ISSN 1404-3041
ISRN MDH-MRTC-116/2003-1-SE

3rd International Workshop on Worst-Case Execution Time Analysis (WCET'2003)

Polytechnic Institute of Porto, Portugal, July 1, 2003

Jan Gustafsson (Workshop Chair)

Department of Computer Science and Engineering
Mälardalen University, Box 883, 721 23 Västerås, SWEDEN

WCET'2003

3rd International Workshop on Worst-Case Execution Time Analysis

(Satellite Event to ECRTS 2003)

Polytechnic Institute of Porto, Portugal
July 1, 2003

Message from the Workshop Chair

Welcome to the 3rd International Workshop on Worst-Case Execution Time Analysis (WCET'2003). The workshop a satellite event to the 15th Euromicro Conference on Real-Time Systems (ECRTS 2003). It was held in Porto, Portugal, July 1, 2003. This is the third in a series of successful events. The first two were held in Delft, Holland, and Vienna, Austria, respectively.

The goal of the workshop is to bring together people from academia, tool vendors and users in industry and that are interested in all aspects of timing analysis for real-time systems. The workshop will provide a relaxed forum to present and discuss new ideas, new research directions and to review current trends in this area. The workshop will be based on short presentations that should encourage discussion by the attendees.

The topics of the workshop include any issue related to timing analysis, in particular:

* Flow analysis for WCET
* Low-level timing analysis
* Calculation methods for WCET
* Timing analysis through measurement
* Modeling and analysis of processor features
* Probabilistic analysis techniques
* Integration of WCET and schedulability analysis
* Evaluation and case studies
* Tools for timing analysis
* Industry experience reports

Statements which are innovative, controversial, or that present new approaches are specially sought.

I would like to thank all the participants, panelists, authors, reviewers and session chairpersons that made this events a successful one. Special thanks to Eduardo Tovar of ECRTS, who was responsible for the local arrangements.

Dr. Jan Gustafsson
Mälardalen University, Västerås, Sweden

FINAL PROGRAM
3rd Intl WORKSHOP ON WORST-CASE EXECUTION TIME (WCET) ANALYSIS
July 1, 2003
Building E ("Edificio E")
School of Engineering of the Polytechnic Institute of Porto (ISEP)
Rua Dr. Antonio Bernardino de Almeida, 431
4200-072 Porto, Portugal

PART I: WCET tools (chair: Jan Gustafsson, Mälardalen University, Västerås, Sweden)

- *Challenges in Calculating the WCET of a Complex On-board Satellite Application*
  M. Rodriguez, N. Silva, J. Estives, L. Henriques and D. Costa, Critical Software SA, Coimbra, Portugal.
- *Convenient User Annotations for a WCET Tool*
  C. Ferdinand, R. Heckmann, H. Theiling, AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany, and R. Wilhelm, Universität des Saarlandes, Saarbrücken, Germany.
- *pWCET, a Tool for Probabilistic WCET Analysis of Real-Time Systems*
  G. Bernat, A. Colin, S. Petters. University of York, United Kingdom.
- Discussion.

PANEL DISCUSSION: "Requirements of WCET tools". Jean Souyris, Airbus Toulouse, France, Jan Lindblad, ENEA OSE, Sweden. Moderator Reinhard Wilhelm, Universität des Saarlandes, Saarbrücken, Germany.

PART II: WCET calculation methods (chair: Peter Puschner, Technische Universität Wien, Austria)

- *On the Design of an Extensible Platform for Flow Analysis of Java using Abstract Interpretation*
  P. Guedes, Cidade Universitária, Recife, Brazil.
- *Elimination of Unstructured Loops in Flow Analysis*
  C. Sandberg, Mälardalen University, Västerås, Sweden.
- *A Survey of Methods to Improve ILP-based WCET Analysis*
  Xianfeng Li, National University of Singapore, Singapore.
- *Discussion of Misconceptions about WCET Analysis*
  R. Kirner and P. Puschner, Technische Universität Wien, Austria.
- Discussion.

PART III: WCET calculation methods, cont. (chair: Reinhard Wilhelm, Universität des Saarlandes, Saarbrücken, Germany)

- *Compiler Support for WCET Analysis: a Wish List*
  G. Bernat, University of York, United Kingdom and N. Holsti, Space Systems Finland, Finland.

- *Impact of Automatic Gain Time Identification on Tree-Based Static WCET Analysis*
  Mathieu Avila, Maxime Glaizot, Isabelle Puaut, IRISA, Campus de Beaulieu, Rennes Cedex, France.
- *Comparison of Trace Generation Methods for Measurement Based WCET Analysis*
  S. Petters, University of York, United Kingdom.
- *Evaluating Reasons for Unexpected Results When Measuring Execution Time of Code*
  V. Lorente, A. Espinosa, A. Terrasa, A. Garcia and A. Crespo, Universitat Politecnica de Valencia, Valencia, Spain.
- Discussion.

PART IV: Low-level analysis (chair: Guillem Bernat, University of York, United Kingdom)

- *Towards Designing WCET-Predictable Processors*
  Christine Rochange, Pascal Sainrat IRIT - UPS, Toulouse, France.
- *A Flexible Tradeoff between Code Size and WCET Employing Dual Instruction Set Processors*
  Sheayun Lee, Jaejin Lee, Chang Yun Park, and Sang Lyul Min, Seoul National University, Seoul, Korea.

PART V: New methods for analyzing WCET (chair: Guillem Bernat)

- *Aspect-Level WCET Analyzer: A Tool for Automated WCET Analysis of the Real-Time Software Composed Using Aspect and Components*
  A. Tesanovic, J. Hansson and P. Uhlin, Linköping University, Linköping, Sweden and D. Nyström and C. Norström, Mälardalen University, Västerås, Sweden.
- *Fully Automatic, Parametric Worst-Case Execution Time Analysis*
  B. Lisper, Mälardalen University, Västerås, Sweden.
- Discussion for the two last sessions.

Contents

Challenges in Calculating the WCET of a Complex On-board Satellite Application

Manuel Rodríguez[1], Nuno Silva[1], João Esteves[1], Luis Henriques[1], Diamantino Costa[1], Niklas Holsti[2], Kjeld Hjortnaes[3]

[1]*Critical Software SA*
Banhos Secos EN1, 3040-032 Coimbra, Portugal
{mrodriguez, nsilva, jesteves, lhenriques, dcosta}@criticalsoftware.com

[2]*Space Systems Finland Ltd*
Kappelitie 6, 02200 Espoo, Finland
holsti@ssf.fi

[3]*ESA/ESTEC*
Noordwijk, Netherlands
kjeld.hjortnaes@esa.int

Abstract

Calculating the WCET of mission-critical satellite applications is a challenging issue. The European Space Agency is currently undertaking the CryoSat mission, consisting of a radar altimetry satellite to be launched in 2005. This paper describes the challenges and the first experimental results of calculating the WCET of the Control and Data Management Unit (CDMU) subsystem of the satellite. This subsystem constitutes the central control unit of all the on-board data handling, as well as the attitude and orbit control system of the satellite, and must guarantee predictable behavior.

1. Introduction

CryoSat is the first satellite of the Living Planet Programme that the European Space Agency (ESA) undertakes in the framework of the Earth Explorer Opportunity Missions (see [1]). It is a three-year radar altimetry mission, scheduled for launch in 2004/2005, dedicated to the observation of the polar regions, particularly the variations in the thickness of the Earth's continental ice sheets and marine ice cover. Its primary objective is to study possible Earth's climate variability and trends, and to predict the thinning of arctic ice due to the global warming.

One of the most important on-board software applications of the CryoSat satellite is the Control and Data Management Unit (CDMU). This application constitutes the central control unit for all the on-board data handling (DH) and the attitude and orbit control system (AOCS) of the satellite. Astrium GmbH (http://www.astrium-space.com) is the prime contractor for the CryoSat mission, and Critical Software (http://www.criticalsoftware.com) is the prime contractor of the Independent Software Verification & Validation activities (ISVV, see [2]), on which the CDMU application is being screened. The ISVV activities encompass a number of static and dynamic analysis techniques (e.g., robustness and stress testing, traceability matrices, code inspections, software failure mode effects and criticality analysis, schedulability analysis, etc.) that are applied by personnel not involved in the development process of the target product to ensure complete independency.

As part of the ISVV activities, schedulability analysis and WCET calculation are also to be performed. The selected tool for the WCET calculation is Bound-T [3], which is based on static code analysis techniques [4]. This paper describes the challenges of calculating the WCET of the CDMU application using this tool.

The structure of the paper is as follows. In Section 2, we present the CDMU onboard software application and the computational model used by the scheduler. The Bound-T tool is described in Section 3. Section 4 provides experimental results and describes the challenges of performing WCET analysis on the CDMU application. Section 5 concludes the paper.

2. The CDMU onboard software application of ESA's CryoSat satellite

The CryoSat's Control and Data Management Unit (CDMU) is the central control unit of all onboard data handling and of the attitude and orbit control system. Data handling functions are mainly constituted of command distribution, telemetry acquisition and timing facilities during all phases of the mission. Furthermore, the CDMU application performs monitoring functions and, depending on detected failures, provides reconfiguration and safe redundancy switchover capabilities.

The scheduler of the CDMU implements a cyclic scheduling policy. In a cyclic scheduling, the tasks are dispatched at predefined intervals. The CDMU scheduler implements a major cycle of 1 second that is divided into 10 minor cycles (called *slots*) of 100 milliseconds each. At each slot, the scheduler dispatches a predefined subset of cyclic tasks according to a task table, which is repeated every major cycle.

The CDMU onboard software consists of 24 cyclic tasks, 12 sporadic tasks and a number of background tasks. The cyclic tasks are allocated periods and offsets multiple of 100 ms (i.e., one slot), and are mainly

responsible for managing telemetry and telecomands, onboard control procedures, housekeeping, and the mission timeline. The sporadic tasks can preempt the cyclic tasks at any time, and are associated both to external events (e.g., the beginning of a slot or the arrival of a telecomand) and to internal error conditions (e.g., single bit-flip error detection). Background tasks are executed during spare time intervals (usually, at the end of a slot), and mainly perform maintenance activities (e.g., memory scrubbing).

The CDMU application is implemented in Ada 95, and the binary code is generated with the XGC Ada compiler. Neither protected objects nor the Ada tasking model are used (all the tasks are implemented in the form of procedures). The target processor used to run the application is the ERC32/SPARC V.7, running at a clock frequency of 24Mhz. Caches, pipelines and branch prediction units are not used.

3. The WCET tool: Bound-T

There are a number of methods developed for the prediction of the WCET of a program. These methods can be grouped into two main categories, namely, *dynamic* methods (e.g., testing and simulation) and *static* methods (e.g., static code analysis). The advantage of static methods is that they do not require input test sets to be defined in order to find the longest execution paths within the program code. Static code analysis [4] has become very popular and has been the object of study of many works in both academy and industry (e.g., see [5]), giving rise to a large variety of methods and supporting tools.

Bound-T [3], from Space Systems Finland (http://www.ssf.fi), is the tool selected by Critical/ESA to perform the WCET analysis of the CDMU onboard software application. Bound-T features static code analysis techniques for estimating the WCET of real-time programs based on their executable binary COFF or ELF code [6]. Apart from the ERC32/SPARC V7 processor targeted by the CDMU application, Bound-T also supports the Intel 8051 and the ADSP21020 processor families. Two of the most outstanding features of the tool are its arithmetic and assertion facilities. The arithmetic facility, based on the non-commercial *Omega Calculator* constraint-solving tool [7], caters for automatically bounding counter-type loops. The assertion facility implements an advanced language that allows the user to manually write assertions and bound those loops that cannot be analyzed fully automatically by the tool.

For more information about the features provided by Bound-T, refer to [3].

4. Challenges and experimental results

The main challenges raised by the analysis of the WCET of the CDMU application are related to the complexity of its loops constructions and to the XGC Ada compiler optimizations. As a consequence:

1. These issues lead to situations unexpected by Bound-T, keeping it from analyzing the application in a fully automatic way.
2. The arithmetic facility of Bound-T systematically blocks (i.e., it runs for too long) when trying to automatically bound the loops of the CDMU application. The arithmetic facility cannot thus be used, and all the loops have to be manually bounded, what can be very time consuming.

In the sequel, we first describe the problems encountered during the analysis of the WCET of the CDMU application's tasks. We then provide experimental results that are compared to the WCET estimations made at early development stages of the application.

4.1. Challenges

The analysis of the WCET of the CDMU code raised many problems that were not expected by the Bound-T tool, and that kept it from analyzing the application in a fully automatic way. The main problems encountered are reported in Figure 1.

1. *Use of "sentinels" in the loops*: A loop terminates only when a particular "sentinel" value is found in an array, rather than when the loop-counter reaches a limit value.
2. *Complex parameter-dependent loops*: The maximum number of iterations of a loop depends on the values of the input parameters. These dependencies in the CryoSat CDMU code are clearly too complex or extensive for the arithmetic facility in the current Bound-T.
3. *Complex loop-arithmetic*: Complex conditions, usually based on logical instructions "and" and "bleu" (*branch if less or equal, unsigned*), appearing in the loop-counter computations or in the loop-termination of the assembly code.
4. *Not explicit loops*: Loops appearing in the assembly code but not in the source code, specially caused by the use of array assignment constructs.
5. *Calls to error-handling code*: Ada run-time checks inserted by the compiler, triggering global error-handlers (e.g., exception *Constraint_Error*) that do not return to the call-site.
6. *Irreducible control-flow*: The longest control-flow path of a function cannot be found because of a compiler optimization or a manual coding of assembly routines leading to poorly structured loops.
7. *Recursive calls*: Recursive call sequences impairing the WCET calculation of the complete set of tasks.

Figure 1. Problems encountered

The assertion facility of Bound-T can be used in order to manually bound the unbounded loops and procedures resulting from these problems. Note that the assertions are not inserted in the application code, but are written as separated files interpreted by Bound-T. Let *P* be an Ada procedure, and *N* a natural number. Examples of typical assertions are the following:

- "`subprogram "P" loop repeats N times; end loop`", which means that the single loop contained by procedure *P* does not repeat for more than *N* times.
- "`subprogram "P" time N cycles`", which means that the execution time of procedure *P* is less than or equal to *N* clock cycles.

Writing assertions is a challenging issue because the user is responsible for finding bounds for the number of iterations of loops and for the execution time of procedures (i.e., value *N* of the previous examples). In particular, concerning the problems reported in Figure 1, the challenges raised are the following:

- For cases 1 to 4, the challenge consists in being able to understand the behavior of the CDMU application and find the maximum number of iterations of the unbounded loops. Bounds for loops in cases 1 to 3 can be found by inspecting the Ada source code. The effort required for this depends on the complexity of the Ada code related to the loops, and requires understanding of the control flow, data flow, and specific mission requirements. Bounding loops in case 4 (i.e., *not explicit loops*) requires also analyzing the assembly code of the application. It adds an additional complexity, since it might not be feasible to actually isolate the Ada instructions of the source code that lead to such assembly loops. It is worth noting that some of the unbounded loops caused by these problems might be automatically bounded by the arithmetic facility of Bound-T. Since the arithmetic facility cannot be used, these and all the other loops of the application have to be manually bounded.
- For cases 5 to 7, the challenge consists in finding a bound for the execution time of the corresponding procedures (i.e., the error-handlers, the irreducible functions and the recursive functions). Another challenging issue for case 5 (i.e., *calls to error-handling code*) consists in making assumptions about the maximum arrival rate of failures, as long as the behavior of the application in presence of faults is to be considered. In general, finding time bounds for cases 5 to 7 can be assisted by the use of techniques other than static code analysis (e.g., see [8]).

In the sequel, we analyze in more detail cases 4 to 7.

Not explicit loops

In some cases, the XGC Ada compiler generates loops that cannot be identified in the source code of the application. It usually concerns the compilation of array assignments constructs, as the one presented in Figure 2.

```
...
Array_A(0..Foo) := Array_B(6..Bar);
...
```

Figure 2. Ada array assignment instruction

Indeed, two loops might be generated in the assembly code for the single array assignment of Figure 2. It usually occurs when the compiler cannot guarantee that the memory addresses (or *slice*) allocated to the target array are different from the slice allocated to the source array. In such a case, the compiler first copies the source array into a temporary location, and then copies the temporary location into the target array.

Apart from array assignments, conditional branches (e.g., *if-else* branches) appearing in the assembly code might be arranged by the compiler in a different order as they appear in the Ada sources. Also, the XGC Ada compiler might generate a huge amount of extra conditional branches for constraint verification purposes. This greatly increases the complexity of the resulting assembly code.

As we explained, the existence of not explicit loops constraints the user to inspect and understand the assembly code, so as to find the maximum number of iterations of the unbounded loops and write the corresponding assertions. This can be very time consuming, as it requires analyzing a large amount of assembly code, and taking into account the specific aspects of the ERC32 architecture. Indeed, some zones in the CDMU application code make a great use of arrays assignments. These array assignments usually occur within other loops or inside conditional structures. Since the compiler might rearrange loops in the assembly code, manually finding bounds can thus become even more difficult.

Calls to error-handling code

This problem is related to run-time checks inserted by the compiler, which trigger global error-handlers not returning to the call-site. More precisely, the code generated for some Ada exceptions contains jumps to addresses outside the text segment.

The XGC Ada compiler is the responsible for most of the calls to exception routines appearing in the assembly code. This is for instance the case of the *Constraint_Error* Ada exception, for which the compiler generates a function labeled "`__raise_constraint_error`", whose implementation is shown in Figure 3.

```
<__raise_constraint_error>:
02096070:       91 d0 20 05     ta  5
02096074:       81 c3 e0 08     retl
02096078:       01 00 00 00     nop
```

Figure 3. Assembly code of the *Contraint_Error* Ada exception

Instruction at line 0x02096070 (*trap 5*) is interpreted by Bound-T as a jump outside the text segment boundaries. The reason is that the information contained in the vector trap is loaded at runtime, and Bound-T cannot keep track of it during static analysis. Static analysis is thus stopped at this point.

As long as the behavior of the application in presence of faults is not to be considered during static analysis, this issue does not represent a significant problem. Indeed, in practice, whenever an exception is raised, the nominal control flow is broken and the CDMU application is restarted. Two workarounds can be envisaged: (i) asserting the execution time of the exception handler with a non significant value (e.g., "`subprogram address "02096070" time 0 cycles;`"), or (ii) asserting that no calls to the error-handling subprogram are executed (e.g., "`all calls to address "020976070" repeat 0 times;`"). It allows Bound-T to proceed calculating the WCET of the application.

Irreducible control-flow

We observed that this problem appeared whenever the call graph contains a jump to a mathematical function in a library responsible for calculating the integer division (e.g., calls to *.div*). These functions are implemented in libraries external to the CDMU application.

The solution consists thus in measuring and asserting the execution time of every library routine leading to an irreducible control flow. For instance, concerning the *.div* routine, the following assertion can be used: "`subprogram ".div" time N cycles;`", where *N* would correspond to the estimated execution time (expressed in cycles) of the function.

Recursive calls

A recursion involving two functions was identified in the code (the depth of this recursion is 2). This inhibits the WCET calculation in almost all the tasks because Bound-T cannot analyze recursion.

A workaround consists in calculating separately the WCET of each function responsible for the recursion. For instance, consider assertion of Figure 4, where "`A`" and "`B`" are the functions responsible for the recursion:

```
subprogram "A" time 0 cycles;
subprogram "B" call to "A" time 0 cycles; end call;
```

Figure 4. Workaround to calculate the WCET of recursive function `B`

Assertion of Figure 4 allows automatically calculating with Bound-T the WCET of function "`B`". A similar assertion can be written to calculate the WCET of function "`A`". Therefore, when the recursion depth is known, WCET bounds for the functions involved in the recursion (e.g., "`A`" and "`B`") can be manually computed via similar assertions, omitting thus the recursion.

4.2. Experimental results

Due to the difficulties encountered while calculating the WCET of the CDMU application with the Bound-T tool, we were not able to obtain definite results yet. Indeed, Bound-T is currently being updated, and some modifications are being performed in the compilation options of the CDMU application.

The preliminary results we obtained are based on assumptions about the execution time of some functions of the CDMU application, specially those functions containing not explicit loops (see Section 4.1). These assumptions were not validated yet. Note also that since many of the asserted functions are common to various tasks, inaccuracies in assumptions are propagated among tasks.

As an example, let us present results concerning task *Mil_Bus_Manager* (Table 1). This task manages the bus that connects the different satellite payloads, and is the largest and most complex task of the CDMU application.

Table 1. WCET calculation results

Task	WCET given by Bound-T (ms)	WCET reported in design documents (ms)
Mil_Bus_Manager	15.79	20

Table 1 reports two different values: the WCET estimated by Bound-T, and the WCET reported in the design documents of the CDMU application. The latter value is mainly based on in-service history information of previous missions of similar applications. As shown in Table 1, the WCET given by Bound-T (about 16 ms) was slightly under the value reported in the design documents (20 ms).

5. Conclusion

Calculating the WCET of mission critical satellite applications is a challenging issue. The Control and Data Management Unit (CDMU) application of the ESA's CryoSat satellite is responsible for all the data exchanged between the satellite and the ground (e.g., telemetry data containing measurements and telecommands containing satellite commands). It is a large and complex satellite application using the XGC Ada compiler and running on an ERC32 architecture-based microprocessor, whose tasks must guarantee predictable worst case execution times.

In the framework of the Independent Software Verification and Validation (ISVV) program promoted by ESA, Critical Software SA is performing (among other testing activities) the WCET analysis of the CDMU application. The tool chosen by ESA for this activity is Bound-T, from Space Systems Finland, based on static code analysis.

The work presented in this paper leads us to state that fully automated tools for analyzing the WCET of large and complex safety critical satellite applications are still not mature enough. Indeed, the various unexpected problems we encountered kept us from accomplishing the WCET calculation of the entire application. These problems were described in detail in the paper: loops in the assembly code not appearing in the source code, loops depending on complex parameters, etc. To overcome these problems, it is necessary (i) estimating loop and time bounds of some parts (usually functions) of the application code by other means (e.g., code inspection of loops, testing techniques, etc.), (ii) writing annotations asserting the bounds of the concerned loops and functions, and (iii) using the annotations as inputs to the WCET tool. Note however that the reported problems can actually be solved by static code analysis techniques, but more mature tools are still needed so as to deal with these problems in an automated way. We also observed that developing complex applications with WCET in mind (e.g., systematically asserting the maximum number of iterations of every loop in the source code) could greatly help automate WCET analysis. Indeed, facilities to automatically bounding loops (e.g., the arithmetic facility of Bound-T) might be of little use for complex and large applications.

As a conclusion, fully automatic tools and good programming practices are highly required in what concerns static code analysis of the WCET of space applications. Indeed, the effort and resources in industry must be planned in advance for every activity, and there is little place for unexpected situations that would require revisiting the planning or allocating extra resources.

References

[1] http://www.esa.int/export/esaLP/cryosat.html

[2] ECSS Secretariat, "ECSS-Q-80B, ECSS Space Product Assurance, Software Product Assurance Draft B", ESA-ESTEC Requirements & Standards Division, Noordwijk, The Netherlands, February 2002 (http://www.ecss.nl/)

[3] http://www.bound-t.com

[4] P. Puschner, C. Koza, "Calculating the Maximum Execution Time of Real-Time Programs", *Real-Time Systems*, vol. 1, pp. 159-176, 1989.

[5] C.M. Bailey, A. Burns, A.J. Wellings, C.H. Forsyth, "A Performance Analysis of a Hard Real-Time System", Control Eng. Practice, Vol. #, No. 4, pp. 447-464, 1995 (http://citeseer.nj.nec.com/burns93olympus.html).

[6] www.pldworld.com/_hdl/1/estec.esa.nl/ftp/pub/ws/wsd/erc32/doc/gcc.pdf

[7] http://www.cs.umd.edu/projects/omega/omega.html

[8] M. Lindgren, H. Hansson, H. Thane, "Using Measurements to Derive the Worst-Case Execution Time", in Proc. of RTAS 2000, Cheju Island, South Korea, 2000 (http://citeseer.nj.nec.com/lindgren00using.html).

Convenient User Annotations for a WCET Tool

Christian Ferdinand, Reinhold Heckmann, Henrik Theiling
AbsInt Angewandte Informatik GmbH
Stuhlsatzenhausweg 69, D-66123 Saarbrücken, Germany
{ferdinand,heckmann,theiling}@absint.com

Reinhard Wilhelm
FR 6.2 Informatik, Universität des Saarlandes
Postfach 15 11 50, D-66041 Saarbrücken, Germany
wilhelm@cs.uni-sb.de

Abstract

*The purpose of **AbsInt**'s WCET tool **aiT** is to obtain upper bounds for the worst-case execution times of specified parts of an executable. Apart from the executable, **aiT** needs some user information. Originally, this information had to refer to program points by their address. Now, user comfort was greatly enhanced by allowing symbolic references and source code annotations.*

1. Introduction and Overview

aiT is a tool for determining upper bounds for the Worst-Case Execution Time (WCET) of code snippets given as routines in executables. These code snippets are for instance tasks called in round-robin fashion by a scheduler, where each task has a specified deadline [11]. **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 hardware realizations.

Currently there are **aiT** versions for three processors: Motorola ColdFire MCF 5307, Motorola PowerPC MPC 755, and ARM7 TDMI. They share a common structure [3]: First, the control flow is reconstructed from the given object code [8, 9]. 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, a *value analysis* computes address ranges for instructions accessing memory. The ColdFire and PowerPC versions use this information in *cache analysis*, which classifies memory references as cache misses or hits [2] (the ARM7 TDMI has no cache). *Pipeline analysis* predicts the behavior of the program on the processor pipeline [6]. The result is an upper bound for the execution time of each basic block in each distinguished execution context. Finally *path analysis* determines a worst-case execution path of the program from the timing information for the basic blocks [10].

