The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, Per Stenström

---

The determination of upper bounds on execution times, commonly called Worst-Case Execution Times (WCETs), is a necessary step in the development and validation process for hard real-time systems. This problem is hard if the underlying processor architecture has components such as caches, pipelines, branch prediction, and other speculative components. This article describes different approaches to this problem and surveys several commercially available tools and research prototypes.

Categories and Subject Descriptors: J.7 [**COMPUTERS IN OTHER SYSTEMS**]: ; C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: REAL-TIME AND EMBEDDED SYSTEMS

General Terms: Verification, Reliability

Additional Key Words and Phrases: Hard real time, worst-case execution times

---

Work reported herein was supported by the European Accompanying Measure ARTIST, Advanced Real Time Systems, and the European Network of Excellence, ARTIST2.
Wilhelm and Thesing are with Fachrichtung Informatik, Saarland University, D-66041 Saarbrücken, Germany. Engblom is with Virtutech AB, Norrtullsgatan 15, SE-113 27 Stockholm. Ermedahl is with Department of Computer Science and Electronics, Mälardalen University, PO Box 883, SE 72123 Västerås, Sweden. Holsti is with Tidorum Ltd, Tiirasaarentie 32, FI-00200 Helsinki, Finland. Whalley is with Computer Science Department, Florida State University, Tallahassee, FL 32306-4530, USA. These authors are responsible for the article and have written the introduction to the problem area and the overview of the techniques. They have also edited the tool descriptions to make them more homogeneous.
Tool descriptions were provided by Guillem Bernat, Christian Ferdinand, Andreas Ermedahl, Reinhold Heckmann, Niklas Holsti, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, Per Stenström and David Whalley. Bernat is with Rapita Systems Ltd., IT Center, York Science Park, Heslington, York YO10 5DG, United Kingdom. Ferdinand and Heckmann are with AbsInt Angewandte Informatik, Science Park 1, D-66123 Saarbrücken. Mitra is with Department of Computer Science, School of Computing, 3 Science Drive 2, National University of Singapore, Singapore 117543. Mueller is with Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206. Puaut is with IRISA, Campus univ. de Beaulieu, F-35042 Rennes Cédex. Puschner is with Inst. für Technische Informatik, TU Wien, A-1040 Wien. Staschulat is with Inst. for Computer and Communication Network Engineering, Technical University Braunschweig, Hans-Sommer-Str. 66, D-38106 Braunschweig. Stenström is with Department of Computer Engineering, Chalmers University of Technology, S-412 96 Göteborg.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0164-0925/20YY/0500-00001 $5.00

1. INTRODUCTION

Hard real-time systems need to satisfy stringent timing constraints, which are derived from the systems they control. In general, upper bounds on the execution times are needed to show the satisfaction of these constraints. Unfortunately, it is not possible in general to obtain upper bounds on execution times for programs. Otherwise, one could solve the halting problem. However, real-time systems only use a restricted form of programming, which guarantees that programs always terminate; recursion is not allowed or explicitly bounded as are the iteration counts of loops. A reliable guarantee based on the worst-case execution time of a task could easily be given if the worst-case input for the task were known. Unfortunately, in general the worst-case input is not known and hard to derive.

We assume that a real-time system consists of a number of tasks, which realize the required functionality. Figure 1 depicts several relevant properties of a real-time task. A task typically shows a certain variation of execution times depending on the input data or different behavior of the environment. The set of all execution times is shown as the upper curve. The shortest execution time is called the *best-case execution time* (BCET), the longest time is called the *worst-case execution time* (WCET). In most cases the state space is too large to exhaustively explore all possible executions and thereby determine the exact worst-case and best-case execution times.

Today, in most parts of industry the common method to estimate execution-time bounds is to measure the *end-to-end* execution time of the task for a subset of the possible executions—test cases. This determines the *minimal observed* and *maximal observed execution times*. These will in general overestimate the BCET and underestimate the WCET and so are not safe for hard real-time systems. This method is often called *dynamic timing analysis*.

Newer measurement-based approaches make more detailed measurements of the execution time of different *parts* of the task and combine them to give better estimates of the BCET and WCET for the whole task. Still, these methods are rarely guaranteed to give bounds on the execution time.

Bounds on the execution time of a task can be computed only by methods that consider all possible execution times, that is, all possible executions of the task. These methods use abstraction of the task to make timing analysis of the task feasible. Abstraction loses information, so the computed WCET bound usually overestimates the exact WCET and vice versa for the BCET. The WCET bound represents the *worst-case guarantee* the method or tool can give. How much is lost depends both on the methods used for timing analysis and on overall system properties, such as the hardware architecture and characteristics of the software. These system properties can be subsumed under the notion of *timing predictability*.

The two main criteria for evaluating a method or tool for timing analysis are thus *safety*—does it produce bounds or estimates?— and *precision*—are the bounds or estimates close to the exact values?

Performance prediction is also required for application domains that do not have hard real-time characteristics. There, systems may have deadlines, but are not required to absolutely observe them. Different methods may be applied and different criteria may be used to measure the quality of methods and tools.

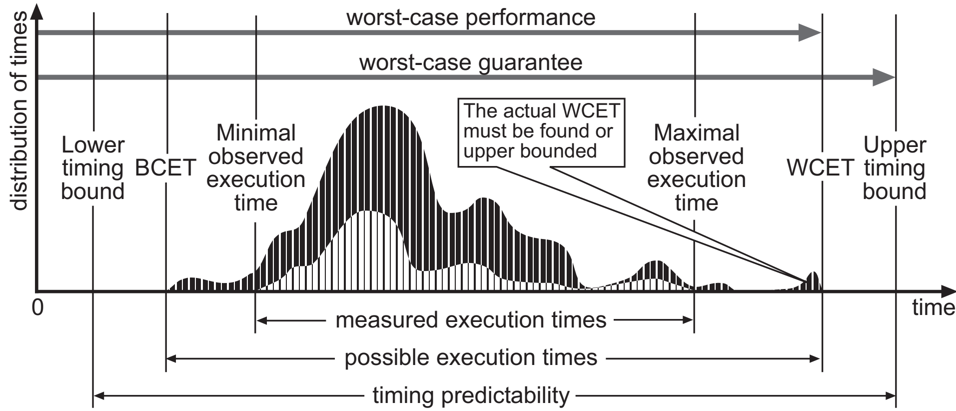

Fig. 1. Basic notions concerning timing analysis of systems. The lower curve represents a subset of measured executions. Its minimum and maximum are the *minimal observed execution times* and *maximal observed execution times*, resp. The darker curve, an envelope of the former, represents the times of all executions. Its minimum and maximum are the *best-case* and *worst-case execution times*, resp., abbreviated BCET and WCET.

The literature on timing analysis has created a confusion by not always making a distinction between worst-case execution times and estimates for them. We will avoid this misnomer in this survey.

We will use the term *timing analysis* for the process of deriving execution-time bounds or estimates. A tool that derives bounds or estimates for the execution times of application tasks is called a *timing-analysis tool*. We will concentrate on the determination of upper bounds or estimates of the WCET unless otherwise stated. All tools described in Section 6 with the exception of SymTA/P offer timing analysis of tasks in uninterrupted execution. Here, a *task* may be a unit of scheduling by an operating system, a subroutine, or some other software unit. This unit is mostly available as a fully-linked executable. Some tools, however, assume the availability of source code and of a compiler supporting a subsequent timing analysis.

Organization of the article

Section 2 introduces the problem and its subproblems and describes methods being used to solve it. Sections 3 and 4 present two categories of approaches, static and measurement-based. Section 6 consists of detailed tool descriptions. Section 7 resumes the state of the art and the deployment and use in industry. Section 8 lists limitations of the described tools. Section 9 gives a condensed overview of the tools in a tabulated form. Section 10 explains, how timing analysis is or should be integrated in the development process. Section 11 concludes the paper by presenting open problems and the perspectives of the domain mainly determined by architectural trends.

2. OVERVIEW OF TIMING-ANALYSIS TECHNIQUES

This section describes the problems that make timing analysis both difficult and interesting as a research topic, presents a decomposition of the problem into subtasks, and categorizes some of the techniques used to determine bounds on execution times. A given timing-analysis method or tool may not address or solve all these subtasks and different methods and tools may solve the same subtask in different ways.

2.1 Problems and Requirements

Timing analysis attempts to determine bounds on the execution times of a task when executed on a particular hardware. The time for a particular execution depends on the path through the task taken by control and the time spent in the statements or instructions on **this path** on **this hardware**. Accordingly, the determination of execution-time bounds has to consider the potential control-flow paths and the execution times for this set of paths. A modular approach to the timing-analysis problem splits the overall task into a sequence of subtasks. Some of them deal with properties of the control flow, others with the execution time of instructions or sequences of instructions on the given hardware.

2.1.1 *Data-Dependent Control Flow.* The task to be analyzed attains its WCET on one (or sometimes several) of its possible execution paths. If the input and the initial state leading to the execution of this worst-case path were known, the problem would be easy to solve. The task would then be started in this initial state with this input, and the execution time would be measured. In general, however, this worst-case input and initial state are **not** known and hard or impossible to determine. A data structure, the task's control-flow graph, CFG, describes a superset of the set of all execution paths. The task's call graph usually is integrated into the CFG.

A first problem that has to be solved is the construction of the control-flow graph and call graph of the task from a source or a machine-code version of the task. They must contain all of the instructions of the task (function closure) under analysis. Problems are created by dynamic jumps and dynamic calls with computed target address. Dynamic jumps are mainly due to switch/case structures and are a problem only when analyzing machine code, because even assembly code usually labels all switch/case branches. Dynamic calls also occur in source code in the form of calls through function pointers and calls to virtual functions. A component of a timing-analysis tool which reconstructs the CFG from a machine program is often called a *Frontend*.

Different paths through the CFG are taken depending directly or indirectly on input data. Some paths in the superset described by the CFG will never be taken, for instance those that have contradictory consecutive conditions. Eliminating such paths may increase the precision of timing analysis. The more the analysis knows about the data flow through the task, the more it knows about the outcome of and the relationship between conditions, the more paths it may recognize as infeasible.

A phase called *Control-Flow Analysis* (CFA) determines information about the possible flow of control through the task to increase the precision of the subsequent analyzes. Control flow analysis may attempt to exclude infeasible paths, determine

execution frequencies of paths or the relation between execution frequencies of different paths or subpaths etc. Control-Flow Analysis has previously been called *High-level Analysis*.

Tasks spend most of their execution time in loops and in (recursive) functions. Therefore, it is an essential task of CFA to determine bounds on the iterations of loops and on the depth of recursion of functions. A necessary ingredient for this are the values of variables, registers, or memory cells occurring in conditions tested for termination of loops or recursion.

It is worth noting that complex processors may actually execute an instruction stream in a different order than the one determined by control-flow analysis. This is due to pipelining (prefetching and delayed branching), branch prediction, and speculative or out-of-order execution.

2.1.2 *Context Dependence of Execution Times.* Early approaches to the timing-analysis problem assumed context independence of the timing behavior; the execution times for individual instructions were independent from the execution history and could be found in the manual of the processor. From this context independence was derived a *structure-based* approach [Shaw 1989]: if a task first executes a code snippet A and then a snippet B, the worst-case bound for $A;B$ was determined as that for A, ub_A, added to that determined for B, ub_B, formally $ub_{A;B} = ub_A + ub_B$. This context independence, however, is no longer true for modern processors with caches and pipelines. The execution time of individual instructions may vary by several orders of magnitude depending on the state of the processor in which they are executed. Thus, the execution time of B can heavily depend on the execution state that the execution of A produced. Any tool should exploit the knowledge that A was executed before B to determine a precise upper bound for B *in the context* A. Determining the upper bound $ub_{A;B}$ for $A;B$ by $ub_{A;B} = ub_A + ub_B$ ignores this information and will in general not obtain precise results.

A phase called *Processor-Behavior Analysis* gathers information on the processor behavior for the given task, in particular the behavior of the components that influence the execution times, such as memory, caches, pipelines, and branch prediction. It determines upper bounds on the execution times of instructions or basic blocks. Processor-Behavior Analysis has previously been called *Low-level Analysis*.

2.1.3 *Timing Anomalies.* The complexity of the processor-behavior analysis subtask and the set of applicable methods critically depend on the complexity of the processor architecture [Heckmann et al. 2003]. Most powerful microprocessors suffer from *timing anomalies* [Lundqvist and Stenström 1999c]. Timing anomalies are contra-intuitive influences of the (local) execution time of one instruction on the (global) execution time of the whole task. This concept is quite complex. So, we will try to explain it in some detail.

We assume that the system under consideration, executing hardware and executed software, are too complex to allow exhaustive execution or simulation. In addition, not all input data are known, so that parts of the execution state are missing in the analysis. Unknown parts of the state lead to non-deterministic behavior, if decisions depend on these unknown parts. For timing analysis, this means that the execution of an instruction or an instruction sequence considered in an initial

abstract state may produce different times based on different assumptions about the missing state components. For example, missing information about whether the next instruction will be in the cache may lead to one execution starting with a cache load contributing the cache-miss penalty to the execution time, while another execution will start with an instruction fetch from the cache. Intuition would suggest that the latter execution would always lead to the shorter execution time of the whole task. On processors with timing anomalies, however, this need not be true. The latter execution may in fact lead to a longer task execution time. This was observed on the Motorola ColdFire 5307 processor [Heckmann et al. 2003]. The reason is the following. This processor speculates on the outcome of conditional branches, that is, it prefetches instructions in one of the directions of the conditional branch. When the condition is finally evaluated it may turn out that the processor speculated in the wrong direction. All the effects produced so far have to be undone. In addition, fetching the wrong instructions has partly ruined the cache contents. Taken together, the costs of the mis-prediction exceed the costs of a cache miss. Hence, the local worst case, the I-cache miss, leads to the globally shorter execution time since it prevents a more expensive branch mis-prediction. This exemplifies one of the reasons for timing anomalies, **speculation-caused anomalies**. One such anomaly is shown in Figure 2.[1]

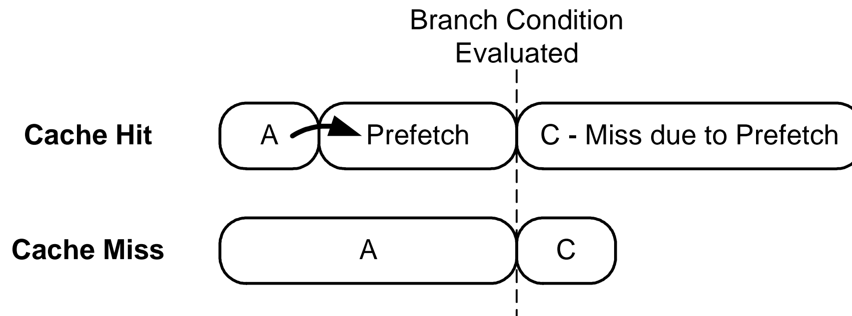

Fig. 2. A timing anomaly caused by speculation.

Another type of timing anomalies are instances of well-known **scheduling anomalies**, first discovered and published by Graham [Graham 1966]. These occur when a sequence of instructions, partly depending on each other, can be scheduled differently on the hardware resources, such as pipeline units. Depending on the selected schedule the execution of the instructions or pipeline phases takes different times. Figure 3 shows an example of a scheduling-caused timing anomaly.

Timing anomalies violate an intuitive, but incorrect assumption, namely that always taking the local worst-case transition when there is a choice produces the global worst-case execution time. This means that the analysis cannot greedily limit its search for upper bounds by choosing the worst cases for each instruction. The existence of timing anomalies in a processor thus has a strong influence on the applicability of methods for timing analysis for that processor [Heckmann et al. 2003].

[1]Figures 2 and 3 are taken from [Reineke et al. 2006].

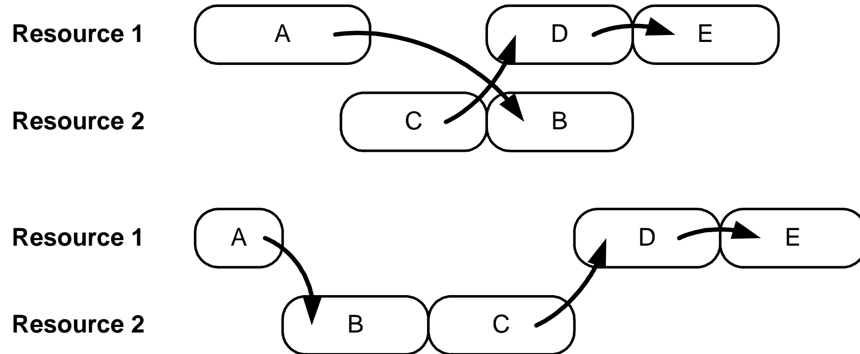

Fig. 3. A scheduling-caused timing anomaly.

