Worst-Case Execution Time Prediction by Static Program Analysis

Reinhold Heckmann *Christian Ferdinand*

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: 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.

Recent advances in the area of abstract interpretation have led to the development of static program analysis tools that efficiently determine upper bounds for the Worst-Case Execution Time (WCET) of code snippets given as routines in executables.

The predicted WCETs can be used to determine an appropriate scheduling scheme for the tasks and to perform an overall schedulability analysis in order to guarantee that all timing constraints will be met (also called *timing validation*) [9]. Some real-time operating systems offer tools for schedulability analysis, but all these tools require the WCETs of tasks as input.

2 Measurement-Based WCET Analysis

Measuring the execution time seems to be a simple alternative to WCET determination by static analysis. Yet the characteristics of modern software and hardware complicate WCET analysis based on measurements.

2.1 Software Features Making Measurement-Based WCET Analysis a Challenge

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.

2.2 Hardware Features Making Measurement-Based WCET Analysis a Challenge

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 [7], which consist of a locally advantageous situation, e.g., a cache hit, resulting in a globally larger execution time. As also demonstrated in [7], 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.

3 Prediction of the Worst-Case Execution Time by Static Program Analysis

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 in the previous sections.

AbsInt's WCET tool **aiT** determines the WCET of a program task in several phases [4] (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 [3];
- **Pipeline Analysis** predicts the behavior of the program on the processor pipeline [6];
- **Path Analysis** determines a worst-case execution path of the program [10].

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

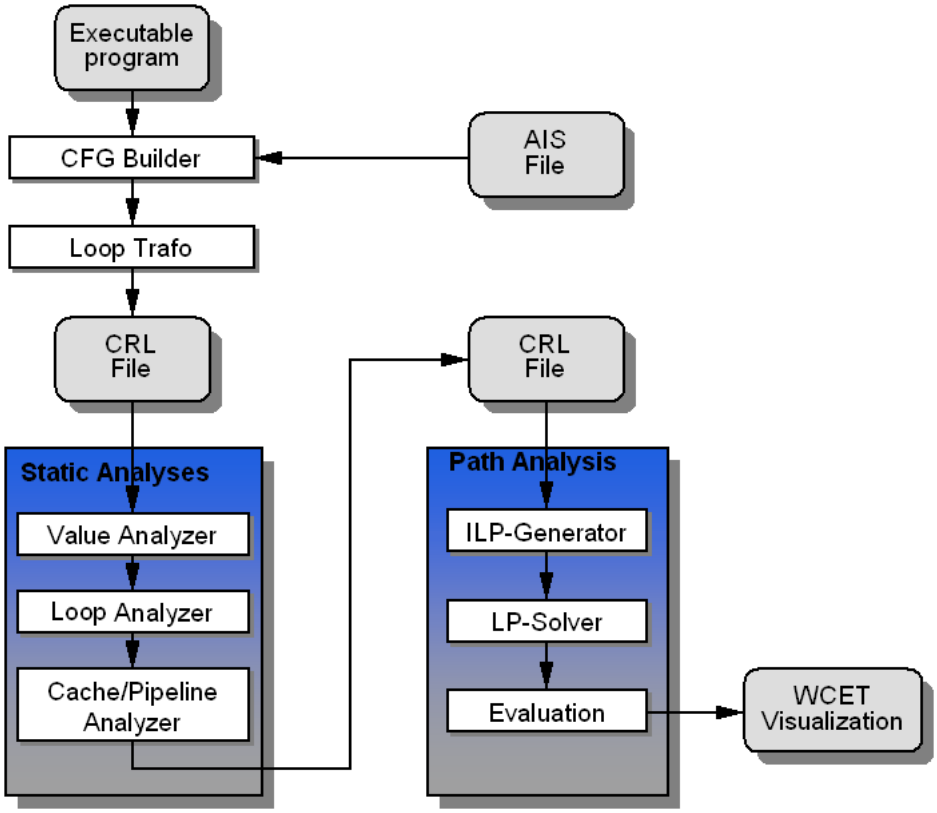

Figure 1: Phases of WCET computation

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 [10]. Value analysis, cache analysis, and pipeline analysis are done by abstract interpretation [2], a semantics-based method for static program analysis. Integer linear programming is used for path analysis.

3.1 Reconstruction of the Control Flow from Binary Programs

The starting point of our analysis framework (see Figure 1) 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. (see section 4).

In the first step a parser reads the executable and reconstructs the control flow [**?**, **?**]. 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 (see section 4). 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.

3.2 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 possible accesses of indirect memory accesses – important for cache analysis – and in loop bound analysis (see section 3.3).