Apart from the executable, **aiT** needs user input to find a result at all, or to improve the precision of the result. The most important user annotations specify the targets of computed calls and branches and the maximum iteration counts of loops (there are many other possible annotations). Originally, program points had to be identified by their address in these annotations. This is cumbersome and error-prone since addresses may change after recompilation. Now a more high-level specification language was introduced for referring to program points symbolically (e.g., the second loop in routine *R*) or via source code annotations (see section 4).

2. Targets of Computed Calls and Branches

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

called AIS file:

```
INSTRUCTION ProgramPoint
   CALLS Target1, ..., Targetn;
INSTRUCTION ProgramPoint BRANCHES
   TO Target1, ..., Targetn;
```

ARM7 TDMI processors do not offer return instructions. Instead, various kinds of computed branches with the return address as target can be employed. **aiT** can recognize most of these branches as returns. The few remaining ones, mostly contained in library code, can be annotated as follows:

```
INSTRUCTION ProgramPoint
   IS A RETURN;
```

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

Example: The library routine `C_MEMCPY` of the ARM7 TDMI consists of hand-written assembler code. It contains 2 computed branches whose targets can be specified as follows:

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

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

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

3. Loop Bounds

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

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

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

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

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

If the qualifier is omitted, **aiT** assumes the worst case of the two possibilities, which is `begin` where the loop test is executed one more time. The `begin`/`end` information refers to the *executable*, not to the source code; the compiler may move the loop test from the beginning to the end, or vice versa.

Example:

```
loop "_prime" + 1 loop end max 10;
```

specifies that the first loop in `_prime` has the loop test at the end and is executed at most 10 times.

4. Source Code Annotations

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

```
/*  ai:  specification1; ...
         specificationn;  */
```

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

Source code annotations admit a special program point or target `here`, which roughly denotes the place where the annotation occurs. More exactly, **aiT** extracts the correspondence between source lines and code addresses from the executable. A `here` occurring in source line *n* then points to the *first* instruction associated with a line number $\geq n$. Since the line information in the executable is created by the compiler, it becomes invalid when lines are added or deleted in the source file. Therefore the application must be recompiled whenever lines are added while annotating.

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

```
for (i=3; i*i <= n; i += 2) {
    /* ai: loop here end max 10; */
    if (divides (i, n))
        return 0; }
```

5. Other Annotations

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

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

  ```
  flow 0x100 / 0x200 is max 4;
  ```

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

6. Related Work

In contrast to most approaches proposed in the literature [7, 5, 4, 1], our annotations may refer to the source code, but do not extend the source language (annotations are comments), nor do they require a special compiler. Instead, **aiT** can analyze code generated by standard compilers. The correspondence between source code annotations and low-level object code is exclusively based on the debug information of the executable. Other than the annotations proposed elsewhere, ours cover the full spectrum between reference to source code lines (`here`) over semantic descriptions (`routine` *R* `+ 1 loop`) till routine-relative or absolute addresses, the latter being useful for annotating optimized code with instructions that cannot be attributed to a particular piece of source code.

The annotation languages proposed in [7, 5, 4, 1] are generally restricted to loop bounds and flow constraints, while ours are more general in that they also admit the specification of targets of computed calls and branches, register values, and addresses of memory accesses. On the other hand, the specialized flow languages, in particular the one proposed in [1], are more expressive and powerful than our flow constraints. Extensions in this direction are intended, but not yet realized to get a working system as soon as possible.

7. Conclusion

aiT is a WCET tool for industrial usage. Information required for WCET estimation such as computed branch targets and loop bounds is determined by static analysis. For situations where **aiT**'s analysis methods do not succeed, a convenient specification and annotation language was developed in close cooperation with **AbsInt**'s customers. This effort has contributed to the good acceptance **aiT** has found among producers of real-time software.

References

[1] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

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

[3] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of EMSOFT 2001, First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, 2001.

[4] R. Kirner. The Programming Language WCETC. Technical report, Technische Universität Wien, Jan. 2002.

[5] L. Ko, C. A. Healy, E. Ratliff, R. D. Arnold, D. B. Whalley, and M. G. Harmon. Supporting the specification and analysis of timing constraints. In *IEEE Real Time Technology and Applications Symposium*, page 170 pp., 1996.

[6] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline Modeling for Timing Analysis. *Proceedings of the 9th International Static Analysis Symposium*, 2002.

[7] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1, 1989.

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

[9] H. Theiling. Generating Decision Trees for Decoding Binaries. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tools for Embedded Systems*, Snowbird, Utah, USA, June 2001.

[10] H. Theiling and C. Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, 1998.

[11] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software. In *Proceedings of 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 625–632. IEEE Computer Society, 2003.

pWCET: a Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems

Guillem Bernat, Antoine Colin, Stefan Petters
Real-Time Systems Research Group
University of York. England, UK
{bernat,acolin,petters}@cs.york.ac.uk

January, 2003

Abstract

This paper describes the tool support for a framework for performing probabilistic worst-case execution time (WCET) analysis for embedded real-time systems. The tool is based on a combination of measurement and static analysis, all in a probabilistic framework. Measurement is used to determine execution traces and static analysis to construct the worst path and effectively providing an upper bound on the worst-case execution time of a program. The paper illustrates the theoretical framework and the components of the tool together with a case study.

1 Introduction

There are two main approaches for the determination of the worst-case execution time of a real-time program. Static analysis and measurement. Static analysis relies on a timing model of the hardware and attempts to determine an upper bound on the longest path of the program. Techniques include tree-based approaches [10, 4], or path based approaches [8, 12, 13]. Efforts on WCET analysis are on determining the effect of advanced processor features like cache, branch prediction and pipelines and their interactions [9, 6, 5, 7], However, these approaches are very complex as the processors themselves become more difficult to predict. An alternative approach to static analysis is by measurement. In this approach the code is run under exhaustive test conditions and the longest execution time recorded.

Both approaches have their advantages and disadvantages. Static analysis provides a safe upper bound guaranteeing that the worst case is never underestimated. This is adequate for simple programs running on simple 8 bit CPUs, however for more complex programs which are data dependent and for advanced CPU's with acceleration features like cache, pipelines, branch prediction buffers and out of order execution the analysis is extremely difficult to perform and

results in unacceptable levels of pessimism. An additional criticism of the approach is that it is based on an abstraction of the processor and may fail to capture effects that occur in the real system. Measurement approaches do observe the real system and therefore are able to account for these phenomena, however they may fail to capture the worst case as the set of test cases that may lead to the worst case may be very difficult to determine. In addition, a safety margin is usually included in the analysis, however there is no scientific process by which such safety factors can be determined.

In addition, traditional static approaches to WCET are too focused on obtaining an absolute upper bound on the execution time of the program. This may be unnecessary pessimistic if the probability of such event happening is extremely small. In probabilistic hard real-time systems the aim is to provide estimates that the probability of missing a deadline is of the same order of magnitude that other dependability estimates. For instance, probabilities smaller than 10^{-12} of missing a deadline should be provided. For such estimates to be made, it is first required to determine the probability distribution of the execution time of individual tasks.

This paper presents the pWCET framework, a theory and its tool support for probabilistic WCET analysis of real-time embedded programs. pWCET combines the best features of both measurement and analysis and allows to draw the benefits from both approaches. The framework is based on determining the execution times of individual blocks by observing the real-system (instead of relying on a processor model) but combining the worst case effects observed locally using static analysis techniques. In this way, no timing model of the processor is needed because the timing information is determined by measurement. There have been some initial approaches for probabilistic timing analysis of systems, [3, 2, 11] use extreme value statistics to model the tail of the distributions.

A different approach is the one presented by the same authors in [1]. This paper presents the general overview of the theory but its main purpose is the description of the tool support. The paper illustrates these concepts with a case study at the end of the paper. The following section provides an introduction of the theory of probabilistic WCET analysis and the description of the tool, its components and features is deferred to section 3.

2 Probabilistic WCET analysis

The aim of probabilistic WCET analysis is to determine the probability distribution of the worst-case execution time of a particular code fragment. The problem is formulated (and solved) in terms of a syntax tree representation of the program and a probabilistic timing schema.

A syntax tree is a representation of a program. It is a tree where the leafs are basic blocks (sequences of instructions that have no control flow instructions except possibly at the end) and inner nodes that correspond to syntactic composition of blocks: Sequential composition, conditional composition and iterative composition.

A timing schema is a set of rules that allow to determine the execution time of a program segment as a function of the execution time of its components. Each rule of the timing schema is associated to a type of node in the tree. For instance a trivial timing schema for static WCET analysis is as follows:

- $W(\mathsf{A}) =$ integer if A is a basic block.
- $W(\mathsf{A;B;}) = W(A) + W(B)$. Sequence of blocks.
- $W(\mathsf{if\ E\ then\ A\ else\ B\ end\ if}) = W(E) + \max(W(A), W(B))$. Conditional.
- $W(\mathsf{for\ E\ loop\ A\ end\ loop;}) = W(E) + n(W(A) + W(E))$. Loop, where n is the maximum number of iterations of the loop.

The aim of the pWCET approach is to provide an equivalent timing schema where integers are replaced by probability distributions and operations on integers are replaced by operations on random variables.

The problem of probabilistic WCET analysis is therefore:

1. To construct a syntax tree representation of the program. For illustrative purposes we consider basic blocks as the smallest execution unit, however there is no reason why other units (larger or smaller) could be used for the purpose,
2. to determine probability distributions of the individual executions of the blocks and their dependency,
3. to determine a probabilistic algebra to manipulate probability distributions,
4. to determine which and when to combine the probability distributions of the individual building blocks to derive the probability distributions of the nodes in the tree,
5. to present and visualise the results to the user.

In the rest of the section we concentrate on items 2 and 3 about the probabilistic issues, the rest of the items are discussed in the following section.

In order to provide a probabilistic timing schema we need to define equivalent operators to the sum and max for random variables. The most important fact is what assumptions about the dependence between blocks can be made.

2.1 Sequential execution $Z = X + Y$

The statistical formulation of the problem is as follows. Let X, Y be random variables that describe the execution time of a program segment. Let $F(x) = P[X \leq x]$, $G(y) = P[Y \leq y]$ be their distribution functions. Consider that situation in which X and Y are the random variables of two code segments A and

B that are executed in sequence: A;B;. Lets denote Z the random variable that describes the execution time of the sequence. The question is to determine what is the probability distribution of Z. Clearly Z can be formulated in statistical terms as $Z = X + Y$, and therefore we are interested in computing $H(z) = P[X + Y \leq z]$.

2.1.1 Dependency

One of the major issues for pWCET analysis is the determination of the dependency between X and Y. This dependence can be:

- X and Y are (assumed to be) independent,
- the joint distribution of X and Y is known and therefore the precise dependence between X and Y is also known,
- the dependency between X and Y is not known (the general case) and it can not be assumed that they are independent.

The hypothesis of independence is commonly assumed in other probabilistic analysis frameworks, however, in the framework of probabilistic WCET analysis this hypothesis is in the general case wrong. As reported in [1] making the hypothesis of independence may lead to severe underestimations of the probability of the worst case, overestimations of various orders of magnitude are possible. This is the case, for example, when the condition that makes one block to run for the worst case is the same that forces the other one for the worst case too.

The joint distribution captures precisely the exact dependence between X and Y. However, capturing such dependence by measurement is very difficult, not only for the computational complexity but also because of the nature of the process. Determining distributions of individual blocks is a difficult task because of the rare occurrence of extreme events, observing combinations of rare events in joint distributions makes the problem much harder.

In any case, there are situations where the joint distribution is not available because it can not be computed and therefore some assumption of the worst dependence between X and Y should be made. The situation when two random variables are positively correlated is called "comonotonicity". This means that both can be expressed as a non-decreasing function of another random variable U ($X = f_1(U)$ and $Y = f_2(U)$) meaning that the values of X are large when the values of Y are large too and as a consequence the probability of the extreme is not the product of probabilities (but the minimum of them).

The determination of $H(z)$ as the cumulative distribution of $Z = X + Y$ can be therefore performed as follows:

If X and Y are independent (or the assumption that they are independent is feasible) then H can be computed by performing the standard convolution between F and G:

$$H(z) = \int_x F(x)G(z - x)dx$$

If the joint distribution between X and Y is known and given by $J(x, y) = P[X \leq x, Y \leq y]$, then the distribution H can be computed as follows:

$$H(z) = \int_{x+y=z} j(x, y)$$

where $j(x, y)$ is the joint probability density function of $J(x, y)$.

Finally, if the dependence between X and Y is not known, we assume that the random variables are comonotonic. The distribution in this case is given by:

$$H(z) = \int_{x+y=z} \frac{\partial^2 \min(F(x), G(y))}{\partial x \partial y}$$

It may be the case that even the comonotonic case is not the adequate hypothesis to make about the dependency between the random variables. In such a case a general bound on the distribution of $H(z)$ should be provided that determines a limiting distribution for Z given any possible dependency. This is one of our current lines of research and a paper describing this analysis is under preparation.

The same results can be extended for an arbitrary sequence of blocks (or random variables): $Z = X_1 + X_2 + \cdots + X_n$. For details see [1].

2.2 Conditional execution $Z = \max(X, Y)$

The above discussion has shown how the distribution of the sequence of blocks can be computed probabilistically. The other main construct in the syntax tree is the conditional execution. In that case the formulation of the problem is very similar. Let X, Y and T be random variables that correspond to the expression, true and false parts of a conditional execution of E, A and B of a program segment of the form if E then A; else B; end if;. Let Z denote the distribution of the sequence. Z can be described as $Z = E + \max(X, Y)$. Where $\max(X, Y)$ is the distribution of the maximum of two random variables. The distribution $H(z)$ is given by:

$$H(z) = \int_{\max(x,y)=z} \frac{\partial^2 \min((F(x), G(y))}{\partial x \partial y}$$

The same approach can handle other types of constructs including other types of conditionals,including case statements.

2.3 Iteration

The operation for loop constructs is a combination of conditional and sequential composition. The only requirement is the identification of the maximum number of iterations of a loop, denoted by n. Then, if X, Y are the random variables that correspond to the expression guard of a loop and the body of a loop of the form for E loop B then, the distribution of the sequence Z is given by

$$Z = E + \overbrace{(E+B) + \cdots + (E+B)}^{n}$$

Other types of loops can be analysed in a similar way. The only requirement is the ability to determine maximum number of iterations of loops. Calls to other subprograms are handled by considering the call as a basic block and using the distribution of the execution time of the subprogram.

As the distributions of individual blocks do not follow standard distributions a numeric approach is the only effective solution.

2.4 Determining probability distributions

The second issue to address is to determine the actual distributions of the execution times of individual units. We use a measurement approach in which the program to be analysed is run under a large number of tests scenarios and the execution time recorded from which the probability is determined. This is a frequentist determination of probability. Other approaches are possible, more in the line of reliability analysis, the distribution could capture the distribution of the execution time on a "per incident" basis instead on a "per run" basis. In this case, the distribution of execution time is determined under particular situations (incidents) only. For example, at the critical instant or when a mode change is requested. This makes it easier to reason on probabilities of missing a deadline on a "per incident" way rather than a "per hour" measure. The method and tool described is transparent to both types of distributions, the only implication is the interpretation of the results.

It is generally easy to determine the distribution of a particular piece of code, however joint distributions are a much harder problem. The most difficult problem is that the number of experiments to perform needed to determine the probabilities grows quadratically. Besides, there are blocks in the tree for which it is not possible to determine the joint distribution because of lack of data or because the elements are not in the same level in the tree.

3 pWCET

A theory is of little use if it can not be put into practice. We have implemented the above framework into a complete toolset for probabilistic WCET analysis that covers the whole process. From automatic code analysis and syntax tree construction, to trace generation and evaluation to an efficient probabilistic calculation engine.

The pWCET toolset has the following features:

- Portable: There is a minimal dependence on the processor architecture. The structure of the program is extracted from the object code representation which requires minimal changes to a parser for different architectures. The determination of timing information is done by trace analysis and therefore a timing model of the processor is not required.

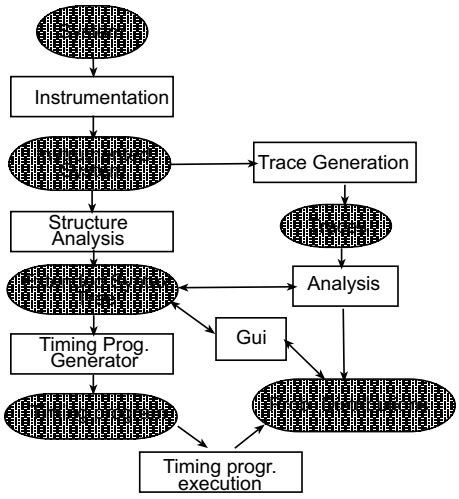

Figure 1: Overview of the pWCET toolset

- Fully flexible timing program generation: The generation process is user programmable and therefore allows different types of timing programs to be produced. The tool is able to generate both static (integer) timing programs, probabilistic programs and symbolic programs. In this paper we describe the probabilistic framework only.
- Generic: The source of the data for tracing analysis can be provided in different ways. For experimental purposes the trace can be generated using a processor simulator or by directly measuring the execution of the code on the target platform.
- Automatic loop analysis: maximum number of iterations of loops are deduced automatically from trace analysis.

The general process of the analysis and tool components is described in Figure 1.

The stages of the analysis are as follows:

- Structure analysis: the program is analysed and a syntax tree representation of the program is generated.
- Instrumentation: insertion of calls of a trace logging mechanism into the program.
- Trace generation: this step produces execution traces which capture the execution times of individual blocks for different runs of the program.
- Trace analysis: traces are parsed and distributions of individual blocks produced. Also joint distributions are captured and loop analysis determined.

- Timing program generation: a traversal of the tree generates a program that will compute the WCET of the program.
- Timing program execution: calculation of the WCET.
- Analysis of results: graphical user interface for browsing the program under analysis and the visualisation of the probability distributions.

The following subsections review each of these stages and the tool support in detail.

3.1 Instrumentation and trace generation

The aim of the instrumentation stage is to enable obtaining execution traces. An execution trace is a list of pairs (instruction,timestamp) that describe the time at which a particular instruction in the program was executed for a particular run. From this execution trace the path that the program followed as well as the different timing of the block is determined.

There are two modes of analysis, using a cycle accurate processor simulator or by directly executing the program on the target architecture. If using a cycle accurate simulator, the program does not need to be instrumented as the trace is produced by the simulator. The structural analysis determines which are the starting and ending addresses of each block of code and by parsing a log of the execution trace produced by the simulator it is able to produce the execution traces.

If the traces are determined by direct observation of the program then a mechanism to determine execution traces has to be embedded into the program and support by the OS included. This is done by manually or automatically inserting instrumentation calls into the source code, or by automatically adding the instrumentation code into the already compiled assembly code. The execution of the code results in a set of observed execution traces. The traces need to be extracted from the target hardware. These traces can then be processed by the pWCET tool (at the time of writing, the automatic program instrumentation is not yet functional).

The current demonstration version uses the simplescalar processor simulator to generate the traces, however the tool also accepts traces generated externally. The simplescalar is a MIPS cycle accurate simulator [1]. The MIPS processor has two levels of cache (second level is an integrated cache) as well as branch target buffers and out-of-order execution. Simplescalar allows the configuration of several processor configurations like changing cache size and arrangement, memory latencies, branch target prediction sizes and algorithms, etc. This is very useful to evaluate the impact of such features on the WCET of a program.

[1] http://www.simplescalar.com

3.2 Structure analysis

The structural analysis reads the non-stripped object code of the program(s) under analysis and builds a control flow graph. The program is first disassembled and the assembly code analysed. By manipulating the code at assembly level, transformations of the code included by the compiler are captured. The control flow graph is then converted into a syntax tree, called the extended syntax tree (XST). It may be the case that there are irreducible constructs (usually generated by the compiler), in this situation the analysis assumes that the whole section of the code is a block. Portability of the whole approach to a different machine architecture requires the rewriting of the lexical and syntactic analyser of assembler code which is a small task. By analysing the code at the object code level, there are no dependencies on the programming language used or only minimal.

The XST is stored as an XML file, structured as a set of trees which are made up of five types of nodes: (a) basic blocks, (b) sequences, (c) conditional code (d)loops and (e) calls. A tree is build for each subprogram, and therefore the XST of a program is made up of a series of such trees, the first one being the main program. The structure analysis also adds into the XST information for each node regarding which sections in the code it corresponds to, as well as annotations present in the source code.

3.3 Trace analysis

The trace analysis computes the distribution functions of each node in the tree from the execution traces. It uses information in the XST to determine the set of addresses that mark the start and end of each block and parses each trace accumulating the result over multiple traces. The result of the analysis is a set of execution time profiles (or ETP for short) which correspond to the discrete probability density function of individual blocks.

For selected pairs of nodes, the trace analysis is also able to determine the joint distribution function of pairs of nodes. The list of pairs of nodes to analyse is indicated before the analysis starts in a configuration file. The result of the analysis is a set of joint execution time profiles (or JEP for sort) which correspond to the discrete joint probability density function of pairs of blocks.

This is a computationally very expensive process. There may be available large number of execution traces, each one holding information of potentially long executions of the program. For example, if an execution trace records in average one every 10 instructions, then a program that runs for 1 second on a 10 MHz machine may generate up to 10^6 sampling points per second. Tests involving several hours of computation should be expected. In order to address the computational complexity the process of trace generation and analysis has been parallelised and the current implementation is able to generate the traces in a local mode (single node) or on a distributed mode using a Beowulf cluster. A special program running on a node of the cluster is responsible to distribute the work to the different nodes and merge the results after the computation has

been performed.

A second component of the trace analysis is loop identification. The information of which blocks form a loop and the nested loop structure is extracted from the syntax tree. From this information, a loop trace can also be generated. A loop trace is an indication of the index counters of each loop hit for a particular run of the program. From this loop trace, the maximum loop iteration for each loop is extracted.

3.4 Timing program generator

pWCET has a powerful mechanism for computing the WCET of programs. This is based on separating the timing analysis into a program generation part and an execution part. This enables different types of analysis to be implemented using the same framework by providing different timing program generators.

The timing program generator traverses the tree in postorder and applies the timing schema rules to each node in the tree. The result of such procedure is a set of commands on how to compute the timing program for the given tree.

The user can direct the way the analysis is performed and which rules are applied by directly manipulating the tree and modifying the rule associated to each node. For example, one common assumption is to rewrite non-rectangular loops to indicate precisely the exact number of iterations of a block, not the (possibly) pessimistic estimate obtained by the loop analysis.

We have experimented with different formats. We currently are able to generate timing programs as Ada programs, matlab scripts and ml programs.

Matlab scripts are very helpful because it allows for fast prototyping and experimentation with different operators, however in general the computation is very slow compared to a custom build solution.

The timing program reads the distributions of its sons and computes the distribution for each node in the tree. We have implemented all probabilistic algebra very efficiently exploiting the properties of the sparse data structure used to capture probability distributions. As an illustration, a convolution of a discrete distribution in Matlab can take as long as 10 times longer than the Ada version for small data sets. For large distributions the difference grows quadratically.

3.5 Analysis of results

The different parts of the tool can be used as either scripts or through a graphical user interface depicted in Figure 2. The set of steps to perform is indicated as a toolbar at the top of the screen. The log of the output of the different phases is recorded in the log screen. Commands can also be typed in directly at the command prompt at the bottom.

The user first selects the main file of the program to analyse, secondly the program is compiled for the MIPS architecture with the necessary libraries. After compilation the program needs to be analysed. The user may select which

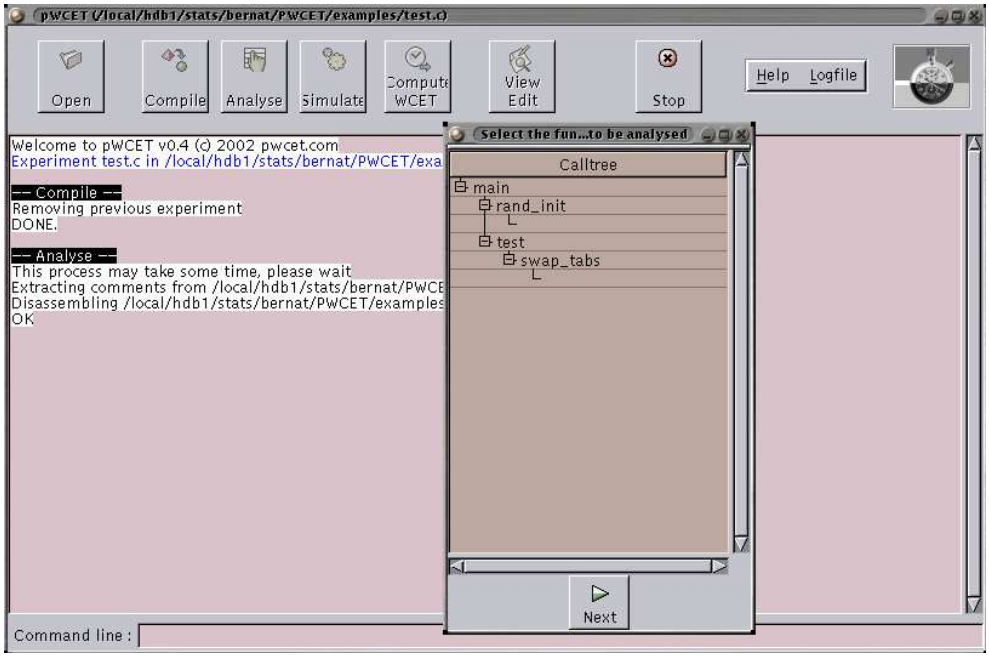

Figure 2: pWCET main window.

functions to include in the analysis. The function selection dialog can be seen in the figure too.

