information about the value is known. A more elaborate form of value analysis computes safe lower and upper bounds for the possible concrete values, i.e. abstract values are intervals that are guaranteed to contain the exact values.

All the value analysis variants described above determine sets of possible values for individual memory locations without any relationship between these sets. Yet if an unknown value is moved from register `r1` to `r2`, then both registers contain unknown values afterward, but these values are known to be equal. So a possible extension of value analysis may record known equalities between values, or more generally, upper and lower bounds for their differences, or even more generally, arbitrary linear constraints between values.

Value analysis, even in its simple form as interval analysis, has various applications as an auxiliary method providing input for other analysis tasks. Some of these applications are listed in the sequel.

3 Stack Usage Analysis

A possible cause of catastrophic failure is stack overflow that usually leads to run-time errors that are difficult to diagnose. The problem is that the memory area for the stack usually must be reserved by the programmer. Underestimation of the maximum stack usage leads to stack overflow, while overestimation means wasting memory resources. Measuring the maximum stack usage with a debugger is no solution since one only obtains a result for a single program run with fixed input. Even repeated measurements with various inputs cannot guarantee that the maximum stack usage is ever observed. Some, but not all compilers provide information about stack usage, but this requires the availability of the source code, and the information becomes invalid when the generated code is optimized by hand or by some automatic tool.

AbsInt's tool **StackAnalyzer** provides a solution to this problem: By concentrating on the value of the stack pointer during value analysis, the tool can figure out how the stack increases and decreases along the various control-flow paths. This information can be used to derive the maximum stack usage of the entire task.

The results of **StackAnalyzer** are presented as annotations in a combined call graph and control-flow graph. Figure 1 shows the call graph of a small application, with stack analysis results at routines and for the entire application (at the top). On this level, the results of stack analysis are displayed in boxes located to the right of the boxes representing the routines of the application. Each result box carries two results: a *global result*, coming first, and a *local result*, following in angular brackets. Each result is an interval of possible stack levels. Intervals of the form $[n,n]$ are abbreviated to n.

The local result at a routine R indicates the stack usage in R considered on its own: It is an interval showing the possible range of stack levels within the routine, assuming value 0 at routine entry. The local result for a routine is derived from the results at individual instructions, which are shown in Figure 2 for one of the routines of this example.

The global result for routine R indicates the stack usage of R in the context of the entire application. It is an interval providing bounds for the stack level while the processor is executing instructions of R, for all call