Value analysis uses the framework of abstract interpretation [2]: 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 [2]. 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.

3.3 Loop Bound Analysis

WCET analysis requires that upper bounds for the iteration numbers of all loops be known. **aiT** determines the number of loop iterations by *loop bound analysis*. This is possible for many loops occurring in typical applications. Bounds for the iteration numbers of the remaining loops must be provided as user annotations (see section 4).

Loop bound analysis relies on a combination of value analysis (see section 3.2) and pattern matching, which looks for typical loop patterns. In general, these loop patterns depend on the code generator and/or compiler used to generate the code that is being analyzed. There are special **aiT** versions adapted to various generators and compilers.

3.4 Cache Analysis

Cache analysis classifies the accesses to main memory. The analysis in our tool is based upon [3], 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 [5].

3.5 Pipeline Analysis

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

3.6 Path Analysis

Path analysis uses the results of pipeline analysis, i.e. the estimated WCETs at the control-flow edges for all contexts, to compute a WCET estimation for the entire code that is currently analyzed.

Let $T(e,c)$ be the estimated WCET for edge e and context c as determined by cache and pipeline analysis. Furthermore, let $C(e,c)$ be the *execution count*, which indicates how often control passes along edge e in context c. If one knows for a specific run of the code the execution counts $C(e,c)$ for each edge e in each context c, then one can get an upper bound for the time of this run by taking the sum of $C(e,c) \cdot T(e,c)$ over all edge-context pairs (e,c). Thus, the task of obtaining a global WCET estimation can be solved by finding a feasible assignment of execution counts $C(e,c)$ to edge-context pairs that maximizes $\sum C(e,c) \cdot T(e,c)$. The value of this sum is then the desired global WCET estimate.

Therefore, path analysis is implemented by *integer linear programming*: The path analyzer sets up a system of linear constraints over the integer variables $C(e,c)$. Then an auxiliary tool looks for a solution of this constraint system that maximizes $\sum C(e,c) \cdot T(e,c)$. The constraints of the constraint system are derived from the control structure and from the loop bounds found by loop bound analysis or provided by the user. Constraints derived from the control structure are for instance those that assert that the sum of the execution counts of the incoming edges of a block equals the sum of the execution counts of the outgoing edges. Additional constraints can be derived from user annotations representing knowledge about dependencies of program parts.

3.7 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) [8].

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 User Annotations

Apart from the executable, **aiT** needs user input to find a result at all, or to improve the precision of the result. The most important user annotations specify the targets of computed calls and branches and the maximum iteration counts of loops (there are many other possible annotations).

4.1 Targets of Computed Calls and Branches

For a correct reconstruction of the control flow from the binary, targets of computed calls and branches must be known. **aiT** can find many of these targets automatically for code compiled from C. This is done by identifying and interpreting switch tables and static arrays of function pointers. Yet dynamic use of function pointers cannot be tracked by **aiT**, and hand-written assembly code in library functions often contains difficult computed branches. Targets for computed calls and branches that are not found by **aiT** must be specified by the user. This can be done by writing specifications of the following forms in a parameter file called AIS file:

INSTRUCTION *ProgramPoint* CALLS *Target*$_1$, ..., *Target*$_n$;
INSTRUCTION *ProgramPoint* BRANCHES TO *Target*$_1$, ..., *Target*$_n$;

Program points are not restricted to simple addresses. A program point description particularly suited for CALLS and BRANCHES specifications is "*R*" + *n* COMPUTED which refers to the *n*th computed call or branch in routine *R* – counted statically in the sense of increasing addresses, not dynamically following the control flow. In a similar way, targets can be specified as absolute addresses, or relative to a routine entry in the form "*R*" + *n* BYTES or relative to the address of the conditional branch instruction, which is denoted by PC.

Example: The library routine C_MEMCPY in TI's standard library for the TMS470 consists of hand-written assembly code. It contains two computed branches whose targets can be specified as follows:

```
instruction "C_MEMCPY" + 1 computed
branches to pc + 0x04 bytes, pc + 0x14 bytes, pc + 0x24 bytes;

instruction "C_MEMCPY" + 2 computed
branches to pc + 0x10 bytes, pc + 0x20 bytes;
```

The advantage of such relative specifications is that they work no matter what the absolute address of C_MEMCPY is.

If the application contains an array P of function pointers, then a call P[i](x) may branch to any address contained in P. **aiT** tries to obtain the list of these addresses automatically: If the array access and the computed call in the executable are part of a small code pattern as it is typically generated by the compiler, **aiT** notices that the computed call is performed via this array. If furthermore the array contents are defined

in a data segment so that they are statically available, and the array is situated in a ROM area so that its contents cannot be modified, then **aiT** automatically considers the addresses in the array as possible targets of the computed call.