After code analysis traces should be generated. By selecting the simulate option the simplescalar simulator is invoked to run the program. Each run is invoked with a different run number which allows to set up a seed for random number generation, for instance. The parameters that determine the configuration of the simulator can be changed in the pWCET configuration file. This enables the evaluation of the effect that particular processor features have on the execution time of the program. The trace generation also performs the trace analysis by merging the results with previous processed traces.

After the code is analysed and traces generated the XST can be browsed using the code browser, shown in Figure 3. The browser allows to select which function to display. It displays a tree of the selected function. Different types of nodes are indicated by different colors. Each node has information of the type, source line and rule for the timing program generator. The same figure shows the screen that allows the modification of the evaluation rule for a loop node. Several operations can be performed for each node, displaying the source code corresponding to the node, the textual representation of the execution profiles, as well as the graphic plot of the distribution of probability of the node. The graph can show both measured and computed distributions.

The final stage is to launch the timing program generation and calculation.

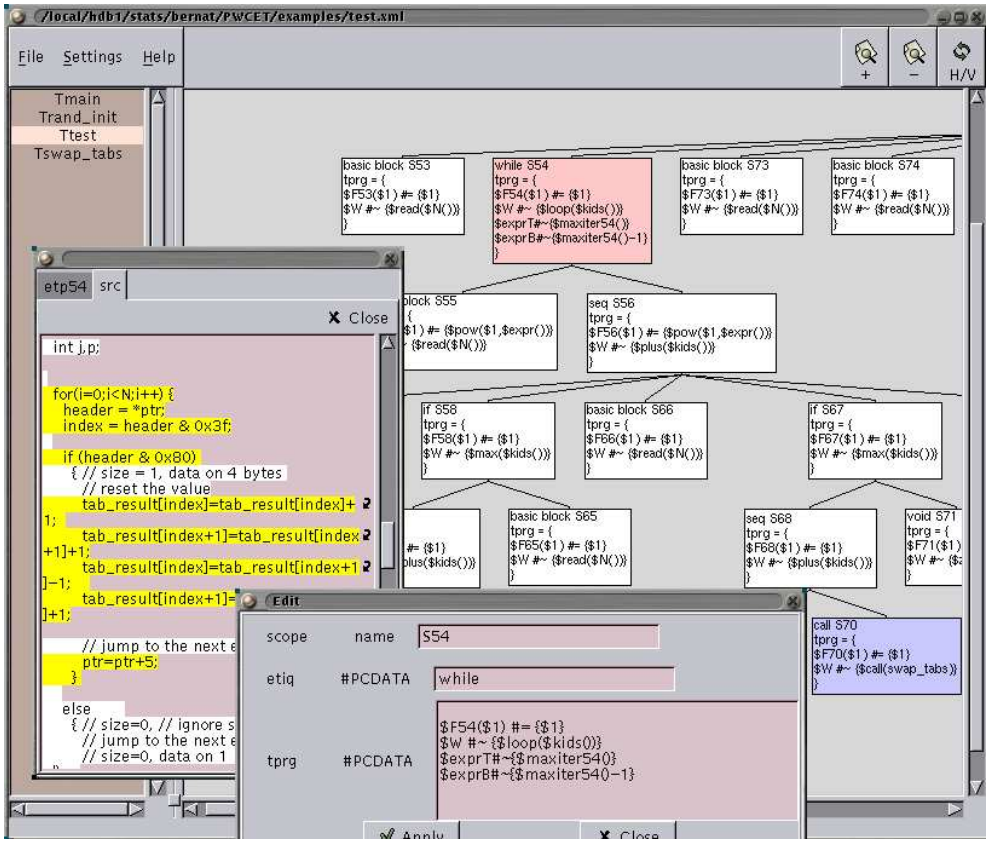

Figure 3: XST Browser showing information window of a node, some graphs and textual output.

The program performs the postorder tree traversal, extracts the rule for performing the WCET calculation from the node attributes section and generates the corresponding program to perform the calculation. The program is then executed by invoking the calculation engine.

After the timing program has been generated and executed, the computed distributions can be viewed with the XST browser. For example when plotting the profiles, both measured and computed profiles are displayed. Examples of such visualisation are shown in the next section.

4 Evaluation

In this section we illustrate the operation of the tool with an example. The program is an implementation of a message processing system. It takes packets from array ptr and decodes them. The type of message and the sign of the data part is encoded in the header. Depending on the different configurations he message is either stored in array tab1 or tab2. A fragment of the program is shown below (for full listing of the program together with other example programs see the pwcet web page).

```
void test() {
 int i,j,p,index;
 char header;
 char * ptr = (char*)data_stream;
 int  * tab_result = tab1;
 int  * tab_error = tab2;
 char * ptr2;
 char * ptr3;

 for(i=0;i<N;i++) {
   header = *ptr;
   index = header & 0x3f;
   if (header & 0x80) {
       tab_result[index]  =tab_result[index]+1;
       tab_result[index+1]=tab_result[index+1]+1;
       tab_result[index]  =tab_result[index+1]-1;
       tab_result[index+1]=tab_result[index]+1;

       // jump to the next element
       ptr=ptr+5;
     }
   else {
       index = header & 0x3f;
       if (header & 0x40) {
         // is positive
         tab_result[index]=-tab_result[index];
       }
       else {
         // is negative
```

```
          tab_result[index]=+tab_result[index];
        }
        // jump to the next element
        ptr+=2;
    }
  }
}
```

The Syntax tree of the fragment of code is shown in Figure 3. Node 54 is the head of the loop. The figure also shows the fragment of the code that corresponds to node 54 as well as the editing window where the expression to calculate the node can be modified by the user. This description is automatically generated.

The simulation generated 1000 traces. The following is a fragment of one of such traces that shows first three iterations of the loop. Note that the trace only shows execution of basic blocks (not of inner nodes in the tree), the analysis part is then responsible to derive the execution of these other nodes. The format is (timestamp node number)*. The timestamp is the cycle number in which the first instruction of the basic block is fetched. For example, in the first iteration node 66 runs for 83 cycles (34943-34860), however in the second iteration it runs for only 13 cycles (and for the rest of iterations in the loop). This is a common behaviour due to cache effects.

```
34399 53 34490 55 34519 57 34631 60 34692 63 34832 64 34860 66
34943 72 34974 55 34978 57 34995 60 35008 62 35098 64 35103 66
35116 72 35123 55 35127 57 35144 60 35157 63 35179 64 35183 66
35196 72 ...
```

The loop analysis determines that in the worst case loop 54 iterates 31 times (this number is the number of times the header is hit). This is indicated with the following maxiter rule:

```
$maxiter54 #= {31}
```

The timing program generated by the tool is shown below. This is an automatically generated ml program. Each node corresponds to an ml function that invokes in its computation recursively the functions that evaluate the sons of the node. The operation of the node is then performed, saved and control returned to the callee. The Following fragment shows the calculation of node 67. Nodes 69, and 71 are basic blocks and its distribution is read from the measured data. Node 70 is a function call to `swap_tabs`. Node 68 is the convolution of the distributions of node 69 and 70. Node 71 was never executed, and therefore is empty. Node 67 is the maximum (probabilistically) of 68 and 71.

```
(*--------- Node 69 -----------*) let w69 () =
  let result = read "ETP69" in
    write (result,"ETP69");
    result;
```

```
;;
(*--------- Node 70 -----------*) let w70 () =
  let result = (ext_call "swap_tabs") in
    write (result,"ETP70");
    result;
;;
(*--------- Node 68 -----------*) let w68 () =
  let result = conv [(w69());(w70())] in
    write (result,"ETP68");
    result;
;;
(*--------- Node 71 -----------*) let w71 () =
  let result = epzero () in
    write (result,"ETP71");
    result;
;;
(*--------- Node 67 -----------*) let w67 () =
  let result = max ((w68())) ((w71())) in
    write (result,"ETP67");
    result;
;;
```

Figure 4 shows the result of the analysis compared to the end to end measurement. The pWCET estimate is an upper bound on the WCET. The distance between the two estimates comes from the fact that the input data does not correspond to the worst possible sequence of data (this is just random messages). The pWCET builds the equivalent of the worst set of input data and plots the profile.

Generation of 1000 traces took 15 minutes on a Pentium 3 at 500 MHz, the generation and evaluation of the timing program was performed in under a minute. The complexity of the timing programs is not greatly affected by the size of the traces.

5 Conclusion

This paper has outlined the theory for probabilistic timing analysis of real-time programs and described the main components of its tool support. The main features are: portability to analyse programs running on different processors and platforms by processing execution traces obtained either by examining the log of a cycle accurate processor simulator or by observing the real system; flexibility: by allowing users to define the way the timing program is generated and therefore enabling different types of analysis. A small case study illustrates the formats of the files involved and the steps of the analysis.

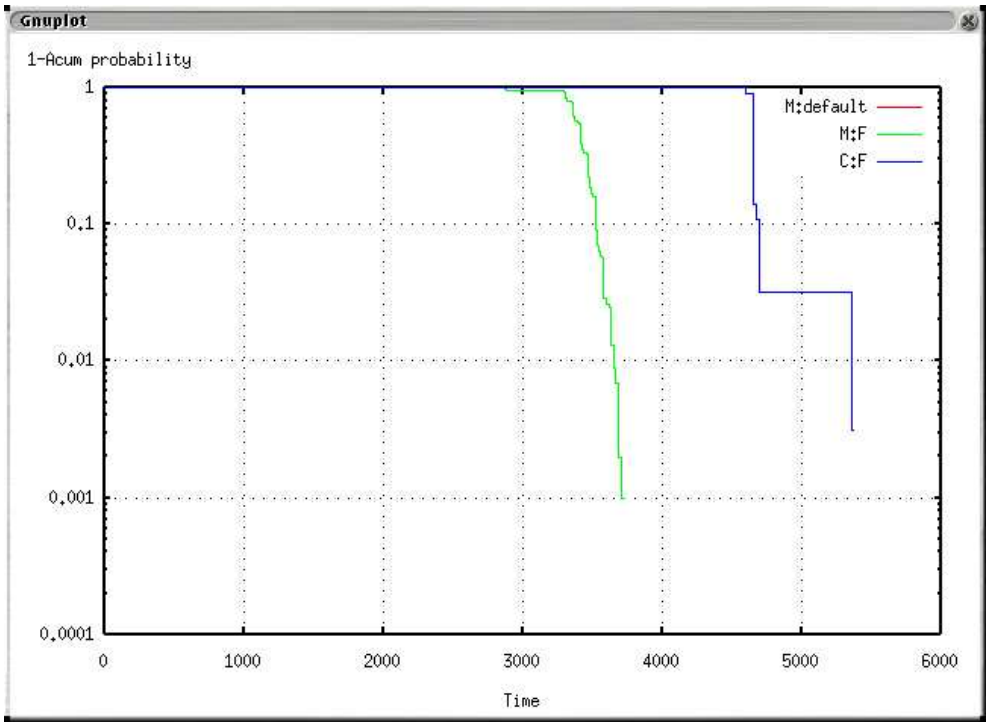

Figure 4: pWCET analysis of node 54. M= Measured, C=Computed. overestimation is due to lack of generating the worst possible input data.

References

[1] G. Bernat, A. Colin, and S. Petters. Wcet analysis of probabilistic hard real-time systems. In *RTSS, Real-Time Systems Symposium*, Austin, TX, USA, December 2002.

[2] Alan Burns and Stewart Edgar. Predicting computation time for advanced processor architectures. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, June 19–21 2000.

[3] Alan Burns and Stewart Edgar. Statistical analysis of WCET for scheduling. In *Proc. of the IEEE Real–Time Systems Symposium (RTSS'01)*, London, United Kingdom, December 4–6 2001.

[4] Antoine Colin and Guillem Bernat. Scope-tree: a program representation for symbolic worst-case execution time analysis. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, Vienna, Austria, June 19–21 2002.

[5] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Journal of Realtime Systems*, 18:249–274, 2000.

[6] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real–time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, Montreal Canada, June 19–20 1998.

[7] Y. A. Liu and G. Gomez. Automatic accurate time–bound analysis for high–level languages. In Frank Müller, Azer Bestravros, et al., editors, *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, Lecture Notes in Computer Science, pages 31–40, Montreal Canada, June 19–20 1998. ACM SIGPlAN, Springer–Verlag.

[8] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Realtime Systems*, 17(2/3):183–207, November 1999.

[9] Frank Müller. Timing analysis for instruction caches. *Journal of Realtime Systems*, 18:217–247, 2000.

[10] C.Y. Park and A.P. Shaw. Experiments with a program timing tool based on source–level timing schema. *IEEE Transactions on Computers*, 24(5):48–57, May 1991.

[11] Stefan M. Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures.* PhD thesis, Institute of Real–Time Computer Systems, Technische Universität München, Munich, Germany, 2002.

[12] F. Stappert, A. Ermedahl, and J. Engblohm. Efficient longest executable path search for programs with complex flows and pipeline effects. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2001)*, pages 132–140, Atlanta, Giorgia, USA, November 16–17 2001.

[13] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by spearated cache and path analysis. *Journal of Realtime Systems*, 18:157–179, 2000.

Industrial Requirements for WCET Tools — Answers to the ARTIST Questionnaire —

Reinhard Wilhelm*
Compiler Design Lab
Saarland University
66041 Saarbrücken, Germany
wilhelm@cs.uni-sb.de

Jakob Engblom*
Department of Information Technology
Uppsala University
SE-751 05 Uppsala, Sweden
jakob.engblom@it.uu.se

Stephan Thesing*
Compiler Design Lab
Saarland University
66041 Saarbrücken, Germany
thesing@cs.uni-sb.de

David Whalley
Computer Science Department
Florida State University
Tallahassee, FL 32306-4530, USA
whalley@cs.fsu.edu

Abstract

This paper presents the results of a questionnaire for WCET tool users undertaken by the ARTIST project. The aim is to get information on the requirements for a WCET tool as seen by the possible users of such a tool.

1 Introduction

ARTIST (Advanced Real-Time Systems) is an IST project of the EU. It's goals are to coordinate the R&D effort in the area of Advanced Real-time Systems so as to:

- Improve awareness of academia and industry in the area, especially about existing innovative results and technologies, standards and regulations.
- Define innovative and relevant work directions, identify obstacles to scientific and technological progress and propose adequate strategies for circumventing them.

ARTIST undertakes several Actions:

- Hard Real-Time Systems: Consolidate and further improve a strong European competence and know-how that is strategic for safety- or mission-critical applications (Synchronous languages, TTA, Fixed priority scheduling).
- Component-based Design and Development: Transfer, enhance interaction between teams working on compositionality/composability problems and software and systems engineering teams involved in the definition of standards e.g. UML, SDL.
- Adaptive Real-Time Systems for Quality of Service (QoS) Management: Soft real-time approaches and technology for telecommunications, large open systems and networks teams with expertise in real-time operating systems and middleware.

The Work Directions are

- to establish a roadmap for future directions in advanced real-time systems.
- to propose curricula for Education and Training in advanced real-time systems.
- to disseminate results and to undertake international collaboration.
- to create strong two-way ties with industry.

ARTIST will

- Focus on system-centric approaches by adapting or further extending them to real-time software and hardware technology.
- Consider generic approaches and will not be biased towards particular application areas.

[1] supported by ARTIST, an IST project of the EU

- Use a diverse selection of suitable applications to evaluate and further specialize the approaches, whenever appropriate.
- Establish good contact and interaction with application-specific projects for essential technologies and infrastructure as well as relevant projects on control theory and dynamic systems.

ARTIST established several Working Groups, one of them on *Timing Analysis*. It was put under the direction of Reinhard Wilhelm. This working group set out to

- identify the requirements of (potential) customers of timing-analysis technology,
- collect information about available implementations of this technology.

This article reports about requirements of industrial users for WCET tools. The commonly used term worst-case execution time (WCET) is a misnomer. In general, worst case execution times cannot be determined, even for terminating programs. The reason is that the worst-case input may be unknown. All existing so-called WCET tools actually compute upper bounds on execution times. These upper bounds may occur for some execution or they may be overestimations, in which case they never occur. The term "upper bound" on execution times indicates that these bounds are *safe*, they never underestimate potential execution times. But they should also be *tight*, i.e. as close as possible to the worst-case execution time.

A questionnaire containing the questions listed below with their answers was put on the web and potential customers in the aeronautics, automotive, and electronics industry asked to fill in their answers. Only 12 persons answered the questionaire. However, as one can see below, they were in quite influential positions taking decisions for quite large groups of developers. Therefore, we feel that the results should be accepted as significant.

2 Results

In this section we present the results as they have been received in the questionnaire. For every question of the questionnaire, we first give the question itself and then the (summarized) answers. In the tables, the column titled '#' gives the number of answers corresponding to the item in the first column.

1. What is your job title? The answers showed that the right people were asked. Jobs title given were *Developer, Engineer, Fellow, Systems Engineer, CTO R&D, Team Leader, Project Manager, Manager System Development, Director, Program Manager, Chief Scientist*.
2. What is the size of the group you are supervising?

Number of People	#
1-5	9
6-10	1
21-100	2

3. For which applications/systems do you need WCET tools in your own development?
   - Automotive (Engine Control), guiding systems, automotive control units
   - automotive applications
   - avionics, on-board SW on satellites
   - Operating systems, but also customers applications.
   - Synchronous programs.
   - Embedded controllers
   - DSP embedded systems development
   - Evaluation of supplier systems
   - Embedded real-time software
4. For which target software platforms (OS, middleware) would you like to have Timing Analysis tools available?

Platform	#
Real-Time OS	11
Hardware	7
Middleware	2

5. What should the functionality of the tools be?

Functionality	#
Very rough first estimate	6
Back annotation of results into the source code	6
Proposals for cache locking	7
Stack-Extent Analysis	9
Best Case Execution Time	7
Annotated Assembly listing	1

   Other analyses that the tools should be able to perform include
   - Execution time coupled to its probability
   - Average Execution Time
   - Assure WCET to safety/criticality level required; distributions/histograms
   - Maybe verify some pre-, postconditions and invariants of functions
   - analysis of code parts with disabled interrupts
   - measure OS and communication impact

Other functionality that should be included into WCET tools include

- Must be possible to exclude "uninteresting" paths (e.g. fatal error handler code)
- Graphical representation,
- stack-frame over-run identification

6. What is the tolerable learning effort for users of the tool? (days)?

Effort	#
2 days	2
3 days	2
5 days	1
5-10 days	1

7. How much effort for annotation of the code is tolerable?

Effort	#
Bound for Loops and Recursion	7
Locked Cache Contents	3

Other answers include

- Loop bounds often obvious, recursion prohibited, at high criticality levels
- As least as possible
- As much annotation as possible

8. What would be tolerable analysis times on realistic code sizes, e.g. 100k instructions? (minutes)

Anwers included

- 1-10 minutes
- 10 minutes
- 10-120 minutes
- 60-120 minutes
- 100000 minutes (*somebody with a lot of time! RW*)

9. Would you adopt a processor with high predictability with some loss in average case performance? Note, this may mean improved WCET!

Answers	#
Yes	9
No	3

10. Which other tools should a WCET tool be integrated with to suit your development flow? Answers were

- Enea ASF/DART, some UML-tool
- Ascet SD, Matlab/Simulink
- Schedulability, CM, traceability, requirements capture
- WCET and flow analysis must be integrated. Other tools less important
- RTOS
- Operating system configuration tool
- Version control for source and binary
- Functional simulation
- Debugger, profilers, RTOS
- Does not apply; We review supplier data.
- High-level compilation

11. Under which formal rules do you work, e.g. DO 178B? Answers included DO178B 5x, US DoD services, NASA, IEC61508, ISO9000, ECSS, ESA rules, DO-248; DO-254; AC 20.115B, usually project-tailored.

12. Do you use coding guidelines to support WCET analysis?

Answer	#
No	9
Yes	3

13. Do you need/plan to use processor architectures with cache memories, complex pipelines or branch prediction hardware for critical applications?

Processor feature	#
Instruction or data cache	8
Branch Prediction	7
Multi-level cache hierarchy	3
Superscalar out-of-order execution	5

14. For which hardware platforms (monoprocessor, multiprocessor) would you like to have Timing Analysis tools available?

Platform	#
Mono and multi processor	4
Mono processor	8

15. Do you use tools for schedulability analysis?

Tool	#
Based on Response Time/Rate Monotonic Analysis	7
Based on Time Triggered Scheduling	4
ARINC 653 (hierarchical model)	1

16. Do you use measuring of the execution time to estimate WCET?

Method	#
Via Code Instrumentation	8
Via Debug Tools (BDM, JTAG, or other interface)	6
Via a Logic Analyzer	4

Other methods mentioned:

- RTOS includes support
- Chip-internal counter.
- Review supppliers' analyses

17. How much effort do you spend in timing validation (% of development)? The 3 answers were
    - 15 % of development
    - 5-10 % of development
    - 1 % of development
18. How much development time is spent in measuring the execution time of code pieces in addition to estimating the worst-case execution time (% of development)? Only two answers were given:
    - 10 % of development
    - 5 % of development
19. Are you willing to adhere to coding guidelines to help WCET analysis?

Answers	#
Yes	10
No	2

20. For which processor architecture (PowerPC, x86, etc) would you like to have Timing tools available?

Architecture	#
PowerPC	7
ARM (ARM7/ARM9)	4
C166/7	4
x86	2
V850	2
Pentium, MIPS, Tricore, Coldfire, TMS, sharc, HC12, M16C, TX49, PPC G4, TS101, 68K	1

21. What is the maximum tolerable price per seat for such a tool (under the assumption that its use saves money spent in validation otherwise)?

Price	#
Up to 5000 $	7
Up to 10000 $	1
Up to 50000 $	1
Don't know	3

22. The following notes have been added by the respondents.
    - I'm not sure answers fully capture issues due to mixed criticality. Different applications have differing criticalities, need different levels of WCET and deadline assurance. Most systems are infeasible when all threads are at guaranteed-to-highest-assurance WCETs. Final timing analysis considers WCET bound versus assurance level, implementations use controlled load shedding to assure any inaccuracies (transient overruns) have no significant impact. Also, development process issues need to be considered. Early in the process, guestimates are used and budgets are constructed. Tools/methods must support an overall performance management process, from early capture of derived timing requirements through certification and into deployment and upgrade.
    - It is hard to give good answers to these questions, since I reply both on behalf of what we would like to use internally and what we would like to offer to the broad market. The broad market has a hard time making up it's mind ;-)
      Some comments: (question number in parantheses) (8) The faster, the more useful it is, of course. If the output made it worthwhile, we could leave a machine crunching for two weeks per MB, computing the WCET. (9) Limiting the scope of the processors limits the market for the tool somewhat. It is quite reasonable that some processors are excluded from the scope however. (19) I believe we and most customers have no trouble with limiting the language. Until you tell us what's not supported... (21) Pricing is very difficult issue.. Higher prices reduces the available market considerably. Higher prices also reduces the license seat count.
    - The tools have to work without requiring intelligence on the part of the user. People that are new to programming in C have to use these tools, so there's no chance of them understanding WCET principles or schedulability analysis techniques.
    - We use the suppliers to provide us with data to review, for certification efforts.
    - We are tool vendors potentially interested in coupling out certified code generators

3 Conclusions

Developers of Embedded Systems and the people managing them are busy people. It is hard to motivate them to

fill out a questionnaire. We can be happy about the number of influential people answering our questions. The answers show that quite a zoo of processors is used by the groups concerned. Many of them have architectural features that make the determination of WCET difficult. The answers show a growing awareness of the problem that run-time guarantees for hard real-time systems are difficult to give for these processors. The answer to question 9, the willingness to go for a processor with predictable timing-behaviour shows that a new research and development area is opening up, namely the design of such processors. The high number of respondents willing to enforce coding guidelines to support WCET analysis proves the same awareness.

Requirements of WCET tools

These are notes written by Jan Lindblad, ENEA OSE, Sweden in conjuction with the panel discussion.

Different Needs

The survey “Industrial requirements for WCET tools” conducted by Reinhard Wilhelm, Jakob Engblom et al. is really a market survey. The answers received were hard to interpret, since there is such a great variation in what different users need.

I think the picture would have be clearer if the answers were plotted against different types of industries. I propose the following simple categorization:

Safety, “Airbus”

- Have money to spend on tools
- Long sales and development cycles (many years)
- Must prove T correctness
- Uses multitude of approaches to do so
- Missing a deadline may cause loss of lives and market (Concorde)
- Small amount of code, have all sources

Motor control, “Volvo”

- Have research budget
- Medium sales and development cycles (a year or many months)
- Improved Q with T correctness
- Q work using some tools, plus testing
- Missing a deadline may cause expensive machine to crash
- Medium amount of code, usually have sources

Communication, “Ericsson”

- Have research budget
- Medium sales and development cycles (a year or many months)
- Improved QoS with T correctness
- Q work by testing
- Missing many deadlines causes penalty fees plus bad market reputation
- Huge amount of code, have some sources

Consumer, “Sony”

- Minimal research budget
- Short cycles, one shot sales
- Improved Q/QoS with T correctness
- Q work by testing
- Missing many deadlines causes bad market reputation
- Medium amount of code, have some sources

Testing vs. Calculation

Some people say testing is useless, some say testing is the only thing we can do. I say both approaches are necessary. Neither of them is good enough to stand on their own today.

Especially in the safety industry, large sets of verification methods are required. Here, both testing and calculation are corner stones.

The Future of WCET

Processing power demands are growing more rapidly than Moores law, so more and more processors have to be connected into a cluster to solve the application needs. This makes the analysis of the system behaviour more challenging since more complex processors and interconnections are being used.

The WCET researchers often tell the users to “use simpler/more predictable processors”, and for a good reason. But to use simpler processors will often increase the number of processors, and we don’t know if that makes the problem easier.

