aiT: WORST-CASE EXECUTION TIME PREDICTION BY STATIC PROGRAM ANALYSIS

Christian Ferdinand and Reinhold Heckmann
AbsInt Angewandte Informatik GmbH,
Stuhlsatzenhausweg 69, D-66123 Saarbrucken, Germany
info@absint.com http://www.absint.com

1. Introduction

Many tasks in safety-critical embedded systems have hard real-time characteristics. Failure to meet deadlines may result in the loss of life or in large damages. Utmost carefulness and state-of-the-art machinery have to be applied to make sure that all requirements are met. To do so lies in the responsibility of the system designer(s). Fortunately, the state of the art in deriving run-time guarantees for real-time systems has progressed so much that tools based on sound methods are commercially available and have proved their usability in industrial practice.

AbsInt's WCET Analyzer **aiT** (`http://www.absint.de/wcet.htm`) is the first automatic tool for checking the correct timing behavior of software in safety-critical embedded systems as found in the aeronautics and automotive industries. To compute automatically upper bounds for the worst-case execution time (WCET), **aiT** first derives safe upper bounds for the execution times of basic blocks and then computes, by integer linear programming, an upper bound on the execution times over all possible paths of the program. These upper bounds are *valid* for all inputs and each task execution, and usually *tight,* i.e. the overestimation of the WCET is small.

2. Challenges of Modern Processor Architecture

In modern microprocessor architectures caches, pipelines, and all kinds of speculation are key features for improving performance. Caches are used to bridge the gap between processor speed and the access time of main memory. Pipelines enable acceleration by overlapping the executions of different instructions. The consequence is that the execution times of individual instructions, and thus the contribution of one execution of an instruction to the

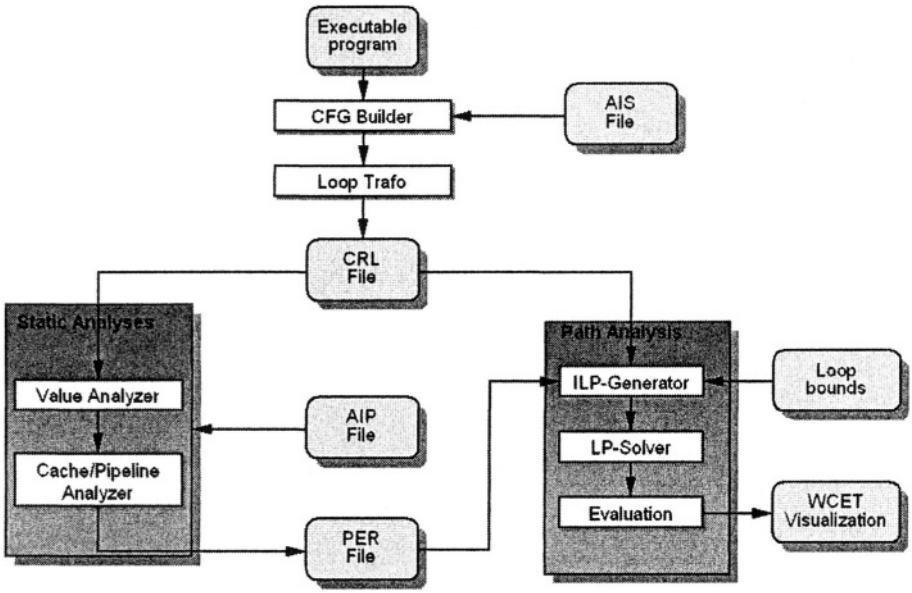

Figure 1. Phases of WCET computation

program's execution time can vary widely. The timing difference between the minimal case (when execution of an instruction goes smoothly through pipeline and cache) and a timing accident (when everything goes wrong) can be in the order of several hundred processor cycles. Since the execution time of an instruction depends on the execution state, e.g., the contents of the cache(s), the occupancy of other resources, and thus on the execution history, the execution time cannot be determined in isolation from the execution history.