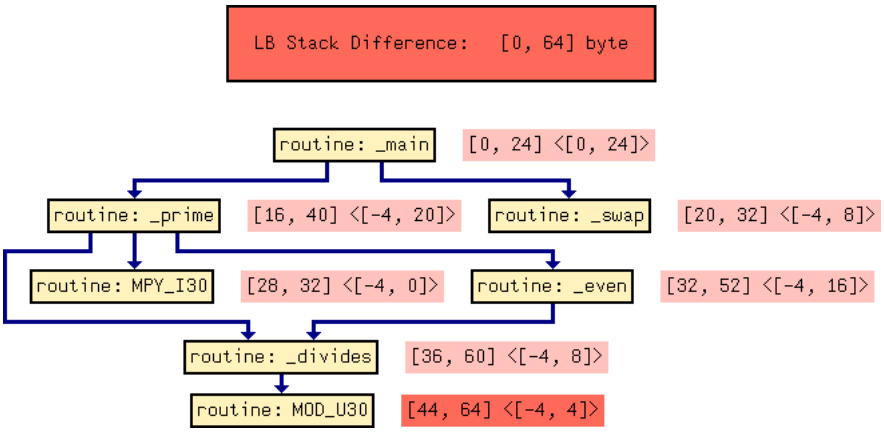

Figure 1: Call graph with stack analysis results

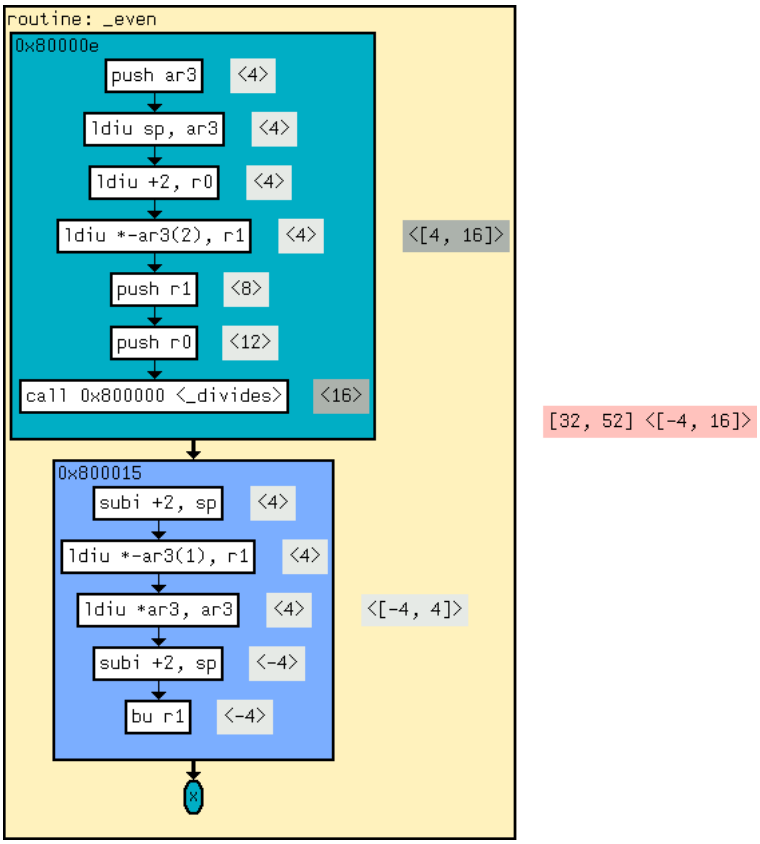

Figure 2: Individual instructions with stack analysis results

paths from the entry point to *R*. Thus, the global result at routine *R* does not include the stack usage of the routines called by *R*.

StackAnalyzer provides automatic tool support to calculate precise information on the stack usage. This not only reduces development effort, but also helps to prevent runtime errors due to stack overflow. Critical program sections are easily recognized thanks to color coding. The analysis results thus provide valuable feedback in optimizing the stack usage of an application. The predicted worst-case stack usages of individual tasks in a system can be used in an automated overall stack usage analysis for all tasks running on one Electronic Control Unit, as described in [7] for systems managed by an OSEK/VDX real-time operating system.

4 WCET Analysis: Worst-Case Execution Time Prediction

Many tasks in safety-critical embedded systems have hard real-time characteristics. Failure to meet deadlines may be as harmful as producing wrong output or failure to work at all. Yet the determination of the Worst-Case Execution Time (WCET) of a task is a difficult problem because of the characteristics of modern software and hardware [17].

Embedded control software (e.g., in the automotive industries) tends to be large and complex. The software in a single electronic control unit typically has to provide different kinds of functionality. It is usually developed by several people, several groups or even several different providers. Code generator tools are widely used. They usually hide implementation details to the developers and make an understanding of the timing behavior of the code more difficult. The code is typically combined with third party software such as real-time operating systems and/or communication libraries.

Concerning hardware, there is typically a large gap between the cycle times of modern microprocessors and the access times of main memory. Caches and branch target buffers are used to overcome this gap in virtually all performance-oriented processors (including high-performance micro-controllers and DSPs). Pipelines enable acceleration by overlapping the executions of different instructions. Consequently the execution behavior of the instructions cannot be analyzed separately since it depends on the execution history.

Cache memories usually work very well, but under some circumstances minimal changes in the program code or program input may lead to dramatic changes in cache behavior. For (hard) real-time systems, this is undesirable and possibly even hazardous. Making the safe yet—for the most part—unrealistic assumption that all memory references lead to cache misses results in the execution time being overestimated by several hundred percent.