Multiple processor cores within one piece of silicon is already happening in industry.

Legal Issues

There is a licensing problem with software you buy today. When a licensing contract is signed, there is usually a paragraph that says something like “you shall not reverse engineer this code”. Users with few or no suppliers may think of this as a minor problem, but for users with many suppliers this is an issue when applying WCET tools.

New Research Field

I propose a new research field: how to combine the results of all the different methods, be that WCET, pWCET or measurement. Use flow information where there is such, inspect the source code when available.

On the Design of an Extensible Platform for Flow Analysis of Java using Abstract Interpretation

Paulo Abadie Guedes Sérgio Vanderlei Cavalcante
Federal University of Pernambuco
Center for Informatics

Abstract

This work presents a tool for flow analysis of Java programs through abstract interpretation. The system under development provides high-level informations that may be used by other modules in order to provide the WCET and BCET for .class files, without the need of code annotations.

WCET analysis tools could then share the same ground for common tasks. This allows the integration of different modules and approaches for the low-level and calculation steps. The main goals are portability, ease of use, extensibility and seamless integration on the platform.

1 Introduction

Real-time systems are those on which the correct temporal behavior is as important as the execution itself. In order to provide guarantees for the execution times, there is the need to model system tasks, and hence the need to perform a temporal analysis and estimate the WCET.

Altough there is an effort towards the creation of a reference implementation for Real-Time Java [1], there is not yet a standard way to estimate the execution time for Java programs, which is a basic need on the development of real-time systems.

This document presents work in progress to the creation of a modular platform for flow analysis of Java methods, which is the first step in the process of building a tool for WCET estimation. The analysis is performed on the bytecode level. The next sections present the tool structure and flow analysis for Java, a small sample and conclusions.

2 Tool structure

The analysis tool is based on an abstract model for the JVM. In order to understand it, a brief introduction is presented and then, the abstraction used.

2.1 The JVM model

The JVM [3] is a stack based machine whose execution sequence is controlled by a call stack[1]. The basic units on the call stack are stack frames, which are responsible for describing the execution state inside each method context. A sequence of frames can describe one possible call sequence inside the program. In our simple model, each stack frame is composed by three components:

1. PC - the program counter
2. The local execution stack
3. The local variable pool

The program counter keep track of the current instuction being executed. Instruction execution is done using the execution stack to hold partial results. Instructions can move data between local variables and the stack. For each method, there are maximum values for both sizes, the local stack and variables.

The machine word is a 32-bit integer value, and bytecodes can move data to and from the stack, change control flow and even operate directly on the local values. Every bytecode is a one byte machine instruction, and may need parameters or not.

2.2 The abstract Java VM

During program analysis, there is the need for extracting information about variable values. This is done using abstract interpretation [2], and hence there is the need for an abstract model for the JVM.

The abstract model is adapted for analysis and can keep on each local variable and on each stack position a set of values. This approach allows an safe but imprecise analysis to be performed on the bytecode level, based on the possible values for method parameters. There is, of course, a trade-off between quality of results and cost of time and memory needed to execute the analysis.

[1] There is one call stack for each Java Thread

The use of abstract interpretation is an advantage, as it allows existing code to be analysed and hence truly reused, since there is no need for mixing additional information with program code. Although some extra information may still be needed, the technique can ask just the strictly necessary in order to perform the analysis. This information doesn't need to be annotated inside the program code. This is an advantage over other techniques, as in [7], which are based on *code annotations*, because it avoids the need to insert extra code to guide the analysis tool.

Among other advantages the method is automatic, can be applied to any program written with the language and yields an approximate and safe description of the program behavior [2].

3 Initial Steps

This section shows some steps to illustrate the basic ideia. Altough the tool is still under development, this shows the basic operations for a simple function. The code illustrates a static "pow" method, which take a double value (base) and an integer exponent and calculate the value of $base^{exponent}$.

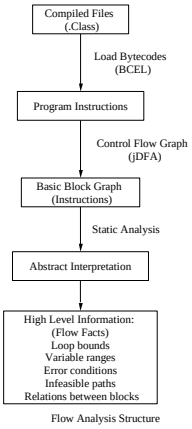

Figure 1: Flow analysis tasks

The basic tasks are: load the code, extract the bytecodes, build the basic block graph and then start the analysis as shown on figure 1. Those are performed using the BCEL [8] and JDFA [9] libraries.

The sample code in Java is shown next. The instruction format used by the libraries is offset, bytecode name, decimal bytecode value and the size in bytes. So, for instance the following string:

```
0: dconst_1[15](1)
```

means a bytecode named dconst_1 at local memory offset zero, whose binary value is 15 and the size is one byte. Some bytecodes as those in offsets 3 or 16 on figure 2 need parameters like arithmetic values.

Java example - the *pow* method

```
1  public class Example{
2    public static double pow(double base,
3                             int exponent){
4      int i;
5      boolean reverse = false;
6      double result = 1.0;
7
8      if (exponent < 0){
9        reverse = true;
10       exponent = - exponent;
11     }
12     else{
13       reverse = false;
14     }
15     for(i = 0; i < exponent; i++) {
16       result = result * base;
17     }
18     if (reverse){
19       result = 1.0 / result;
20     }
21     return result;
22   }
23 }
```

Conditional jump instructions change control flow as shown by the *true* and *false* labels in the arcs. A *null* label mean sequential execution to a bytecode that is a jump target. Execution start at position 0 and flow toward the last block, at 46.

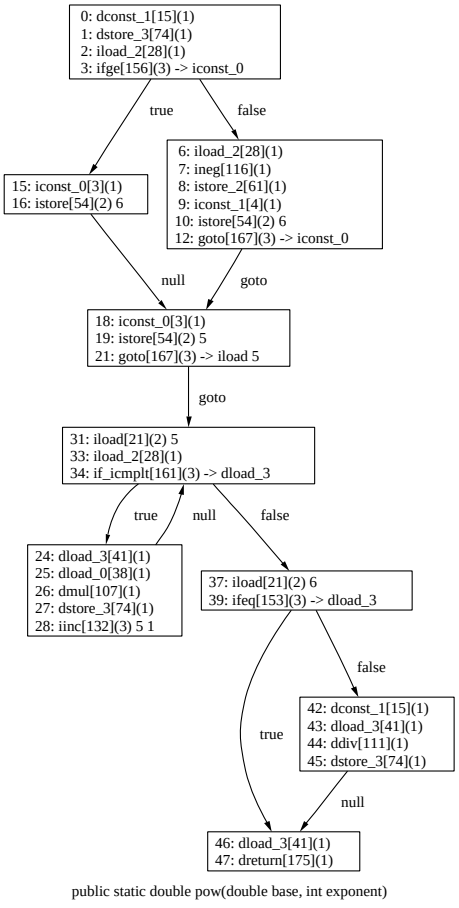

Figure 2: Basic Block Graph for the Pow method

3.1 Abstract interpretation

The abstract interpretation step use the abstract virtual machine to traverse the method structure on the basic block graph. During evaluation, abstract values are created and stored automatically [4].

The table in figure 3 contains the values for program counter (PC), conditions to the *if* and *for* blocks, the abstract stack and the result *value*. This values are calculated during the analysis[2]. In this sample, the parameters are limited to $base = \{2.0, 5.0\}$ and $pow = \{2, 3\}$. At each point, the variables and the stack slots of the abstract machine can hold abstract values. Each analysis step then evaluate a set of values during execution.

	PC	For	If	Stack	Result	i
				{}	{}	{}
1	0			{{1.0}}	{}	{}
2	1			{}	{1.0}	{}
3	2			{{2, 3}}	{1.0}	{}
4	3		f	{empty}	{1.0}	{}
5	3		t	{2, 3}	{1.0}	{}
6	15			{{0}}	{1.0}	{}
7	16			{}	{1.0}	{}
8	18			{{0}}	{1.0}	{}
9	19			{}	{1.0}	{0}
10	21			{}	{1.0}	{0}
11	31			{{0}}	{1.0}	{0}
12	33			{{0}, {2, 3}}	{1.0}	{0}
13	34	f		{empty}	{1.0}	{0}
14	34	t		{{0}, {2, 3}} -> {}	{1.0}	{0}
15	24			{{1.0}}	{1.0}	{0}
16	25			{{1.0},{2.0,5.0}}	{1.0}	{0}
17	26			{{2.0, 5.0}}	{1.0}	{0}
18	27			{}	{2.0, 5.0}	{0}
19	28			{}	{2.0, 5.0}	{1}
20	31			{{1}}	{2.0, 5.0}	{1}
21	33			{{1},{2,3}}	{2.0, 5.0}	{1}
22	34	f		{{1}, empty}	{2.0, 5.0}	{1}
23	34	t		{{1},{2,3}} -> {}	{2.0, 5.0}	{1}
24	24			{{2.0,5.0}}	{2.0, 5.0}	{1}
25	25			{{2.0,5.0},{2.0,5.0}}	{2.0, 5.0}	{1}
26	26			{{4.0,10.0,25.0}}	{2.0, 5.0}	{1}
27	27			{}	{4.0, 10.0, 25.0}	{1}
28	28			{}	{4.0, 10.0, 25.0}	{2}
29	31			{{2}}	{4.0, 10.0, 25.0}	{2}
30	33			{{2},{2, 3}}	{4.0, 10.0, 25.0}	{2}
31	34	f		{{2},{2}} -> {}	{4.0, 10.0, 25.0}	{2}
32	34	t		{{2},{3}} -> {}	{4.0, 10.0, 25.0}	{2}
33	37	f	f	empty	{4.0, 10.0, 25.0}	
34	37	f	t	{{0}}	{4.0, 10.0, 25.0}	
35	39	f	t	{}	{4.0, 10.0, 25.0}	
36	46	f	t	{{4.0, 10.0, 25.0}}	{4.0, 10.0, 25.0}	
37	47	f	t	{{4.0, 10.0, 25.0}}	{4.0, 10.0, 25.0}	
38	24	t		{{4.0, 10.0, 25.0}}		{2}
39	25	t		{{4.0,10.0,25.0},{2.0,5.0}}		{2}
40	26	t		{{8.0,20.0,50.0,125.0}}		{2}
41	27	t		{}	{8.0,20.0,50.0,125.0}	{2}
42	28	t		{}	{8.0,20.0,50.0,125.0}	{3}
43	31			{{3}}	{8.0,20.0,50.0,125.0}	
44	33			{{3},{3}}	{8.0,20.0,50.0,125.0}	
45	34	t		{{3}2empty}	{8.0,20.0,50.0,125.0}	
46	34	f		{{3},{3}} -> {}	{8.0,20.0,50.0,125.0}	
47	37			{{0}}	{8.0,20.0,50.0,125.0}	
48	39			{}	{8.0,20.0,50.0,125.0}	
49	46			{{8.0,20.0,50.0,125.0}}	{8.0,20.0,50.0,125.0}	
50	47			{{8.0,20.0,50.0,125.0}}	{8.0,20.0,50.0,125.0}	

Figure 3: Abstract interpretation

Line 1 is the first instruction of the method. Lines 4 and 5 are the first choice, on which there is only one possible branch for the values of *exponent*. There are two lines for the same instruction because it must be analysed on the environment where it may become true and false [4]. At 13 and 14, there is the first time the loop condition is evaluated. At this time, there is not yet any value that can make the loop finish. This means that the loop will be executed at least once.

The points at lines 21 and 22 are the next time the loop condition is evaluated, as well as in 31 and 32. Lines 31 and 32 are the first time where there is a value that allow the loop to stop. Then, now the information about the minimum value for the loop count (2) is known. Here it is necessary to analyse both sides: the branch where the loop will stop and the other where the loop continues.

Lines 33 to 37 are the execution of the branch for loop count 2. The method may return here, so the final values for the result are partially known. From 38 to 45 there is one more iteration and then the loop finishes. In this point the maximum loop count is known (3). The method can finish at line 50, so here all the final values for the result are known.

It is possible to notice that during analysis, the code from 6 to 12 and from 42 to 45 never executes. This is interesting because it means that for this particular input, those blocks can be excluded from the WCET calculation.

3.2 Analysis results

There are two points on the code where the method can return, at lines 37 and 50. The final estimated value for the *result* variable will be calculted based on the values at those points. This value is $result = \{4.0, 10.0, 25.0\} \cup \{8.0, 20.0, 50.0, 125.0\}$. In the same way, there is information available about the final values of the other variables.

The real values for these input parameters are $result = \{4.0, 25.0\} \cup \{8.0, 125.0\}$, which happens when exponent is 2 and 3. It can be noticed that *result* is safely overestimated, as all possible values were found and there are some values ($\{10.0, 20.0, 50.0\}$) which actually never happens during the execution.

The loop count is correctly calculated, as there is not any statements which may introduce extra values during evaluation. In the same way, the other values are correctly calculated for this case.

In this simple example, the analysis finished as expected. This is not the general behavior, because there may be structures that may not stop such as unbounded loops. In order to deal with other types of programs it may be necessary to limit the analysis in some way [4].

Another issue is thar each selection double the state space. In the same way, each loop double at every iteration. This may lead to an exponential

[2]The others are constants or were ommited

explosion on the size of the state space. Dealing with state space explosion can be achieved through the use of merging, possibly at the cost of a less precise final answer.

After the execution, the system will present the flow information calculated, such as the existing paths. Some false paths may also be presented, like executing the *true* branch on the first *if* and the *false* branch on the second.

4 Conclusions

This work presented a system for flow analysis of java programs, which is the first step towards a complete WCET estimation tool. The system is flexible due to the ability to work with bytecodes directly. This is a good choice for several reasons, including the fact that there are many languages and tools that generate class files for the Java Virtual Machine, and the possibility to reuse and analyze classes whose source code is not available.

A prototype tool is under development, to analyse class files and extract the flow information needed to the other steps of the timming estimation. Further extensions involve the creation of modules to execute low-level analysis and the WCET final calculation, based on a specific platform.

References

[1] Bollela, G.; Gosling, J.; Brosgol, B. M.; Dibble, P.; Furr, S.; Hardin, D.; and Turnbull, M. **"The Real Time Specification for Java"**. Addison Wesley, 2000.

[2] Gustafsson, J. **"Analyzing Execution Time of Object-Oriented Programs Using Abstract Interpretation"**. PhD Thesis, Department of Computer Systems, information Technoogy, Uppsala University, Swedden. May 2000.

[3] Lindholm, T.; and Yellin, F. **"The Java Virtual Machine Specification"**. 2nd ed. Addison-Wesley, 1999.

[4] Gustafsson, J.; and Ermedhal, A. **"Automatic derivation of Path and Loop Annotations in Object-Oriented Real-Time Proframs"**. Workshop on parallel and distributed real-time systems, 1997. 11th IEEE International Parallel Processing Symposium (IPPS'97).

[5] Altenbernd, P. **"On the false path problem in hard real-time programs"**. In Proceedings of the 8th EUROMICRO workshop on Real-Time Systems, 1996.

[6] Ermedahl, A.; Engblom, J. and Stappert, F. **"A Unified Flow Information Language for WCET Analysis"**. 2o. International Workshop on Worst-Case Execution Time Analysis. Technical University of Vienna, Austria. June 18, 2002.

[7] Bernat, G.; Burns, A.; and Wellings, A. **"Portable Worst-Case Execution Time Analysis Using Java Bytecode"**. In Proceedings of the 12th EUROMICRO conference on Real-Time Systems, 2000.

[8] Dahm, M. **Byte Code Engineering**. Proceedings J1T 99, 1999.

[9] Mohnen, M. **An Open Framework for Data-Flow Analysis in Java**. Workshop on Intermediate Representation Engineering for Virtual Machines. June 2002, Dublin, Ireland.

Elimination of Unstructured Loops in Flow Analysis

Christer Sandberg
Department of Computer Science
Mälardalen University, Västerås, Sweden
christer.sandberg@mdh.se

June 25, 2003

Abstract

A static WCET analysis should ideally be able to handle most kinds of program constructs. Unstructured loops are usually avoided nowadays in code written by humans, but they may be generated by tools, e.g. state machine code generators. They may also be introduced by an optimizing compiler. Therefore, the WCET analysis should be prepared to handle unstructured loops.

Most of todays flow analysis requires code without unstructured loops. This paper proposes that unstructured loops should be transformed into structured loops as a preprocessing step to the actual flow analysis. The advantages with elimination of unstructured loops are:

- The various steps of the analysis does not need to handle unstructured code separately. This includes e.g.
  - Syntactical analysis for finding loops that are possible to reduce
  - Finding merge points for flow information
  - A single upper bound of a loop
- Tree-based calculation can be used without modifications, since it assumes all loops to be structured

1 Introduction

The task of estimating WCET using the static approach can be divided into three phases:

- Flow analysis
- Low level analysis
- Calculation

We are developing a tool that implements these phases in a modular manner [GLB+02, Erm03]. The goal is to estimate the WCET for all program constructs (provided that the program terminates) based on an automatic analysis, i.e. without the need for manual annotations. The purpose of the flow analysis is to detect infeasible paths and to calculate upper loop bounds. There are different approaches with different properties to such an analysis, see [Erm03]. We have chosen to use abstract interpretation on an intermediate code level. One advantage of working with intermediate code is that a certain amount of optimizations may have been done before the flow analysis take place.

The abstract interpretation calculates safe values of variables with respect to loop bounds. Abstract interpretation may require a lot of computational power, the amount depending on the analyzed program as well as its input data. In order to simplify the calculations we can do a number of approximations, e.g. merging of states and simplified calculations of loop bounds. If approximations are made this may lead to overestimations (a less tight WCET) due to the requirement of being safe.

Two of the problems that a WCET tool that is useful in practice has to face are: the computational requirements of the analysis and the accuracy of the result. The occurrence of unstructured loops in the analysis may enlarge these problems.

1.1 Reducing Computational Requirements

In order to reduce computational requirements, we use the following methods [GBS03]: representing abstract values by a single interval, merging states, collapsing loops and reducing

the information needed for iteration count of loops.

The abstract values are represented by a single interval. An abstract value is the set of all possible distinct values that a variable can hold at a certain point of the execution. In order to decrease the amount of data to process we store these values as a single interval in the current implementation of our tool. In case a value of a variable controls the number of iterations of a loop this approximation does not need to decrease the tightness of the WCET value.

Merging of states. Suppose there are two distinct sub paths in the flow graph, p_1 and p_2, that have at least the start and end nodes in common. For each of these paths, the abstract interpreter will calculate different states, i.e. different abstract values of the involved variables. To reduce the amount of data to be processed, these states can be merged to a single state in one of the common nodes of p_1 and p_2, meaning that for each variable in any of the states the abstract value in the new state is calculated as the interval containing all values. We call such common nodes merge points. In general a merge may lead to a larger overestimation but on the other hand it can speed up the analysis time considerably. For example, the following positioning of merge points can be considered:

- at function end
- at loop termination
- at each loop iteration
- after if statements

The alternative is selectable in our tool. A suitable merge point that is expected to have a big effect is once in each iteration of a loop in the flow graph. Since the loop header node is the only node in a loop that is guaranteed to be taken in every iteration, it can be used as a merge point.

Collapsing loops. By use of syntactical analysis we can identify a certain set of loop constructs that can be transformed to a closed form expression. Depending on the analyzed program, this may have a big influence on the need of computation.

Iteration count of loops. The iteration counts of inner loops may be either calculated separately from the outer loop or as a sum of all iterations of the outer loop. Again, the reduction of data needed to be handled in calculation when choosing the latter is payed for as a possibly less tight final result. However, for simple loops there need not be any reduction of the tightness.

1.2 Scope Graph

We create a *scope graph* to provide our tool with a structure of the program on a higher level than the flow graph [GBS03]. The scope graph is a directed acyclic graph of scopes. Each scope is a container for a certain possibly looping program construct like a loop or a function. The scope graph is therefore based on a call graph merged with the flow graphs [EE00]. The scope graph can support both the flow analysis and the low level analysis with structure information of the program. A scope contains a non-empty set of nodes that refers to basic blocks in the flow graph.

Figure 1 depicts a loop with the single node 5, constituting a scope. The function itself will constitute a scope, which subordinates the loop scope. Scopes forms a tree hierarchy.

The flow information like iteration counts of loops and edges is expressed using so called flow facts, see [EE00]. Each loop needs to be annotated with a flow fact giving the upper bound of the iteration count. In addition other flow facts can be created to obtain a more tight final WCET. The flow facts can also be bound to certain conditions.

An important feature of the scope graph is that the nodes in the scopes are not the actual nodes of the flow graph, but rather pointers to these (in figure 3 these nodes are annotated with the numbers of the nodes in the control flow graph that they refer to). This gives a flexibility to express the flow information in a context sensitive manner, since a single flow graph node can be pointed to from different scope nodes. This feature will be used in the following.

2 The Problem of Unstructured Loops

As explained in the introduction we have to manage unstructured loops. An unstructured loop is a loop without a single header, i.e. a node (basic block) in the flow graph that dominates all nodes in the loop body [ASU86].

To analyze unstructured loops one of the following may be a solution:

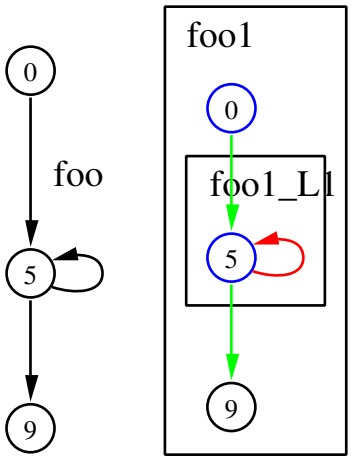

Figure 1: The flow graph and the corresponding scope graph for a function containing a loop.

- [GBS03] describes a method based on attaching one iteration couter to each loop header. This may lead to overestimation.
- Attaching one iteration counter to each basic block. This will give a lot of flow information.

Both solutions have clear disadvantages. Therefore, we have developed a method to transform unstructured loops into structured.

```
int irr(int x)
{
   if (x <  0)
      goto L1;
   do {
      x++;
     L1:
      x+=2;
   } while (x<10);
   return x;
}
```

Figure 2: A simple C-function containing one unstructured loop.

3 Eliminating Unstructured Loops

Unstructured loops in a control flow graph can be seen as loops with entry edges to more than one node in the loop. Actually, an unstructured loop can be seen as different loops sharing (parts of) the same code. In some cases the unstructured loop may actually have been created from different structured loops, merged together by a code size optimizing compiler. Therefore a straightforward way to eliminate unstructured loops is to reverse that step: for each entry of an unstructured loop we create a scope that contains only that entry edge ignoring the other, hence resulting in a scope that has a single header, see Figure 3. This can easily be done because of the scope graph construct proposed by [EE00]. Since each scope node is a pointer to a basic block rather than the basic block itself, it is possible to have duplicate scopes refering to the same piece of code.

Elimination of unstructured loops in this way can be done in a straightforward manner by use of well-known techniques. The advantages with elimination of unstructured loops are:

- Merge points in abstract interpretation can be found without extra effort in the analysis. Since the unstructured loops will be transformed to structured loops there is no difference in the handling.

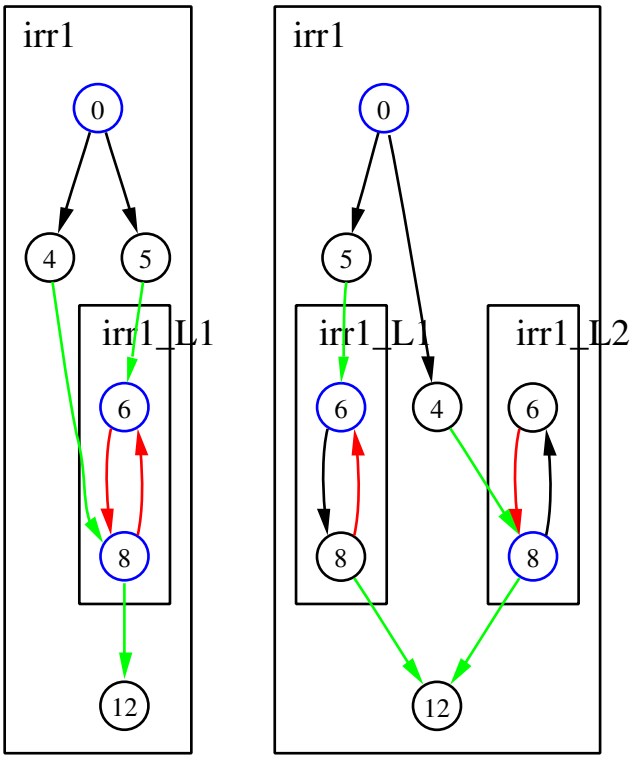

Figure 3: Scope graphs for the unstructured program in Figure 2. The left graph maps directly to the unstructured control flow graph. On the the right hand side there is a scope graph for the same program where the unstructured loop is replaced by two structured.

- The upper bound of originally unstructured loops will not be less tight, which they would have been if the loop had been analyzed in unstructured form.
- It makes tree-based calculation possible to use. The tree-based calculation is not able to handle unstructured loops.
- The various steps of the analysis will be simpler if they can assume that all loops are structured.

There exists several algorithms that can be used to identify unstructured loops [Ram02]. In our implementation we use DJ-graphs, but another can be used as well. One advantage of DJ-graphs is that they are relatively simple to implement. One disadvantage is that the elimination of multiple entry edges of an unstructured loop may expose nested loops, structured or unstructured, that was not discovered by the algorithm.

4 Conclusions

This paper suggests transforming unstructured loops to structured in the flow analysis part of a WCET calculation tool. The method is based on known technologies. We have implemented this as a preprocessing step in our tool for WCET calculation and verified that the abstract interpretation is capable of analyzing originally unstructured code, without any special modification except the preprocessing step. We have also verified that the final WCET calculation, including path-based calculation, works as expected.