3. Phases of WCET Computation

AbsInt's WCET tool **aiT** determines the WCET of a program task in several phases [Ferdinand et al., 2001] (see Figure 1):

- **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 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 [Ferdinand, 1997];

- **Pipeline Analysis** predicts the behavior of the program on the processor pipeline [Langenbach et al., 2002];
- **Path Analysis** determines a worst-case execution path of the program [Theiling and Ferdinand, 1998].

Cache Analysis uses the results of value analysis to predict the behavior of the (data) cache. The results of cache analysis are used within pipeline analysis allowing the prediction of pipeline stalls due to cache misses. The combined results of the cache and pipeline analyses are the basis for computing the execution times of program paths. Separating WCET determination into several phases makes it possible to use different methods tailored to the subtasks [Theiling and Ferdinand, 1998]. Value analysis, cache analysis, and pipeline analysis are done by abstract interpretation [Cousot and Cousot, 1977]. Integer linear programming is used for path analysis.

Value Analysis

Value analysis tries to determine the values in the processor registers for every program point and execution context. Often it cannot determine these values exactly, but only finds safe lower and upper bounds, i.e. intervals that are guaranteed to contain the exact values. The results of value analysis are used to determine loop bounds and possible addresses of indirect memory accesses (important for cache analysis).

Value analysis uses the framework of abstract interpretation: an abstract state maps registers to intervals of possible 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 register can only hold finitely many different values. Practically, value analysis becomes only efficient by applying suitable widening and narrowing operators as proposed in [Cousot and Cousot, 1977]. The results of value analysis 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.

Pipeline Analysis

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

Ferdinand, 1999]. 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. Usage of aiT

aiT reads 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.

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

In addition to the raw information about the WCET, detailed information delivered by the analysis can be visualized by **AbsInt**'s **aiSee** tool (`http://www.aisee.com`). Figure 2 shows the graphical representation of the call graph for some small example. The calls (edges) that contribute to the worst-case running time are marked by the color red. The computed WCET is given in CPU cycles and in microseconds.

Figure 3 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

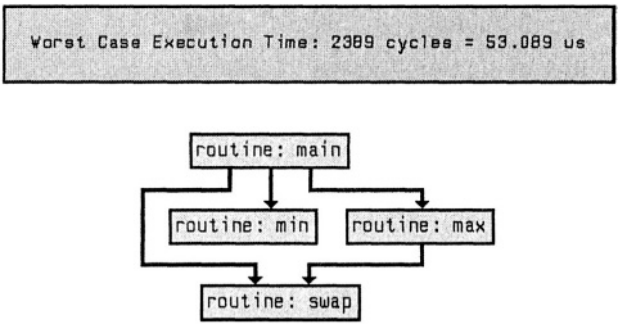

Figure 2. Call graph with WCET results

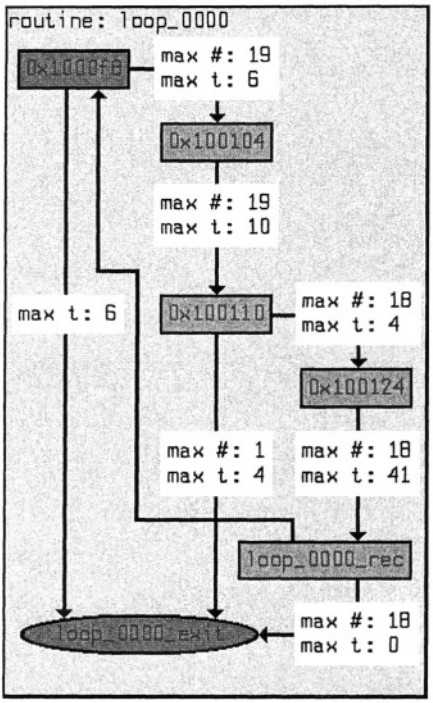

Figure 3. Basic block graph in a loop, with timing information

`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 4 shows the development of possible pipeline states for a basic block in this example. Such pictures are shown by **aiT** upon special demand. The grey boxes correspond to the instructions of the basic block, and the smaller