—The assumption that only local worst cases have to be considered to safely determine upper bounds on global execution times is unsafe.

—The assumption that one could identify a worst initial execution state, to safely start measurement or analysis of a piece of code in, is unsafe.

The consequences for timing analysis of systems to be executed on processors with timing anomalies are as follows:

—The analysis may be forced to follow execution through several successor states, whenever it encounters an abstract state with a non-deterministic choice between successor states. This may lead to a quite large state space to consider.

—The analysis has to be able to express the absence of state information instead of assuming some worst initial state. Absent information in abstract states stands for all potential concrete instances of these missing state components, thus do not wrongly exclude any possible execution.

2.2 Classification of Approaches

We present two different classes of methods.

Static methods. These methods do not rely on executing code on real hardware or on a simulator. They rather take the task code itself, maybe together with some annotations, analyze the set of possible control flow paths through the task, combine control flow with some (abstract) model of the hardware architecture, and obtain upper bounds for this combination. One such static approach is described in detail in [Wilhelm 2005].

Measurement-based methods. These methods execute the task or task parts on the given hardware or a simulator for some set of inputs. They then take the measured times and derive the maximal and minimal observed execution times, see Figure 1, or their distribution or combine the measured times of code snippets to results for the whole task.

Static methods emphasize *safety* by producing bounds on the execution time, guaranteeing that the execution time will not exceed these bounds. The bounds allow safe schedulability analysis of hard real-time systems.

Measurements can be used in different ways. End-to-end measurements of a subset of all possible executions produce estimates, not bounds. They may be useful for applications that do not require guarantees, typically non-hard real-time systems. They may give the developer a feeling about the execution time in common cases and the likelihood of the occurrence of the worst case. Measurement can also be applied to code snippets after which the results are combined to estimates for the whole program in similar ways as used in static methods. Guarantees that safe bounds are obtained can currently only be given for rather simple architectures due to the reasons given in Section 2.1.

2.3 Methods for Subtasks of Timing Analysis

We briefly describe some methods that are being used to solve the above mentioned subtasks of timing analysis. These methods are imported from other fields of computer science such as compiler construction, computer architecture, performance estimation, and optimization.

These methods can be categorized according to several properties: whether they are *automatic* or *manual*, whether they are *generic*, i.e., stand for a whole class of methods, or are *specific instances* of such a generic method, and whether they are applied at *analysis time* or at *tool-construction time*.

2.3.1 *Static Program Analysis.* Static program analysis is a generic method to determine properties of the dynamic behavior of a given task without actually executing the task [Cousot and Cousot 1977; Nielson et al. 1999]. These properties are often undecidable. Therefore, sound approximations are used; they have to be correct, but may not necessarily be complete. An example from our domain is instruction-cache analysis, which attempts to determine for each point in the task which instructions will be in the cache every time execution reaches this program point. For straight-line programs and known initial cache contents this is easy and can be done by a standard simulator. However, it is in general undecidable for tasks whose control flow depends on input data. A sound analysis will compute a subset of the instructions that will definitely be in the instruction cache every time execution reaches the program point. More instructions may actually be in the cache, but the analysis may not be able to find this out. Several instances of static program analysis are being used for timing analysis.

2.3.2 *Simulation.* Simulation is a standard technique to estimate the execution time for tasks on hardware architectures. A key advantage of this approach is that it is possible to derive rather accurate estimations of the execution time for a task for a given set of input data and assuming sufficient detail of the timing model of the architectural simulator. [Desikan et al. 2001] compares timing measurements with runs on different simulators and gives indication of the errors obtained for an Alpha architecture. The Simplescalar [Austin et al. 2002] simulator among others is used. Simplescalar is also used by some WCET groups. The results show large differences in timing compared to the measured values. The measurement method used might be questioned, but at least it shows that the Simplescalar simulator should not be trusted as a clock-cycle accurate simulator for all types of architectures.

Unfortunately, standard cycle-accurate simulators cannot be used off-hand in static methods for timing analysis, since static methods should not simulate exe-

cution for particular input data, but rather for all input data. Thus, input data is assumed to be unknown. Unknown input data leads to unknown parts in the execution state of the processor and non-deterministic decisions at control-flow branches. Simulators modified to cope with these problems are being used in several of the tools described later.

2.3.3 *Abstract Processor Models.* Processor-behavior analysis needs a model of the architecture. This need not be a concrete model implementing all of the functionality of the target hardware. A simplified model that is conservative with respect to the timing behavior is sufficient. Such an abstract processor model either is a part of the engine for processor-behavior analysis or is input to the construction of such an engine. In any case, the construction of an abstract processor model is done at tool-construction time.

One inherent limitation of all the approaches that are based on some model of the hardware architecture is that they rely on the timing accuracy of the model. In general, computer vendors do not disclose enough information about the microarchitecture so that one can develop and safely validate the accuracy of a timing model. Without such validation, any WCET tool based on an abstract model of the hardware cannot be trusted without further assurance. Additional means for model validation have to be taken. This could be done by measurements. Measured execution times are compared against predicted bounds. Another method is trace validation checking whether externally observable traces are projections of traces as predicted by the model. Not all events predicted by the model are externally observable. However, both methods are similar to testing; they can discover the presence of errors, but not prove their absence. Stronger guarantees can be given by equivalence checking between different abstraction levels. An ongoing research activity is the formal derivation of abstract processor models from concrete models.

2.3.4 *Integer Linear Programming (ILP). Linear programming* [Chvatal 1983] is a generic methodology to code the requirements of a system in the form of a system of linear constraints. Additionally given is a goal function that has to be maximized or minimized to obtain an optimal assignment of values to the system's variables. One speaks of *Integer Linear Programming* if these values are required to be integers. While linear programs can be solved in polynomial time, requiring the solution to be integer makes the problem NP-hard. This indicates that the use of ILP should be restricted to small problem instances or to subproblems of timing analysis generating only small problem instances.

In the timing-analysis domain, ILP is used in the IPET approach to bounds calculation, see Section 3.4. The control flow of tasks is translated into integer linear programs, essentially by coding Kirchhoff's rule about the conservation of flow. Extra information about the control flow can often be coded as additional constraints. The goal function expresses the execution time of the program under analysis. Its maximal value is then an upper bound for all execution times.

An escape from the exponential complexity that is often taken in other application domains is to use heuristics. These heuristics will in general only arrive at suboptimal solutions. A suboptimal solution in timing analysis represents an unsafe estimate for the WCET. Thus the escape of resorting to heuristics is barred.

ILP has been used for a completely different purpose, namely to model (very simple) processors [Li et al. 1995b; Li et al. 1995a]. However, the complexity of solving the resulting integer linear programs did not allow this approach to scale [Wilhelm 2004].

2.3.5 *Annotation.* The annotation of tasks with information available from the developer is a generic technique to support subsequently applied automatic validation techniques. The developer using a WCET tool may have to supply some information that the tool needs in separate files or by annotating the task. This information describes

—the memory layout and any needed characteristics of memory areas,

—ranges for the input values of the task,

—information about the control flow of the task if not determined automatically, e.g. loop bounds, shapes of nested loops, if iterations of inner loops depend on iteration variables of outer loops, frequencies of paths or branches taken,

—deviations from the standard function-calling conventions, and

—directives as to the desired precision of the result, which often depends on the invested effort for differentiating contexts.

2.3.6 *Frontend.* Most WCET tools analyze software at the executable level, since only at this level is all necessary information available. The first phase in timing analysis is thus the decoding of the executable and the reconstruction of its control flow. This can be quite involved depending on the instruction set of the processor and the code-generation patterns of the compiler. Some timing analysis tools are integrated with a compiler which emits the necessary CFG and call graph for the analysis.

2.3.7 *Visualization of Results.* The results of timing analysis are presented in human-readable form, best in the form of an informative visualization. This usually shows the call and control-flow graphs annotated with computed timing information and possibly also information about the processor states.

The following two sections present the two categories of approaches, static and measurement-based approaches. Section 6 describes the available tools from these two categories in more detail.

3. STATIC METHODS

This class of methods does not rely on executing code on real hardware or on a simulator, but rather takes the task code itself, combines it with some (abstract) model of the system, and obtains upper bounds from this combination.

Figure 4 shows the core components of a static timing-analysis tool and the flow of information.

3.1 Value Analysis

This is a static program analysis. Any method for data-cache behavior analysis needs to know effective memory addresses of data, in order to determine where a memory access goes. Effective addresses are only available at run time. However, a *value analysis*, as implemented in aiT, see Section 6.1, Bound-T, see Section 6.2, and in SWEET, see Section 6.7, is able to determine many effective addresses in

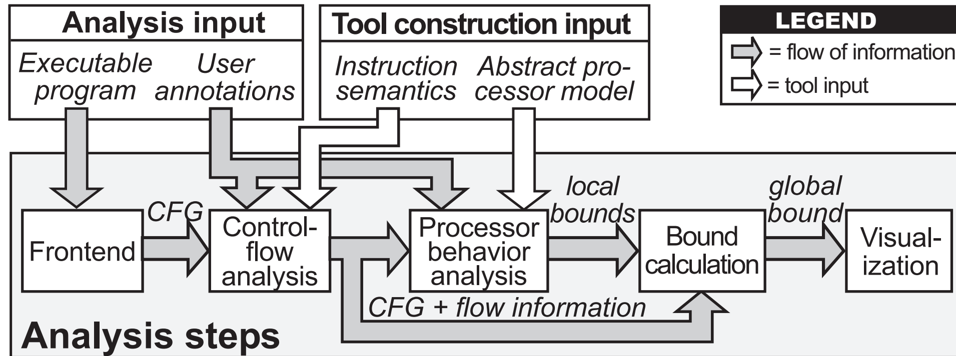

Fig. 4. Core components of a timing-analysis tool. The flow of information is shown by arrows filled in grey. The arrows filled white represent tool-construction input.

disciplined code statically [Thesing et al. 2003]. It does so by computing ranges for the values in the processor registers and local variables at every program point. This analysis is also useful to determine loop bounds and to detect infeasible paths.

3.2 Control-Flow Analysis

The purpose of control-flow analysis is to gather information about possible execution paths. The set of paths is always finite, since termination must be guaranteed. The exact set of paths can in general not be determined. Any superset of this set will be a safe approximation. The smaller this superset is the better. The execution time of any safely eliminated path can be ignored in computing the upper bound and thus will not contribute to it.

The input of flow analysis consists of a task representation, e.g. the call graph and the control-flow graph of the task and possibly additional information such as ranges for the input data and iteration bounds of some loops. The latter are either determined by a preceding value analysis or provided by the user. The result of the flow analysis can be seen as constraints on the dynamic behavior of the task. This includes information on which functions may be called, on dependencies between conditionals, and on the (in)feasibility of paths, etc.

There are a number of approaches to automatic flow analysis. Some of the methods are general, while others are specialized for certain types of code constructs. The methods also differ in the type of codes they analyze, i.e., source-, intermediate- (inside the compiler) or machine code.

Control-flow analysis is generally easier on source code than on machine code, but it is difficult to map the results to the machine-code program because compilation, in particular code optimization and linking may change the control-flow structure.

Gustafsson et al. [Gustafsson et al. 2003] uses a combination of flow-analysis methods. For simple loops, pattern matching is used. The pattern matching uses the results of a value analysis. For more complex loop constructs, an abstract interpretation-based method is used [Gustafsson 2000; Gustafsson et al. 2005]. The analysis is performed on the intermediate code level. Pattern-matching methods are based on the fact that for most loops the supported compilers use the same or similar groups of machine instructions to initialize, update and test loop counters.

Pattern-matching finds occurrences of such instruction groups in the code and analyzes the values of the instruction operands to find the counter range, for example in terms of the initial value, the increment or decrement and the final value of the counter. The drawback of this method is that it can be defeated by compiler optimizations, or by evolution of the compiler itself, if this changes the emitted instruction patterns so much that the matching fails.

Bound-T, see Section 6.2, finds loop bounds by modelling the computation, instruction by instruction, using affine equations and inequalities (Presburger Arithmetic). Bound-T then examines the model to find variables that act as loop counters. If Bound-T also finds bounds on the initial and final values of the variable, a simple computation gives a bound on the number of loop iterations.

Whalley et al. [Healy et al. 1998; Healy and Whalley 1999] use data flow analysis and special algorithms to calculate bounds for single and nested loops in conjunction with a compiler. [Stappert and Altenbernd 2000] uses symbolic execution on the source code level to derive flow information. aiT's loop-bound analysis, see Section 6.1, is based on a combination of an interval-based abstract interpretation and pattern-matching [Thesing 2004] working on the machine code.

The result of control-flow analysis is an annotated syntax tree for the structure-based approaches, see Section 3.4, and a set of flow facts about the transitions of the control-flow graph, otherwise. These flow facts are translated into a system of constraints for the methods using implicit path enumeration, see Section 3.4.

3.3 Processor-Behavior Analysis

As stated in Section 2.1.2, a typical processor contains several components that make the execution time context-dependent, such as memory, caches, pipelines and branch prediction. The execution time of an individual instruction, even a memory access depends on the execution history. To find precise execution-time bounds for a given task, it is necessary to analyze what the occupancy state of these processor components for all paths leading to the task's instructions is. Processor-behavior analysis determines invariants about these occupancy states for the given task. In principle, no tool is complete that does not take the processor periphery into account, i.e., the full memory hierarchy, the bus, and peripheral units. In so far, an even better term would be *hardware-subsystem behavior analysis*. The analysis is done on a linked executable, since only this contains all the necessary information. It is based on an abstract model of the processor, the memory subsystem, the buses, and the peripherals, which is conservative with respect to the timing behavior of the concrete hardware, i.e., it never predicts an execution time less than that which can be observed on the concrete processor.

The complexity of deriving an abstract processor model strongly depends on the class of processor used.

—For simpler 8bit and 16bit processors the timing model construction is rather simple, but still time consuming, and rather simple analyses are required. Complicating factors for the processor behavior analysis include instructions with varying execution time due to argument values and varying data reference time due to different memory area access times.

—For somewhat more advanced 16bit and 32bit processors, like the NEC V850E,

possessing a simple (scalar) pipeline and maybe a cache, one can analyze different hardware features separately, since there are no timing anomalies, and still achieve good results. Complicating factors are similar as for the simpler 8- and 16-bit processors, but also include varying access times due to cache hits and misses and varying pipeline overlap between instructions.

—More advanced processors, which possess many performance enhancing features that can influence each other, will exhibit timing anomalies. For these, timing-model construction is very complex. Also the analyses to be used are less modular and more complex [Heckmann et al. 2003].

In general, the execution-time bounds derived for an instruction depend on the states of the processor at this instruction. Information about the processor states is derived by analyzing potential execution histories leading to this instruction. Different states in which the instruction can be executed may lead to widely varying execution times with disastrous effects on precision. For instance, if a loop iterates 100 times, but the worst case of the body, e_{body}, only really occurs during one of these iterations and the others are considerably faster (say twice as fast), the over-approximation is $99 * 0.5 * e_{\text{body}}$. Precision can be gained by regarding execution in classes of execution histories separately, which correspond to *flow contexts*. These flow contexts essentially express by which paths through loops and calls control can arrive at the instruction. Wherever information about the processor's execution state is missing a conservative assumption has to be made or all possibilities have to be explored.

Most approaches use Data Flow Analysis, a static program-analysis technique based on the theory of Abstract Interpretation [Cousot and Cousot 1977]. These methods are used to compute invariants, one per flow context, about the processor's execution states at each program point. If there is one invariant for each program point, then it holds for all execution paths leading to this program point. Different ways to reach a basic block may lead to different invariants at the block's program points. Thus, several invariants could be computed. Each holds for a set of execution paths, and the sets together form a partition of the set of all execution paths leading to this program point. Each set of such paths corresponds to what sometimes is called a *calling context*, *context* for short. The invariants express static knowledge about the contents of caches, the occupancy of functional units and processor queues, and of states of branch-prediction units. Knowledge about cache contents is then used to classify memory accesses as definite cache hits (or definite cache misses). Knowledge about the occupancy of pipeline queues and functional units is used to exclude pipeline stalls. Assume that one uses the following method: First accept Murphy's Law, that everything that can go wrong, actually goes wrong, assuming worst cases all over. Then both types of "good news" of the type described above can often be used to reduce the upper bounds for the execution times. Unfortunately, this approach is not safe for many processor architectures with timing anomalies, see Section 2.1.3.

3.4 Estimate Calculation

The purpose is to determine an estimate for the WCET. In dynamic approaches the WCET estimate can underestimate the WCET, since only a subset of all executions

is used to compute it. Combining measurements of code snippets to end-to-end execution times can also overestimate the WCET, if pessimistic estimates for the snippets are combined. In static approaches this phase computes an upper bound of all execution times of the whole task based on the flow and timing information derived in the previous phases. It is then usually called *Bound Calculation*. There are three main classes of methods combining analytically determined or measured times to end-to-end estimates proposed in literature: *structure-based*, *path-based*, and techniques using *implicit path enumeration* (IPET).