If array access and computed call are too far apart or realized in an untypical way, **aiT** cannot recognize that they belong together. Similar remarks apply to computed branches via switch tables. In both cases, the array or table belonging to the computed call or branch can be declared in the AIS file. The declaration starts like the ones described above:

```
INSTRUCTION ProgramPoint CALLS VIA ArrayDescriptor;
INSTRUCTION ProgramPoint BRANCHES VIA ArrayDescriptor;
```

Here, the *ArrayDescriptor* describes the location and the format of the table that contains the call or branch targets. These targets are extracted from the table according to the given format rules.

4.2 Loop Bounds

WCET analysis requires that upper bounds for the iteration numbers of all loops be known. **aiT** determines the number of loop iterations by *loop bound analysis*, but succeeds in doing so only for loops with constant bounds whose code matches certain patterns typically generated by the supported compilers. Bounds for the iteration numbers of the remaining loops must be provided by user annotations. A maximum iteration number of j is specified in the AIS parameter file as follows:

```
LOOP ProgramPoint Qualifier MAX j;
```

A *ProgramPoint* is either an address or an expression of the form `"`*R*`"` + *n* `LOOPS` which means the nth loop in routine R counted from 1. *Qualifier* is an optional information:

`begin` indicates that the loop test is at the beginning of the loop, as for C's `while`-loops.

`end` indicates that the loop test is at the end of the loop, as for C's `do-while`-loops.

The qualifier is only needed if the loop structure is so complicated that **aiT** cannot identify the position of the loop test by itself. If the qualifier is missing in this case, **aiT** assumes the worst case of the two possibilities, which is `begin` where the loop test is executed one more time. The `begin`/`end` information refers to the *executable*, not to the source code; the compiler may move the loop test from the beginning to the end, or vice versa.

Example: `loop "_prime" + 1 loop end max 10;`
specifies that the first loop in `_prime` has the loop test at the end and is executed at most 10 times.

4.3 Source Code Annotations

Specifications can also be included in C source code files as special comments marked by the key string `ai:`

```
/* ai: specification_1; ... specification_n; */
```

The names of the source files are extracted from the debug information in the executable.

Source code annotations admit a special program point or target `here`, which roughly denotes the place where the annotation occurs (due to compiler optimizations the debug information is not always precise). More exactly, **aiT** extracts the correspondence between source lines and code addresses from the executable. A `here` occurring in source line n then points to the *first* instruction associated with a line number $\geq n$.

For loop annotations, it is not required that `here` exactly denotes the loop start address. It suffices that it resolves to an address anywhere in the loop as in the following example:

```
for (i=3; i*i <= n; i += 2) {
    /* ai: loop here end max 10; */
    ... }
```

4.4 Other Annotations

Apart from branch targets and loop bounds, many other properties can be declared in parameter or source files.

- To get any WCET results at all, you must specify upper bounds for the recursion depths of all recursive routines. These specifications are similar to the loop bound specifications described above.
- Flow constraints relate the execution counts of any two basic blocks. For instance,

  ```
  flow (0x100) <= 4 (0x200);
  ```

  means that the number of executions of the block starting at address `0x100` is at most 4 times the number of executions of the block starting at `0x200`. As always, relative addresses or semantic program point descriptions may be used instead of these absolute addresses.
- **aiT** can be informed about the clock rate of the microprocessor. Knowing the clock rate, **aiT** can display its results in real time units such as milliseconds. Without this information, all results are displayed in processor cycles.
- End specifications instruct **aiT** to stop reading the executable at a certain program point. A possible application is for instance to inform **aiT** that an interrupt routine called by a software interrupt does not return.
- Value analysis tries to determine register values and addresses of memory accesses. In cases it fails, information about exact addresses or address ranges may be supplied by annotations.
- You may specify that a memory area is read-only or write-only, contains data or code.
- You may exclude certain routines from WCET analysis and supply their WCET directly.
- You may specify that a routine never returns (like exit).
- You may specify that a certain basic block is never executed or a certain condition always evaluates to true or to false.
- Program points can be given symbolic names for later reference.