5 Future Work

In case the elimination of unstructured loops exposes loops that was not found by the DJ-algorithm, our calculations will fail. To solve all cases of unstructured loops we need to refine the implementation of the loop analysis in our tool. One straightforward method would be to apply the DJ-algorithm recursively, once for each copy of an unstructured loop. Another solution can be to implement the node splitting method, see [ASU86] and [UM02].

References

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986. Generally known as the "Dragon Book".

[EE00] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS'00)*, November 2000.

[Erm03] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Faculty of Science and Technology, Uppsala University, June 2003.

[GBS03] J. Gustafsson, N. Bermudo, and L. Sjöberg. Flow Analysis for WCET calculation. Technical Report 0547, ASTEC Competence Center, Uppsala University, URL: `http://www.mrtc.mdh.se/publications/0547.ps`, March 2003.

[GLB+02] Jan Gustafsson, Björn Lisper, Nerina Bernmudo, Christer Sandberg, and Linus Sjöberg. A Prototype Tool for Flow Analysis of C Programs. In *Proc. 14th Euromicro Conference of Real-Time Systems, (ECRTS'02)*, pages 9–12, 2002.

[Ram02] G. Ramalingam. On Loops, Dominators, and Dominance Frontiers. *ACM Transactions of Programming Languages and Systems*, 24(5):455–490, September 2002.

[UM02] S. Unger and F. Mueller. Handling Irreducible Loops: Optimized Node Splitting versus DJ Graphs. *ACM Transactions of Programming Languages and Systems*, 24(4):299–333, 2002.

A Survey of Methods to Improve ILP-based WCET Analysis

Xianfeng Li
School of Computing
National University of Singapore , Singapore 117543
lixianfe@comp.nus.edu.sg

Abstract

Integer Linear Programming technique has been used in WCET analysis. Its nature of performing the components of WCET analysis integrately reduces pessimisms that happen to separated approaches. However, it has its own problems in terms of both accuracy and performance. In this paper, we discuss the two issues and survey/propose several methods that can possibly improve either or both of them.

1 Introduction

Integer Linear Programing (ILP) technique has been adopted in WCET analysis. Li, Malik and Wolfe used it to model instruction cache [2]. In our past work, we have successfully modeled branch prediction as well as its interaction with instruction cache [1, 3]. As the ILP-based approach performs low level instruction timing analysis and longest path calculation in a integrated manner, pessimism due to lack of sufficient information between the two tasks in seperated analysis is reduced. Despite the benefit that ILP-based approach has, there are several problems we need to deal with.

2 Improving the Quality of Program Path Analysis.

As discussed in [2], there are two groups of constraints in the ILP-based framework: *structural constraints* and *functional constraints*. Structural constraints are obtained from the program's Control Flow Graph (CFG) and low-level timing analysis, which typically builds a set of graphs where constraints on instruction timing are derived (from now on we use the term *timing graph* for this sort of graphs). In [2], a Cache Conflict Graph (CCG) is constructed for each cache line to capture cache misses related to that cache line. Then corresponding constraints are derived from those CCGs. Functional constraints are either provided manually or derived from program analysis to bound loop iterations, recursion depth, or infeasible paths. These constraints enumerate the paths *implicitly*, thus avoid the path explosion problem that happens to approaches relying on *explicit* enumeration of program paths.

However, this implicit enumeration has its own disadvantage. It often cannot represent a program path exactly. In other words, given the constraints derived from the path information, the ILP solver can explore a larger set of paths than the actually feasible ones. Consider the following example.

```
for (i=0; i<100; i++) {
  if (i < 25 || i >= 75)
    A;      // A maps to cache line X
  else
    foo();  // B in foo() maps to X
}
```

Clearly, A and B in $foo()$ are mutual exclusive and there are only two iterations where one displaces the other from cache line X (when $i = 25, 75$). However, given the constraint $A+B = 100$, the ILP solver can explore an infeasible path where A and B execute alternatively throughout the 100 iterations, hence the cache misses will be much more than the actual situation.

We discuss two possible methods to improve the quality of program path analysis for the ILP-based WCET analysis:

1. To put functional constraints on timing graphs, e.g., constraints on CCG edges.
2. To use analysis elements other than basic blocks, e.g., acyclic paths accross a loop body.

The idea of method 1 is that, once we discover a program segment whose behavior is independent of the input data, or there is an infeasible path which involves several blocks, we can perform program flow analysis and derive constraints from the hardware states we are modeling. In the above example, a path through both A and B in one iteration is an infeasible path, by doing data flow analysis, it can be found that A and B displace each other just twice: once that B displaces A and another time that A displaces B. These two constraints are put on the

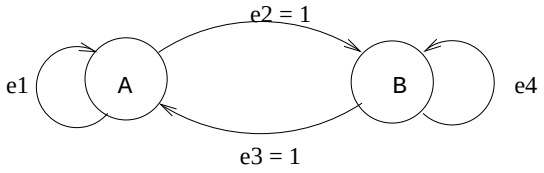

Figure 1: Cache Conflict Graph

edges in Figure 1. They prevent the ILP solver from assigning $e_1 = 50$ and $e_2 = 50$ for the worst case.

The motivation for method 2 is that, given a loop body, the path representation contains more global information than the block representation does. Consider the same example again: with block representation, the constraint can be generated is: $A + B = 100$. By using path representation, we have a better constraint. Suppose the infeasible path through both A and $foo()$ is labelled as P, then we have $P = 0$, which means that this path is infeasible. Another benefit of path representation is to reduce the problem's complexity by removing some of the feasible paths which cannot contribute to the worst case. Given two paths P_1 and P_2 acrossing the same loop body. If P_1's maximal possible cost is less than P_2's minimal possible cost and P_1's impact on furture execution cannot make up this gap, then it is safe to conclude that P_1 has no chance to contribute to the worst case because P_2 always contributes more than P_1 does. Therefore P_1 can be practically deemed as an infeasible path.

Method 2 has the drawback of increasing the complexity of the timing graphs. For example, Cache Conflict Graphs will have more nodes and edges than the original ones using basic blocks as elements.

3 Reducing ILP Solving Time

As many researchers in this community have observed, there is a conflict between ILP-based approach's tight WCET results and its high computational complexity. This problem becomes serious when analyzing real-life programs, which normally have much larger sizes than benchmark programs, or when more advanced micro-architectures are modeled, even small programs will result in fairly complex analysis and long computation time.

To reduce the computation time of the ILP approach, Theiling and Ferdinand combined *abstract interpretation* (AI) and ILP for WCET analysis in [4]. They used AI to compute properties of programs. After that, the execution time of each analysis unit becomes a constant so that the complicated timing graphs modeling micro-architectures are not necessary anymore. It greatly reduces the complexity of ILP formulation and the WCET problem can

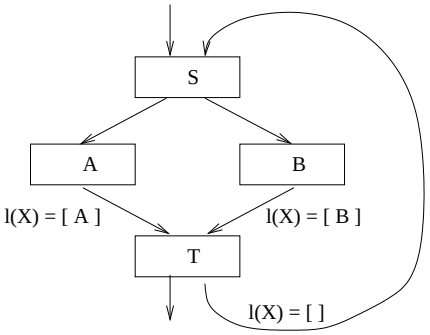

Figure 2: **must analysis** of Abstract Interpretation

be solved fast. However, this is still a separated approach and the pessimism discussed in section 1 remains. For example, in Figure 2, suppose both block A and B map to cache line X. Then the content of line X will be A/B after executing A/B respectively. Upon the control flows merging at block T, a *must analysis* (refer [4] for details) results in an empty entry of X. Therefore in the next iteration, there will always be a cache miss no matter the previously visited block is A or B. This is not true if A or B is traversed in consecutive iterations.

Our observation here is that the trade-off between accuracy and performance is too aggressive and not adaptive. We propose a solution that makes accuracy-performance trade-off *adaptively* and is also aware of the potential overestimation it introduced. The basic idea can be described as follows. Given a program and the target hardware, we first build an ILP framework which produces WCET result as accurate as possible (like what were done in [1, 2, 3]). Then with the framework, we try to identify elements (i.e., basic blocks) which are "*unimportant*". The intuition of "unimportant element" is that, if we make pessimistic assumptions on the execution of the element in question, the overestimation resulted in is neglectable or modest (depend on our needs). Consider a typical situation, where a basic block B is visited with two possibilities: cache hit or miss. This uncertainty is the consequence of some earlier branch instruction. Now suppose the path which leads to the cache hit of B has an execution cost which is much more significant than a cache miss, then assuming B always misses introduces trivial overestimation to the overall execution time.

Unfortunately, this idea is hard to work with current ILP frameworks, which typically consist of a CFG (or CFGs related by a procedure call graph) and a set of timing graphs (i.e., Cache Conflict Graphs). Any simplification of one element's execution breaks the integrity of the corresponding timing graph, i.e., in Figure 1, suppose A can be simplified

Approaches	#Variables	#Constraints
CFG/TGs	67	114
ECFG	43	32

Table 1: ILP formulations of `matsum` (branch prediction modeling) with the two approaches

as always miss, then the self loop edge e_1 of A, which results in cache hit, is trimed off from the CCG. This results in the flow increase of e_2 and consequently more cache misses for B. Another reason that makes current ILP frameworks inconvenient for complexity reduction is that the elements (basic blocks) are not so fine-grained for finding enough chances where simplification can be made. Instead, basic blocks annotated with hardware states are better candidates because the future executions are more predictable with hardware information being carried along.

In response to the above problem with current ILP frameworks, we propose an alternative approach using single graph other than a set of graphs. This approach achieves the same accuracy as current ILP frameworks. In addition, it is convenient to perform the adaptive complexity reduction with it. This single graph is obtained by expanding the original CFG (we call the *expanded CFG* ECFG). We first introduce a concept: *block instance*. A block instance is a block under a specific hardware state (i.e., instruction cache content). If a block B can possibly execute under N hardware states, then it has N instances. An interesting property of the block instance is that its execution time is statically determined because the hardware state, which affects the execution time of instruction, is known. To build the expanded CFG, we traverse the original CFG iteratively with the hardware state being populated/updated. If a block B is traversed under a hardware state which did not appear in any of its previous instances, a new instance is created for it; otherwise the control flow is passed to the previous instance which has this hardware state. This expanding process is guranteed to terminate because hardware states are finite (quite limited with each block in practice).

Figure 3 illustrates a simple CFG and its expanded CFG modeling direct-mapped instruction cache. For simplicity, we assume block A, B and C are mapped to three distinct cache lines and the cache content related to them is annotated beside each block instance in the ECFG, i.e., $(?, ?, ?)$ of instance $A0$ means the contents of the three cache lines are unknown at the very begining. After $A0$ is traversed, the corresponding cache line is A and the cache content is updated to $(A, ?, ?)$. As mentioned ealier, all instances are free of execution time ambiguity.

We empirically compared the complexity of the two approaches: the one using CFG/TGs (timing

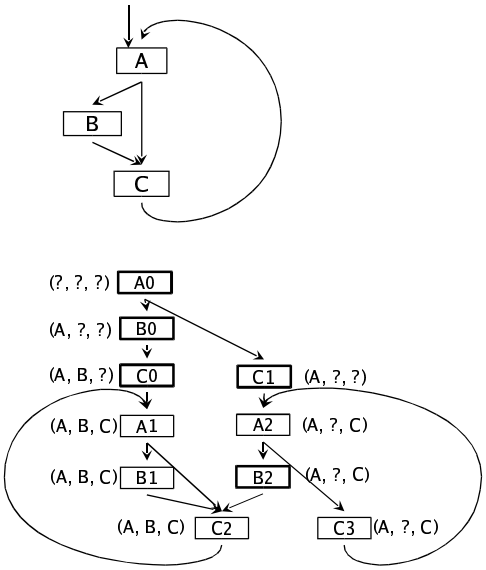

Figure 3: CFG and the expanded CFG (model direct-mapped instruction cache)

benchmarks	#Vars	#Cons	Time
fft (bpred)	594	764	0.06s
fft (bpred+cache)	1063	1254	0.12s
des (cache)	1356	1534	0.06s
djpeg	2153	1812	0.06s

Table 2: Impact of ILP problem size and nature on solving time (results of the CFG/TGs approach)

graphs) and the other one using single ECFG. In Table 1, the numbers of variables and constraints of the `matsum` benchmark (modeling branch prediction) are presented. Obviously, the ECFG has less variables and constraints than the CFG/TGs.

Another advantage is that single graph ECFG fits in the paradigm of network flow problem better than CFG/TGs does. Therefore it is likely that the ILP problem formulated from an ECFG will be solved faster than the ILP problem formulated from CFG/TGs if the two ILP formualtions have similar sizes (in terms of variables and constraints). Our assumption is supported in practice. We give the ILP problems' sizes and their solving times for several benchmarks in Table 2. The `fft (bpred)` means benchmark `fft` with modeling of branch prediction, `(bpred+cache)` is the combined branch prediction

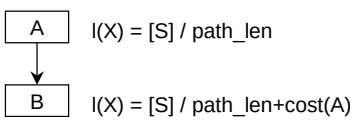

Figure 4: CFG traversal with hardware state/time information

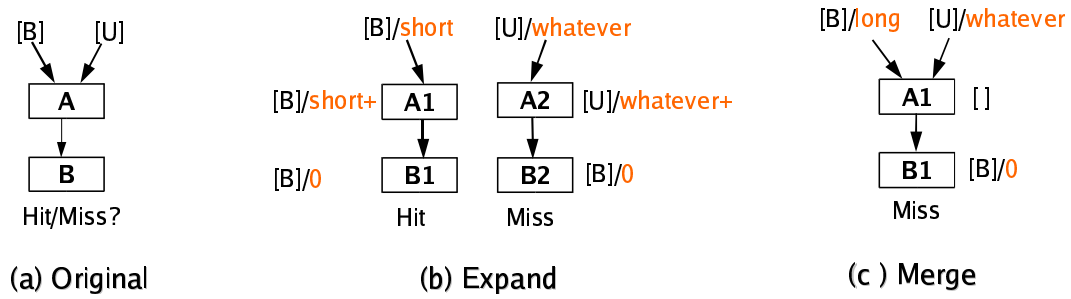

Figure 5: Adaptive complexity reduction example

and instruction cache modeling and `(cache)` is instruction cache modeling. `Djpeg` is a fairly large benchmark and we give its ILP problem size and solving time without micro-architecture modeling . So the ILP variables and constraints of it are just from the CFG plus several functional constraints. We can see that even though `djpeg` (single CFG) has much bigger size of ILP problem, its solving time is roughly the same as those of `fft (bpred)` (CFG/TGs) and `des(cache)` (CFG/TGs). Branch prediction modeling yields more complicated timing graphs than cache modeling. This partly explains why `fft (bpred)` needs the same solving time as `des (cache)` does despite the big difference in problem sizes.

The approach of using ECFG works with adaptive complexity reduction as follows. When we traverse the original CFG for building ECFG, we populate/update not only the hardware state, but also the time information for each update. In Figure 4, suppose when A is traversed, the cache line X has content S and S was fetched into cache $path\_len$ clock cycles ago, or in other words, the path traversed from S to A has the length of $path\_len$ cycles. If A does not map to line X, then after the execution of A, the path length is updated to $path\_len + cost(A)$, where $cost(A)$ is the clock cycles for executing A. With the time information, we are aware that whether a simplification, which merges two instances of a block , sacrifices neglectable/modest accuracy or not.

Figure 5(a) is an original sub-CFG. There are two incoming paths which carry different cache states: if the flow comes from the left path, block B is in cache and a cache hit is resulted when B is visited; while if the flow comes from the right path, B would be miss from the cache. In Theiling and Ferdinand approach [4], the cache line where B maps to is empty after the merge at block A and a cache miss is always resulted for B. In our approach, we make decisions according to the time information. In Figure 5(b), suppose the left path is short (the path started from the last time B is visited), then a cache miss will contribute significantly to the overall execution time of the path. In this case, we keep two block instances, which have different cache contents, for each block along the path. While in Figure 5(c), since the left path is a long path, a cache miss does not contribute significantly. Therefore we merge the two paths and only one instance is kept for each block along the path. In this case, we trade a modest accuracy loss for complexity reduction.

Besides the information of path length, the number of loop iterations is another effective time information. A path that is short but crosses several iterations is unlikely to be part of the WCET path if some longer paths across the loop body exist. Therefore simplification can be made in such situation.

The idea of adaptive complexity reduction and the single graph approach for WCET analysis are still ongoing work and we are trying to mature, implement and verify them in the near future.

References

[1] X. Li, T. Mitra, and A. Roychoudhury. Accurate timing analysis by modeling caches, speculation and their interaction. In *ACM Design Automation Conf. (DAC)*, 2003.

[2] Y-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3), 1999.

[3] T. Mitra, A. Roychoudhury, and X. Li. Timing analysis of embedded software for speculative processors. In *ACM SIGDA International Symposium on System Synthesis (ISSS)*, 2002.

[4] Henrik Theiling and Christian Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, 1998.

Discussion of Misconceptions about WCET Analysis *

Raimund Kirner, Peter Puschner
Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1
A-1040 Wien, Austria
{raimund,peter}@vmars.tuwien.ac.at

Abstract

Worst-case execution time (WCET) analysis tools are needed for the development of hard real-time systems. Despite the theoretic advances in academic research in WCET there has been hardly any impact on the industrial practice of timing analysis. The essential question is why it was not possible to provide more influential research over the last one-and-a-half decades. This paper gives constructive answers to this question. It presents a number of misconceptions about current WCET analysis. These discussions will help to guide research to the development of more useful WCET analysis techniques. This paper deals with WCET analysis techniques for hard real-time systems.

1 Introduction

The knowledge of the worst-case execution time (WCET) of tasks is crucial for the design of real-time systems. Since about more than one and a half decades, research in WCET analysis has been done to support the industry by concepts for the development of WCET analysis tools. Still there is hardly any impact on the industrial practice of timing analysis. The numerous published WCET analysis techniques and several prototype tool implementations did not trigger any ground-breaking improvements for the wide-spread industrial use of more advanced WCET analysis techniques. But still there is a strong need for useful WCET analysis tools: simple runtime measurements or manual counting of instructions are no feasible solutions assessing the code timing of increasingly complex real-time systems. This leads to the question why there is still a lack of industrial-strength WCET analysis tools.

The focus of his paper is on WCET analysis techniques for *hard real-time systems* (HRTS). The construction of HRTS requires a validation that shows that the system meets all timing constraints under guarantee [5]. In contrast, *soft real-time systems* (SRTS) do not to fulfill such strict requirements.

In this paper we highlight misconceptions about WCET analysis to present starting points for future research in this area. One of the main challenges in WCET analysis is the increasing hardware complexity of processors. The variance between optimal and worst-case performance of processors is growing significantly. The advanced hardware features make the timing prediction of modern processors quite complex. As a result, approximations in static WCET analysis produce steady increasing pessimism in the calculated WCET bound. If it is not possible to test all relevant execution scenarios, the consequences for measurement-based WCET analysis approaches are similar. The implementation of precise WCET analysis tools becomes more and more complex and the computation time needed to analyze all variations for modern processors becomes tremendously long.

To overcome the problem of the increasing complexity in WCET analysis it is necessary to make useful restrictions that lead to more predictable systems. To achieve this, the fundamental misconceptions about current WCET analysis approaches have to be analyzed. Based on these elaborations one can identify WCET analysis approaches that are more promising for practical usability.

The rest of the paper presents current misconceptions about WCET analysis. Section 2 discusses the main misconceptions about WCET analysis. Section 3 concludes this document.

*This work has been supported by the IST research project "High-Confidence Architecture for Distributed Control Applications (NEXT TTA)" under contract IST-2001-32111.

2 Discussion of Misconceptions about WCET Analysis

For a better understanding of the existing problems, a short overview about some basic properties of static WCET analysis and runtime measurements is given.

Static WCET analysis methods usually provide safe upper bounds for the WCET. To guarantee safeness, any piece of information that is not available for the analysis has to be modelled in a conservative way. Therefore, overestimation becomes the price for the safeness of the calculated upper WCET bound. In a static WCET analysis framework, calculating the concrete execution time for fractions of the code is called *exec-time modeling*. The implementation of exec-time modeling for modern processors with features like caches or pipelines becomes quite complex. The advantage of measurement-based WCET analysis techniques is that they do not require exec-time modeling. However, the drawback of using simple measurements is that measured execution times may vary depending on the concrete values of the input data.

Misconception I: "Safe Upper WCET Bounds Need to be Known for Every Real-Time Task"

It is often argued that strict static WCET analysis has to be used to analyze the timing of any real-time system. In reality, only the design of *hard* real-time systems (HRTS) really requires the provision of safe upper WCET bounds. HRTS are only a small category of real-time systems, having usually simple software structures.

The timeliness of *soft* real-time systems (SRTS) is only a question of quality of service, as sporadic deadline misses usually do not cause serious consequences. Therefore, SRTS are built to handle only typical system load scenarios. Since the accurate timing analysis of SRTS is less stringend than for HRTS, SRTS tend to have relatively complex software structures, e.g, MPEG-based video streaming. As a consequence, for modern processors with pipelines or caches, the application of strict static WCET analysis techniques to SRTS may cause too much pessimism. Furthermore, for SRTS that use modern processors, the precision obtained by runtime measurements tends to be more precise than strict static WCET analysis techniques. And the common drawback of measurement-based analysis methods – the potential underestimation of the WCET – is not necessarily so critical for SRTS.

Misconception II: "Measurement is not an Adequate Technique for WCET Analysis"

It is often argued that runtime measurements are not an adequate technique to obtain the WCET for HRTS as they typically provide only a lower bound of the WCET. To discuss properties of runtime measurements in further detail, it is necessary to distinguish between pure runtime measurements and hybrid WCET analysis methods.

Performing pure runtime measurements with exhaustive search over the value space of the input data is in general not feasible and as a consequence, only a lower bound for the WCET can be found. But things become much more easier on programs with relatively few input-data dependent control flow.

For target architectures where instruction timing only depends on the previous program control flow and the values of the operands, it is sufficient to perform the measurements for all combinations of the input data that influence the control flow. For example, the instruction timing of an architecture having a pipeline but no instruction delays due to hierarchic memory depends only on the previous control-flow dependent and the parameters. Target architectures with features like caches have an instruction timing that depends on the previous control flow and instruction parameters. For these architectures it is required to perform the measurements for combinations of all input data.

Also hybrid WCET analysis methods based on static analysis and runtime measurements can be used to calculate safe upper bounds for the WCET. Hybrid methods are relatively new and they are typically designed to exploit available control-flow information.

As a consequence, runtime measurements are an adequate WCET analysis method for hybrid analysis methods or for the analysis of systems with strongly constrained input-data dependent control flow.

Misconception III: "WCET Analysis Is Simple To Use!"

The optimal WCET analysis tool would not require any special knowledge from the user about the analyzed code. Due to undecidability, the realization of such a tool is not possible. However, it is typically discussed whether static WCET analysis or a measurement-based approach can be provide more transparency to the user. In fact, both methods have their inherent limitations and, in general, will require additional knowledge about the runtime behavior of the code.

From the theoretic point of view, static WCET analysis has various advantages over measurement-based approaches. Also, the calculated WCET bound is automatically a safe upper bound if only partial knowl-

edge about the possible control flow of a code is available. In practice, static WCET analysis has numerous limitations: One of them is due to *flow facts*, that describe the possible control flow paths (CFP) of a program. In general, flow facts cannot be fully automatically extracted from the program code by semantic analysis. Code inspection and manual code annotation by the programmer is required to specify the possible CFP more precisely. The flow facts together with the program code are used by the static WCET analysis tool to calculate a WCET bound. In practice, concrete flow facts specifications are not powerful enough to express the possible CFP of generic programs in a precise way. For relative simple processors without caches or pipelines it is sufficient to specify flow facts as restrictions over the execution frequencies of program blocks. For modern processors this information is not sufficient to calculate precise WCET bounds. As the footprints in pipelines and caches depend on the concrete execution *order* of instructions, flow facts need to have a semantics much closer to the program execution. The calculation of flow facts about the execution order of instructions would be even more complex than flow facts about the execution frequency, which might lead to additional pessimism.

Measurement-based approaches do not directly rely on flow facts as the knowledge about the control flow is not required to perform a runtime measurement. However, to obtain WCET bounds for hard real-time systems requires to test all relevant execution scenarios of the code. A concrete execution scenario for a code is determined by the initial state of the target hardware and the values for the input parameter. The key question is how to find the relevant values for the input data so that it is ensured that all relevant execution scenarios are tested. An exhaustive search over the whole value space of the input data is in general not feasible. Missing a relevant value instantiation of the input data can result into an underestimation of the WCET. Therefore, measurement-based approaches have an analogous limitation to static WCET analysis methods. As static WCET analysis methods require flow facts to describe the control flow of a given code, measurement-based approaches require the provision of precise information about execution scenarios to be tested.

For program code with limited complexity, static WCET analysis methods as well as measurement-based approaches can be designed to be simple to use. Due to undecidability, the analysis of generic code structures will, however, always require the provision of additional information about the execution behavior of the code.

Misconception IV: "Static WCET Analysis Provides Accurate Results"

An important factor for the accuracy of a WCET analysis tool is the construction of an accurate exec-time model. To calculate a precise WCET bound, the WCET analysis tool has to use the underlying exec-time model to consider all possible execution combinations - a task that becomes quite expensive and complex for modern processors. Static WCET analysis methods therefore use safe approximations, that inherently cause pessimism. For example, when modeling the behavior of a cache, it can happen due to approximations that the number of cache misses is highly overestimated. In practice, this means that the "effective cache size" is only a fraction of the real cache size. There exist numerous work about modeling of different hardware features by static WCET analysis tools. However, one has to be aware that the support of a certain hardware feature by a static WCET analysis tool in general cannot be done without inducing overestimations. Though a WCET analysis tool promises the support of a certain hardware feature, the user may not be satisfied by the provided accuracy.