Fig. 5 taken from [Ermedahl 2003] shows the different methods. Fig. 5(a) shows an example control-flow graph with timing on the nodes and a loop-bound flow fact.

In structure-based bound calculation as used in Heptane, cf. [Colin and Puaut 2000] and Section 6.6, an upper bound is calculated in a bottom-up traversal of the syntax tree of the task combining bounds computed for constituents of statements according to combination rules for that type of statement [Colin and Bernat 2002; Colin and Puaut 2000; Lim et al. 1995]. Fig. 5(d) illustrates how a structure-based method would proceed according to the task syntax tree and given combination rules. Collections of nodes are collapsed into single nodes, simultaneously deriving a timing for the new node. As stated in Section 2.1.2, precision can only be obtained if the same code snippet is considered in a number of different *flow contexts*, since the execution times in different flow contexts can vary widely. Taking flow contexts into account requires transformations of the syntax tree to reflect the different contexts. Most of the profitable transformations, e.g. loop unrolling, are easily expressed on the syntax tree [Colin and Bernat 2002].

Some problems of the structure-based approach are that not every control flow can be expressed through the syntax tree, that the approach assumes a very straightforward correspondence between the structures of the source and the target program not easily admitting code optimizations, and that it is in general not possible to add additional flow information as can be done in the IPET case.

In path-based bound calculation, the upper bound for a task is determined by computing bounds for different paths in the task, searching for the overall path with the longest execution time [Healy et al. 1999; Stappert and Altenbernd 2000; Stappert et al. 2001]. The defining feature is that possible execution paths are represented *explicitly*. The path-based approach is natural within a single loop iteration, but has problems with flow information extending across loop-nesting levels. The number of paths is exponential in the number of branch points, possibly requiring heuristic search methods.

Fig. 5(b) illustrates how a path-based calculation method would proceed over the graph in Fig. 5(a). The loop in the graph is first identified and the longest path within the loop is found. The time for the longest path is combined with flow information about the loop bound to extract an upper bound for the whole task.

In IPET, program flow and basic-block execution time bounds are combined into sets of arithmetic constraints. The idea was originally proposed in [Li and Malik 1995] and adapted to more complex flows and hardware timing effects in [Puschner and Schedl 1995; Engblom 2002; Theiling 2002a; Theiling 2002b; Ermedahl 2003]. Each basic block and program flow edge in the task is given a time coefficient (t_{entity}), expressing

the upper bound of the contribution of that entity to the total execution time every time it is executed and a count variable (x_{entity}), corresponding to the number of times the entity is executed. An upper bound is determined by maximizing the sum of products of the execution counts and times ($\sum_{i \in entities} x_i * t_i$), where the execution count variables are subject to constraints reflecting the structure of the task and possible flows. The result of an IPET calculation is an upper timing bound and a worst-case count for each execution count variable.

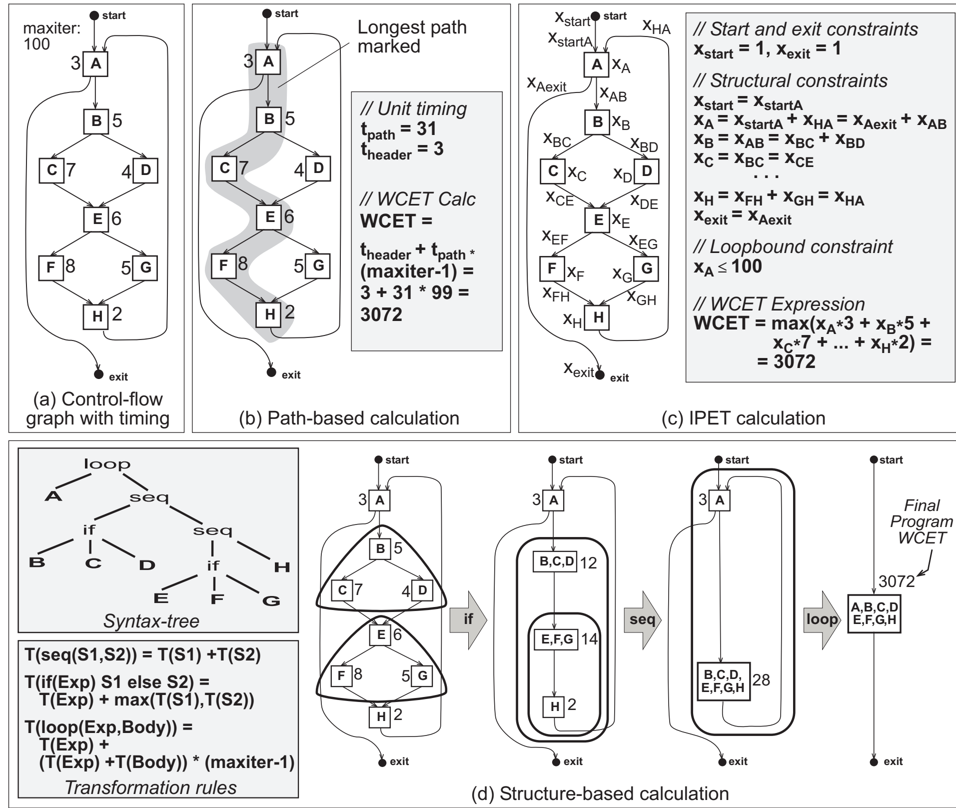

Fig. 5. Bound calculation

Fig. 5(c) shows the constraints and formulae generated by an IPET-based bound-calculation method for the task illustrated in Fig. 5(a). The *start* and *exit constraints* state that the task must be started and exited once. The *structural constraints* reflect the possible program flow, meaning that for a basic block to be executed it must be entered the same number of times as it is exited. The *loop bound* is specified as a constraint on the number of times the loop-head node `A` can be executed.

IPET is able to handle different types of flow information. It has traditionally been applied in a global fashion treating the whole task and all flow information

together as a unit. IPET-based bound calculation uses integer linear programming (ILP) or constraint programming (CP) techniques, thus having a complexity potentially exponential in the task size. Also, since flow facts are converted to constraints, the size of the resulting constraint system grows with the number of flow facts.

3.5 Symbolic Simulation

Another static method is to simulate the execution of the task in an abstract model of the processor. The simulation is performed without input. The simulator thus has to be capable to deal with partly unknown execution state. This method combines flow analysis, processor-behavior prediction, and bound calculation in one integrated phase [Lundqvist 2002]. One problem with this approach is that analysis time is proportional to the actual execution time of the task. This can lead to a very long analysis since simulation is typically orders of magnitudes slower than native execution.

4. MEASUREMENT-BASED METHODS

These methods attack some parts of the timing-analysis problem by executing the given task on the given hardware or a simulator, for some set of inputs, and measuring the execution time of the task or its parts.

End-to-end measurements of a subset of all possible executions produce estimates or distributions, not bounds for the execution times, if the subset is not guaranteed to contain the worst case. Even one execution would be enough if the worst-case input were known.

Other approaches measure the execution times of code segments, typically of CFG basic blocks. The measured execution times are then combined and analyzed, usually by some form of bound calculation, to produce estimates of the WCET or BCET. Thus, measurement replaces the processor-behaviour analysis used in static methods. Thus, the path-subset problem can be solved in the same way as for the static methods, using control flow analysis to find all possible paths and then using bound calculation to combine the measured times of the code segments into an overall time bound. This solution would include all possible paths, but would still produce unsafe results if the measured basic-block times were unsafe. Another problem is that only a subset of the possible contexts (initial processor states) is used for each measured basic block or other kind of code segment.

The context-subset problem could be attacked by running more tests to measure more contexts or by setting up a worst-case initial state at the start of each measured code segment. The first method (more tests) only decreases, but does not eliminate the risk of unsafe results and is expensive unless intensive testing is already done for other reasons. Exhaustive testing of all execution paths is usually impossible. The second method (use worst-case initial state) would be safe if one could determine a worst-case initial state. However, identifying worst-case initial states is hard or even impossible for complex processors, see below. Measurement-based tools can compute execution-time bounds for processors with simple timing behaviour, but produce only estimates of the BCET and WCET for more complex processors, as long as this problem is not convincingly solved. Other tools collect and analyse multiple measurements to provide a picture of the variability of the execution time

of the application, in addition to estimates of the BCET and WCET.

There are multiple ways in which measurement can be performed. The simplest approach is by extra instrumentation code that collects a timestamp or CPU cycle counter (available in most processors). Mixed HW/SW instrumentation techniques require external hardware to collect timings of lightweight instrumentation code. Fully transparent (non-intrusive) measurement mechanisms are possible using logic analyzers. Also hardware tracing mechanisms like the NEXUS standard and the ETM tracing mechanism from ARM are non-intrusive, but don't necessarily produce exact timings. For example, NEXUS buffers its output, and time stamps are produced when events leave the buffer, i.e. with a delay. Measurements can also be performed from the output of processor simulators or even VHDL simulators.

The results of measurement-based analysis can be used to provide a picture of the actual variability of the execution time of the application. They can also be used to provide validation for the static analyis approaches. Measurement should also not produce execution times that are far lower than the ones predicted by analytical methods, because this would indicate that the latter are imprecise.

5. COMPARISON OF STATIC AND MEASUREMENT-BASED METHODS

In this section, we attempt to compare the two classes of timing-analysis methods—static and measurement-based—to highlight the differences and similarities in their aims, abilities, technical problems and research directions. The next section will describe some timing-analysis tools in more detail to show the state of the practice of both classes of methods.

Static methods compute bounds on the execution time. They use control-flow analysis and bound calculation to cover all possible execution paths. They use abstraction to cover all possible context dependencies in the processor behaviour. The price they pay for this safety is the necessity for processor-specific models of processor behaviour, and possibly imprecise results such as overestimated WCET bounds. In favour of static methods is the fact that the analysis can be done without running the program to be analysed—which often needs complex equipment to simulate the hardware and peripherals of the target system.

Measurement-based methods replace processor behaviour analysis by measurements. Therefore, unless all possible execution paths are measured or the processor is simple enough to let each measurement be started in a worst-case initial state, some context-dependent execution-time changes may be missed and the method is unsafe. For the estimate-calculation step, these methods may use control-flow analysis to include all possible execution paths, or they may simply use the observed execution paths (observed number of loop iterations, for example) which again makes the method unsafe. The advantages claimed for these methods are that they are simpler to apply to new target processors, because they do not need to model processor behaviour, and that they produce WCET and BCET estimates that are more precise—closer to the exact WCET and BCET—than the bounds from static methods, especially for complex processors and complex applications.

Still, since the exact WCET or BCET is usually not known, there is really no way to check how precise an estimate or bound is. Studies of precision often compare the estimates or bounds not to the exact WCET or BCET but to the extreme

observed times from a large but not exhaustive set of tests.

Users can help to improve precision for both classes of methods. For the static methods, users can improve the precision (tighten the bounds) by annotations that exclude infeasible executions from the analysis. For the measurement-based methods, users can improve the precision by adding test cases to include more possible executions in the measurements. Some measurement-based tools also allow annotations for the estimate calculation to exclude infeasible executions or to include more executions by defining larger loop bounds than have been observed.

Both classes of methods share some technical problems and solutions. The front-ends are similar when both use executable code as input; control-flow analysis is similar; and bound/estimate calculation can be similar. For example, the IPET calculation is used by some static tools and by some measurement-based tools.

The main technical problem for static methods is modelling processor behaviour. This is not a problem for most measurement-based methods, where the main problem is to measure the execution time accurately, with fine granularity, and without perturbing the program being measured. The solution is often processor- or platform-specific, but implementing a measurement method for a new processor is usually less work than creating an abstract model of the processor behaviour.

The handling of timing anomalies offers an interesting comparison of the methods. For measurement-based methods, timing anomalies make it very hard to find a worst-case initial state for a measurement. To be safe, the measurement should now be done from all possible initial states, which is impractical. Measurement-based methods then use only a subset of initial states and so are not safe.

Static methods based on abstract interpretation have ways to express the absence of information and can therefore analyse large state sets, including all possible states for a safe analysis. Here, timing anomalies make it hard to define state abstractions that give a precise abstract interpretation of each instruction and of the execution time spent in the instruction—the processor behaviour tends to depend on unknown aspects of the state, forcing the abstract simulation to follow many possible executions of each basic block. Still, this laborious exploration is limited to basic blocks, because the abstract simulation considers the global flow of the task for the processor-behavior analysis by propagating simulation results between basic blocks. Thus, no worst case assumptions need to be made by the analysis on the basic block level.

Current research in these methods addresses some shared problems, while each class of methods also has its own research directions. Clearly, research into improved abstract processor models is relevant only to static methods, while the development of better measurement methods—in particular, standard interfaces for measurement for many processor types—is of interest mainly for the measurement-based methods.

Control-flow analysis, on the other hand, is a common subject of research that applies to both static methods and measurement-based meethods. Another common subject is the separation of contexts to improve the precision of the analysis. This means that a given part of the task under analysis, for example a subroutine or a loop body, can be analysed or measured separately depending on its context, for example the call-path to the subroutine, or the ieration number of the loop. Here,

the question that is common to static methods and measurement-based methods is when to separate between contexts and how—automatically or by manual annotations.

6. COMMERCIAL WCET TOOLS AND RESEARCH PROTOTYPES

The tool providers and researchers participating in this survey have received the following list of questions:

—What is the functionality of your tool?

—What methods are employed in your tool?

—What are the limitations of your tool?

—Which hardware platforms does your tool support?

This section has the following line-up of tools, from completely static tools such as aiT in Subsection 6.1, Bound-T in Subsection 6.2, and the prototypes of Florida (Subsection 6.3), Vienna (Subsection 6.4), Singapore (Subsection 6.5), and IRISA (Subsection 6.6), through mostly static tool with a small rudiment of measurement in SWEET (very controlled pipeline measurements on a simulator), in Subsection 6.7, and the Chalmers prototype (Subsection 6.8), through SymTA/P (cache analysis and block/segment measurement starting from a controlled cache state and bound calculation), in Subsection 6.9, to the most measurement-based tool, RapiTime (block measurement from an uncontrolled initial state and bound calculation), in Subsection 6.10.

6.1 The aiT Tool of AbsInt Angewandte Informatik, Saarbrücken, Germany

Functionality of the Tool. The purpose of AbsInt's timing-analysis tool aiT is to obtain upper bounds for the execution times of code snippets (e.g. given as subroutines) in executables. These code snippets may be tasks called by a scheduler in some real-time application, where each task has a specified deadline. aiT works on executables because the source code does not contain information on register usage and on instruction and data addresses. Such addresses are important for cache analysis and the timing of memory accesses in case there are several memory areas with different timing behavior.

Apart from the executable, aiT might need user input to be able to compute a result or to improve the precision of the result. User annotations may be written into parameter files and refer to program points by absolute addresses, addresses relative to routine entries, or structural descriptions (like the first loop in a routine). Alternatively, they can be embedded into the source code as special comments. In that case, they are mapped to binary addresses using the line information in the executable.

Apart from the usual user annotations (loop bounds, flow facts), aiT supports annotations specifying the values of registers and variables. The latter is useful for analyzing software running in several different modes that are distinguished by the value of a mode variable.

The aiT versions for all supported processors share a common architecture as shown in Fig. 6:

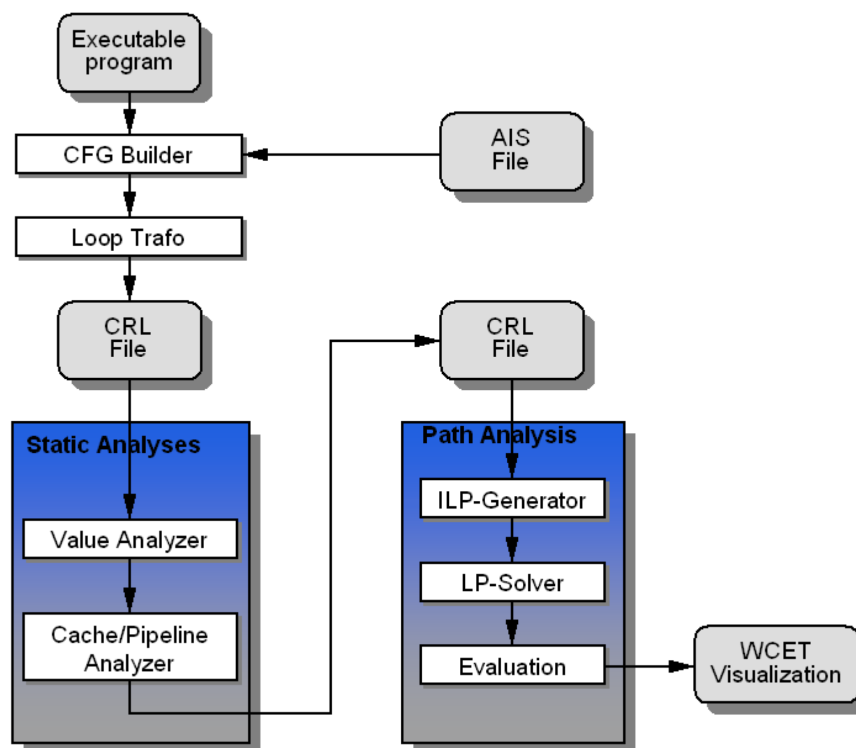

Fig. 6. Architecture of the aiT WCET analysis tool

—First, the control flow is reconstructed from the given object code by a bottom up approach. 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 the following analysis steps.