5 aiT – WCET Analyzers

The techniques described above have been incorporated into **AbsInt**'s **aiT** WCET analyzer tools. As input they get an executable, user annotations as described in section 4, 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

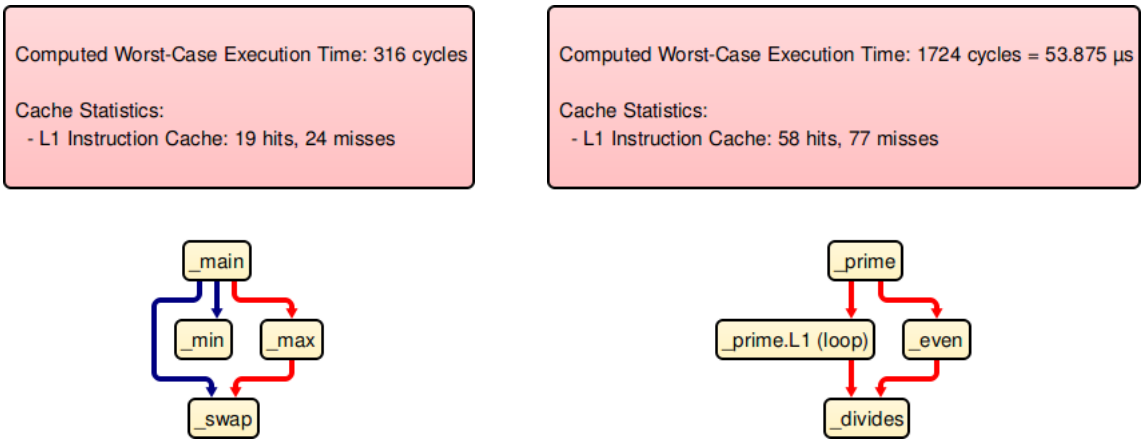

Figure 2: Call graph with WCET results

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 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 the aiSee tool [1]. Figure 2 shows graphical representations of the call graphs for some small examples. 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 provided that the cycle time of the processor has been specified (as in the picture on the right).

The routines in the call graph can be opened to see their basic block graphs; Figure 3 shows an example. The worst-case path through the routine is indicated by the light red edges. `sum #` describes the number of traversals of an edge in the worst case. `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. Such pictures are shown by **aiT** upon special demand. The grey boxes correspond to the instructions of the basic block, and the smaller 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 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.

6 Advantages

aiT allows one 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).

6 Advantages

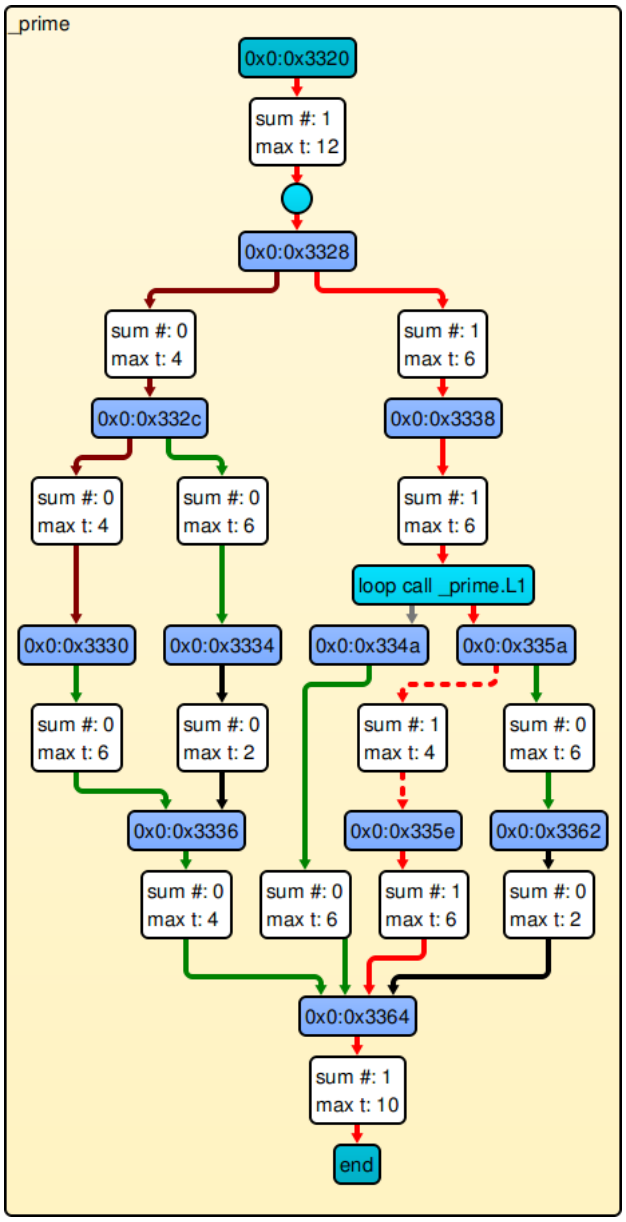

Figure 3: Basic block graph of a routine, with timing information

- **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.
- **aiT** is a WCET tool for industrial usage. Information required for WCET estimation such as computed branch targets and loop bounds is determined by static analysis. A convenient specification and annotation language was developed in close cooperation with **AbsInt**'s customers for situations where **aiT**'s analysis methods do not succeed. Annotations for library functions (RT, communication) and RTOS functions can be provided in separate files by the respective developers (on source level or separately).
- **aiT** works on optimized code.
- **aiT** saves development time by avoiding the tedious and time consuming (instrument and) execute (or