Misconception V: "WCET Analysis Has to Consider Task Preemptions"

It is often argued that intra-task WCET analysis introduces too much pessimism. However, widening the WCET analysis to the inter-task level creates additional complexity in the analysis as the number of variable analysis parameters increases. The alternative approach is to construct more predictable systems that support a hierarchical timing analysis. Such an approach allows for the calculation of accurate results. There exist already research in the area of separating the execution context of tasks to make the execution time of a single task more predictable [3, 1, 4]. Further research in hardware and software paradigms is required to develop practicable solutions for constructing more predictable systems.

Misconception VI: "Too Much Reserved Time due to Pessimism in WCET Analysis can be Recycled as Gain Time by Soft Real-Time Tasks"

Due to undecidability, the calculation of safe upper WCET bounds often induces pessimism. It has been argued in literature that pessimism is not such a key problem for WCET analysis methods, since a waste of resources due to pessimism could be recycled as gain time by soft real-time tasks.

It is in general questionable whether it is a good strategy to mix hard and soft real-time computation patterns. An argument from the community of fault-tolerant computing is that it is a better strategy to split systems into smaller, redundant distributed parts to increase fault tolerance.

Another point is that such a combination increases the complexity of the system, as non-real-time tasks influence the predictability of the hard real-time tasks. The existence of non-real-time tasks also hampers the process of software certification as it becomes more difficult to argue about the predictability of a system that includes soft real-time tasks.

The lucid separation of hard real-time and soft real-time tasks may be also a system requirement. As hard real-time tasks typically have a quite simple software structure, their calculated WCETs have few possibilities for allocation of gain time. Therefore, the time budget for soft real-time tasks in most cases has to be allocated statically.

It is also a basic question whether the overestimation of WCET analysis tools is a real problem as computer systems used a safety-critical environment often have quite simple code. The overestimation of the WCET for the simple software in safety-critical systems tend to be significantly lower than that for generic software with more complicated code structures.

Misconception VII: "WCET Analysis Tools Have to Support Generic Programs"

It is often claimed that a WCET analysis tool has to support generic software structures. For example, some WCET analysis projects address the full support of a programming language like ANSI C.

A more promising strategy is to develop WCET analysis methods for specific application domains. As already mentioned in misconception I, hard real-time systems typically have a simple program control flow. Another point is that code generated automatically by a code generator often has a restricted shape that simplifies WCET analysis. The simplified structure code of programs targeting these application-specific domains makes WCET analysis easier. In contrast to this, WCET analysis tools are typically designed for generic programs, where their analysis limitations become apparent.

There are various ways for a WCET analysis tool to exploit simplifications from the concrete application context. As a potential benefit, the precision of the WCET analysis tool will improve and also the implementation complexity for the analysis tool will be reduced.

3 Summary and Conclusion

This paper discussed misconceptions in current WCET analysis approaches. An important result is that one has to analyze which activities of a real-time system are really time-critical. Only for these hard real-time activities a safe WCET analsis is required. For the soft real-time activities a probabilistic timing analysis is sufficient to guarantee aspects like quality of service.

To enable safe and precise WCET analysis for hard real-time tasks, mechanisms are required to ensure the predictability of them. A promising technique to achieve this is "WCET-oriented programming", i.e., reducing the number of input-dependent control flow paths in the code [6, 7, 8]. Development tools like an intelligent editor can assist the software developer in using this technique [2].

References

[1] B. Cogswell and Z. Segall. Macs: A predictable architecture for real time systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 296–305, 1991.

[2] J. Fauster, R. Kirner, and P. Puschner. Intelligent editor for writing wcet-oriented programs. Research Report 30/2003, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2003. submitted to EMSOFT'03.

[3] D. B. Kirk and J. K. Strosnider. Smart (strategic memory allocation for real-time) cache design using the mips r3000. pages 322–330, Lake Buena Vista, Florida, USA, Dec. 1990.

[4] M. Lee, S. L. Min, C. Y. Park, Y. H. Bae, H. Shin, and C. S. Kim. A Dual-mode Instruction Prefetch Scheme for Improved Worst Case and Average Case Program Execution Times. pages 98–105, 1993.

[5] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 1st edition, 2000. ISBN: 0130996513.

[6] P. Puschner. Is worst-case execution-time analysis a non-problem? – towards new software and hardware architectures. In *Proc. 2nd Euromicro International Workshop on WCET Analysis*, Technical Report, York YO10 5DD, United Kingdom, Jun. 2002. Department of Computer Science, University of York.

[7] P. Puschner. Transforming execution-time boundable code into temporally predictable code. In B. Kleinjohann, K. K. Kim, L. Kleinjohann, and A. Rettberg, editors, *Design and Analysis of Distributed Embedded Systems*, pages 163–172. Kluwer Academic Publishers, 2002. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).

[8] P. Puschner. Algorithms for Dependable Hard Real-Time Systems. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2003.

Compiler Support for WCET Analysis: a Wish List

G. Bernat
Real-Time Systems Research Group
University of York, England, UK
bernat@cs.york.ac.uk

N. Holsti
Space Systems Finland Ltd
Espoo, Finland
niklas.holsti@iki.fi

Abstract

Static timing analysis of a computer program needs both high-level information from the source code of the program, and low-level information from the compiled object code. Compilers and linkers could support such analysis by providing more and better information about the structure and behaviour of the source and object code and about the relationship between source and object code. Moreover, some parts of timing analysis would be eased by more control over the code generation process. Finally, timing analysis often depends on annotations or assertions embedded in the source code, or referring to the source code. Compilers and linkers could help us make use of annotations by translating the annotations from the source domain to the object domain.

To make these needs known to compiler developers and vendors, we propose the collection of a "wish list" of requirements from academic and industrial groups working in timing analysis. We discuss how such a list should be selected, presented and motivated, with emphasis on finding other users with similar needs, for example other kinds of static analysis, debugging or program verification.

1. Introduction

Static analysis of a program's timing behaviour, such as WCET analysis, needs both high-level information from the source code of the program, and low-level information from the compiled and perhaps even linked machine code. Some parts of WCET analysis are easier on the source-code level, for example path analysis and pointer analysis, but obviously the actual machine code must be analyzed to find the actual execution time. Conversely, static analysis of the machine code alone can be quite difficult. For example, if the compiler has generated branch instructions with dynamically computed target addresses, it is hard to build the machine-level control-flow graph although the source-level control-flow may be quite static and simple, such as a switch/case statement.

The information on the high and low levels must also be correlated, for example to find the correspondence between source-code control flow and machine-code branches, or between source-code variables and machine registers or memory locations.

Since the compiler and linker generate the machine code from the source code, they are best placed to create the correlation between the two levels, and to some extent already do so by emitting debugging information such as symbol tables and memory maps. Present-day compilers and linkers also perform quite a lot of program analysis themselves, but usually do not make the results available for other tools such as static code analysers and WCET analyzers. WCET analysis tools have to reconstruct this information from source code and object code alone; this is a challenging task and sometimes difficult to perform. WCET researchers and tool developers would often like more and better information and support from the compiler and linker. At WCET 2002 it was proposed the creation of a "wish list" of better compiler support for WCET analysis.

The purpose of this paper is to collect the requirements from the WCET analysis community from individual efforts with the long term objective of influencing tool manufacturers (specially compiler vendors) to generate intermediate data formats useful for timing analysis.

2. The role of compilers in WCET analysis

The kind of WCET tools we are considering are those that analyze machine code (or code in a low-level intermediate language) and perhaps also source code, but are not integrated with a compiler. For correlating the source-code with machine code the tools must thus depend on the additional information generated by the target compiler and linker, whch is usually just the debugging information. This information is often insufficient and creates unnecessary problems for the WCET analysis.

Some WCET researchers have modified existing compilers or built their own compilers and even new programming languages with better support for WCET analysis. They have studied how a compiler can generate useful information and also how this information, derived from the source-code, can be maintained and translated through the compilation and linking process to apply to the machine code. This is a valid research area but we feel that it is unlikely to yield useful production compilers with WCET support. We believe that most WCET R&D groups, and certainly most software developers who are potential users of WCET tools, would prefer to use the common target languages, compilers and linkers, mainly for customer support and certification issues. This reduces the threshold for users to adopt WCET tools but it requires us to persuade the compiler and linker suppliers to change their tools to support WCET analysis better. The experience from the development of special languages and special compilers with support for WCET analysis will be useful here.

The ultimate goal is the definition of a standard format of code transformations and code properties that is produced by compiler tools. The standardisation would allow the seamless integration of this data across tool chains.

3. The clients of the compiler and linker

The proposed wish-list for improved compiler and linker support must of course serve the needs of WCET analysis, but to make the list persuasive, we should use the fact that there are many other users (clients) of the outputs from the compiler and linker. The target processor that executes the machine code is only of these users, which include at least:

- The linker (as a client of the compiler and of a previous run of the linker),
- The loader (as a client of the compiler and linker),
- The compiler itself, in several possible ways: separate compilation of module specifications and header files; interfaces between compiler passes; use of run-time monitoring results for optimization, etc.,
- The debugger (tool and human) and disassembler,
- Machine-code instrumentation, translation and verification tools,
- Manual machine-code review and tools to support such review,
- The target program itself, for reflection or intro-spection purposes such as exception handling, stack unwinding, garbage collection, run-time verification, etc.,
- Other code analysis tools, including profilers, memory usage analysis tools, static code analysers (for example like Spark), etc
- And, last but in our view not least, WCET analysis tools.

Any wish for added WCET support is more likely to be implemented if it benefits other clients, too. Such *collateral benefits* should be actively sought and clearly presented.

4. General guidelines

We invited the WCET03 workshop to discuss and collect a list of requirements from the community. We have initially classified the set of requirements according to the following categories:

- Properties of source code level: Including tree structure of the code, implicit type conversions, results of pointer analysis, dead code analysis, variable sizes of arrays, value-range analysis, loop induction variable analysis, annotations, type analysis and range analysis for automatic deduction of ranges of loop bounds, multiple language support, Virtual method invocations in OO languages, etc..
- Properties of machine code. For example, the list of the possible targets of a dynamic branch instruction that corresponds to a switch/case statement.
- Mapping between the source-code, intermediate code and machine-code levels. For example, the location in the machine code that corresponds to a WCET annotation in the source. Automatic extraction of code annotations.
- Map of code transformations, mostly code optimi-sations so that one-to-one mappings between source code and object code can be derived.
- New compiler controls or options to make the machine code easier to analyze. For example, special restrictions on optimization such as creating irreducible loops.
- New functionality for the compiler tool chain: automatic instrumentation of programs for coverage analysis and for timing instrumentation.
- Standard ways to annotate real-time and WCET aspects in the source code, with translation to the machine-code level. For example, annotations for loop bounds and path constraints.

For each wish, the list should explain clearly what is desired (taking into account that the audience are not WCET analysis experts) and why it will be useful to WCET analysis *and other tools*. The list should suggest

how the wish could be implemented in a compiler or linker, with reference to any existing implementation in a research context. For new information to be provided by the compiler or linker, the list should suggest the format and medium, for example how the information could be encoded in ELF or DWARF or in a separate file. All information about the machine code should also be traced back to the source code.

5. Discussion

At the WCET03 workshop, we started the discussion on these requirements recording current developments by individuals, as well as desired functionality. The issues discussed include:

- What information is required from compilation tools
- Format of such information
- New compiler functionality for WCET analysis.

6. Conclusions

To enable WCET analysis detailed information already available in compiler tool chains is required. We propose to collect an agreed set of requirements from the WCET Community on the information needed to perform WCET analysis with the aim of producing a white paper to influence compiler manufacturers and vendors to make such information available.

Table 1, below, lists the items we have collected so far. ordered by requirement category. This list is of course not yet complete, and also the columns "Examples" and "Supported analyses" are incomplete. The authors would be most grateful for comments on this list and suggested additions to this list, dealing with the issues raised in this paper. Other future work includes finding collateral benefits, prioritizing the requirements, and defining the data formats and other interfaces for implementing the requirements.

Table 1. Compiler and linker support for timing analysis

Requirement category	*Property, mapping or control*	*Examples*	*Supported analyses*
Properties on source-code level	Tree structure of the code	Intra-procedural control structures: sequence, conditional, switch/case, loop, exception. Inter-procedural control structures: call, return (normal/alternate), exception.	Control flow.
	Implicit type conversions	Address to or from integer. Integer to long.	Values and arithmetic. Loop bounds.
	Types and value ranges or value sets of variables and expressions	Range of loop counters. Range of actual parameters. Pointer analysis results ("points-to" properties).	Feasible paths, loop bounds. Cache timing (memory access patterns, dynamic addresses).
	Array sizes	Dynamically created (heap) arrays. Local (stack) arrays with dynamic size. Formal array parameters to subprograms where actual parameter determines size.	Loop bounds. Cache timing. Stack usage bounds.
	Loop induction variables	Loop counters. Index expressions that depend on iteration count.	Loop bounds. Cache timing.
	Content and location of source-code annotations	Loop-bound annotations. Memory timing annotations.	Potentially all.

Requirement category	*Property, mapping or control*	*Examples*	*Supported analyses*
	Feasible paths and loop bounds (as deduced by compiler).	Dead code detected e.g. by constant propagation. Loop bounds for compiler-generated loops (e.g. copying loops).	Control flow, feasible paths, loop bounds.
Mapping source code to object code	Source lines to code instructions	As currently implemented in "debug" information.	Support other mappings, e.g. mapping of path constraints or annotations.
	Source tree to code instructions and branches	Altered order of then/else in conditional statement. Altered order of cases in case/switch statement. Changes in the placement of a loop termination test (test at start/middle/end of loop). Other loop transformations, unrolling etc..	Support other mappings.
	Source annotations to code	Map a loop-bound annotation to the loop (head) in the object code. Adapt a loop-bound annotation to the transformations applied to the loop.	Loop bounds.
Properties on object code level	Possible targets of dynamic branches.	Switch/case structures implemented with jump tables or address tables	Intra-procedural control flow.
	Possible callees for dynamic calls.	Late-bound method calls in object-oriented programming. Interrupt handlers and trap handlers, called via vector tables.	Inter-procedural control-flow.
	Code that violates target-processor standards and needs special analysis	Library routines or compiler-generated routines that have non-standard calling sequences.	Control-flow and others.
	How target-processor standards are used, when there are alternatives	For each subprogram or call: which of the alternative calling sequences and parameter-passing methods is used.	Any aspect of analysis influenced by target-processor standards, for example inter-procedural control flow and data flow.
	Logical role of multi-purpose instructions	Whether an instruction that loads the Program Counter represents a jump, call or return.	Control flow and others.
	Operand type information for polymorphic instructions.	Signed versus unsigned interpretation of integer arithmetic and comparison instructions and of immediate (literal) operands.	Values and arithmetic. Feasible paths, loop bounds, pointers.
	Logical effect of code subsequences	On processors with small word size, e.g. 8 bits, the fact that a certain sequence of 8-bit computations has the effect of adding two 16-bit quantities.	Values and arithmetic. Feasible paths, loop bounds, pointers.

Requirement category	*Property, mapping or control*	*Examples*	*Supported analyses*
	Code that relies on over-flow or other exceptions for nominal operation	A loop from 0 to 255 using an 8-bit counter might rely on overflow from 255 to 0 in the last iteration.	Values and arithmetic. Loop bounds.
	Memory locations that are initialized dynami-cally at program start but are constant during run.	Trap vector tables. Constants copied from PROM to RAM.	Values and arithmetic. Control-flow analysis when the values enter dynamic branch or call computations. Feasible paths, loop bounds, pointers.
	Memory locations with special semantics	Volatile variables (consecutive reads may give different values). Control registers with different read/write roles (a read does not return the last written value).	Values and arithmetic. Feasible paths, loop bounds.
Control over object code generation	Control the generated loop structures	Generate only reducible loops. Prevent loop unrolling or other specific loop transformations.	Loop analysis, loop bounds. Support source-to-object mappings.
	Control the generated inter-procedural transfers	Prevent or enforce inlining. Enforce target-standard procedure calling protocols.	Inter-procedural control-flow. Support source-to-object mappings.

Impact of automatic gain time identification on tree-based static WCET analysis

Mathieu Avila, Maxime Glaizot, Isabelle Puaut
IRISA, Campus de Beaulieu, 35042 Rennes Cédex, FRANCE
e-mail: puaut@irisa.fr

Abstract

WCET estimates obtained using static analysis methods are getting increasingly pessimistic as the complexity of hardware and software increases. The difference between the WCET of one task (estimated off-line) and its actual execution time (only known on-line) is known as gain time. *Identifying gain time as soon as possible is important because it increases the number of tasks that can be accepted dynamically. While some research has already been undertaken for the identification of gain time, few work has considered the impact of gain time identification and reclaiming on static WCET analysis methods. This is the objective of this paper, in which we introduce three classes of methods for gain time identification, and discuss their impact on tree-based static WCET analysis methods.*

1 Motivations for automatic gain time identification

Most scheduling algorithms for hard real-time tasks assume that the WCET estimation of each task is known. A number of dynamic scheduling algorithms have also been proposed to dynamically accept soft real-time tasks when spare capacity is left by hard-real-time tasks. Spare capacity is either *extra time* (time known to be left by the hard real-time tasks during the design phase) or *gain time* (spare time appearing at run-time when hard real-time tasks execute in less than their WCET).

Static WCET analysis techniques return an upper bound on the execution time of a task on a given hardware, based on its source code. Having an upper bound on all possible execution times (safety) is of prime importance in hard real-time systems to have confidence in the schedulability analysis methods. But despite the important progress made in static analysis methods, safety comes at the cost of pessimistic WCET estimations. Two sources of pessimism can be identified: (i) analysis of the execution paths, or *high level analysis* (when it is not known statically which path will be executed, the longest path is selected), (ii) *low-level analysis* (when the execution time of an instruction is not known *a priori* due to the use of complex processors with performance enhancing features such as caching or branch prediction, the most pessimistic execution time is selected). As the complexity of software and hardware increases, the degree of pessimism of WCET estimates also increases. In such situations, identifying and reclaiming gain time is getting increasingly important. In this paper we concentrate on the estimation of gain time, and not on its reclaiming.

In our opinion, the methods for gain time identification should have the following properties:

- *Early detection*. The presence of gain time should be detected before the tasks finish their execution. The sooner the gain time is detected, the earlier new tasks can be dynamically accepted.
- *Predictable cost*. Early identifying gain time requires to monitor the progression of the tasks, which has a cost in terms of execution time. This cost has to be predictable and the designer should have means to control the cost of gain time identification.
- *High efficiency.* All gain time should be detected, should it come from the low-level or the high-level sources of pessimism of static WCET analysis.
- *Transparency.* No support (or very low support) from the designer should be required.
- *Predictable and low memory requirements.*

Several techniques have been proposed for gain time identification. [3] consists in measuring the execution time of tasks between so-called *gain points* using specific hardware, the gain points being placed by the programmer. A software evolution of [3] is presented in [2]. In this proposal, the gain points are automatically determined, but the target language is very simple. Both [3] and [2] identify all gain times because they use measurements to monitor the tasks progress. Other methods, that only identify gain time coming from the pessimistic identification of worst-case execution paths, have been proposed in [1] and [4] (the latter tackles object-oriented hard real-time programs). The principle of these two methods is to know statically the WCET of the different paths in the program. Then, each time a path decision is taken, the gain time can be estimated, but

because there are no measures of the actual execution times, the *low-level* analysis pessimism will not be identified.

In the following, we briefly propose three classes of techniques aiming at reaching all the above-identified desirable properties and study the support that static WCET analysis should provide in order for these techniques to be implemented.

2 Three methods of gain time identification and their impact on WCET analysis

All three methods use some general principles. Instrumentation code is inserted in the tasks at specific points called gain points (GP) in which the time actually consumed by the task is measured. Measurements are used to identify *all* sources of pessimism of WCET analysis. All three methods identify gain time on-line by subtracting the measured execution time of *segments* of the task code from the WCET of the same segments. The methods differ by the rules governing GP placement and the definition of a segment. Our discussion hereafter concentrates on the impact of gain time identification on tree-based WCET analysis tools.

A simple example is used hereafter to illustrate the pros and cons of each method for GP placement, as well as their impact on WCET analysis. The source code of the example is presented in figure 1. Two important things can be noticed about this code: (i) the maximum number of iterations of the loop (three, as indicated in the annotation [3] in the source code) may be overestimated (the loop may execute once or twice only); (ii) the most time-consuming execution path within the loop is the "else" path, although the "then" path can be actually executed too.

```
int i;
int j;
for (i=0;i<N;i++) { [3]
  /* Known to iterate at most 3 times */
  if (j==1) {
    j=3;
  }
  else {
    j=j+i;
  }
}
```

Figure 1. Source code sample

We can extract two data structures from this source code: the program control flow graph (left part of figure 2) and its syntax tree (right part of figure 2). The latter data structure is used in so-called *tree-based* WCET analysis tools to compute the WCET of a piece of code through a bottom-up traversal of its syntax tree.

Figure 3 depicts for this sample program the differences that may exist between the off-line and a given on-line timeline. It shows that the actual execution time is lower than the WCET. More precisely, it identifies the different sources of

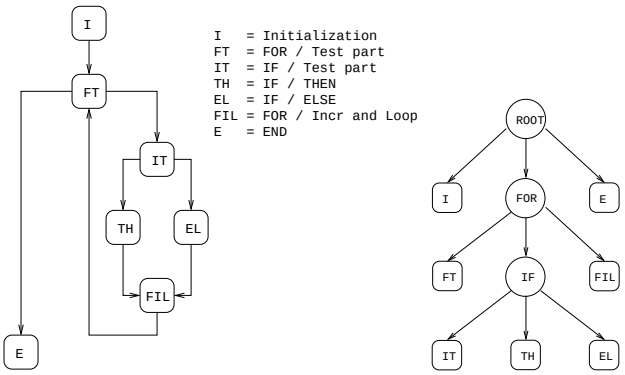

Figure 2. Control-flow graph (left) and syntax tree (right)

gain time: (i) gain time coming from the pessimism of low-level analysis (in the figure, the actual execution time of basic block I is lower than its worst-case counterpart identified off-line); (ii) gain time due to the pessimistic evaluation of the worst-case execution path (e.g. the loop iterates two times instead of three at worst; within the loop the "else" branch – basic block EL – may be executed whereas it is not the longest branch).

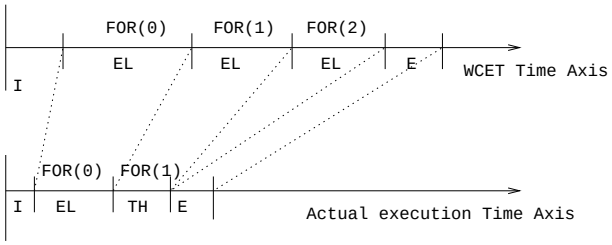

Figure 3. Off-line and on-line timelines

2.1 Segment-based method

This method puts almost no constraint on the locations of GPs. It reasons on *segments* defined as intervals between successive GPs in the task control flow. This method is flexible since the length and location of segments can be tailored so as to find an appropriate tradeoff between cost and earliness of gain time identification. However, the off-line overheads of the method are high. Indeed, the static WCET analyzer has to generate partial WCETs for any pair of points that can be consecutive in the task control flow, leading to a potentially high number of partial WCETs to be computed.

For instance, if three GPs are placed in our sample code as shown in figure 4, six partial WCETs must be computed to cover all possible paths between successive GPs in the control flow graph. Furthermore, the integration of the computation of WCETs of segments is not natural in tree-based WCET analyzers because of the mismatch between the location of GPs and the data structures used by such analyzers.

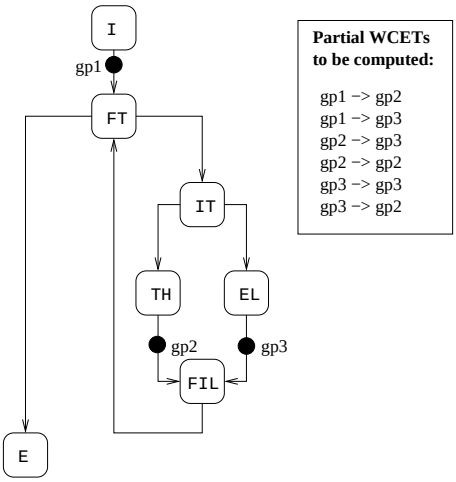

Figure 4. Example of GP locations in the segment-based GP placement method

The large number of partial WCETs, in addition to increasing the complexity of the computation of partial WCETs, also increases the complexity of the on-line part of gain time identification, since all partial WCETs have to be accessible on-line. Another problem of this method is that, when a GP is placed in a loop body, it only allows to identify gain time within the loop but is unable to identify the gain time arising when the loop iterates less than expected.

2.2 Structural method

This method restricts the locations of GPs to the control structures in the syntax tree like loops, conditional constructs (segments do not cross control structures boundaries as in the first method). A control structure in which it is interesting to reclaim gain time is enclosed by a pair of GPs (immediately before and after the control structure). Partial WCETs are then needed for all "instrumented" control structures. Figure 5 shows on our example all possible pairs of GPs (e.g. GPs ga_b and ga_e to define a segment corresponding to the loop).