—Next, value analysis computes ranges for the values in the processor registers at every program point. Its results are used for loop bound analysis, for the detection of infeasible paths depending on static data, and to determine possible addresses of indirect memory accesses. An extreme case of control depending on static data is a virtual machine program interpreting abstract code given as data. [Souyris et al. 2005] report on a successful analysis of such an abstract machine.

—aiT's cache analysis relies on the addresses of memory accesses as found by value analysis and classifies memory references as sure hits and potential misses. It is based upon [Ferdinand and Wilhelm 1999], which handles LRU caches, but 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 deviation from perfect LRU is the reason for the reduced predictability of the cache contents in case of these two processors compared to processors with

perfect LRU caches [Heckmann et al. 2003].

—Pipeline analysis predicts the behavior of the task on the processor pipeline. The result is an upper bound for the execution time of each basic block in each distinguished execution context.

—Finally, bound calculation (called path analysis in the aiT framework) determines a worst-case execution path of the task from the timing information for the basic blocks.

Employed Methods. The structuring of the whole task of determining upper bounds into several phases allows different methods tailored to the subtasks to be used. In aiT's case, value analysis and cache/pipeline analysis are realized by abstract interpretation, a semantics-based method for static program analysis [Ferdinand and Wilhelm 1999; Ferdinand et al. 2001; Langenbach et al. 2002]. Path analysis is implemented by integer linear programming. Reconstruction of the control flow is performed by a bottom-up analysis [Theiling et al. 2000]. Detailed information about the upper bounds, the path on which it was computed, and the possible cache and pipeline states at any program point are attached to the call graph / control-flow graph and can be visualized in AbsInt's graph browser aiSee.

Limitations of the Tool. aiT includes automatic analysis to determine the targets of indirect calls and branches and to determine upper bounds of the iterations of loops. These analyses do not work in all cases. If they fail, the user has to provide annotations.

aiT relies on the standard calling convention. If some code doesn't adhere to the calling convention, the user might need to supply additional annotations describing control flow properties of the task.

Supported Hardware Platforms. Versions of aiT exist for the Motorola PowerPC MPC 555, 565, and 755, Motorola ColdFire MCF 5307, ARM7 TDMI, HCS12/STAR12, TMS320C33, C166/ST10, Renesas M32C/85 (prototype), and Infineon TriCore 1.3.

6.2 The Bound-T Tool of Tidorum, Helsinki, Finland

The Bound-T tool was originally developed at Space Systems Finland Ltd under contract with the European Space Agency (ESA) and intended for the verification of on-board software in spacecraft. Tidorum Ltd is extending Bound-T to other application domains.

Functionality of the Tool. The tool determines an upper bound on the execution time of a subroutine, including called functions. Optionally, the tool can also determine an upper bound on the stack usage of the subroutine, including called functions.

The input is a binary executable program with (usually) an embedded symbol table (debug information). The tool is able to compute upper bounds on some counter-based loops. For other loops the user provides annotations, called *assertions* in Bound-T. Annotations can also be given for variable values to support the automatic loop-bounding.

The output is a text file listing the upper bounds etc. and graph files showing call-graphs and control-flow graphs for display with the DOT tool [Gansner and North 2000].

As a further option, when the task under analysis follows the ESA-specified HRT ("Hard Real Time") tasking architecture, Bound-T can generate the HRT Execution Skeleton File that contains both the tasking structure and the computed bounds and can be fed directly into the ESA-developed tools for Schedulability Analysis and Scheduling Simulation.

Employed Methods. Reading and decoding instructions is hand-coded based on processor manuals. The processor model is also manually constructed for each processor. Bound-T has general facilities for modelling control flow and integer arithmetic, but not for modelling complex processor states. Some special-purpose static analyses have been implemented, for example for the SPARC register-file overflow and underflow traps and for the concurrent operation of the SPARC Integer Unit and Floating Point Unit. Both examples use (simple) abstract interpretation followed by ILP.

The control-flow graph (CFG) is often defined to model the processor's instruction-sequencing behaviour, not just the values of the program counter. A CFG node typically represents a certain pipeline state, so the CFG is really a pipeline-state graph. Instruction interactions (e.g. data-path blocking) are modelled in the time assigned to CFG edges.

Counter-based loops are bounded by modelling the task's loop-counter arithmetic as follows. The computational effect of each instruction is modelled as a relation between the "before" and "after" values of the variables (registers and other storage locations). The relation is expressed in Presburger Arithmetic as a set of affine (linear plus constant term) equations and inequalities, possibly conditional. Instruction sequences are modelled by concatenating (joining) the relations of individual instructions. Branching control-flow is modelled by adding the branch condition to the relation. Merging control-flow is modelled by taking the union of the inflowing relations.

Loops are modelled by analysing the model of the loop-body to classify variables as loop-invariant or loop-variant. The whole loop (including an unknown number of repetitions) is modelled as a relation that keeps the loop-invariant variables unchanged and assigns unknown values to the loop-variant variables. This is a first approximation that may be improved later in the analysis when the number of loop iterations is bounded. With this approximation, the computations in an entire subprogram can be modelled in one pass (without fixpoint iteration).

To bound loop iterations, Bound-T first re-analyses the model of the loop body in more detail to find loop-counter variables. A loop counter is a loop-variant variable such that one execution of the loop body changes the variable by an amount that is bounded to a finite interval that does not contain zero. If Bound-T also finds bounds on the initial and final values of the variable, a simple computation gives a bound on the number of loop iterations.

Bound-T uses the Omega Calculator from Maryland University [Pugh 1991] to create and analyze the equation set. Loop-bounds can be context-dependent if they depend on scalar pass-by-value parameters for which actual values are provided at the top (caller end) of a call-path.

The worst-case path and the upper bound for one subroutine are found by the Implicit Path Enumeration Technique, (see Section 3.4) applied to the control-flow graph of the subroutine. The lp_solve tool is used [Berkelaar 1997]. If the subroutine has context-dependent loop bounds, the IPET solution is computed separately for each context (call path).

Annotations are written in a separate text file, not embedded in source-code. The program element to which an annotation refers is identified by a symbolic name (subroutine, variable) or by structural properties (loops, calls). The structural properties include nesting of loops, location of calls with respect to loops, and location of variable reads and writes.

Limitations of the Tool. The task to be analyzed must not be recursive. The control-flow graphs must be reducible. Dynamic (indexed) calls are only analyzed in special cases, when Bound-T's data-flow analysis finds a unique target address. Dynamic (indexed) jumps are analyzed based on the code patterns that the supported compilers generate for switch/case structures, but not all such structures are supported.

Bound-T can detect some infeasible paths as a side-effect of its loop-bound analysis. There is, however, no systematic search for such paths. Points-to analysis (aliasing analysis) is weak, which is a risk for the correctness of the loop-bound analysis.

The bounds of an inner loop cannot depend on the index of the outer loop(s). For such "non-rectangular" loops Bound-T can often produce a "rectangular" upper bound. Loop-bound analysis does not cover the operations of multiplication (except by a constant), division or the logical bit-wise operations (and, or, shift, rotate).

The task to be analyzed must use the standard calling conventions. Furthermore, function pointers are not supported in general, although some special cases such as statically assigned interrupt vectors can be analyzed.

No cache analysis is yet implemented (the current target processors have no cache or very small and special caches). Any timing anomalies in the target processor must be taken into account in the execution time that is assigned to each basic block in the CFG. However, the currently supported, cacheless processors probably have no timing anomalies. As Bound-T has no general formalism (beyond the CFG) for modelling processor state, it has no general limitations in that regard, but models for complex processors would be correspondingly harder to implement in Bound-T.

Supported Hardware Platforms. Intel-8051 series (MCS-51), Analog Devices ADSP-21020, ATMEL ERC32 (SPARC V7), Renesas H8/300, ARM7 (prototype) and ATMEL AVR and ATmega (prototypes).

6.3 Research Prototype from Florida State University, North Carolina State University, Furman University

The main application areas for our timing analysis tools are hard real-time systems and energy-aware embedded systems with timing constraints. We are currently working on using our timing analyzer to provide QoS for soft real-time systems.

Functionality of the Tool. The toolset performs timing analysis of a single task or a subroutine.

A user interacts with the timing analyzer in the following manner. First, the user compiles all of the files that comprise the task. The compiler was modified to produce information used by the timing analyzer, which includes number of loop iterations, control flow, and instruction characteristics. The number of iterations for simple and non-rectangular loop nests are supported. The timing analyzer produces lower bounds and upper bounds for each function and loop in the task. This entire process is automatic.

Employed Methods. The tool uses data-flow analysis for cache analysis to make caching categorizations for each instruction [Arnold et al. 1994]. It supports direct-mapped and set-associative caches [Mueller 2000]. Control-flow analysis is used to distinguish paths at each loop and function level in the task [Arnold et al. 1994]. The pipeline is simulated to obtain the upper bound of each path, caching categorizations are used during this time so that pipeline stalls and cache-miss delays can be properly integrated [Healy et al. 1995]. The loop analysis iteratively finds the worst-case path until the caching behavior reaches a fixed point that is guaranteed to remain the same [Arnold et al. 1994; Mueller 2000]. Loop bounds analysis is performed in the compiler to obtain the number of iterations for each loop. The timing analyzer is also able to address non-rectangular loop nests, which is modeled in terms of summations [Healy et al. 2000]. Parametric timing analysis support is also provided for run-time bound loops by producing a bounds formula parameterized on loop bounds rather than cycles [Vivancos et al. 2001]. Branch constraint analysis is used to tighten the predictions by disregarding paths that are infeasible [Healy and Whalley 2002]. A timing tree is used to evaluate the task in a bottom-up fashion. Functions are distinguished into instances so that caching categorizations for each instance can be separately evaluated [Arnold et al. 1994].

Limitations of the Tool. Loop bounds for numeric timing analysis are required to be statically known, or there has to be a known loop bound in the outer loop in a non-rectangular loop nest. Loop bounds need not be statically known when using parametric timing analysis support. Like most other timing analysis tools, no support is provided for pointer analysis or dynamic allocation. No calls through pointers are allowed since the call graph must be explicit to analyze the task. We also do not allow recursion since we do not currently provide any support to automatically determine the maximum number of recursive calls that can be made in a cyclic call graph. We provide limited data cache support wrt. access patterns [White et al. 1999]. We also provide limited support for data cache analysis for array accesses in loop nests using cache miss equations [Ramaprasad and Mueller 2005]. The timing analyzer is only able to determine execution-time bounds of applications on simple RISC/CISC architectures. The tool has limited scalability in terms of analyzing small codes in seconds, medium-sized codes in minutes. But entire systems may take hours/days, which we do not deem feasible. Scalability depends on the system/target device and is less of a problem with 8-bit systems, but a more significant problem with 32-bit systems.

Supported Hardware Platforms. The hardware platforms include a variety of uniprocessors (multiprocessors should be handled in schedulability analysis). These

include the MicroSPARC I, Intel Pentium, StarCore SC100, PISA/MIPS, and Atmel Atmega [Anantaraman et al. 2003; Mohan et al. 2005]. Experiments have been performed with the Force MicroSPARC I VME board. The timing analyzer WCET predictions have been validated on the Atmel Atmega to cycle-level accuracy [Mohan et al. 2005].

6.4 Research Prototypes of TU Vienna

The TU Vienna real-time group has developed a number of tool prototypes for experimenting with different approaches to execution-time analysis. Three of these are presented in this paper: The first is a prototype tool for *static timing analysis* that has been integrated into a Matlab/Simulink tool chain and can analyze C code or Matlab/Simulink models. Second we present a *measurement-based tool* that uses genetic algorithms to direct input-data generation for timing measurements in the search for the worst case or long program execution times. The third tool is a *hybrid tool* for timing analysis that uses both measurements and elements from static analysis to assess the timing of C code.

6.4.1 *TU Vienna Research Prototype for Static Analysis.*

Functionality of the Tool. The timing analysis for C programs performs timing analysis for software coded in WCETC, where WCETC is a subset of C with extensions that allow users or tool components for flow analysis to make annotations about (in)feasible execution paths [Kirner 2002]. The tool cooperates with a C compiler. The compiler translates the WCETC code into object code and some information for the WCET analyzer. This object code is then analyzed to compute an upper bound. Back annotation of bound information for single statements, entire functions, and tasks are possible.

A component of the static tool has been built into the Matlab/Simulink tool chain. This component generates code from the block set that includes all path annotations necessary for timing analysis, i.e., there is no need to perform any additional flow analysis or annotate the code. In that way the tool supports fully automatic timing analysis for the complete Matlab/Simulink block set defined within the European IST project SETTA, see [Kirner et al. 2002]. In addition, the tool supports back annotation of detailed execution-time information for individual Matlab/Simulink blocks and entire Matlab/Simulink application models.

Employed Methods. A number of adaptations had been made to the Matlab/Simulink tool chain. First, the code-generation templates used by the target language compiler (TLC) were modified. The templates for our block set were changed so that TLC generates Code with WCETC macros instead of pure C code. Second, a GNU C compiler was adapted to translate WCETC code and cooperate with the WCET analyzer. The modified C compiler uses abstract co-interpretation of path information during code translation in order to trace changes in the control structure as made by code optimization. This co-transformation of path information is the key to keeping path information consistent in optimizing compilers, thus facilitating timing analysis of highly optimized code [Kirner and Puschner 2003; Kirner 2003]. It computes execution-time bounds for the generated code by means of integer linear programming adding information about infeasible execution paths [Puschner and Schedl 1997].

Back annotation of timing-analysis results to the Matlab/Simulink specification level is done via dedicated WCET blocks that represent time bounds for blocks and entire tasks.

Limitations of the Tool. If used together with the SETTA Matlab/Simulink block set, the static WCET tool provides a fully automated timing analysis, i.e., there is no need for annotations or help from the user to calculate execution-time bounds. So there are no limitations besides the fact that the programmer must only use blocks from the block set.

In case the tool is to be used to analyze C code it may be necessary to annotate the code with information about (in)feasible paths (either by using a flow-annotation tool or by providing manual annotations). In the latter case the quality of the computed bounds strongly depends on the quality of the annotations.

Supported Hardware Platforms. M68000, M68360, and C167.

6.4.2 *TU Vienna Research Prototype for Measurement-based Analysis.*

Functionality of the Tool. The measurement-based tool for dynamic execution-time analysis yields optimistic approximations to the worst-case execution times of a piece of code.

Employed Methods. Genetic algorithms are used to generate input data for execution-time measurements [Puschner and Nossal 1998] as follows: At the beginning, the task or program under observation is run with a number of random input-data sets. For each of the input data sets the execution time is measured and stored. The execution-time values are then used as fitness values for their respective input data sets (i.e., longer execution times imply higher fitness). The fitness values, in turn, form the basis for the generation of a new population of input-data sets by the genetic algorithms. The GA-based input-data generation strategy is repeated until the termination criterion as specified for the analysis is reached [Atanassov et al. 1999].

Limitations of the Tool. GA-based bounds search can in general not guarantee to produce safe results, as measurement-based techniques approach the upper bound from the side of lower execution times.

Supported Hardware Platforms. The targets supported include the C167, and PowerPC processors.

6.4.3 *TU Vienna Research Prototype for Hybrid Analysis.*

Functionality of the Tool. The hybrid timing analysis tool combines static program analysis techniques and execution time measurements to calculate an estimate of the WCET. The main features of the tool are the automatic segmentation of the program code into segments of reasonable size and the automatic generation of test data used to measure the execution times of all subpaths within each program segment. The tool has been designed with a special focus on analyzing automatically generated code, e.g., code generated from Matlab/Simulink models.