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 has impact on the cache behavior. Hardware simulation, emulation, or direct measurement with logic analyzers can only determine the execution time for one input. They cannot be used to infer the execution times for all possible inputs in general.

Furthermore, the execution time depends on the processor state in which the execution is started. Modern processor architectures often violate implicit assumptions on the worst start state. The reason is that they exhibit timing anomalies as defined in [9], which consist of a locally advantageous situation, e.g., a cache hit, resulting in a globally larger execution time. As also demonstrated in [9], processor pipelines may exhibit so-called domino effects where—for some special pieces of code—the difference between two start states of the pipeline does not disappear over time, but leads to a difference in execution time that cannot be bounded by a constant.

4.1 Structure of WCET Computation

Abstract interpretation can be used to efficiently compute a safe approximation for all possible cache and pipeline states that can occur at a program point. These results can be combined with ILP (Integer Linear Programming) techniques to safely predict the worst-case execution time and a corresponding worst-case execution path. This approach can help to overcome the challenges listed above.

AbsInt's WCET tool **aiT** determines the WCET of a program task in several phases [5] (see Figure 3):

- **CFG Building** decodes, i.e. identifies instructions, and reconstructs the control-flow graph (CFG) from a binary program;
- **Value Analysis** computes value ranges for registers and memory cells, and address ranges for instructions accessing memory;
- **Loop Bound Analysis** determines upper bounds for the number of iterations of simple loops;
- **Cache Analysis** classifies memory references as cache misses or hits [4];
- **Pipeline Analysis** predicts the behavior of the program on the processor pipeline [8];
- **Path Analysis** determines a worst-case execution path of the program [16].

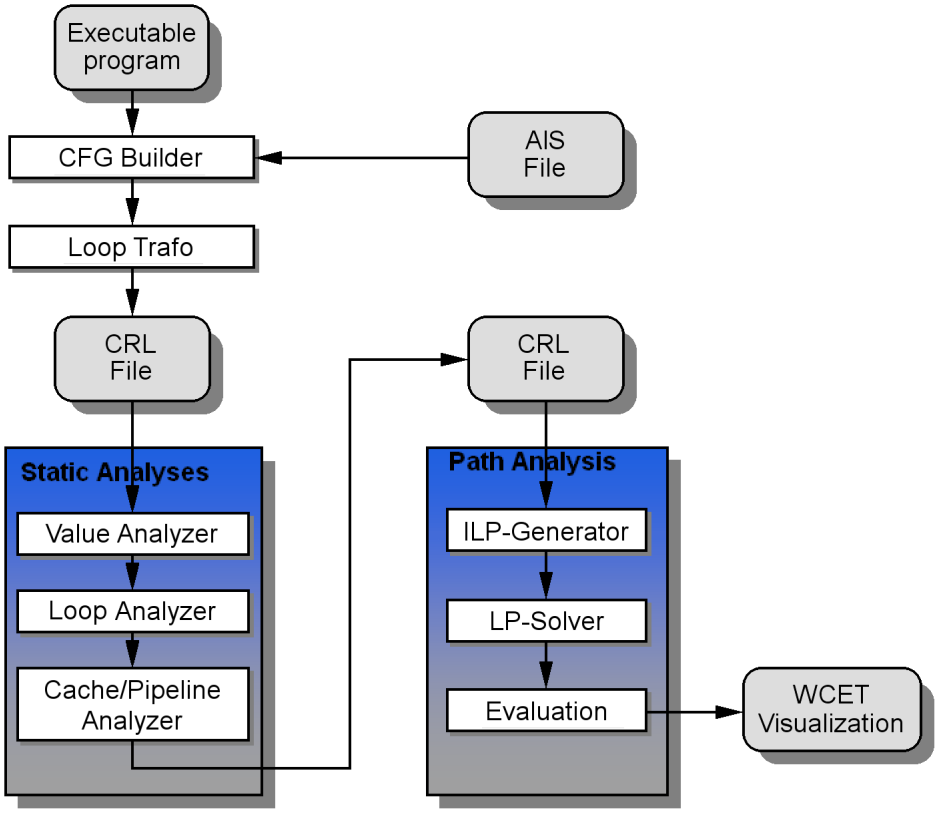

Figure 3: Phases of WCET computation

Separating WCET determination into several phases makes it possible to use different methods tailored to the subtasks [16]. Value analysis, cache analysis, and pipeline analysis are based on abstract interpretation [2]. Integer linear programming is used for path analysis.