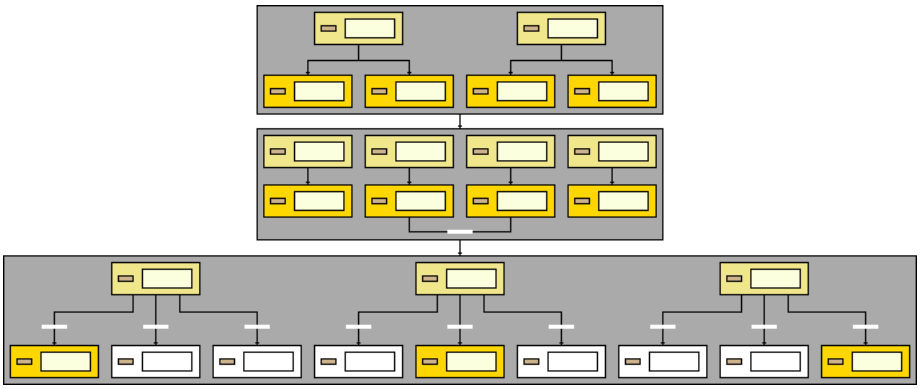

Figure 4: Possible pipeline states in a basic block

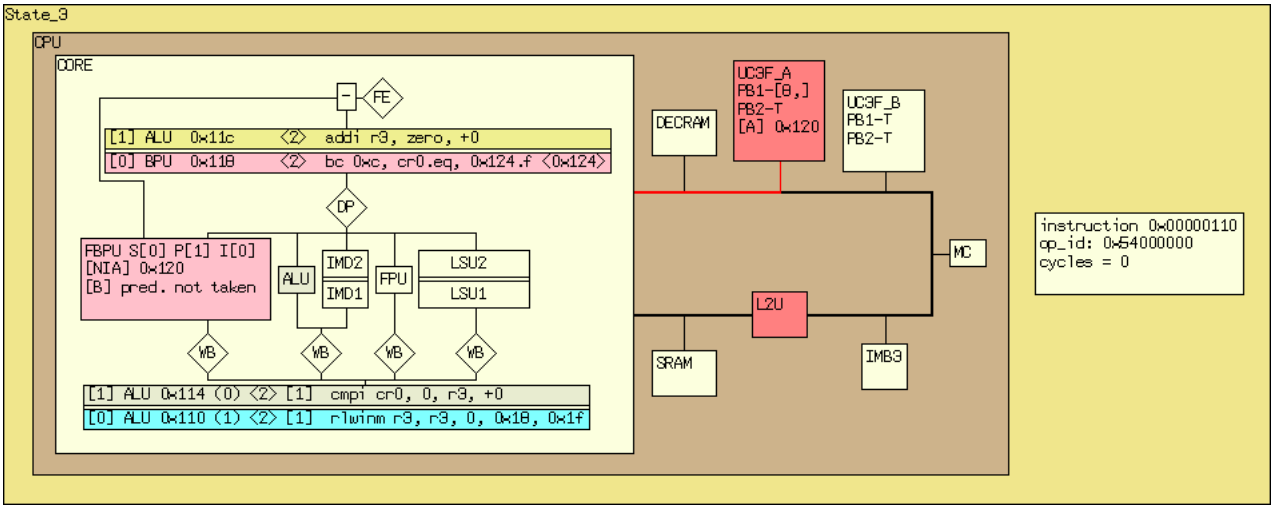

Figure 5: Individual pipeline state

emulate) and measure cycle for a set of inputs over and over again.

- **aiT**'s visualization features give a precise insight of the inner working of the processors. This information can be used for program optimization. Comparable detailed information is not available by other existing development tools. Measurements only allows to access dedicated debug interfaces or the outside of modern processors. The internal working is usually not captured.
- **aiT** is based on the theory of Abstract Interpretation thus offering a provably correct relation to the architecture's semantics.

7 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

[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*, pages 238–252, Los Angeles, California, 1977.

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] Henrik Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing Systems and Applications*, pages 23–30, Cheju Island, South Korea, 2000.

[12] 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.

[13] 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.