The architecture of the hybrid analysis tool is given in Fig. 7. The tool takes a C program as input, partitions the program into code segments, and extracts path

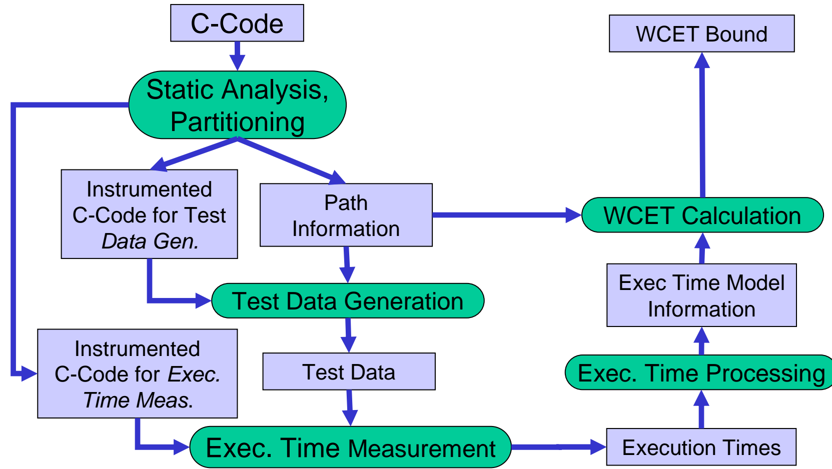

Fig. 7. Architecture of the TU-Vienna hybrid timing analysis tool

information that is used during the final bounds calculation to identify infeasible paths within the program. Automatic test-data generation is used to derive the required input data for the execution-time measurements. The measurements are typically performed remotely on the real target hardware. Measurement results are processed to construct a timing model specific to the analyzed program, which is used together with the path information to calculate a WCET estimate.

Employed Methods. The central technique of the hybrid timing-analysis tool is the automatic test-data generation used to derive a WCET estimate by means of execution time measurements. To approach a full subpath coverage within each program segment, a formal test-data generation method based on model checking is used [Wenzel et al. 2005a; Wenzel et al. 2005b]. However, to compensate the high computation cost of formal test-data generation, a three-step approach is used:

(1) Random search is used to generate the majority of test data. The path coverage of these test data is derived using an automatically, entirely instrumented version of the program.
(2) Heuristic search methods like genetic algorithms allow to improve the segment coverage already achieved by step 1.
(3) The remaining test data are generated using formal test-data generation based on *model checking* [Clarke et al. 1999]. The formal model of the program needed for the model checker is automatically derived from the source code. Code optimizations have to be performed to improve the performance of the formal test-data generation [Wenzel et al. 2005a]. This approach provides for a given subpath in a program segment either the test data to trigger its execution or the information that this subpath is infeasible.

Based on the measurement results and the path information of the program, the overall WCET estimate is calculated using integer linear programming.

Limitations of the Tool. As the tool performs static program analysis at the source code level, it has to be assured that the compiler does not significantly change the structure of the program code. The hybrid analysis tool guarantees path coverage for each program segment, therefore every execution scenario is covered by the analysis. However, the tool does not provide state coverage, thus the WCET estimate, though being quite precise, is not guaranteed to be a safe upper bound of the execution times for complex processors having pipelines or caches. Furthermore, the calculation of the tool supports program-flow annotations only at the granularity of entire program segments.

Supported Hardware Platforms. The targets supported currently include the HCS12, and Pentium processors.

However, as the hybrid approach of the tool does not rely on a model of a processor, it is relatively easy to adapt it to other processors. To port the tool to a new processor one has to modify the instrumentation code used to measure execution times and provide a mechanism to communicate the measurement results to the host computer.

6.5 The Chronos Research Prototype from National University of Singapore

Chronos[2] is an open-source static WCET analysis tool.

Functionality of the Tool. The purpose of Chronos is to determine a tight upper bound for the execution times of a task running on a modern processor with complex micro-architectural features. The input to Chronos is a task written in C and the configuration of the target processor. The frontend of the tool performs data flow analysis to compute loop bounds. If it fails to obtain certain loop bounds, user annotations have to be provided. The user may also input infeasible-path information to improve the accuracy of the results. The frontend maps this information from the source code to the binary executable.

The core of the analyzer works on the binary executable. It disassembles the executable to generate the control flow graph (CFG) and performs processor-behavior analysis on this CFG. Chronos supports the analysis of (i) out-of-order pipelines, (ii) various dynamic branch prediction schemes, (iii) instruction caches, and the interaction among these different features to compute a tight upper bound on the execution times.

Employed Methods. Chronos employs several innovative techniques to efficiently obtain safe and tight upper bounds on execution times.

The core of the analyzer determines upper bounds of execution times of each basic block under various execution contexts such as correctly predicted or mispredicted jump of the preceding basic blocks and cache hits/misses within the basic block [Li 2005]. Determining these bounds is challenging for out-of-order processor pipelines due to the presence of timing anomalies. It requires the costly enumeration of all possible schedules of instructions within a basic block. Chronos avoids this enumeration via a fixed-point analysis of the time intervals (instead of concrete time instances) at which the instructions enter/leave different pipeline

[2] Chronos, according to Greek mythology, is the personification of time.

stages [Li et al. 2004].

Next, the analyzer employs Integer Linear Programming (ILP) to bound the number of executions corresponding to each context. This is achieved by bounding the number of branch mispredictions and instruction cache misses. Here ILP is used to accurately model branch prediction, instruction cache as well as their interaction [Li et al. 2005]. The analysis of branch prediction is generic and parameterizable w.r.t. the commonly used dynamic branch prediction schemes including GAg and gshare [Mitra et al. 2002]. Instruction caches are analyzed using the ILP-based technique proposed by Li, Malik, and Wolfe [Li et al. 1999]. However, the integration of cache and branch prediction requires analyzing the constructive and destructive timing effects due to cache blocks being "prefetched" along the mispredicted paths. This complex interaction is accounted for in the analyzer [Li et al. 2003] by suitably extending the instruction cache analysis.

Finally, bounds calculation is implemented by the IPET technique (see Section 3.4) by converting the loop bounds and user provided infeasible-path information to linear flow constraints.

Limitations of the Tool. Chronos currently does not analyze data caches. Since the focus is mainly on processor-behavior analysis, the tool performs limited data flow analysis to compute loop bounds. The tool also requires user feedback for infeasible program paths.

Supported Hardware Platforms. Chronos supports the processor model of SimpleScalar [Austin et al. 2002] `sim-outorder` simulator, a popular cycle-accurate micro-architectural simulator. The tool deliberately targets a simulated processor model so that the processor can be easily configured with different pipeline, branch prediction, and instruction cache options. This allows the user to evaluate the efficiency, scalability, and accuracy of the WCET analyzer for various processor configurations without requiring the actual hardware. Moreover, the source code of the entire tool is publicly available allowing the user to easily extend and adapt it for new architectural features and estimation techniques.

6.6 The Heptane tool of IRISA, Rennes

Heptane is an open-source static WCET analysis tool released under GPL license.

Functionality of the Tool. The purpose of Heptane is to obtain upper bounds for the execution times of C programs by a static analysis of their code (source code and binary code). The tool analyses the source and/or binary format depending on the calculation method the tool is parameterized to work with.

Employed Methods. Heptane embeds in the same analysis tool a timing schema-based and an ILP-based method for bound calculation (see Sections 3.4 and 2.3.4). The former method produces quickly safe, albeit in some circumstances overestimated upper bounds for a code snippet, while the latter requires more computing power, but yields tighter results. The two calculation methods cannot be used simultaneously on fragments of the same task. The timing-schema method operates on the task's syntactic structure, which is extracted from the source code. The ILP-based method exploits the task's control-flow graph extracted from the task's binary.

Finding the upper bound of a loop requires the knowledge of the maximum number of loop iterations. Heptane requires the user to give this information through *symbolic annotations* in the source program. Annotations are designed to support non-rectangular and even non-linear loops (nested loops whose number of iterations arbitrarily depends on the counter variables of outer loops) [Colin and Puaut 2000]. The final bound is computed using an external evaluation tool (Maple and Maxima for the computation method based on timing schemata, lp_solve and CPLEX for the method based on ILP).

Heptane integrates mechanisms to take into account the effect of *instruction caches*, *pipelines* and *branch prediction*.

—Pipelines are tackled by an off-line simulation of the flow of instructions through the pipelines.

—An extension of Frank Mueller's so-called *static cache simulation* [Mueller 2000], based on data flow analysis is implemented in the tool. It classifies every instruction according to its worst-case behavior with respect to the instruction cache. Instruction categories take into account loop nesting levels. The Heptane tool takes as input the memory map of the code (cacheable vs. uncacheable code regions, address range of scratchpad memory, if any) as well as the contents of locked cache regions if a cache locking mechanism is used [Puaut and Decotigny 2002].

—An approach derived from static cache simulation is used to integrate the effect of branch predictors based on a cache of recently taken branches [Colin and Puaut 2000].

The modeling of the *instruction cache*, *branch predictor* and *pipeline* produce results expressed in a microarchitecture-independent formalism [Colin and Puaut 2001a], thus allowing Heptane to be easily modified or retargeted to a new architecture.

Limitations of the Tool

—No automatic flow analysis (loop bounds are given manually as annotations at the source code level), no detection of mutually exclusive or infeasible paths, resulting in pessimistic upper bounds for some tasks (e.g. [Colin and Puaut 2001b]).

—The bound-calculation method based on timing schemata currently does not support compiler optimizations that cause a mismatch between the task's syntax-tree and control flow graph.

—No support for data cache analysis.

—Limited number and types of target processors (currently limited to scalar processors with in-order execution) and only the gcc compiler.

Supported Hardware Platforms. Heptane is designed to produce timing information for in-order monoprocessor architectures (Pentium1 - accounting for one integer pipeline only, StrongARM 1110, Hitachi H8/300, and MIPS as a virtual processor with an overly simplified timing model).

6.7 SWEET (SWEdish Execution Time tool)

SWEET was previously developed in Mälardalen University, C-Lab in Paderborn, and Uppsala University. The development has now fully moved to Mälardalen University [**?**], with a main research focus on the flow analysis.

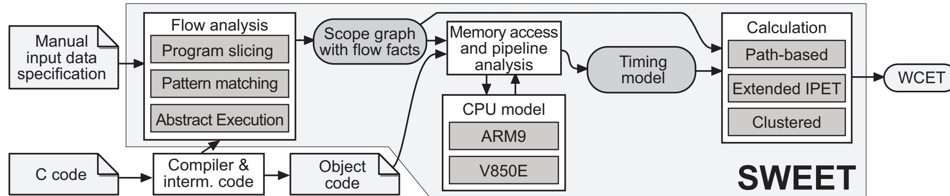

Fig. 8. Architecture of the SWEET timing-analysis tool

Functionality of the Tool. SWEET has been developed in a modular fashion, allowing for different analyses and tool parts to work rather independently [Gustafsson 2000; Engblom 2002; Ermedahl 2003]. The tool architecture of SWEET is shown in Fig. 8. In essence, it conforms to the general scheme for WCET analysis presented in Section 2, consisting of three major phases: a *flow analysis*, a *processor-behavior analysis* and an *estimate calculation*. The analyses communicate through two well-defined data structures, the *scope graph with flow facts* [Ermedahl 2003], (representing the result of the flow analysis), and the *timing model* [Engblom 2002], (representing the result of the processor-behavior analysis). In essence, SWEET offers the following functionality:

—Automatic flow analysis on the intermediate code level.

—Integration of flow analysis and a research compiler.

—Connection between flow analysis and processor-behavior analysis.

—Instruction cache analysis for level one caches.

—Pipeline analysis for medium-complexity RISC processors.

—A variety of methods to determine upper bounds based on the results of flow- and pipeline analysis.

Employed Methods. Unlike most WCET analysis tools, SWEET's flow analysis is integrated with a research compiler. The flow analysis is performed on the intermediate code (IC) of the compiler, after structural optimizations. Thus, the control structure of the IC and the object code is similar, and the flow analysis results for the IC are valid for the object code as well.

SWEET's flow analysis is based on a multi-phase approach. A *program slicing* is used to restrict the flow analysis to only those parts of the program that may affect the program flow [?]. A value analysis, see Subsection 3.1, combined with *pattern-matching* catches "easy cases" such as simple loops. More complicated codes are handled by the *abstract execution* [Gustafsson 2000; Gustafsson et al. 2005], a form of symbolic execution based on abstract interpretation. The analysis uses abstract interpretation to derive safe bounds on variables values at different points in the program. However, rather than using traditional fixed-point iteration [Cousot and Cousot 1977], loops are "rolled out" dynamically and each iteration is analysed individually in a fashion similar to symbolic execution. The abstract execution is able to automatically calculate both loop bounds and infeasible path information.

SWEET's processor-behavior analysis is highly decoupled from the flow analysis, and based on a two-phase approach. In the first phase, the *memory access analysis*, memory areas accessed by different instructions are determined. If the

target hardware has an instruction cache, an instruction-cache analysis similar to [Ferdinand et al. 1999] is also performed. The result of the analysis is a set of "execution facts" which are used in the pipeline analysis. Such facts specify the memory area(s) that an instruction may reference or if the instruction may hit and/or miss the cache. Execution facts can also specify other factors like assumptions on branch prediction outcomes, and the precise set available depends on the target processor.

The *pipeline analysis* is performed by simulating object code sequences through a trace-driven cycle-accurate CPU model. The CPU model needs to be controllable so that the execution facts can correctly be accounted for in each instruction, and the instruction trace followed as provided (branch instructions have to follow the trace, for example). The execution facts are used to enforce worst-case timing behaviour of the instructions. For data dependent instructions the worst-case timing is assumed, unless execution facts specify otherwise. The pipeline analysis has been explicitly designed to allow standard CPU simulators to be used as CPU models. However, this requires that the simulator is clock-cycle accurate, can be forced to perform its simulation according to given instruction sequences and corresponding execution facts, and does not suffer from timing anomalies.

Consecutive simulation runs starting with the same basic block in the code are combined to find timing effects across sequences of two or more blocks in the code [Engblom 2002]. The analysis assumes that there is a known upper bound on the length of block sequences that can exhibit timing effects; this value can be greater than two even on quite simple processors.

SWEET's estimate calculation phase support three different type of calculation techniques, all taking the same two data structures as input. A *fast path-based* technique [Stappert et al. 2001; Ermedahl 2003; **?**], a *global IPET* technique [Ermedahl 2003], and a hybrid *clustered* technique [Ermedahl et al. 2005; Ermedahl 2003]. The clustered calculation can perform both local IPET and/or local path-based calculations (the decision on what to use is based on the flow information available for the specific program part under analysis).

SWEET uses DOT from GraphViz [Gansner and North 2000] to graphically visualize its results.

Limitations of the Tool. Each part of the tool, flow analysis, processor-behavior analysis, and bound calculation have their individual limitations.

The flow analysis can handle ANSI-C programs including pointers, unstructured code, and recursion. However, to make use of the automatic flow analysis the program must be compiled with the research compiler that SWEET is integrated with, otherwise flow facts must be manually given. There are also some limitations inherent to the used research compiler, e.g. the use of dynamically allocated memory is currently not supported, and annotations will be needed in such cases.

The current memory access analysis does not handle data caches. Only one-level instruction caches are supported. The pipelines that are amenable to SWEET's pipeline analysis are limited to in-order pipelines with bounded long-timing effects and no timing anomalies. In particular, out-of-order pipelines are not handled.

The path-based bound calculation requires that the code of the task is well-structured. The IPET-based and clustered calculation methods can handle arbi-

trary task graphs.

Supported Hardware Platforms. SWEET's processor-behavior analysis currently supports the ARM9 and the NEC V850E. The V850E model has been validated against actual hardware, which the ARM9 has not.

6.8 Research Prototype from Chalmers University of Technology

This tool is a research prototype that was developed to evaluate new concepts for the determination of execution-time bounds for tasks executed on high-performance microprocessors featuring pipelining and multi-level caching techniques.

Functionality of the Tool. The developed tool is capable of automatically deriving safe upper bounds for tasks' binaries using a subset of the Power-PC instruction set architecture. It is sometimes possible to derive the exact WCET, in case the worst-case input of the task is known.

Additional functionality:

- Integration of path and timing analysis through symbolic cycle-level execution of the task on a detailed architectural simulation model extended to handle unknown input data values [Lundqvist and Stenström 1999b; Lundqvist 2002].
- Binary code transformation techniques that eliminate timing anomalies, which may occur when tasks are run on processors where instructions are scheduled dynamically [Lundqvist and Stenström 1999c; Lundqvist 2002].
- A data cache analysis method that can identify data structures that can be safely cached - called predictable - so as to improve the worst-case cache performance. Under this method, data structures with unpredictable access patterns are identified and tagged as non-cacheable [Lundqvist and Stenström 1999a].
- A method that can determine the worst-case data-cache performance for data accesses to predictable data structures whose exact location in the address space is statically unknown. This method can be applied to improve worst-case cache performance for e.g. procedures whose input parameters are pointers to data structures whose access patterns are predictable, but whose locations in the memory space are not known until run-time [Lundqvist 2002].

Employed Methods. Simulation models have been used for some time to estimate the execution time for tasks on future architectures making use of advanced pipelining and caching techniques. One such example is the SimpleScalar toolset [Austin et al. 2002]. A key advantage of this approach is that it is possible to derive arbitrarily accurate estimations of the execution time for a task for a given set of input data and assuming sufficient detail of the timing model of the architectural simulator. Unfortunately, as the input data is unknown for a WCET analyzer, such simulation models cannot be used off-hand.