aiT allows to inspect the timing behavior of (time-critical parts of) program tasks. The analysis results are determined without the need to change the code and hold for all executions (for the intrinsic cache and pipeline behavior). **aiT** takes into account the combination of all the different hardware characteristics while still obtaining tight upper bounds for the WCET of a given program in reasonable time.

4.2 Reconstruction of the Control Flow from Binary Programs

The starting point of our analysis framework (see Figure 3) is a binary program and a so-called *AIS file* containing additional user-provided information about numbers of loop iterations, upper bounds for recursion, etc. In the first step a decoder reads the executable and reconstructs the control flow [13, 14]. 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 micro-architecture analysis.

The decoder can find the target addresses of absolute and `pc`-relative calls and branches, but may have difficulties with target addresses computed from register contents. Thus, **aiT** uses specialized decoders that are adapted to certain code generators and/or compilers. They usually can recognize branches to a previously stored return address, and know the typical compiler-generated patterns of branches via switch tables. Yet non-trivial applications may still contain some computed calls and branches (in hand-written assembly code) that cannot be resolved by the decoder; these unresolved computed calls and branches are documented by appropriate messages and require user annotations. Such annotations may list the possible targets of computed

calls and branches, or tell the decoder about the address and format of an array of function pointers or a switch table used in the computed call or branch.

4.3 Value Analysis

Value analysis as described in Section 2 tries to determine the values in the processor memory for every program point and execution context. Its results are used to determine possible addresses of indirect memory accesses—important for cache analysis—and in loop bound analysis. They are usually so good that only a few indirect accesses cannot be determined exactly. Address ranges for these accesses may be provided by user annotations.

4.4 Loop Bound Analysis

WCET analysis requires that upper bounds for the iteration numbers of all loops be known. **aiT** tries to determine the number of loop iterations by *loop bound analysis*, but succeeds in doing so for simple loops only. Bounds for the iteration numbers of the remaining loops must be provided as user annotations.

aiT employs two different methods for loop bound analysis. The older method relies on a combination of value analysis and pattern matching, which looks for typical loop patterns. In general, these loop patterns depend on the code generator and/or compiler used and sometimes even on the optimization level.

The newer method described in [3] uses an interprocedural data-flow analysis to derive loop invariants from the semantics of the instructions. This new analysis does not depend on the compiler used or optimization level, but only on the semantics of the instruction set for the target machine. It is able to handle loops with multiple exits and multiple modifications of the loop counter per iteration including modifications in procedures called from the loop.

4.5 Cache Analysis

Cache analysis classifies the accesses to main memory. The analysis in our tool is based upon [4], which handles analysis of caches with LRU (Least Recently Used) replacement strategy. However, it had to be modified to reflect the non-LRU replacement strategies of common microprocessors: the pseudo-round-robin replacement policy of the ColdFire MCF 5307, and the PLRU (Pseudo-LRU) strategy of the PowerPC MPC 750 and 755. The modified algorithms distinguish between sure cache hits and unclassified accesses. The deviation from perfect LRU is the reason for the reduced predictability of the cache contents in case of ColdFire 5307 and PowerPC 750/755 compared to processors with perfect LRU caches [6].

4.6 Pipeline Analysis

Pipeline analysis models the pipeline behavior to determine execution times for sequential flows (basic blocks) of instructions, as done in [11]. 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.