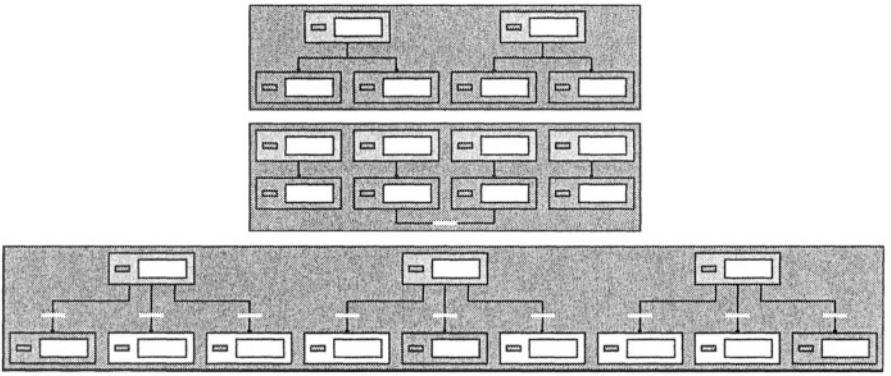

Figure 4. Possible pipeline states in a basic block

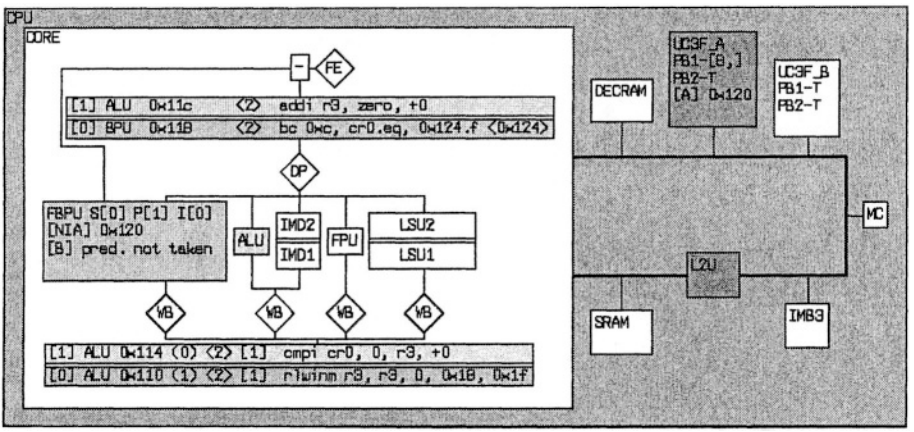

Figure 5. Individual pipeline state

rectangles are individual pipeline states. Their cycle-wise evolution is indicated by the edges 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 the details they differ in leave the pipeline. This happened for instance at the end of the second instruction.

Figure 5 shows part of the top left pipeline state from Figure 4 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.

5. Conclusion

aiT enables one to develop complex hard real-time systems on state-of-the-art hardware, increases safety, and saves development time. It has been applied to real-life benchmark programs containing realistically sized code modules. Precise timing predictions make it possible to choose the most cost-efficient hardware. Tools like **aiT** are of high importance as recent trends, e.g., X-by-wire, require the knowledge of the WCET of tasks.

References

Cousot, Patrick and Cousot, Radhia (1977). 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.

Ferdinand, Christian (1997). *Cache Behavior Prediction for Real-Time Systems.* PhD thesis, Saarland University.

Ferdinand, Christian, Heckmann, Reinhold, Langenbach, Marc, Martin, Florian, Schmidt, Michael, Theiling, Henrik, Thesing, Stephan, and Wilhelm, Reinhard (2001). 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.

Langenbach, Marc, Thesing, Stephan, and Heckmann, Reinhold (2002). 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.

Schneider, **Jern** and Ferdinand, Christian (1999). 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.

Theiling, Henrik and Ferdinand, Christian (1998). 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.