Concepts have been developed that leverage on the accuracy of architectural timing models to make tight, but still safe estimates of the execution-time bounds. One key concept developed is cycle-level symbolic program simulation. First, loop bounds, branch conditions etc., which are not input-data dependent will be calculated as the task is executed symbolically on the architectural simulator. However, in order to handle unknown input data, the instruction-set sim-

ulator was extended with the capability of handling unknown parts of execution states. For example, if a branch condition depends on unknown input data, both paths are executed. Inevitably, this may result in the well-known path explosion problem in program loops with a large number of iterations. A path-merging approach excludes the paths that may not be part of the worst-case execution path through the task. The analysis that determines which paths to exclude must take into account what timing effect they may have in the future. This involves analysis of worst-case performance taking microarchitectural features such as pipelining and caching into account. Methods have been developed and integrated to do this analysis for a range of architectural features including multi-level instruction and data caches [Lundqvist and Stenström 1999b; Lundqvist and Stenström 1999a] and multiple-issue pipelines with dynamic instruction scheduling [Lundqvist and Stenström 1999c]. How often path merging is carried out is a tradeoff between computational speed and accuracy of the analysis and is investigated in [Lundqvist 2002]. The described symbolic execution of tasks has been shown to exclude some infeasible paths thereby making the execution-time bounds tighter.

Limitations of the Tool. The user has to provide annotations for loops with unknown termination properties.

An inherent limitation of the approach is the computational complexity of the analysis. While the path-merging method partly addresses this concern, it also introduces over-estimation of the WCET. Therefore, we believe that this approach is most relevant in the final stage of the development cycle, when the design space has been narrowed down so that one can afford the long-running simulations for limited aspects of the entire system.

Supported Hardware Platforms. Our tool currently supports a fairly rich subset of the PowerPC instruction set architecture. We have also integrated architectural timing models for some implementations of the PowerPC featuring dual-instruction issue, a dynamically scheduled pipeline, a parameterized cache hierarchy with split first-level set-associative instruction and data caches and a unified second-level cache. The cache and block size as well as the associativity of each cache is parameterized.

6.9 SymTA/P Tool of TU Braunschweig, Germany

Functionality of the Tool. The purpose of SymTA/P is to obtain upper and lower execution time bounds of C programs running on microcontrollers. SymTA/P is an acronym of SYMbolic Timing Analysis for Processes. The key idea of SymTA/P is to combine platform independent path analysis on source code level and platform dependent measurement methodology on object code level, using an actual target system. The main benefit is that this hybrid analysis can easily be re-targeted to a new hardware platform.

The execution time measurement can be obtained by an off-the-shelf cycle-accurate processor simulator or by an evaluation board. Instruction cache as well as data cache behavior is analyzed for a single uninterrupted task execution. Furthermore, the cache related preemption delay for fixed priority preemptive real-time systems for direct mapped and associative caches is analyzed and integrated in a

cache-aware response time analysis. To be efficient, cache analysis requires that a task execution trace can be generated that is uninterrupted by cache misses, either because the task is small enough to fit in the cache or because the evaluation system offers sufficient on-chip fast memory, e.g. scratchpad RAM. If that is not available, then an appropriate simulator must be used instead.

Employed Methods. The hybrid approach consists of a static program path analysis and a measurement for the execution time of program segments. In most static analyses of execution times, a basic block has been assumed as the smallest entity. However often a program consists of a single feasible execution path (SFP) only. Such a SFP is a sequence of basic blocks where the execution sequence is invariant to input data [Wolf 2002; Wolf et al. 2001; Ye and Ernst 1997].

SymTA/P uses symbolic analysis on the abstract syntax tree to identify such single feasible paths (SFP) at the source code level. The result is a control-flow graph with nodes containing single feasible paths or basic blocks that are part of a multiple feasible path. A single feasible path can reach beyond basic block boundaries, for example, a fast Fourier transformation or a FIR filter. In these cases, the program contains loops with several if-then-else statements which are input independent. This means that the branch direction depends only on local variables with known values, for example the loop iteration count. Therefore the entire loop represents an SFP and is represented by a single node. The main benefit of SFPs is a smaller number of instrumentation points.

In a second step, the execution time for each node is estimated. Off-the-shelf processor simulators or standard cost-efficient evaluation boards can be used. The C source code is instrumented with measurement points that mark the beginning and the end of each node. Such a measurement point is linked to a platform dependent measurement procedure, such as accessing the system clock or internal timer. For a processor simulator the instrumentation can use a processor debugger command to stop the simulation and store the internal system clock. Then, the entire C-program with measurement points is compiled, linked and executed on the evaluation board (or simulated on the processor simulator). During this measurement, a safe initial state cannot be assured in all cases. Therefore an additional time delay is added that covers a potential underestimation during such a measurement.

At this step, SymTA/P assumes that each memory access takes constant time. For timing analysis, input data for a complete branch coverage must be supplied. A complete branch coverage means, that with a given set of input data all branches and other C statements are at least executed once. This criterion requires the user to specify a considerably fewer number of input data than a full path coverage because combinations of execution paths need not be considered. This advantage comes with the drawback that it adds a conservative overhead to cover pipelining effects between nodes.

The constant memory access time assumption is revised by analyzing the instruction-cache and the data-cache behavior [Wolf et al. 2002]. When using a processor simulator, the memory access trace for each node is generated using a similar methodology as the execution time measurements. The traced memory accesses are annotated to the corresponding node in the control flow graph. A data flow analysis technique is used to propagate the information which cache lines are available at

each node. Pipelining effects between nodes are not directly modeled, instead a conservative overhead corresponding to starting with an empty pipeline is assumed.

The longest and shortest path in the control flow graph are found by IPET (see Section 3.4) and Fig. 5. The time for each node is given by the measured execution time and the statically analyzed cache access behavior. This framework has also been used to calculate the power consumption of a program [Wolf et al. 2002]. Loop bounds have to be specified by the user, if the loop condition is input dependent.

If preemptive scheduling is used, cache blocks might be replaced by higher priority tasks. SymTA/P considers the cache behavior of the preempted and preempting task to compute the maximum cache related preemption delay. This delay is considered in cache-aware response time analysis [Staschulat and Ernst 2004] [Staschulat et al. 2005].

SymTA/P uses a static analysis approach for data cache behavior, which combines symbolic execution to identify input dependent memory accesses and uses integer linear programming to bound the worst case data cache behavior for input dependent memory accesses [Staschulat and Ernst 2006].

Limitations of the Tool. The measurement on an evaluation board is more accurate if the program paths between measurements points are longer. If many basic blocks are measured individually the added time delays to cover pipelining effects would lead to an overestimation of the total worst case execution time. Data-dependent execution times of single instructions are not explicitly considered. It is assumed that input data covers the worst case regarding data-dependent instruction execution time. Input data has to be provided that generates complete branch coverage. Such patterns are usually available from functional test. The precision of the final analysis depends on the measurement environment. Especially for evaluation boards the interference of the measurement instrumentation has to be a constant factor to obtain sufficiently accurate results. Currently, the approach assumes a sequential memory access behavior where the CPU stalls during a memory access.

Supported hardware platforms. C programs on the following micro-controllers have been analyzed: ARM architectures (e.g. ARM9), TriCore, StrongARM, C167, and i8051. The software power analysis has been applied to SPARClite. Furthermore, an open interface for processor simulators and a measurement framework for evaluation boards is provided.

6.10 The RapiTime tool of Rapita Systems Ltd., York, UK

RapiTime aims at medium to large real-time embedded systems on advanced processors. The RapiTime tool targets the automotive electronics, avionics and telecommunications industries.

Functionality of the Tool. Rapitime is a measurement-based tool, i.e., it derives timing information of how long a particular section of code (generally a basic block) takes to run from measurements. Measurement results are combined according to the structure of the program to determine an estimate for the longest path through the program,

RapiTime not only computes an estimate for the WCET of a program as a single (integer) value, but also the whole probability distribution of the execution time

of the longest path in the program (and other subunits). This distribution has a bounded domain (an absolute upper bound, the WCET estimate, and a lower bound).

The input of RapiTime is either a set of source files (C or Ada) or an executable. The user also has to provide test data from which measurements will be taken. The output is a browsable HTML report with description of the WCET prediction and actual measured execution times, split for each function and subfunction.

Timing information is captured on the running system by either a software instrumentation library, a lightweight software instrumentation with external hardware support, purely non-intrusive tracing mechanisms (like Nexus and ETM) or even traces from CPU simulators.

The user can add annotations in the code to guide how the instrumentation and analysis process will be performed, to bound the number of iterations of loops, etc.

The RapiTime tool supports various architectures, adapting the tool for new architectures requires porting the object code reader (if needed) and determining a tracing mechanism for that system.

RapiTime is the commercial quality version of the pWCET tool developed at the Real-Time Systems Research Group at the University of York.

Employed Methods. The RapiTime tool is structure-based and works on a tree representation of the program. The structure is derived from either the source code or from the direct analysis of executables.

The timing of individual blocks is derived from extensive measurements extracted from the real system. RapiTime not only computes the maximum of the measured times but whole probability distributions [Bernat et al. 2002; Bernat et al. 2005]. The WCET estimates are computed using an algebra of probability distributions.

The timing analysis of the program can be performed on different contexts, therefore allowing to analyze, for instance, each different call to a function individually. The level of detail and how many contexts are analyzed is controlled by annotations. RapiTime also allows to analyze different loop iterations by virtually unrolling loops. For each of these loop contexts, loop bounds are derived from actual measurements (or annotations).

Limitations of the Tool. The RapiTime tool does not rely on a model of the processor, so in principle it can model any processing unit (even with out-of-order execution, multiple execution units, various hierarchies of caches etc). The limitation is put on the need to extract execution traces which require some code instrumentation and a mechanism to extract these traces from the target system. Regarding source code level, RapiTime cannot analyze programs with recursion and with non-statically analyzable function pointers.

Supported Hardware Platforms. Motorola processors (including MPC555, HCS12, etc), ARM, MIPS, NecV850.

7. EXPERIENCE

Three commercial WCET tools are available, aiT, Bound-T, and RapiTime. There are extensive reports about industrial use [Thesing et al. 2003; Sandell et al. 2004; Souyris et al. 2005; Byhlin et al. 2005; Holsti et al. 2000b; Holsti et al. 2000a]. In

fact, tools are under routine use in the aeronautics and the automotive industries. They enjoy very positive feedback concerning speed, precision of the results, and usability. The aeronautics industry uses them in the development of the most safety-critical systems, e.g. the fly-by-wire systems for the Airbus A380. The used WCET tool will be qualified as verification tool in the sense of the international avionics standard for safety-critical software, RTCA/DO-178B (Software Considerations in Airborne Systems and Equipment Certification Requirements) for Level A Code.

There are not many published benchmark results about timing-analysis tools. [Lim et al. 1995] is a study done by the authors of a method carefully explaining the reasons for over-estimation. [Thesing et al. 2003; Souyris et al. 2005] report experiences made by developers. The developers are experienced, and the tool is integrated into the development process. We summarize these three comparable benchmarks in Table I. They seem to exhibit a surprising paradox, benchmarks published earlier offer better results regarding the degree of overestimation, although significant methodological progress has been made in the meantime. The reason lies in the fact that the advancement of processor architectures and in particular the divergence of processor and memory speeds have made the challenge of timing analysis harder. Both have increased the timing variability and thus the penalty for the lack of knowledge in analysis results. Let's take the cache-miss penalty as an example, the single cause with highest weight. In [Lim et al. 1995], published in 1995, a cache-miss penalty of 4 cycles was assumed. In [Thesing et al. 2003], a cache-miss penalty of roughly 25 was given, and finally in the setting described in [Souyris et al. 2005], the cache-miss penalty was between 60 internal cycles for a worst-case access to an instruction in SDRAM and roughly 200 internal cycles for an access to data over the PCI bus. Thus, any overestimation should be considered in the context of the given architecture. An overestimation of 30% in 2005 as reported in [Souyris et al. 2005] means a huge progress compared to an overestimation of 30% reported in 1995 [Lim et al. 1995]! On the other hand, static methods are capable of predicting exact WCETs on simple microcontrollers without caches, deep pipelines, and speculation.

The Mälardalen University WCET-research group has performed several industrial WCET case-studies as M.Sc. Thesis projects using the SWEET [Carlsson et al. 2002] and aiT [Sandell et al. 2004; Byhlin et al. 2005; Eriksson 2005; Zhang 2005; Sehlberg 2005] tools. The students were experts neither on the used timing-analysis tool nor on the analyzed system. However, they were assisted both by WCET analysis experts from academia and industrial personel with detailed system knowledge.

The case-studies show that it is possible to apply static WCET analysis to a variety of industrial systems and codes. The tools used performed well and derived safe upper timing bounds. However, detailed knowledge of the analyzed code and many manual annotations were often required to achieve reasonably tight bounds. These annotations were necessary to give the analysis enough knowledge about program flow constraints and in some cases constraints on addresses of memory accesses. A higher degree of automation and support from the tools, e.g. automatic loop bounds calculation, would in most cases have been desirable.

The case-studies also show that single timing bounds, covering all possible scenarios, are not always what you want. For several analyzed systems it was more

Reference	Year	Cache-miss penalty	Overestim.
[Lim et al. 1995]	1995	4	20-30%
[Thesing et al. 2003]	2002	25	15%
[Souyris et al. 2005]	2005	60 for accessing instructions in SDRAM 200 for access over PCI bus	30-50%

Table I. Cache-miss penalties and degrees of overestimation.

interesting to have different WCET bounds for different running modes or system configurations rather than a single WCET bound. The latter would in most cases have been a gross overapproximation. In some cases, it was also possible to manually derive a parametrical formula [Sandell et al. 2004; Byhlin et al. 2005], showing how the WCET estimate depends on some specific system parameters.

The case-studies were done for processors without cache. The overestimations were mostly in the range 5-15% as compared with measured times. Measurements were in some cases done with emulators, and in some cases directly on the hardware using oscilloscopes and logical analyzers.

Performance and Size. Table IV in Section 9 lists the maximal size of tasks analyzed by the different tools. They vary between 10 kByte and 80 kByte of code. Analysis times vary widely depending on the complexity of the processor and its periphery and the structure and other characteristics of the software. Analysis of a task for a simple microprocessor such as the C166/ST10 may finish in a few minutes. Analysis of a complex software, an abstract machine and the code interpreted by it, and a complex processor has been shown to take in the order of a day, see [Souyris et al. 2005].

Also of interest is the size of the abstract processor models underlying some static analysis approaches. They range from 3000 to 11000 lines of C code. This C code is the result of a translation from a formal model.

8. LIMITATIONS OF THE TOOLS

Determining safe and precise bounds on execution times is a very difficult problem, undecidable in general as is known, but still very complex for programs with bounded iteration and recursion. There are several features whose use will easily ruin precision. Among these are pointers to data and to functions that cannot statically be resolved, and the use of dynamically allocated data. Most tools will expect that function calling conventions are observed. Some tools forbid recursion. Currently, only mono-processor targets are supported. Most tools only consider uninterrupted execution of tasks.

9. TOOL COMPARISON

This section shows in a condensed form the most relevant information about the different tools. The following abbreviations for the Vienna tools are used, Vienna M. for Vienna Measurement, Vienna S. for Vienna Static, and Vienna H. for Vienna Hybrid. Table II lists the methods used for the different subtasks. The methods are the ones described in Section 2. The abbreviation *n.a.* stands for *not applicable*, while a dash (-) is used when no such method or analysis is employed.

Table III describes which architectural features, e.g., caches and pipelines, may

Tool	Flow	Proc. Behavior	Bound Calc.
aiT	value analysis	static program analysis	IPET
Bound-T	linear loop-bounds and constraints by Omega test	static program analysis	IPET per function
RapiTime	n.a.	measurement	structure-based
SymTA/P	single feasible path analysis	static program analysis for I/D cache, measurement for segments	IPET
Heptane	-	static prog. analysis	structure-based, IPET
Vienna S. Vienna M. Vienna H.	- Genetic Algorithms Model Checking	static program analysis segment measurements segment measurements	IPET n.a. IPET
SWEET	value analysis, abstract execution, syntactical analysis	static program analysis for instr. caches, simulation for the pipeline	path-based, IPET-based, clustered
Florida		static program analysis	path-based
Chalmers		modified simulation	
Chronos		static prog. analysis	IPET

Table II. Analysis methods employed

Tool	Caches	Pipeline	Periphery
aiT	I/D, direct/set associative, LRU, PLRU, pseudo round robin	in-order/out-of-order	PCI bus
Bound-T	-	in-order	-
RapiTime	n.a.	n.a.	n.a.
SymTA/P	I/D, direct/set-associative, LRU	n.a.	n.a.
Heptane	I-cache, direct, set associative, LRU, locked caches	in-order	-
Vienna S. Vienna M. Vienna H.	jump-cache n.a. n.a.	simple in-order n.a. n.a.	- n.a. n.a.
SWEET	I-cache, direct/set associative, LRU	in-order	-
Florida	I/D, direct/set associative	in-order	-
Chalmers	split first-level set-associative, unified second-level cache	multi-issue superscalar	-
Chronos	I-cache, direct, LRU	in-order/out-of-order, dyn. branch prediction	-

Table III. Support for architectural features

be present in processor architectures for which instances of the tools are available. Instruction caches are supported by most of the tools. However, data caches need a resolution of effective memory addresses at analysis time. This is currently not supported by many tools. Of particular interest is, whether only in-order execution pipelines are supported. Out-of-order execution almost unavoidably introduce timing anomalies [Lundqvist and Stenström 1999c], which require integrated analyses of the cache and the pipeline behavior.