Like value and cache analysis, pipeline analysis is based on the framework of abstract interpretation. Pipeline analysis of a basic block starts with a set of pipeline states determined by the predecessors of the block and lets this set evolve from instruction to instruction by a kind of cycle-wise simulation of machine instructions. In contrast to a real simulation, the abstract execution on the instruction level is in general non-deterministic since information determining the evolution of the execution state is missing, e.g., due to non-predictable cache contents. Therefore, the abstract execution of an instruction may cause a state to split into several successor states. All the states computed in such tree-like structures form the set of entry states for the successor instruction. At the end of the basic block, the final set of states is propagated to the successor

blocks. The described evolution of state sets is repeated for all basic blocks until it stabilizes, i.e. the state sets do not change any more.

The output of 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 path analysis to obtain the WCET for the entire task.

4.7 Path Analysis

Using the results of the micro-architecture analyses, path analysis determines a safe estimate of the WCET. The program's control flow is modeled by an integer linear program [16, 15] so that the solution to the objective function is the predicted worst-case execution time for the input program. A special mapping of variable names to basic blocks in the integer linear program enables execution and traversal counts for every basic block and edge to be computed.

4.8 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.

Frequently the first execution of the loop body loads the cache, and subsequent executions find most of their referenced memory blocks in the cache. Because of speculative prefetching, cache contents may still change considerably during the second iteration. Therefore, the first few iterations of the loop often encounter cache contents quite different from those of later iterations. Hence it is useful to distinguish the first few iterations of loops from the others. This is done in the VIVU approach (virtual inlining, virtual unrolling) [10].

Using upper bounds on the number of loop iterations, the analyses can virtually unroll not only the first few iterations, but all iterations. The analyses can then distinguish more contexts and the precision of the results is increased—at the expense of higher analysis times.

4.9 Usage of aiT

The techniques described above have been incorporated into **AbsInt**'s **aiT** WCET analyzer tools. They get as input an executable, user annotations, a description of the (external) memories and buses (i.e. a list of memory areas with minimal and maximal access times), and a task (identified by a start address). A task denotes a sequentially executed piece of code (no threads, no parallelism, and no waiting for external events). This should not be confused with a task in an operating system that might include code for synchronization or communication.

The WCET analyzers compute an upper bound of the runtime of the task (assuming no interference from the outside). Effects of interrupts, IO and timer (co-)processors are not reflected in the predicted runtime and have to be considered separately (e.g., by a quantitative analysis).

The task WCETs predicted by **aiT** can be used to determine an appropriate scheduling scheme for the tasks and to perform an overall schedulability analysis in order to guarantee that the application meets all timing constraints (also called *timing validation*) [12]. Some real-time operating systems offer tools for schedulability analysis, but all these tools require the WCETs of tasks as input.

4.10 Visualization of aiT's Results

aiT's results are written into a report file. In addition, **aiT** produces a picture description that can be visualized by the **aiSee** tool [1] to view detailed information delivered by the analysis.