One can note that the number of partial WCETs to be computed off-line tend to be less numerous than with the

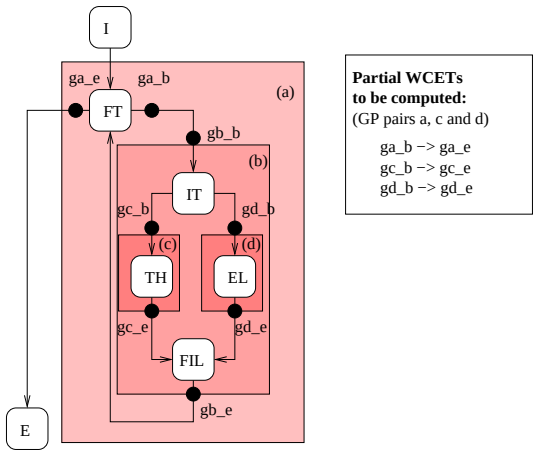

Figure 5. Possible GP locations in the structural GP placement method

first method. On our example, if the TH, EL and FOR control structures are instrumented (pairs of GPs a, c and d, which is roughly equivalent to placing GPs $gp1$, $gp2$ and $gp3$ in the first method), only three partial WCETs have to be computed, compared to six in the first method. The off-line computation of the partial WCET of segments is rather straightforward, as tree-based tools actually compute a WCET for each level of the syntax-tree. However, the method is less flexible than the first one because of the imposed restrictions on the locations of GPs.

2.3 Path based method

This last method is an hybrid one which is halfway between the first two ones. As in the segment-based method, no restriction is put on the locations of GPs, thus ensuring the flexibility of the method (ability to find an appropriate trade-off between cost and earliness of gain time identification). But instead of defining segments as intervals between successive GPs in the task control flow, it defines segments as intervals between the *beginning* of execution of the task and the different GPs (see figure 6).

Since a GP can be encountered several times if it is enclosed in a loop (for instance $gp2$ in the figure), several partial WCETs have to be generated depending on the loop counters. Instead of generating all possible WCETs of segments, we propose to represent the WCET as parametric values depending on the loop counters (functions with the loops counters as parameters). These functions should be simple enough to be evaluated on-line, but expressive enough to represent the WCET time elapsed since the beginning of the program.

On the on-line part, the task must keep track of the values

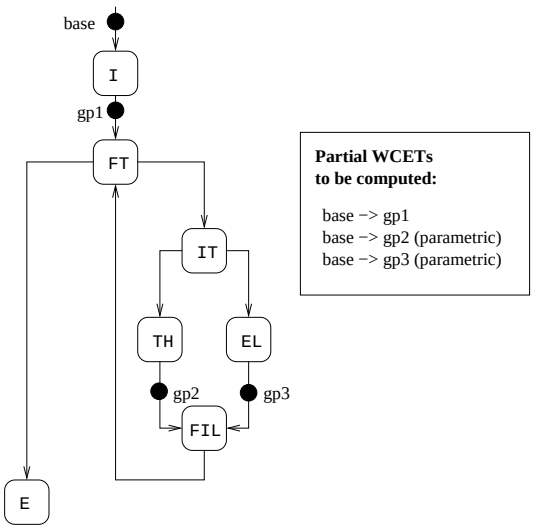

Figure 6. Example of GP locations in the path-based GP placement method

needed by the evaluation functions (essentially loop counters). Each time a GP is encountered, these values, the evaluation function and the elapsed time are stored. The actual computation of gain time (especially the calls to the functions that evaluate the segments' WCETs) can be deferred until gain time is requested by the dynamic scheduler. The impact of this method on tree-based WCET analysis is bigger when GPs are placed inside loops than outside, because then parametric WCET representations must be generated.

2.4 Degree of pessimism of partial WCET estimates

A common issue to be addressed is the degree of safety of WCET estimations of segments (partial WCETs) in order for the estimation of gain time not to be overly optimistic.

This issue is illustrated in figure 7, which depicts gain time identification during the execution of a task made of a sequence of two blocks A and B. Due to the consideration of pipelining effects, the sum of the partial WCETs of A and B (6 time units each in the figure) may exceed the WCET of the sequence A;B (10 time units). Assume that the actual execution of both A and B is 4 time units (the gain time is 2 time units). If the partial WCETs of 6 are used for gain time computation, the gain time is overestimated (4 time units instead of 2), which can cause new tasks to be accepted dynamically whereas too few spare time is available.

More generally, the requirement is that the partial WCETs be *consistent* with the global WCET of the task. Thereby we mean that the value of the WCET of a segment of code is lower or equal to its equivalent in the WCET of

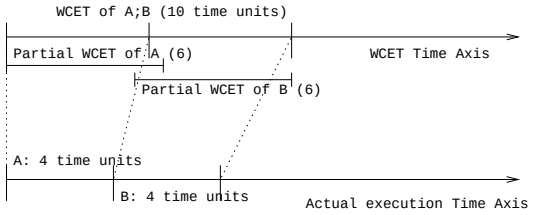

Figure 7. Pessimism of partial WCET estimates

the whole task. Such consistency problems could occur because of low-level analysis in architectures with pipelines (as shown in the example) and in symbolic WCET estimation methods. Possible directions to address this issue would be to provide "optimistic" partial WCETs or to change the WCET computation method for the whole task so that partial WCETs are consistent with the global one.

3 Concluding remarks

Early identification of gain time requires to obtain WCETs of *segments* of the task code instead of considering the code as a whole. In this paper, we have proposed three classes of methods for identifying gain time, differing by their definition of segments. We have further examined their impact on tree-based WCET analysis methods.

Except for the second proposed method, which can be integrated naturally in tree-based WCET analyzers, we are convinced that the need to compute partial WCETs has a non negligible impact on the structure of the WCET analysis tools. Earliness of gain time identification comes at the price of a further increase in complexity of WCET analysis techniques.

References

[1] N. C. Audsley, R. I. Davis, and A. Burns. Mechanisms for enhancing the flexibility and utility of hard real-time systems. In *IEEE Real-time systems symposium*, pages 12–21, December 1994.

[2] P. Gopinath and R. Gupta. Applying compiler techniques to scheduling in real-time systems, 1990. Philips Laboratories.

[3] D. Haban and K. Shin. Application of real-time monitoring to scheduling tasks whith random execution times. In *IEEE Transaction on software enginneering*, December 1990.

[4] E. Y.-S. Hu, A. Wellings, and G. Bernat. A novel gain time reclaiming framework integrating wcet abalysis for object-oriented real-time systems. In *Second workshop on WCET analysis*, June 2002.

Comparison of Trace Generation Methods for Measurement Based WCET Analysis *

Stefan M. Petters
Department of Computer Science
University of York
United Kingdom
Stefan.Petters@cs.york.ac.uk

Abstract

Recent work on a measurement based worst case execution time estimation method uses observations of small units of the program. These observations are called execution traces and contain information of the execution path as well as the execution time of the units observed. This paper gives an overview on available options to extract the traces and highlights the advantages and disadvantages of these options.

1 Motivation

The measurement based worst case execution time (WCET) estimation method presented by Bernat et al. in [1] and [2] relies on the measurement of the execution time of small sections of code called traces as basic unit of the analysis. Within the approach, the observed traces are translated into execution time frequencies of the units. These execution time frequencies are interpreted as probability mass distributions and combined with a timing schema to provide a execution time profile of the worst case path through the program. The paper [1] focuses on the translation of traces to profiles and the combination of profiles rather than the production of the traces.

Within this paper we are addressing the different options for obtaining execution traces. Special attention will be given the factors of applicability, *overestimation* and *prolonged execution time*. Prolonged execution time indicates the impact of the method on the final executable code. For example added code will extend the execution time, if it stays in place. Opposed to this overestimation assesses the extra time within the measurements, which is not reflected in the execution time of the final executable code.

2 Discussion of Methods

Before going into the discussion of the methods a number of terms need to be defined. A measurement is made up of two observations. The points in the code where these observations are made are called *observations points* throughout this paper. The *observation interval* is the time which has passed between two observation points.

The exact meaning of a time stamp has to be considered as well. Within the paper a time stamp reflects usually a certain level of the execution pipeline. In out-of-order ex-

*The work presented in this paper is supported by the *European Union* under Grant Next TTA "IST-2001-32111".

ecution units of some processors[1], the observation point has to be either guarded by serialising instructions, or it has to be accepted, that the results of the measurements are fuzzy. Both cases add to the overall estimation of the execution time profiles. The serialising instruction disables the execution acceleration of the out-of-order execution engine, thus leading to a prolonged execution time, if the serialising instruction stays in place for the final production executable, or adds an unknown overestimation if it is somehow removed from the code. In the case of accepting more fuzzy results, a potentially considerable overestimation comes from the fact, that jitter of the time stamp at an observation point is added as well to the finished observation interval as well as the started observation interval.

2.1 Simulation

A simulator may be used to execute the program (cf. [1]). In this context we are only discussing *cycle accurate simulators* i.e. a simulator, which simulates not only the functional but also the temporal features of a processor accurately.

If such a simulator is available very accurate and usually perfectly reproducible measurements are possible. The granularity of measurements allows usually the time stamping of any individual assembler instruction. The missing requirement of instrumenting the code is a further plus for simulating the code. However, the cycle accurate simulator suffers from one of the major problems of static analysis. Additionally the cycle accuracy of a simulator usually only applies to the CPU itself and does not imply accurate timing of peripheral hardware e.g. anything from SDRAM to PCI bus hardware. As the simulation takes enlarges the execution depending on the complexity of the modelled processor by up to two orders of magnitude on similar hardware, the problem of time needed for the trace generation becomes an issue. This can be somewhat relieved by using a cluster of computers to do the simulations.

[1] These effects obviously only occur, if the time stamp is taken within the out-of-order execution core of the processor. The fetch and commit stages are always in-order execution units.

2.2 Light Weight Software Monitoring

Software monitoring relies on instrumentation code placed in the software to be investigated. The code inserted may record various data of the system. In our case we are interested in event triggered software monitoring in which the instrumentation code is placed at the desired observation points in the application code and is executed whenever the observation point is reached, opposed to periodic software monitoring (cf. [3]), in which the application code is interrupted in merely period distances.

While the term software encoding encompasses all kind of data collection, we are only interested in identification of the code executed and the time stamp corresponding the execution of this code. The prefix *light weight* indicates, that the code inserted tries to minimise its impact on the execution time. As opposed to *heave weight* software monitoring, which tries to make a safe estimate by establishing the worst case state of a processor at the start of each observation interval. The time passed is usually taken from an internal cycle counter. The POSIX tracing standard as used by Terrasa et al. in [4] shows a implementation of such a light weight software monitor.

On the positive side it can be noted that the instrumentation code will be left in place after the measurements are completed. The instrumentation code may be quite simple, which makes a comparable quick port of the method to another architecture possible. A major drawback of this method is the additional variability introduced by this method especially on high performance processors with caches. Either the memory area is mapped as non-cachable, which makes the access time in storing the sampled data very long or the cache access patterns add to the temporal variability of the code. The amount of memory necessary to store the trace data can be considerable, which adds to the effort of bounding the impact on code

variability.

The code added may be quite computational expensive. The POSIX tracing standard allows for interfaces to actual collect the trace. This functionality leads to a prolongation of the execution time. This prolongation may be up to one order of magnitude if the distance between two observation points is just one basic block in control intensive applications[2].

2.3 Heavy Weight Software Monitoring

As has been explained in the previous section heavy weight software monitoring establishes the worst possible state at the beginning of each measurement. This has been used by Petters in [5]. This implies that an observation point implies two time stamps. One for the completion of an observation interval, prior to the disruption of the working sets of caches, branch prediction etc. and a second one starting a new observation interval after the disruption.

The code has to be replaced after the measurements have been completed. In order to limit the code to be replaced the main code of the instrumentation should be implemented as a function which is called with an identifier from all observation points. In this case only the calls to the measurement routine have to be masked or the measurement routine is replaced with an return. Depending on the architecture the first renders the final operational executable usually faster than the second solution.

As an advantage the disruption of execution units may be preceded by a write out of the obtained measurement data to disc. This limits the amount of memory necessary for the storage of the trace data. The overestimation by this method may be seriously. Depending on the size two orders of magnitude are possible, if one tries to trace individual assembler instructions. By following coding style and tracing larger code ($\tilde{4}$-10 basic blocks) the overestimation may be reduced to a factor of two. However, this raises the issue of test coverage. In [5] this has been solved by enforcing paths within the observation interval. Adding information about the path taken within the observation interval by introducing very small additional instrumentation code may solve this problem as well.

More than with the other methods this method raises the question on the validity of the obtained results under the fact that the code in the final production executable is not identical to that one under investigation during the measurements.

2.4 Hardware Supported Software Monitoring

As with the previous methods, instrumentation code is added to the application at the observation points. This code delivers the location data to a reserved external port. The timing is taken either by the hardware device probing the port or is taken from in internal cycle counter and written in a separate access. The necessary port pins for this method are usually quite costly and in the general case this will only be applicable on micro controller and similar complex processors. High performance processors generally do not have free accessible pins and the ones accessible via buses raise questions on the time needed for transactions. The major advantage of monitoring method is the small impact of the code on the execution time.

2.5 (Software Supported) Hardware Monitoring

This option comes in two flavours. On one hand are bus monitors applied on one or more buses of the processor, on the other hand is hardware built in by processor manufactures to support debugging.

The first case has been used in the past by tracking instruction flow on the address bus of some processors. This has been inhibited by the use of instruction caches and has almost completely vanished as a tracing mechanism for the software.

[2] Control intensive applications consist of small basic blocks and are therefore vulnerable to heavy prolongation

However, in recent years debugging interfaces were equipped with additional features to allow for timing information to be extracted out of the code. Namely the Tricore OCDS and the more generally available NEXUS2 (cf. [6]). These interfaces allow the sampling of time at given points. The NEXUS2, for example, takes a time stamp at every taken branch instruction. Thus a complete trace is available for further analysis. The major challenge is to tap into these ports and extract the data. Commercial tools initially intended for debugging provide interfaces to do this. The problem consists here of getting the raw data out of the tools.

One major advantage is, that no or only minimal software instrumentation is needed[3]. A problem lies in the potential bandwidth problem. If the observation intervals are too short, the trace data may not be completely transmitted over the debugging interface. In this case, the debugging tools try a interpolation of the measurements, which defeats the purpose of taking the traces in the first place. Some debugging tools offer to hold the processor, if two observation points are too close. However, this can only be applied to the processor itself and peripherals clocked directly with the CPU clock. More remote hardware may behave differently in the temporal domain, if the execution is put on hold at an arbitrary instant.

3 Conclusion

As it is, there is no one-fits-all solution to the problem of trace generation. While especially commercially supported hardware monitoring has some appealing advantages over the other methods, the still limited availability of processors makes it necessary to look at the alternatives.

[3] In some cases one might want to enforce a observation point. This is generally possible by adding hand crafted code

References

[1] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real–time systems," in *Proceedings of the 23rd Real-Time Systems Symposium RTSS 2002*, (Austin, Texas, USA), pp. 279–288, Dec. 3–5 2002.

[2] G. Bernat, A. Colin, and S. M. Petters, "pWCET: a tool for probabilistic worst case execution time analysis of real–time systems," technical report YCS353 (2003), University of York, Department of Computer Science, York, YO10 5DD, United Kingdom, Apr. 2003.

[3] L. Svobodova, *Computer Performance Measurement and Evaluation Methods: Analysis and Applications*. No. 2 in Elsevier Computer Science Library, New York: American Elsevier Publishing Company, Inc; 1976.

[4] A. Terrasa, I. Paches, and A. Garcia-Fornes, "An evaluation of the posix trace standard implemented in rt-linux," in *Proceedings of the IEEE International Symposium on Performance Analysis and Software*, 2000.

[5] S. M. Petters, *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Institute for Real–Time Computer Systems, Technische Universität München, Munich, Germany, Sept. 2002.

[6] IEEE-ISTO, *IEEE-ISTO 5001-1999, The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface*, 1999. Available at `http://www.nexus5001.org/`

Evaluating reasons for unexpected results when measuring execution time of code.

V. Lorente A. Espinosa A. Terrasa A. Garcia
A. Crespo

1 Summary

In our research group, we are working on developing and testing the POSIX-Trace standard, that defines a common application interface for trace management. POSIX-Trace standard has been defined to cover all the requirements for testing and debugging a real-time system in an efficient and portable way. So, POSIX-Trace standard provides an efficient an portable way of monitoring. One of the advantages of this approach is that it can be used from the source code level, which facilitates its use. POSIX-Trace approach can be also used for measuring execution time of code, providing additional information besides the time.

Currently, we are developing this standard for two Real-Time operating systems : MarteOS and RT-Linux.

Testing and debugging real-time systems requires the accurate timing of the process being studied. However, most of the best current approaches are expensive and ad-hoc solutions. Nowadays, however, processors provide a very accurate way (Time Stamp Counter) of knowing how long a piece of code takes to execute without the need for external hardware. This feature can be exploited at source code level to measure the execution time of code by counting the clock cycles that a piece of code takes to execute.

In one of our last accepted papers, we tested some software approaches in order to study their bounds and resolution, from the most intrusive way of measuring time (POSIX Trace) to the least intrusive way (Time Stamp Counter). Two types of tests was made:

- One to measure how long each approach takes to execute, that is, the intrusive bounds of each approach.
- The other to test the accuracy of each approach, measuring with each approach different amounts of time known in advance (from nano seconds to mili seconds) and calculating the difference between the actual time spent and the time measured by the approach.

The main conclusions we showed in this paper was :

- POSIX-Trace mechanism can be used for measuring execution time of code, achieving as good results as the CPU-Time Clock mechanism.
- Reading the Time Stamp Counter (TSC) register, it is possible to reach a very good resolution when measuring execution time of code. However, it can not be used for monitoring during system operation and its use depends on if the processor used has this feature available or not.
- It would be possible to measure the execution time of code obtaining a quite good resolution using software approaches from a high-level language. This would allow us to instrument the source code, eliminating drawbacks such as finding the right map between the source code and the assembler code in order to make accurate measures.

When we studied the results obtained in the tests made for this paper, we realised some unexpected results.

The first unexpected result was that the time spent by the `rdtsc` instruction was not constant, observing from 20 cicles to 150 cicles. Take in account that this result not only reflects the cicles needed to execute the assembler `rdtsc` instruction but there are also some other instruction involved (assignment instruction, etc).

The second unexpected result was the increase of time consumed by the method itself with load (second type of test) and without load (first type of test). The difference of time between the actual time and the measured time should be close to the results obtained in the first test (execution time of the approach), but it was not. When using cache, the results showed that the time consumed by the method itself was twice the time obtained in the first test. This might be attributed to cache effects, and it is almost true. However, when no cache was used there was still a small difference of time between these two tests. Without cache, the time consumed was around 4 percent higher with load than without load.

Now, we are working on evaluating the sources of this unexpected results, testing several features of the pc architecture that can affect the results mentioned before.

The features we are testing are :

- Out of order execution when using `rdtsc` assembler instruction. [2]
- Several methods to define cache policies. [1]
- TLB entry invalidation. [1]

2 RDTSC and the out-of-order execution.

RDTSC instruction allows users to obtain the current value of the Time Stamp Counter register (TSC). The Intel TSC is a 64-bit model specific register (MSR) that is incremented every clock cycle. On reset, the time-stamp counter is set

to zero. This counter provides the best resolution when measuring execution time of code.

Although RDTSC instruction will always give back a proper cycle counter, a user could obtain different execution time when measuring the same piece of code. The variations when using this instruction to monitor perfermance, appear because there are many things that happen inside the system, invisible to the application programmer, that can affect the cycle count returned in a specific situation.

The main problem when using this instruction to monitor performance is the out-of-order execution, that means that instructions are not necessarily performed in the order they appear in the source code. This feature is supported by Intel processors since the Pentium®Pro architecture and it can be a very big issue when using the RDTSC instruction, because it could potentially be executed before or after its location in the source code, giving a misleading cycle count.

In order to keep the RDTSC instruction from being performed out-of-order, a serializing instruction must be used. A serializing instruction will force every preceding instruction in the source code to complete before allowing the program to continue. One such instruction is the CPUID instruction, which is normally used to identify the processor on which the program is being run. For the purposes of this paper, the CPUID instruction will only be used to force the in-order execution of the RDTSC instruction.

When using the CPUID instruction, however, the programmer must also take into account the cycles it takes for the instruction to complete, and subtract this from the recorded number of cycles. A strange quirk of the CPUID instruction is that it can take longer to complete the first couple of times it is called. Thus, the best policy is to call the instruction three times, measure the elapsed time on the third call, then subtract this measurement from all future measurements.

Example of this technique:

```
static inline void measure(long long *time){
  long long t1,t2;

  __asm__ __volatile__ ("cpuid;cpuid;cpuid; rdtsc" : "=A"(t1): :"ebx","ecx");


           /********* Code to measure *********/


  __asm__ __volatile__ ("cpuid; rdtsc" : "=A"(t2): :"ebx","ecx");

  *time=t2-t1;
}
```

3 Cache

Currently, in Intel Processors there are several methods to specify cache policies :

- CD and NW bits.
- Memory type range register (MTRR).
- Page attribute table (PAT).

CD and NW bits are global flags that control overall caching behaviour. However, while the MTRR allow mapping of memory types to regions of the physical address space, the PAT allows mapping of memory types to pages within the linear address space.

MTRR and PAT use Model Specific Registers (MSR's). That means that not all the processors have these functionalities. We need be sure if the processor we are using has these functionalities before we use them. One way to know it is through the instruction CPUID.

3.1 CD and NW bits.

CD and NW flags in CR0 register control overall caching of system memory.

CD Flag: Controls caching of system memory locations. If the CD flag is **clear**, **caching is enabled** for the whole of system memory, but may be restricted for individual pages or regions of memory by other cache-control mechanisms. If the CD flag is **set**, **caching is restricted** in the processor's caches

NW Flag: Controls the write policy for system memory locations. If the NW and CD flags are clear, write-back is enabled for the whole of system memory, but may be restricted for individual pages or regions by other cache-control mechanisms.

- Disable Cache.

```
__asm__ __volatile__ ("movl %%cr0, %%eax \n\t");
__asm__ __volatile__ ("orl $0x60000000, %%eax \n\t");
__asm__ __volatile__ ("movl %%eax, %%cr0 \n\t");
```

- Enable Cache

```
__asm__ __volatile__ ("movl %%cr0, %%eax \n\t");
__asm__ __volatile__ ("orl $0x60000000, %%eax \n\t");
__asm__ __volatile__ ("movl %%eax, %%cr0 \n\t");
```

3.2 Memory Type Range Register

The memory type range registers (MTRR's) provide a mechanism for associating the memory types with physical-address ranges in system memory. These memory types and its values can be seen in the kernel *mtrr.h* file, (*/usr/include/asm/mtrr.h* in my system)

```
/*  These are the region types  */
#define MTRR_TYPE_UNCACHABLE 0
#define MTRR_TYPE_WRCOMB     1
/*#define MTRR_TYPE_          2*/
/*#define MTRR_TYPE_          3*/
#define MTRR_TYPE_WRTHROUGH  4
#define MTRR_TYPE_WRPROT     5
#define MTRR_TYPE_WRBACK     6
#define MTRR_NUM_TYPES       7
```

The MTRR mechanism allows up to 96 memory ranges to be defined in physical memory, but only 8 of these ranges are variable ranges. Rest of them are fixed ranges, being these fixed ranges below the address 0x100000 (1 Mbyte).

We can define any of these 8 variable ranges by means of a pair of registers for each of these ranges. MTRRphysBase*n* and MTRRphysMask*n*. MTRRphysBase defines the base address and memory type for the range and MTRRphysMask contains a mask that is used to determine the address range.

In the file *mtrr.c* we can see correspondences between Model Specific Registers (MSR's) and MTRR registers.

```
#define MTRRcap_MSR     0x0fe
#define MTRRdefType_MSR 0x2ff

#define MTRRphysBase_MSR(reg) (0x200 + 2 * (reg))
#define MTRRphysMask_MSR(reg) (0x200 + 2 * (reg) + 1)

#define NUM_FIXED_RANGES 88
#define MTRRfix64K_00000_MSR 0x250
#define MTRRfix16K_80000_MSR 0x258
#define MTRRfix16K_A0000_MSR 0x259
#define MTRRfix4K_C0000_MSR 0x268
#define MTRRfix4K_C8000_MSR 0x269
#define MTRRfix4K_D0000_MSR 0x26a
#define MTRRfix4K_D8000_MSR 0x26b
#define MTRRfix4K_E0000_MSR 0x26c
#define MTRRfix4K_E8000_MSR 0x26d
#define MTRRfix4K_F0000_MSR 0x26e
#define MTRRfix4K_F8000_MSR 0x26f
```

Variable range registers count (VCNT) field, in the MTRRcap register, indicates the number of variable ranges implemented on the processor. MTRRcap

register

MTRRdefType sets the default properties of the regions of physical memory that are not encompassed by MTRR's.

MTRR ranges programming can be done by means of *WDMSR* and *RDMSR* assembler instructions or by the */proc/mtrr* file. For example, this is the result of reading */proc/mtrr* in my system with 256 Mbytes.

```
reg00: base=0x00000000 (   0MB), size= 256MB: write-back, count=1
reg01: base=0xf8000000 (3968MB), size=  64MB: write-combining, count=2
reg07: base=0xfc000000 (4032MB), size=  32MB: write-combining, count=1
```

An interesting test is to delete the *reg00* line, observing an slower behaviour of our machine. This test can be done by executing `echo "disable=0" > /proc/mtrr`

3.3 Page table and Page attribute table(PAT)

PCD and **PWT** flags in control register CR3 control the global caching and write policy for the page directory. The PCD flag enables caching of the page directory when clear and prevents caching when set. The PWT flag enables write-back caching of the page directory when clear and write-through caching when set. These flags do not affect the caching and write policy for individual page tables. These flags only have effect when paging is enabled and the CD flag in control register CR0 is clear.

PCD and **PWT** flags in the page-directory and page-table entries control caching for individual page tables and pages, respectively. The PCD flag and the PWT flag have the same effect than in the CR3 register.