Table IV gives additional information such as the language level of the analyzed software, how the results are presented, the size of the biggest programs analyzed,

Tool	Lang. level	Result repr.	Max anal.	Integration
aiT	object	text, graphical	80 KByte	some versions adapted to code generated by STATEMATE, Ascet/SD, Scade, MATLAB/Simulink, or Targetlink
Bound-T	object	text, graphical	30 KByte	stack-extent analysis, HRT Schedulability Analyzer
RapiTime	source (C, Ada), object	graphical, html based	50 KLOC	Matlab/Simulink, Targetlink, SimpleScalar simulator
SymTA/P	source (C)	text, graphical	7KLOC	Tasking Compiler, Schedulability analysis (cache-related preemption delay)
Heptane	source, object	graphical	14 KLOC	
Vienna S.	source, object	text, graphical		Matlab/Simulink, optimizing compiler
Vienna M.	object	text		
Vienna H.	source, object	text, graphical		Matlab/Simulink, Targetlink
SWEET	flow analysis on intermediate, proc. beh. anal. on object	text, graphical		IAR compiler, SUIF compiler, LCC compiler
Florida	object			cycle-accurate simulators, power-aware schedulers, compiler
Chalmers	object			
Chronos	object	graphical	10 KByte	GCC compiler, SimpleScalar simulator

Table IV. Additional features. The first column gives the language level of the systems analyzed. The second column shows how the results are presented to the user. The third column gives the maximum size of tasks analyzed in one run. The total size of the analyzed system may be larger. The last column lists other tools integrated with the timing-analysis tool.

etc.

Table V lists the hardware platforms that have been targeted with the tools.

10. INTEGRATION WITH OTHER TOOLS

Timing Analysis can be performed as an isolated activity. However, most of the time it is done for a subsequent schedulability analysis, whose job it is to check, whether the given set of tasks can all be executed on the given hardware satisfying all their constraints. Timing analysis should be integrated with such a schedulablity analysis for improved results, because schedulability analysis can profit from information about context-switch costs incurred by cache and pipeline damages that result from preemption. Information about the amount of damage can be made available by WCET tools.

On the other hand, timing precision can profit from integration of WCET tools with compilers. Compilers have semantic information that is hard to recover from the generated code. If this information were passed on to the WCET tool, the precision could be increased. For instance, the integration with the compiler and

Tool	Hardware platform
aiT	Motorola PowerPC MPC 555, 565, and 755, Motorola ColdFire MCF 5307, ARM7 TDMI, HCS12/STAR12, TMS320C33, C166/ST10, Renesas M32C/85, Infineon TriCore 1.3
Bound-T	Intel-8051, ADSP-21020, ATMEL ERC32, Renesas H8/300, ATMEL AVR, ATmega, ARM7
RapiTime	Motorola PowerPC family, HCS12 family, ARM, NecV850, MIPS3000
SymTA/P	various ARM (RealView Suite), TriCore, i8051, C167
Heptane	Pentium1, StrongARM 1110, Hitachi H8/300
Vienna S.	M68000, M68360, C167
Vienna M.	C167, PowerPC
Vienna H.	HCS12, Pentium
SWEET	ARM9 core, NEC V850E
Florida	MicroSPARC I, Intel Pentium, StarCore SC100, Atmel Atmega, PISA/MIPS
Chalmers	PowerPC
Chronos	SimpleScalar out-of-order processor model with MIPS-like instruction-set-architecture (PISA)

Table V. Supported hardware platforms

linker would be desirable to supply the WCET tool with the possible targets for dynamic calls and jumps and the possible memory locations for dynamic data accesses (pointer analysis). A standard format for this information, perhaps embedded in the linked executable, would be preferable to a real-time interaction between these tools. A closer integration between the WCET tool and the compiler is desirable, so that the compiler can utilize feedback about temporal properties of code from the WCET tool in order to identify the best code optimization strategy for each section of code it generates.

For automatically synthesized code, integration with the semantic information available on the model level would be very helpful. Timing information about dynamically called library functions is necessary to bound the time for their calls.

The current symbol-table structures in executable files are also an important practical problem, although trivial in theory. The symbol-table structures are poorly defined and differ across compilers and targets.

It is obvious that a source-file browsing tool should be integrated with the WCET tool, to help the user to understand the results and to control the WCET tool. The browsing tool may, in turn, need to interact with a version-control tool.

An exciting extension is to integrate a WCET tool into a tool for the performance analysis of distributed and communication-centric systems.

11. CONCLUSIONS

The problem of determining upper bounds on execution times for single tasks and for quite complex processor architectures has been solved. Several commercial WCET tools are available and have experienced very positive feedback from extensive industrial use. This feedback concerned speed, precision of the results, and usability. Several research prototypes are under development.

11.1 Remaining problems and Future Perspectives

What significant problems or novel directions is timing analysis facing?

Tool	Contact person	Contact information
aiT	Christian Ferdinand AbsInt Angewandte Informatik GmbH Science Park 1 D-66123 Saarbrücken Germany	Email: info@AbsInt.com Phone: +49 681 383 60 0 Fax: +49 681 383 60 20 Web: www.AbsInt.com
Bound-T	Niklas Holsti Tidorum Ltd Tiirasaarentie 32 FI-00200 Helsinki Finland	Email : info@tidorum.fi Phone: +358 (0) 40 563 9186 Web: www.tidorum.fi Web: www.bound-t.com
RapiTime	Guillem Bernat Rapita Systems Ltd. IT Center, York Science Park Heslington York YO10 5DG United Kingdom	Email: enquiries@rapitasystems.com Phone: +44 1904 567747 Fax: +44 1904 567719 Web: www.rapitasystems.com
SymTA/P	Rolf Ernst, Jan Staschulat Institute for Computer and Communication Network Engineering Technical University Braunschweig Hans-Sommer-Str. 66 D-38106 Braunschweig, Germany	Email: ernst|staschulat@ida.ing.tu-bs.de Phone: +49 531 391 3730 Fax: +49 531 391 3750 Web: www.ida.ing.tu-bs.de/research/projects/symta
Heptane	Isabelle Puaut IRISA, ACES Research Group Campus univ. de Beaulieu 35042 Rennes Cédex, France	Email: puaut@irisa.fr Phone: +33 02 99 84 73 10 Fax: +33 02 99 84 25 29 Web: www.irisa.fr/aces/work/heptane-demo/heptane.htm l
Vienna	Peter Puschner Inst. für Technische Informatik TU Wien A-1040 Wien, Austria	Email: peter.puschner@tuwien.ac.at Phone: 01 58 801 18 227 Fax: 01 58 69 149
SWEET	Björn Lisper Mälardalen University P.O. Box 883 SE-721 23 SE 72123 Västerås Sweden	Email: bjorn.lisper@mdh.se Email: andreas.ermedahl@mdh.se Email: jan.gustafsson@mdh.se Phone: +46 21 151 709 Web: www.mrtc.mdh.se/projects/wcet/
Florida	David Whalley Florida State University Frank Mueller North Carolina State University Chris Healy Furman University, USA	Email: whalley@cs.fsu.edu Email: mueller@cs.ncsu.edu Email: chris.healy@furman.edu
Chalmers	Per Stenström Department of Computer Engineering Chalmers University of Technology S-412 96 Göteborg, Sweden	Email: pers@ce.chalmers.se Phone: +46 31 772 1761 Fax: +46 31 772 3663
Chronos	Tulika Mitra, Abhik Roychoudhury, Xianfeng Li, National University of Singapore	Email: tulika@comp.nus.edu.sg, Email: abhik@comp.nus.edu.sg Email: lixianfe@comp.nus.edu.sg www.comp.nus.edu.sg/˜rpembed/chronos

Table VI. Contact information for the tools.

Increased support for flow analysis. Most problems reported in timing-analysis case studies relate to setting correct loop-bounds and other flow annotations. Stronger static program analyses are needed to extract this information from the software.

Verification of abstract processor models. Static timing-analysis tools based on abstract processor models may be incorrect if these models are not correct. Strong guarantees for correctness can be given by equivalence checking between different abstraction levels. Ongoing research activities attempt the formal derivation of abstract processor models from concrete models. Progress in this area will not only improve the accuracy of the abstract processor models, but also reduce the effort to produce them. Measurement-based methods can also be used to test the abstract models.

Integration of timing analysis with compilation. An integration of static timing analysis with a compiler can provide valuable information available in the compiler to the timing analysis and thereby improve the precision of analysis results.

Integration with scheduling. Preemption of tasks causes large context-switch costs on processors with caches; the preempting task may ruin the cache contents, such that the preempted task encounters considerable cache-reload costs when resuming execution. These context-switch costs may be even different for different combinations of preempted and preempting task. These large and varying context-switch costs violate assumptions underlying many real-time scheduling methods. A new scheduling approach considering the combination of timing analysis and preemptive scheduling will have to be developed. SymTA/P provides an integration of WCET calculation and cache-related preemption delay for schedulability analysis [Staschulat et al. 2005].

Interaction with energy awareness. This may concern the trade off between speed and energy consumption. [Jayaseelan et al. 2006] presents a static analysis technique to estimate the worst-case energy consumption of a task on complex micro-architectures. Estimating a bound on energy is non-trivial as it is unsafe to assume any direct correlation with the bound on execution time. On the other hand, information computed for WCET determination, e.g., cache behavior, is of high value for the determination of energy consumption [Seth et al. 2003].

Design of systems with time-predictable behavior. This is a particularly well-motivated research direction, because several trends in system design make systems less and less predictable [Thiele and Wilhelm 2004].

There are first proposals in this area. [Anantaraman et al. 2003] propose a virtual simple architecture (VISA). A VISA is the pipeline timing specification of a hypothetical simple processor. Upper bounds for execution times are derived for a task assuming the VISA. At run-time, the task is executed speculatively on an unsafe complex processor, and its progress is continuously gauged. If continued safe progress appears to be in jeopardy, the complex processor is reconfigured to a simple mode of operation that directly implements the VISA, thereby explicitly bounding the task's overall execution time. Progress is monitored at intra-task checkpoints, which are derived from upper bounds and unsafe predictions of execu-

tion times for subtasks [Anantaraman et al. 2004]. Hence, VISA shifts the burden of bounding the execution times of tasks, in part, to hardware by supporting two execution modes within a single processor. [Puschner 2005] proposes to transform sets of program paths to single paths by predicate conversion as in code generation for processors with predicated execution. The disadvantage is the loss in performance resulting from the need to execute predicated paths that would originally not be executed.

Any proposal increasing predictability plainly at the cost of performance will most likely not be accepted by developers.

Extension to component-based design. Timing-Analysis methods should be made applicable to component-based design and systems built on top of real-time operating systems and using real-time middleware.

11.2 Architectural Trends

The hardware used in creating an embedded real-time system has a great effect on the ease of predictability of the execution time of programs.

The simplest case are traditional 8-bit and 16-bit processors with simple architectures. In such processors, each instruction basically has a fixed execution time. Such processors are easy to model from the hardware timing perspective, and the only significant problem in WCET analysis is how to determine the program flow.

There is also a class of processors with simple in-order pipelines, which are found in cost-sensitive applications requiring higher performance than that offered by classic 8-bit and 16-bit processors. Examples are the ARM7 and the recent ARM Cortex R series. Over time, these chips can be expected to replace the 8-bit and 16-bit processors for most applications. With their typically simple pipelines and cache structures, relatively simple and fast WCET hardware analysis methods can be applied.

At the high-end of the embedded real-time spectrum, performance requirements for applications like flight control and engine control force real-time systems designers to use complex processors with caches and out-of-order execution. Examples are the PowerPC 750, PowerPC 7448, and ARM11 families of processors. Analyzing such processors requires more advanced tools and methods, especially in the hardware analysis.

The mainstream of computer architecture is steadily adding complexity and speculative features in order to push the performance envelope. Architectures such as the AMD Opteron, Intel Pentium 4 and Pentium-M, and IBM Power5 use multiple threads per processor core, deep pipelines, and several levels of caches to achieve maximum performance on sequential programs.

This mainstream trend of ever-more complex processors is no longer as dominant as it used to be, however. In recent years, several other design alternatives have emerged in the mainstream, where the complexity of individual processor cores has been reduced significantly.

Many new processors are designed by using several simple cores instead of a single or a few complex cores. This design gains throughput per chip by running more tasks in parallel, at the expense of single-task performance. Examples are the Sun Niagara chip which combines 8 in-order four-way multithreaded cores on a

single chip [Olukotun and Hammond 2005], and the IBM-designed PowerPC for the Xbox 360, using three two-way multithreaded in-order cores [Krewell 2005]. These designs are cache-coherent multiprocessors on a chip, and thus have a fairly complex cache- and memory system. The complexity of analysis moves from the behavior of the individual cores to the interplay between them as they access memory.

Another very relevant design alternative is to use several simple processors with private memories (instead of shared memory). This design is common in mobile phones, where you typically find an ARM main processor combined with one or more DSPs on a single chip. Outside the mobile phone industry, the IBM-Sony-Toshiba Cell processor is a high-profile design using a simple in-order PowerPC core along with eight synergistic processing elements (SPEs) [Hofstee 2005]. The Cell will make its first appearance in the Sony PlayStation 3 gaming console, but IBM and Mercury Computing systems are pushing the Cell as a general-purpose real-time processor for high-performance real-time systems. The SPEs in the Cell are designed for predictable performance, and use local program-controlled memories rather than caches, just like most DSPs. Thus, this type of architecture provides several easy-to-predict processors on a chip as an alternative to a single hard-to-predict processor.

Field-programmable gate arrays (FPGAs) are another design alternative for some embedded applications. Several processor architectures are available as “soft cores” that can be implemented in an FPGA together with application-specific logic and interfaces. Such processor implementations may have application-specific timing behavior which may be challenging for off-the-shelf timing analysis tools, but they are also likely to be less complex and thus easier to analyze than general-purpose processors of similar size. Likewise, some standard processors are now packaged together with FPGA on the same chip for implementing application-specific logic functions. The timing of these FPGA functions may be critical and need analysis, separately or together with the code on the standard processor.

There is also work on application-specific processors or application-specific extensions to standard instruction sets, again creating challenges for timing analysis. Here there is also an opportunity for timing analysis: to help find the application functions that should be speeded up by defining new instructions.

ACKNOWLEDGMENTS

ARTIST (Advanced Real-Time Systems) is an IST project of the EU. Its goals are to coordinate the research and development effort in the area of Advanced Real-time Systems. ARTIST established several Working Groups, one of them on *Timing Analysis*, under the direction of Reinhard Wilhelm. The cluster includes the leading European WCET researchers and tool vendors. The goal of the cluster is to combine the best components of existing European timing-analysis tools. This working group has collected industrial requirements for WCET tools by asking (potential) users of this technology to fill out a questionnaire. The results were condensed into an article that has appeared in the proceedings of the EUROMICRO WCET workshop 2003 [Wilhelm et al. 2003].

The same working group has questioned tool providers and researchers about their commercial and research tools, respectively. This article contains an intro-

duction into the methods for the determination of execution-time bounds and a survey of the existing tools.

Many thanks go to the anonymous reviewers for extensive and very helpful comments on the first version of the paper. These comments led to substantial improvements of the presentation.

REFERENCES

Anantaraman, A., Seth, K., Patil, K., Rotenberg, E., and Mueller, F. 2003. Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-Time Systems. In *International Symposium on Computer Architecture.* 250–261.

Anantaraman, A., Seth, K., Patil, K., Rotenberg, E., and Mueller, F. 2004. Enforcing Safety of Real-Time Schedules on Contemporary Processors using a Virtual Simple Architecture (VISA). In *Proceedings of the IEEE Real-Time Systems Symposium.* 114–125.

Arnold, R., Mueller, F., Whalley, D., and Harmon, M. 1994. Bounding Worst-Case Instruction Cache Performance. In *Proc. of the IEEE Real-Time Systems Symposium.* Puerto Rico, 172–181.