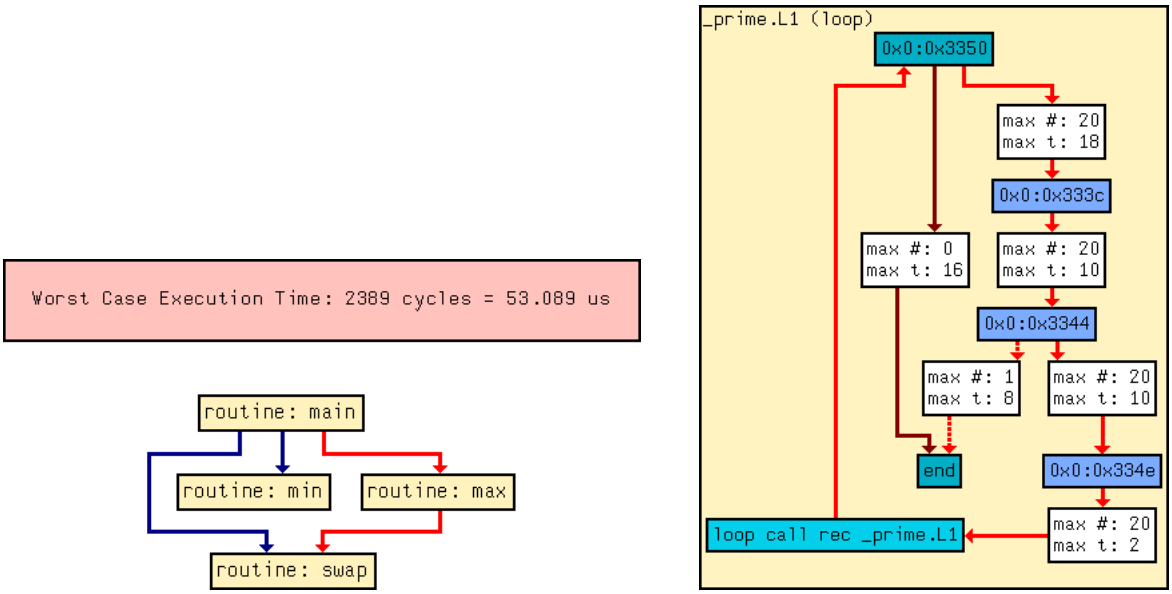

Figure 4: Call graph and control-flow graph with WCET results

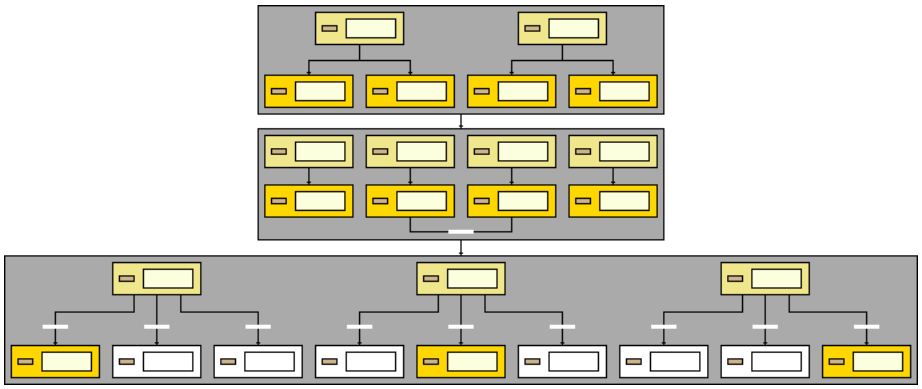

Figure 5: Possible pipeline states in a basic block

Figure 4, left, shows the graphical representation of the call graph for some small example. The calls (edges) that contribute to the worst-case runtime are marked by the color red. The computed WCET is given in CPU cycles and in microseconds provided that the cycle time of the processor has been specified.

Figure 4, right, shows the basic block graph of a loop. The number `max #` describes the maximal number of traversals of an edge in the worst case, while `max t` describes the maximal execution time of the basic block from which the edge originates (taking into account that the basic block is left via the edge). The worst-case path, the iteration numbers and timings are determined automatically by **aiT**.

Figure 5 shows the possible pipeline states for a basic block in this example. Such pictures are shown by **aiT** upon special demand. The large dark grey boxes correspond to the instructions of the basic block, and the smaller rectangles in them stand for individual pipeline states. Their cyclewise evolution is indicated by the strokes connecting them. Each layer in the trees corresponds to one CPU cycle. Branches in the trees are caused by conditions that could not be statically evaluated, e.g., a memory access with unknown address in presence of memory areas with different access times. On the other hand, two pipeline states fall together when details they differ in leave the pipeline. This happened for instance at the end of the second instruction, reducing the number of states from four to three.

Figure 6 shows the top left pipeline state from Fig. 5 in greater magnification. It displays a diagram of the architecture of the CPU (in this case a PowerPC 555) showing the occupancy of the various pipeline stages with the instructions currently being executed.

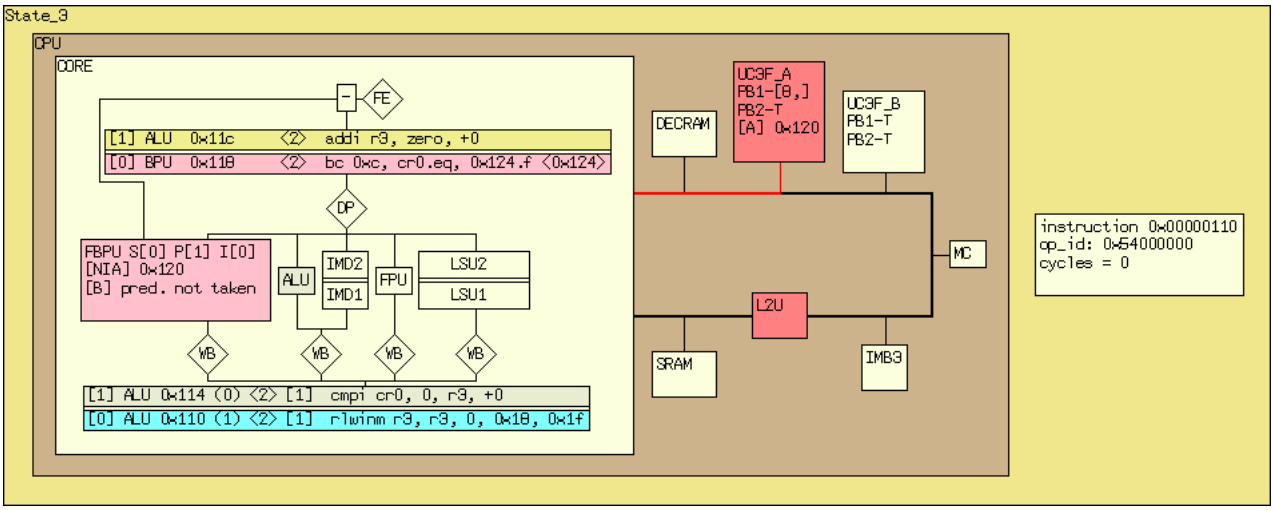

Figure 6: Individual pipeline state

5 Dependence on Target Architectures

There are **aiT** versions for PowerPC MPC 555, 565, and 755, ColdFire 5307, ARM7 TDMI, HCS12/STAR12, TMS320C33, C166/ST10, Renesas M32C/85, and Tricore 1.3.

Decoders are automatically generated from processor specifications defining instruction formats and operand meaning. The CRL format used for describing control-flow graphs is machine-independent. *Value Analysis* must interpret the operations of the target processor. Hence, there is a separate value analyzer for each target, but features shared by many processors (e.g., branches based on condition bits) allowed for considerable code sharing among the various value analyzers.

There is only one cache analyzer with a fixed interface to pipeline analysis. It is parameterized on cache size, line size, associativity, and replacement strategy. Each replacement strategy supported by **aiT** is implemented by a table for line age updates that is interpreted by the cache analyzer.

The pipeline analyzers are the most diverse part of **aiT**. The supported target architectures are grouped according to the complexity of the processor pipeline. For each group a common conceptual and coding framework for pipeline analysis has been established, in which the actual target-dependent analysis must be filled in by manual coding.

6 Precision of aiT

Since the real WCET is not known for typical real-life applications, statements about the precision of **aiT** are hard to obtain. For an automotive application running on MPC 555, one of **AbsInt**'s customers has observed an overestimation of 5–10% when comparing **aiT**'s results and the highest execution times observed in a series of measurements (which may have missed the real WCET). For an avionics application running on MPC 755, Airbus has noted that **aiT**'s WCET for a task typically is about 25% higher than some measured execution times for the same task, the real but non-calculable WCET being in between. Measurements at **AbsInt** have indicated overestimations ranging from 0% (cycle-exact prediction) till 10% for a set of small programs running on M32C, TMS320C33, and C16x/ST10. Table 1 shows the results for C166. The analysis times were moderate—a few seconds till about 3 minutes for edn.