Atanassov, P., Haberl, S., and Puschner, P. 1999. Heuristic Worst-Case Execution Time Analysis. In *Proc. 10th European Workshop on Dependable Computing.* Austrian Computer Society (OCG), 109–114.

Austin, T., Larson, E., and Ernst, D. 2002. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer 35,* 2.

Berkelaar, M. 1997. lp solve: A Mixed Integer Linear Program Solver. Tech. rep., Eindhoven University of Technology.

Bernat, G., Colin, A., and Petters, S. M. 2002. WCET Analysis of Probabilistic Hard Real–Time Systems. In *Proceedings of the 23rd Real-Time Systems Symposium RTSS 2002.* Austin, Texas, USA, 279–288.

Bernat, G., Newby, M., and Burns, A. 2005. Probabilistic timing analysis: an approach using copulas. *Journal of Embedded Computing 1,* 2, 179–194.

Byhlin, S., Ermedahl, A., Gustafsson, J., and Lisper, B. 2005. Applying Static WCET Analysis to Automotive Communication Software. In *Proc. 17th Euromicro Conference of Real-Time Systems, (ECRTS'05).* 249–258.

Carlsson, M., Engblom, J., Ermedahl, A., Lindblad, J., and Lisper, B. 2002. Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system. In *Proc. 2nd International Workshop on Real-Time Tools (RT-TOOLS'2002).*

Chvatal, V. 1983. *Linear Programming.* Freeman.

Clarke, E. M., Grumberg, O., and Peled, D. A. 1999. *Model Checking.* MIT Press.

Colin, A. and Bernat, G. 2002. Scope-tree: a Program Representation for Symbolic Worst-Case Execution Time Analysis. In *Proc. 14th Euromicro Conference of Real-Time Systems, (ECRTS'02).* 50–59.

Colin, A. and Puaut, I. 2000. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *The Journal of Real-Time Systems 18,* 2-3 (May), 249–274.

Colin, A. and Puaut, I. 2001a. A Modular and Retargetable Framework for Tree-based WCET Analysis. In *Proc. of the 13th Euromicro Conference on Real-Time Systems.* Delft, The Netherlands, 37–44.

Colin, A. and Puaut, I. 2001b. Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System. In *Proc. of the 13th Euromicro Conference on Real-Time Systems.* Delft, The Netherlands, 191–198.

Cousot, P. and Cousot, R. 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, 238–252.

Desikan, R., Burger, D., and Keckler, S. 2001. Measuring Experimental Error in Microprocessor Simulation. In *Proc. 28th International Symposium on Computer Architecture (ISCA 2001)*. ACM Press.

Engblom, J. 2002. Processor Pipelines and Static Worst-Case Execution Time Analysis. Ph.D. thesis, Uppsala University, Dept. of Information Technology, Box 337, Uppsala, Sweden.

Eriksson, O. 2005. Evaluation of Static Time Analysis for CC Systems. M.S. thesis, Mälardalen University, Västerås, Sweden.

Ermedahl, A. 2003. A Modular Tool Architecture for Worst-Case Execution Time Analysis. Ph.D. thesis, Uppsala University, Dept. of Information Technology, Box 325, Uppsala, Sweden. ISBN 91-554-5671-5.

Ermedahl, A., Stappert, F., and Engblom, J. 2005. Clustered Worst-Case Execution- Time Calculation. *IEEE Transactions on Computers 54,* 9 (9).

Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., and Wilhelm, R. 2001. Reliable and Precise WCET Determination for a Real-Life Processor. In *EMSOFT*. LNCS, vol. 2211. 469–485.

Ferdinand, C., Martin, F., and Wilhelm, R. 1999. Cache Behavior Prediction by Abstract Interpretation. *Science of Computer Programming 35*, 163 – 189.

Ferdinand, C. and Wilhelm, R. 1999. Fast and Efficient Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems 17*, 131 – 181.

Gansner, E. R. and North, S. C. 2000. An Open Graph Visualization System and its Applications to Software Engineering. *Software — Practice and Experience 30,* 11, 1203–1233.

Graham, R. L. 1966. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal 45*, 1563–1581.

Gustafsson, J. 2000. Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation. Ph.D. thesis, Department of Computer Systems, Information Technology, Uppsala University.

Gustafsson, J., Ermedahl, A., and Lisper, B. 2005. Towards a Flow Analysis for Embedded System C Programs. In *Proc. 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005)*. Sedona, USA.

Gustafsson, J., Lisper, B., Sandberg, C., and Bermudo, N. 2003. A Tool for Automatic Flow Analysis of C-programs for WCET Calculation. In *8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003), Guadalajara, Mexico.*

Healy, C., Arnold, R., Müller, F., Whalley, D., and Harmon, M. 1999. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers 48,* 1 (Jan).

Healy, C., Sjödin, M., Rustagi, V., and Whalley, D. 1998. Bounding Loop Iterations for Timing Analysis. In *Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98).*

Healy, C., Sjodin, M., Rustagi, V., Whalley, D., and van Engelen, R. 2000. Supporting Timing Analysis by Automatic Bounding of Loop Iterations. *The Journal of Real-Time Systems*, 121–148.

Healy, C. and Whalley, D. 1999. Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99).* 79–88.

Healy, C. and Whalley, D. 2002. Automatic Detection and Exploitation of Branch Constraints for Timing Analysis. *IEEE Transactions on Software Engineering*, 763–781.

Healy, C., Whalley, D., and Harmon, M. 1995. Integrating the Timing Analysis of Pipelining and Instruction Caching. In *Proc. of the IEEE Real-Time Systems Symposium.* Pisa, Italy, 288–297.

Heckmann, R., Langenbach, M., Thesing, S., and Wilhelm, R. 2003. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *IEEE Proceedings on Real-Time Systems 91,* 7, 1038–1054.

Hofstee, P. 2005. Power Efficient Processor Architecture and The Cell Processor. In *11th Symposium on High-Performance Computing Architecture (HPCA-11).*

HOLSTI, N., LÅNGBACKA, T., AND SAARINEN, S. 2000a. Using a Worst-Case Execution-Time Tool for Real-Time Verification of the DEBIE Software. In *Proc. DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457)*.

HOLSTI, N., LÅNGBACKA, T., AND SAARINEN, S. 2000b. Worst-Case Execution-Time Analysis for Digital Signal Processors. In *Proc. EUSIPCO 2000 Conference (X European Signal Processing Conference)*.

JAYASEELAN, R., MITRA, T., AND LI, X. 2006. Estimating the worst-case energy consumption of embedded software. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

KIRNER, R. 2002. The Programming Language WCETC. Tech. rep., Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria.

KIRNER, R. 2003. Extending Optimising Compilation to Support Worst-Case Execution Time Analysis. Ph.D. thesis, Technische Universität Wien, Vienna, Austria.

KIRNER, R., LANG, R., FREIBERGER, G., AND PUSCHNER, P. 2002. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. In *Proc. 14th Euromicro International Conference on Real-Time Systems*. 31–40.

KIRNER, R. AND PUSCHNER, P. 2003. Transformation of Meta-Information by Abstract Co-Interpretation. In *Proc. 7th International Workshop on Software and Compilers for Embedded Systems*. Vienna, Austria, 298–312.

KREWELL, K. 2005. IBM Speeds Xbox 360 to Market. *Microprocessor Report*.

LANGENBACH, M., THESING, S., AND HECKMANN, R. 2002. Pipeline Modelling for Timing Analysis. In *Static Analysis Symposium SAS 2002*, M. V. Hermenegildo and G. Puebla, Eds. Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, 294–309.

LI, X. 2005. Microarchitecture Modeling for Timing Analysis of Embedded Software. Ph.D. thesis, School of Computing, National University of Singapore.

LI, X., MITRA, T., AND ROYCHOUDHURY, A. 2003. Accurate Timing Analysis by Modeling Caches, Speculation and their Interaction. In *ACM Design Automation Conf. (DAC)*.

LI, X., MITRA, T., AND ROYCHOUDHURY, A. 2005. Modeling Control Speculation for Timing Analysis. *Journal of Real-Time Systems 29,* 1.

LI, X., ROYCHOUDHURY, A., AND MITRA, T. 2004. Modeling Out-of-Order Processors for Software Timing Analysis. In *IEEE Real-Time Systems Symposium (RTSS)*.

LI, Y.-T. S. AND MALIK, S. 1995. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proc. of the 32:nd Design Automation Conference*. 456–461.

LI, Y.-T. S., MALIK, S., AND WOLFE, A. 1995a. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proceedings of the IEEE Real-Time Systems Symposium*. 298–307.

LI, Y.-T. S., MALIK, S., AND WOLFE, A. 1995b. Performance Estimation of Embedded Software with Instruction Cache Modeling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 380–387.

LI, Y.-T. S., MALIK, S., AND WOLFE, A. 1999. Performance Estimation of Embedded Software with Instruction Cache Modeling. *ACM Transactions on Design Automation of Electronic Systems 4,* 3.

LIM, S.-S., BAE, Y. H., JANG, G. T., RHEE, B.-D., MIN, S. L., PARK, C. Y., SHIN, H., PARK, K., MOON, S.-M., AND KIM, C. S. 1995. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering 21(7)*, 593–604.

LUNDQVIST, T. 2002. A WCET Analysis Method for Pipelined Microprocessors with Cache Memories. Ph.D. thesis, Dept. of Computer Engineering, Chalmers University of Technology, Sweden.

LUNDQVIST, T. AND STENSTRÖM, P. 1999a. A Method to Improve the Estimated Worst-Case Performance of Data Caching. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. 255–262.

LUNDQVIST, T. AND STENSTRÖM, P. 1999b. An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. *In Real-Time Systems 17,* 2/3 (November).

Lundqvist, T. and Stenström, P. 1999c. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*. 12–21.

Mitra, T., Roychoudhury, A., and Li, X. 2002. Timing Analysis of Embedded Software for Speculative Processors. In *ACM SIGDA International Symposium on System Synthesis (ISSS)*.

Mohan, S., Mueller, F., Whalley, D., and Healy, C. 2005. Timing Analysis for Sensor Network Nodes of the Atmega Processor Family. *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 405–414.

Mueller, F. 2000. Timing Analysis for Instruction Caches. *The Journal of Real-Time Systems 18,* 2 (May), 217–247.

Nielson, F., Nielson, H. R., and Hankin, C. 1999. *Principles of Program Analysis.* Springer-Verlag Heidelberg.

Olukotun, K. and Hammond, L. 2005. The Future of Microprocessors. *ACM Queue*.

Puaut, I. and Decotigny, D. 2002. Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS02).* Austin, Texas, 114–123.

Pugh, W. 1991. The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing.* ACM Press, New York, NY, USA, 4–13.

Puschner, P. 2005. Experiments with WCET-Oriented Programming and the Single-Path Architecture. In *Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems.* 205–210.

Puschner, P. and Nossal, R. 1998. Testing the Results of Static Worst-Case Execution-Time Analysis. In *Proc. 19th IEEE Real-Time Systems Symposium.* 134–143.

Puschner, P. and Schedl, A. 1997. Computing Maximum Task Execution Times – A Graph-Based Approach. *Journal of Real-Time Systems 13,* 1 (Jul.), 67–91.

Puschner, P. P. and Schedl, A. V. 1995. Computing Maximum Task Execution Times with Linear Programming Techniques. Tech. rep., Technische Universität Wien, Institut für Technische Informatik. Apr.

Ramaprasad, H. and Mueller, F. 2005. Bounding Worst-Case Data Cache Behavior by Analytically Deriving Cache Reference Patterns. In *Proc. of the IEEE Real-Time Embedded Technology and Applications Symposium.* 148–157.

Reineke, J., Wachter, B., Thesing, S., Wilhelm, R., Polian, I., Eisinger, J., and Becker, B. 2006. A definition and classification of timing anomalies. In *WCET 2006.* Dresden.

Sandell, D., Ermedahl, A., Gustafsson, J., and Lisper, B. 2004. Static Timing Analysis of Real-Time Operating System Code. In *Proc. 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04).*

Sehlberg, D. 2005. Static WCET Analysis of Task-Oriented Code for Construction Vehicles. M.S. thesis, Mälardalen University, Västerås, Sweden.

Seth, K., Anantaraman, A., Mueller, F., and Rotenberg, E. 2003. FAST: Frequency-Aware Static Timing Analysis. *Proc. of the IEEE Real-Time Systems Symposium*, 40–51.

Shaw, A. C. 1989. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering 15,* 7, 875–889.

Souyris, J., le Pavec, E., Himbert, G., Jégu, V., Borios, G., and Heckmann, R. 2005. Computing the WCET of an Avionics Program by Abstract Interpretation. In *WCET 2005.* 15–18.

Stappert, F. and Altenbernd, P. 2000. Complete Worst-Case Execution Time Analysis of Straight-line Hard Real-Time Programs. *Journal of Systems Architecture 46,* 4, 339–355.

Stappert, F., Ermedahl, A., and Engblom, J. 2001. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proc. 4th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES'01).*

Staschulat, J. and Ernst, R. 2004. Multiple process execution in cache related preemption delay analysis. In *EMSOFT.* Pisa, Italy.

STASCHULAT, J. AND ERNST, R. 2006. Worst case timing analysis of input dependent data cache behavior. In *ECRTS*. Dresden, Germany.

STASCHULAT, J., SCHLIECKER, S., AND ERNST, R. 2005. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *ECRTS*. Palma de Mallorca, Spain.

THEILING, H. 2002a. Control Flow Graphs For Real-Time Systems Analysis. Ph.D. thesis, Universität des Saarlandes, Saarbrücken, Germany.

THEILING, H. 2002b. ILP-Based Interprocedural Path Analysis. In *Embedded Software (EMSOFT)*. Lecture Notes in Computer Science, vol. 2491. Springer, 349–363.

THEILING, H., FERDINAND, C., AND WILHELM, R. 2000. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Real-Time Systems 18,* 2/3 (May), 157–179.

THESING, S. 2004. Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models. Ph.D. thesis, Saarland University.

THESING, S., SOUYRIS, J., HECKMANN, R., RANDIMBIVOLOLONA, F., LANGENBACH, M., WILHELM, R., AND FERDINAND, C. 2003. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software Systems. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*. IEEE Computer Society, 625–632.

THIELE, L. AND WILHELM, R. 2004. Design for Timing Predictability. *Real-Time Systems 28*, 157 – 177.

VIVANCOS, E., HEALY, C., MUELLER, F., AND WHALLEY, D. 2001. Parametric Timing Analysis. In *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*. 88–93.

WENZEL, I., RIEDER, B., KIRNER, R., AND PUSCHNER, P. 2005a. Automatic Timing Model Generation by CFG Partitioning and Model Checking. In *Proc. Design, Automation and Test in Europe (DATE'05)*. Munich, Germany.

WENZEL, I., RIEDER, B., KIRNER, R., AND PUSCHNER, P. 2005b. Measurement-Based Worst-Case Execution Time Analysis. In *Proc. 3rd IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*. Seatlle, Washington, 7–10.

WHITE, R., MUELLER, F., HEALY, C., AND WHALLEY, D. 1999. Timing Analysis for Data Caches and Wrap-Around Fill Caches. *Real-Time Systems*, 209–233.

WILHELM, R. 2004. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *VMCAI 2004*. LNCS, vol. 2937. Springer, 309–322.

WILHELM, R. 2005. Determining Bounds on Execution Times. In *Handbook on Embedded Systems*, R. Zurawski, Ed. CRC Press, 14–1,14–23.

WILHELM, R., ENGBLOM, J., THESING, S., AND WHALLEY, D. 2003. Industrial Requirements for WCET Tools — Answers to the ARTIST Questionnaire. In *EUROMICRO Workshop on WCET (WCET 2003)*, J. Gustafsson, Ed.

WOLF, F. 2002. *Behavioral Intervals in Embedded Software*. Kluwer Academic Publishers.

WOLF, F., ERNST, R., AND YE, W. 2001. Path clustering in software timing analysis. *IEEE Transactions on VLSI Systems 9(6)*, 773–782.

WOLF, F., KRUSE, J., AND ERNST, R. 2002. Timing and power measurement in static software analysis. In *Microelectronics Journal, Special Issue on Design, Modeling and Simulation in Microelectronics and MEMS*. Vol. 6(2). 91–100.

WOLF, F., STASCHULAT, J., AND ERNST, R. 2002. Hybrid cache analysis in running time verification of embedded software. *Journal of Design Automation for Embedded Systems 7,* 3, 271–295.

YE, W. AND ERNST, R. 1997. Embedded program timing analysis based on path clustering and architecture classificaton. In *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD '97) San Jose, USA*. 598–604.

ZHANG, Y. 2005. Evaluation of Methods for Dynamic Time Analysis for CC Systems AB. M.S. thesis, Mälardalen University, Västerås, Sweden.