Table 1: Precision of **aiT** for some C166 programs

Example		from external RAM			from flash		
Program	Size	measured cycles	predicted cycles	over-estimation	measured cycles	predicted cycles	over-estimation
fac	2.9k	949	960	1.16 %	810	832	2.72 %
fibo	3.4k	2368	2498	5.49 %	2142	2228	4.01 %
coverc1	16k	5670	5672	0.04 %	3866	4104	6.16 %
coverc	4.3k	7279	7281	0.03 %	5820	6202	6.56 %
morswi	5.9k	17327	17332	0.03 %	8338	8350	0.14 %
coverc2	24k	18031	18178	0.82 %	12948	14054	8.54 %
swi	24k	18142	18272	0.72 %	13330	14640	9.83 %
edn	13k	262999	267643	1.77 %	239662	241864	0.92 %

7 Conclusion

Tools based on abstract interpretation can perform static program analysis of embedded applications. Their results hold for all program runs with arbitrary inputs. Employing static analyzers is thus orthogonal to classical testing, which yields very precise results, but only for selected program runs with specific inputs.

aiT allows to inspect the timing behavior of (time-critical parts of) program tasks. It takes into account the combination of all the different hardware characteristics while still obtaining tight upper bounds for the WCET of a given program in reasonable time. **StackAnalyzer** and **aiT** are used among others by Airbus in the development of various safety-critical applications for the A380. They will be used as verification tools in the sense of DO178b. The qualification requirements for such verification tools are lighter than for code generation tools. In contrast to code generation tools, verification tools cannot introduce any errors into the safety-critical system. Failure of a verification tool may only lead to an overlooked error.

In a less regulated application area, **StackAnalyzer** and **aiT** can be used for optimizing the system. For instance, the results of **StackAnalyzer** are useful when optimizing the assignment of priorities to tasks in order to minimize the memory consumption in an OSEK-like operating system. The results of **aiT** can be used to find an optimal schedule in a time-triggered operating system.

The usage of static analyzers enables one to develop complex systems on state-of-the-art hardware, increases safety, and saves development time. Precise stack usage and timing predictions enable the most cost-efficient hardware to be chosen.

References

[1] AbsInt Angewandte Informatik GmbH. *aiSee Home Page*. http://www.aisee.com, 2006.

[2] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, Los Angeles, California, 1977.

[3] Christoph Cullmann. Statische Berechnung sicherer Schleifengrenzen auf Maschinencode. Master's thesis, Universität des Saarlandes, 2006.

[4] Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[5] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of EMSOFT 2001, First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer-Verlag, 2001.

[6] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, July 2003. Special Issue on Real-Time Systems.

[7] Winfried Janz. Das OSEK Echtzeitbetriebssystem, Stackverwaltung und statische Stackbedarfsanalyse. In *Embedded World*, Nuremberg, Germany, February 2003.

[8] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann. Pipeline modeling for timing analysis. In *Proceedings of the 9th International Static Analysis Symposium SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 294–309. Springer-Verlag, 2002.

[9] Thomas Lundquist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, December 1999.

[10] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In Kai Koskimies, editor, *Proceedings of the International Conference on Compiler Construction (CC'98)*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94. Springer-Verlag, March /April 1998.

[11] Jörn Schneider and Christian Ferdinand. Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 34, pages 35–44, May 1999.

[12] John A. Stankovic. *Real-Time and Embedded Systems*. ACM 50th Anniversary Report on Real-Time Computing Research, 1996. http://www-ccs.cs.umass.edu/sdcr/rt.ps.

[13] Henrik Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing Systems and Applications*, Cheju Island, South Korea, 2000.

[14] Henrik Theiling. Generating Decision Trees for Decoding Binaries. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 112–120, Snowbird, Utah, USA, June 2001.

[15] Henrik Theiling. ILP-based interprocedural path analysis. In Alberto L. Sangiovanni-Vincentelli and Joseph Sifakis, editors, *Proceedings of EMSOFT 2002, Second International Conference on Embedded Software*, volume 2491 of *Lecture Notes in Computer Science*, pages 349–363. Springer-Verlag, 2002.

[16] Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, Madrid, Spain, December 1998.

[17] Reinhard Wilhelm. Determining bounds on execution times. In R. Zurawski, editor, *Handbook on Embedded Systems*, pages 14–1 – 14–23. CRC Press, 2005.