This mechanism offers an advantage compared to the MTRR method, not limiting the number of pages where we can define a cache policy.

The Page Attribute Table (PAT) extends the IA-32 architecture's page-table format to allow memory types to be assigned to regions of physical memory based on linear address mappings. The PAT is a companion feature to the MTRR's. The PAT was introduced into the IA-32 architecture in the Pentium III processor and is also available in the Pentium IV processor.

MSR 0x277 is a 64 bit register that contains eight page attribute fields: PA0 through PA7. Each of the eight attribute fields can contain any of the memory type that can be encoded with PAT, being the same that those used for encoding memory types in the MTRR mechanism.

To select a memory type for a page from the PAT, a 3-bit index made up of the PAT,PCD, and PWT bits most be encoded in the page-table or page-directory entry for the page.

TABLE 1: *Selection of PAT entries with PAT, PCD, and PWT flags.*

PAT	PCD	PWT	PAT entry
0	0	0	PA0
0	0	1	PA1
0	1	0	PA2
0	1	1	PA3
1	0	0	PA4
1	0	1	PA5
1	1	0	PA6
1	1	1	PA7

4 Translation Lookaside Buffer (TLB)

To minimize the number of bus cycles required for address translation, the most recently accessed page-directory and page-table entries are cached in the processor in devices called translation lookaside buffers (TLBs). The CPUID instruction can be used to determine the sizes of the TLBs provided in the Pentium processors.

The TLBs are inaccessible to application programs and tasks (privilege level greater than 0); that is, they cannot invalidate TLBs. Only operating system or executive procedures running at privilege level of 0 can invalid TLBs or selected TLB entries.All of the (nonglobal) TLBs are automatically invalidated any time the CR3 register is loaded (unless G flag for a page or page-table entry is set)

The INVLPG instruction is provided to invalidate a specific page-table entry in the TLB. Normally, this instruction invalidates only an individual TLB entry; however, in some cases, it may invalidate more than the selected entry and may even invalidate all of the TLBs. This instruction ignores the setting of the G flag in a page-directory or page-table entry.

The Page Global Enable (PGE) flag in register CR4 and the global (G) flag of a page-directory or page-table entry can be used to prevent frequently used pages from being automatically invalidated in the TLBs on a task switch or a load of register CR3.

References

[1] *Intel Architecture Software Developer's Manual. Volume 3:System Programming.*

[2] Pentium II processor application notes. *Using the RDTSC Instruction for Performance Monitoring.*

Towards designing WCET-predictable processors

Christine Rochange and Pascal Sainrat
Institut de Recherche en Informatique de Toulouse
118, route de Narbonne
31062 Toulouse cedex 4, France
{rochange, sainrat}@irit.fr

Abstract

Several methods based on a static analysis of the executable code have been proposed in the past to estimate the worst-case execution time of programs. Their main advantage is to limit measurements to small parts of code (e.g. basic blocks). However these methods have been designed for basic processor architectures and recent work by Engblom has shown that they would not be safe for more advanced designs. Actually, the execution of a basic block could have an impact on the execution of a distant subsequent basic block. Ignoring this impact could result in an under-estimated WCET, which could be dramatic in a hard real-time context. In this paper, we suggest that advanced architectures could include specific hardware that would eliminate all possible long timing effects. We show how this idea could permit to make superscalar pipelines analyzable for the WCET.

1. Introduction

1.1 Evaluating the Worst-Case Execution Time

For the calculation of the Worst-Case Execution Time of a program, the ideal thing would be to measure (or simulate) all the possible execution paths. This is generally not affordable because it would be very expensive in time. Moreover, while measuring all the complete paths, parts of code that belong to several paths would be evaluated several times. Thus, the objective is to limit measurements, as much as possible, to small parts of code.

The behaviour of some components of the processor architecture (like the cache memories or the branch predictor) is very dependent on the execution history and the timing analysis has to consider complete execution paths. To fasten the analysis, techniques like *static simulation* examine several paths in parallel, which might require a large storage capacity.

Some other parts of the processor, like the execution pipeline, are expected to exhibit a more «local» behaviour. So, as far as they are concerned, it is possible to measure parts of code, instead of complete paths, which limits the redundancy of measurements. These parts of code can be basic blocks, or bodies of algorithmic structures. The calculation of the WCET then consists in combining the individual execution times of the parts to obtain the execution time of the longest possible path.

Some WCET computation methods are based on a bottom-up traversal of the program syntax tree, while others are based on the control flow graph. In this paper, we focus on a method of this second category: the Implict Path Enumeration Technique or IPET [LiMa95], which consists in representing the control flow graph and the results of the flow analysis by a set of constraints on the numbers of executions of each part of code (basic block), and then in maximizing the total execution time (which is the sum of the products of the number of executions by the execution time of each basic block).

1.2 Modeling pipelined processors

To be able to model both correctly and precisely pipelined processors, the above method has to be adapted to take into account the pipeline effect that makes the execution of a sequence of two basic blocks shorter than the addition of the two individual execution times.

In section 2, we show how this effect can be included in the model. We also outline Engblom's analysis of long timing effects associated to sequences of more than two basic blocks and we argue that the IPET method cannot easily take into account these effects. Instead of restricting the choice for a real-time system to very simple architectures that cannot generate long timing effects, we think that high-performance processors could include a hardware mechanism to eliminate them, as defended in section 3. In section 4, we apply this principle to a perfect superscalar processor and show that the performance loss is very small. Section 5 concludes the paper.

2. Inter-block timing effects

Processor pipelines generate two kinds of timing effects:

- *pairwise effects* due to the overlapping of two adjacent basic blocks in the pipeline
- *long timing effects* that represent the impact of this overlap when sequences of three and more blocks are considered.

In this section, we examine how these effects could be modeled in the IPET method.

2.1 Pairwise timing effects

Timing effects between two adjacent blocks can be modeled by defining an execution time for the edge that connects them in the control flow graph (as illustrated in Figure 1). The execution time of an edge represents the gain due to the overlap of the two blocks in the pipeline. It is always negative. Then, the set of constraints that

express the structure of the control flow graph is rewritten to link the execution times of blocks and edges.

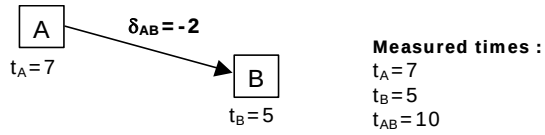

Figure 1. *Pairwise timing effects*

2.2 Long timing effects: Engblom's timing model

Engblom has shown in his PhD thesis [Engb02] that there could exist timing effects between distant basic blocks: «*A long timing effect for a sequence of instructions $I_1 \ldots I_m$, $m \geq 3$, occurs whenever I_1 has the effect of disturbing the execution in such a way that the execution of the instructions $I_2 \ldots I_m$ is different compared to if I_1 had not been present*». An example of a long timing effect is given in Figure 2. Among the sources of long timing effects, Engblom mentions parallel pipelines, long latency instructions, dynamic scheduling, … Engblom has highlighted that long timing effects could occur for sequences of any length (potentially infinite), and that they could be either negative, null or positive.

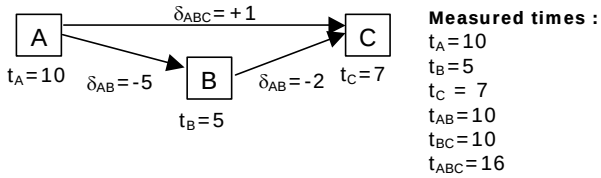

Figure 2. *Long timing effects*

Engblom proposed a timing model where a timing effect $\delta_{i\ldots j}$ is associated to each sequence of basic blocks $B_i \ldots B_j$. The execution time of a sequence of basic blocks $B_1 \ldots B_n$ is then given by :

$$t_{1\ldots n} = \sum_{i=1}^{n} t_i + \sum_{1 \leq j \leq k \leq n} \delta_{j\ldots k}$$

Whenever a long timing effect is negative, it can be ignored, which might lead to WCET over-estimation. But when it is positive, it has to be taken into account for a safe WCET analysis. To model long timing effects when applying the IPET method, one should add some weighted edges between non-adjacent blocks. The main difficulty is then to obtain the weight of these edges because it requires to measure the execution time of the corresponding sequences. Since a long timing effect can exist between two blocks that are very far from each other, all the possible sequences of blocks have to be measured, which could be even longer than measuring all the possible execution paths and obviously goes against the principle of the IPET method (that is to limit measures to small parts of code). Expressing constraints on the number of executions of long edges might be difficult too.

3. Towards a high-performance processor without long timing effects

Pipelined processors can be safely analysed using the IPET method if they are guaranteed not to generate positive long timing effects.

What has been proposed until now is to restrict the choice of a processor architecture to one that does not exhibit any possibility of positive long term effects. Engblom has shown that a single in-order pipeline has this property.

However, this kind of architecture might not meet higher and higher performance requirements. This is why we suggest a new approach that consists in adding to the processor hardware the ability to prevent the appearance of positive long timing effects, as illustrated in Figure 3. This mechanism analyses the instruction flow to detect conditions that could engender long timing effects (LTEs): occupation of a resource during several clock cycles, access to the memory hierarchy with default in the first-level cache memory, … Then, the mechanism controls the pipeline to prevent the next basic block to enter the pipeline until the end of the risk of long timing effects.

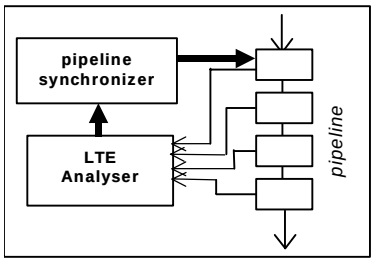

Figure 3. *A mechanism to eliminate long timing effects*

We feel that this approach could be implemented in high-performance processors, and the challenge is to limit the induced performance degradation.

In the next section, we show how this principle could be applied to a processor with a superscalar in-order pipeline (that was shown by Engblom to possibly generate infinite positive long term effects).

4. Case study: superscalar pipeline

4.1 Evaluation methodology

In order to focus on the impact of parallel pipelines, we make the following assumptions: perfect multiple branch prediction, perfect cache memories, no interlocks due to data dependencies, all instructions executed in a single cycle, … This is what we call a «perfect» pipeline.

Performance results that we will give were obtained using a simulator of a 6-stage perfect pipelined processor that we developped within the MicroLib project [MIC] based on SystemC.

The benchmark codes we used are listed in Table 1. They were taken from the SNU benchmark suite [SNU], and were slightly modified to inline function calls. We

only excuted the main function (i.e. not the starting and ending code).

		# insts
`crc`	*CRC computation*	54 790
`fft1`	*FFT using Cooly-Turkey algorithm*	3163
`jfdctint`	*JPEG slow-but-accurate integer implementation of the forward DCT*	5828
`lms`	*LMS adaptive signal enhancement*	535 985
`ludcmp`	*LU decomposition*	7 797

Table 1. *Benchmark applications (from the SNU suite)*

4.2 Long timing effects in a superscalar pipeline

Figure 3 shows how a sequence of 4 basic blocks (A-B-C-D) would be executed in a perfect 4-stage 2-way superscalar pipeline.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
IF	A_1 A_2	A_3 B_1	B_2 C_1	C_2 C_3	C_4 C_5	D_1			
D		A_1 A_2	A_3 B_1	B_2 C_1	C_2 C_3	C_4 C_5	D_1		
EX			A_1 A_2	A_3 B_1	B_2 C_1	C_2 C_3	C_4 C_5	D_1	
WB				A_1 A_2	A_3 B_1	B_2 C_1	C_2 C_3	C_4 C_5	D_1

starting with block A

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
IF	B_1 B_2	C_1 C_2	C_3 C_4	C_5 D_1					
D		B_1 B_2	C_1 C_2	C_3 C_4	C_5 D_1				
EX			B_1 B_2	C_1 C_2	C_3 C_4	C_5 D_1			
WB				B_1 B_2	C_1 C_2	C_3 C_4	C_5 D_1		

starting with block B

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
IF	C_1 C_2	C_3 C_4	C_5 D_1						
D		C_1 C_2	C_3 C_4	C_5 D_1					
EX			C_1 C_2	C_3 C_4	C_5 D_1				
WB				C_1 C_2	C_3 C_4	C_5 D_1			

starting with block C

Figure 3. *Executing a sequence of 4 basic blocks in a 2-way superscalar pipeline*

From this figure, we can compute the execution times and the timing effects of each sub-sequence of A-B-C-D, as shown in Table 2.

More generally, we can compute that, whatever the number of parallel pipelines is, all the long timing effects (even the timing effects for infinite-length basic block sequences) equal -1, 0 or +1. Proof is given in Appendix. As mentioned before, a positive timing effect constitutes a difficulty for computing the WCET using the IPET method.

sub-sequence	***execution time***	***timing effect***
A	5	-
B	4	-
C	6	-
D	4	-
A-B	6	-3
B-C	7	-3
C-D	6	-4
A-B-C	8	-1
B-C-D	7	0
A-B-C-D	9	**+1**

Table 2. *Computing inter-block timing effects*

We have measured the frequency of positive timing effects. Results are given in Table 3. They show that positive timing effects are frequent: on a mean, 14.39% of the 6-block sequences exhibit a positive effect (+1). This confirms the importance of the problem, even with a quite simple pipeline.

seq. length	***3***	***4***	***5***	***6***
`crc`	21.28%	13.18%	22.03%	18.33%
`fft1`	7.85%	10.83%	10.91%	11.02%
`jfdctint`	38.10%	24.00%	28.13%	23.08%
`lms`	19.05%	19.10%	18.45%	9.09%
`ludcmp`	7.07%	11.72%	10.18%	10.41%
MEAN	**18.67%**	**15.76%**	**17.94%**	**14.39%**

Table 3. *Percentage of positive timing effects, as a function of the sequence length.*

4.3 Synchronizing the pipeline to eliminate long timing effects

We propose to include in the processor a mechanism that resynchronizes the pipeline whenever a basic block enters in the fetch stage (see Appendix 2) while the first slot of this stage is occupied by one instruction of the previous block. In our example, this would produce the scheduling shown in Figure 4. Instructions that belong to the same basic block can then be processed in parallel but no timing effect can occur between basic blocks.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
IF	A_1 A_2	A_3	B_1 B_2	C_1 C_2	C_3 C_4	C_5	D_1			
D		A_1 A_2	A_3	B_1 B_2	C_1 C_2	C_3 C_4	C_5	D_1		
EX			A_1 A_2	A_3	B_1 B_2	C_1 C_2	C_3 C_4	C_5	D_1	
WB				A_1 A_2	A_3	B_1 B_2	C_1 C_2	C_3 C_4	C_5	D_1

Figure 4. *Executing a sequence of 4 blocks in a 2-way superscalar pipeline with a resynchonizing mechanism*

In the previous example, resynchronizing increments the execution time of the sequence by one clock cycle. Table 4 gives measures of the performance degradation in terms of instruction throughput (number of committed instructions per cycle) over a non-synchronized pipeline.

pipeline width	*2-way*	*4-way*	*8-way*
`crc`	-8.55%	-15.98%	-39.67%
`fft1`	-3.11%	-10.76%	-22.44%
`jfdctint`	-1.52%	-4.07%	-10.15%
`lms`	-5.76%	-8.32%	-25.13%
`ludcmp`	-2.72%	-7.65%	-23.32%
MEAN	**-4.33%**	**-9.36%**	**-24.14%**

Table 4. *Instruction throughput degradation due to the synchronization of the pipeline*

The reduction of the instruction throughput appears not to be so high and synchronizing the pipeline does not give up the benefit of parallel pipelines. For example, the speedup of a 2-way synchronized superscalar processor over a scalar processor is, on a mean over the five benchmarks, 1.91.

5. Conclusion

This paper comes after Englom's thesis which has highlighted that the execution of a given basic block can have an influence on the execution of a distant subsequent block. This influence can be expressed as a value, called *long timing effect*, that can either be positive, negative or null. A long timing effect has to be taken into account in the computation of the execution time of any sequence that includes those two blocks.

Now, several methods for WCET computation, like IPET, aim for restricting measurements to small parts of code (basic blocks). This does not permit to take long timing effects into account, and then this class of methods can only be applied to processors that cannot generate long timing effects. Engblom has shown that some simple pipelined processors have this property.

In this paper, we proposed another approach that consists in including in the processor a mechanism that dynamically detects conditions that might engender a long timing effect and synchronizes the pipeline in such a way that this effect cannot appear. To illustrate this approach, we applied it to a perfect superscalar pipeline.

Simulation results show that the performance degradation is limited: the *synchronized* superscalar processor is competitive compared to a scalar processor.

References

[Engb02] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. Uppsala Dissertations from the Faculty of Science and Technology 36, April 2002.

[LM95] Y.-T. S. Li, S. Malik. *Performance Analysis of Embedded Software Using Implicit Path Enumeration*. 32nd Design Automation Conference (DAC), 1995.

[MIC] `http://www.microlib.org`

[SNU] `http://archi.snu.ac.kr/realtime/benchmark/`

Appendix 1. Computing long time effects in a superscalar pipeline.

Let us consider an n_i-instruction basic block. Its execution time in a perfect (without any stall) s-stage scalar pipeline is given by $t_i = s + n_i - 1$.

To express its execution time in a perfect w-way s-stage superscalar pipeline, we write $n_i = x_i.w + y_i$.
Then, we obtain:

$$t_i = s + \left\lceil \frac{n_i}{w} \right\rceil - 1 \qquad \text{or} \qquad t_i = s + x_i + \left\lceil \frac{y_i}{w} \right\rceil - 1$$

where $\lceil q \rceil$ is the upper integer value of q.

Now, the execution time of a sequence of basic blocks $B_i \ldots B_j$ can be expressed as:

$$t_{i\ldots j} = s + \left\lceil \frac{\sum_{k=i}^{j} n_k}{w} \right\rceil - 1 = s + \sum_{k=i}^{j} x_k + \left\lceil \frac{\sum_{k=i}^{j} y_k}{w} \right\rceil - 1$$

We can then compute the long timing effect $\delta_{1\ldots m}$ associated to the sequence of basic blocks $B_1 \ldots B_m$. In Engblom's thesis, it is defined as:

$$\delta_{1\ldots m} = t_{1\ldots m} - t_{2\ldots m} - t_{1\ldots m-1} + t_{2\ldots m-1}$$

So we can write:

$$\delta_{1\ldots m} = \sum_{1}^{m} x_i - \sum_{2}^{m} x_i - \sum_{1}^{m-1} x_i + \sum_{2}^{m-1} x_i + \left\lceil \frac{\sum_{1}^{m} y_i}{w} \right\rceil - \left\lceil \frac{\sum_{2}^{m} y_i}{w} \right\rceil - \left\lceil \frac{\sum_{1}^{m-1} y_i}{w} \right\rceil + \left\lceil \frac{\sum_{2}^{m-1} y_i}{w} \right\rceil$$

Since $0 \le y_i < d, \forall i$, we have:

$$\left\lceil \frac{\sum_{1}^{m} y_i}{w} \right\rceil - \left\lceil \frac{\sum_{2}^{m} y_i}{w} \right\rceil = \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \quad (0 \text{ or } 1)$$

and then $\delta_{1\ldots m} = \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} - \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} = \begin{Bmatrix} -1 \\ 0 \\ 1 \end{Bmatrix}$

Appendix 2. Detecting basic blocks.

Hardware designers usually define a basic block as a sequence of code preceded by a branch and ending with the next branch. But our mechanism has to detect compiler basic blocks, defined as sequences of code that can only be entered at their first instruction and exited at their last instruction. Their detection requires to have a full knowledge of the static code, which is not the case of the hardware.

Then we suggest that the compiler could help the hardware by marking the beginning of basic blocks. It could exploit unused bits of the instruction codes, if such bits are available in the target instruction set. Otherwise, the compiler could make sure that every basic block ends with a control flow instruction, by adding a branch to the next instruction whenever it should not be the case.

A Flexible Tradeoff between Code Size and WCET by Employing Dual Instruction Set Processors

Sheayun Lee[†] Jaejin Lee[†] Chang Yun Park[*] Sang Lyul Min[†]

[†]School of Computer Science and Engineering
Seoul National University
Seoul 151-742, Korea

[*]Department of Computer Science and Engineering
Chungang University
Seoul 156-756, Korea

E-mail: sylee@archi.snu.ac.kr

Abstract

Code size is an important design issue in cost-sensitive embedded systems. A dual instruction set processor, which supports a reduced instruction set in addition to a full instruction set, can be used to reduce the code size by using smaller instructions in generating application program code. In general, however, a program compiled into the reduced instruction set typically runs slower than its full instruction set counterpart. Motivated by this observation, we propose a technique that enables a flexible tradeoff between a program's code size and its WCET (worst-case execution time) by selectively using the two different instruction sets for different sections within a single program.

1. Introduction

Embedded systems are often characterized by stringent constraints imposed on code size, due to a small amount of available memory. Therefore, a number of techniques have been proposed to reduce the memory space occupied by application program code. One promising technique is to employ a dual instruction set, where the processor supports both a full (normal) instruction set and a reduced instruction set, in which an instruction has a smaller number of bits [3]. Examples include the ARM with the 16-bit Thumb instruction set [2], the MIPS 32/16-bit TinyRISC [5], and the ARC Tangent [4] processors.

When such a dual instruction set processor is used, the same program compiled into different instruction sets have distinguished characteristics in terms of code size and execution time. Specifically, a program compiled into the reduced instruction set is typically smaller but runs slower than the same program compiled into the full instruction set. The main reason behind this performance gap is that the full instruction set program executes fewer instructions, since a single full instruction can perform more operations than a single reduced instruction does.

This motivates us to exploit the tradeoff relationship between a program's code size and its WCET (worst-case execution time) for real-time embedded systems, where the amount of available memory is often limited and tasks must meet timing constraints. We propose a technique that enables a flexible tradeoff between code size and WCET by selectively using the two instruction sets for different sections within a given program. The technique begins with the smallest code possible for a given program, and estimates the reduction of the WCET that can be achieved by utilizing a certain amount of additional code space. We repeat this procedure by gradually increasing the code size limit, until no more reduction of the WCET is possible by using more code space. The proposed technique is centered around selective code transformation, where a given program is first compiled into the reduced instruction set, and a selected subset of basic blocks are then transformed into the full instruction set, in a way that the reduction of the WCET is maximized within a given code size budget.

2. Overall approach

Our technique for selective code transformation consists of three steps. First, we compile the whole program into the reduced instruction set, and estimate its WCET using a hierarchical analysis technique [7]. This will serve as the baseline for the code size, i.e., the smallest code possible for the given program. Second, we determine the set of basic blocks to be transformed into the full instruction set that gives the maximum reduction of the WCET, while maintaining the code size under a given upper bound. Finally, the selected basic blocks are actually transformed into the full instruction set, and a mixed instruction set code is generated as a result.

The procedure of selective code transformation is illustrated in Figure 1. In order to determine the set of basic blocks to be transformed into the full instruction set, we

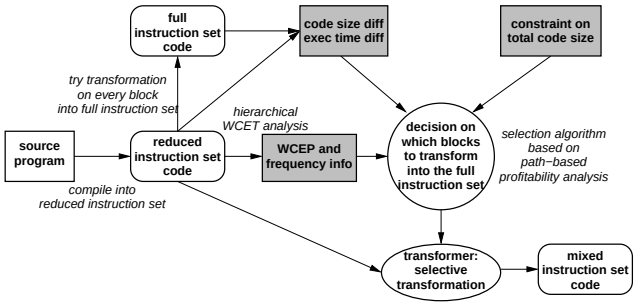

Figure 1. Overview of selective code transformation.

need information about the code size and execution time of each basic block compiled into both the reduced and the full instruction sets. The code size of each block can be estimated in a straightforward manner because it can be statically determined by examining the instruction sequence in that block. On the other hand, we assume that the worst-case execution time of each basic block is estimated by an existing analysis technique such as the one presented in [6]. We obtain the code size and execution time information of each basic block in the full instruction set by performing the transformation without generating code.

In addition to the code size and execution time of each block, the selection algorithm requires information about their execution frequency corresponding to the worst-case execution scenario of the given program. This frequency information can be derived from the hierarchical WCET analysis of the given program, since the WCEP (worst-case execution path) and the corresponding execution count for each basic block can be extracted from the syntax tree used in the analysis. The execution frequency information for each block combined with the code size and the execution time differences is input to the selection algorithm. Based on its results, the selected blocks are transformed into the full instruction set and the final mixed instruction set code is generated.

3. Selective code transformation

One complication in determining the basic blocks to be transformed is that the mixed use of the dual instruction set requires proper handling of transitions between execution of the different instruction sets. These mode switches are typically triggered by executing a special instruction or sequence of instructions. Since the insertion of such mode switch instructions incurs overhead in terms of both code size and execution time, our technique should take the mode transitions into account.

Intuitively, the blocks to be transformed into the full instruction set are those on the WCEP that are frequently executed under the worst-case execution scenario. We cannot, however, consider the basic blocks individually, since transforming a single block will usually degrade the WCET due to the mode switch overhead, while nonetheless increasing the code size. Therefore, we define a cost-benefit model based on *acyclic subpaths* [1], which can capture the set of basic blocks executed together. Section 3.1 describes in detail the path-based cost-benefit model and the greedy heuristic that iteratively selects the subpath with the maximum ratio of its benefit to its cost.

3.1. Cost-benefit model and selection algorithm

A program is given by a control flow graph $P = \langle V, E \rangle$, where $V = \{v_i \mid i = 1, 2, \cdots, n\}$ is the set of basic blocks and $E = \{e_{ij} = \langle v_i, v_j \rangle\}$ is the set of edges which represent the control flow in the program. Assume that a subset of basic blocks are in the full instruction set and the remaining blocks are in the reduced instruction set. That is, the set of blocks V is partitioned into two disjoint subsets F and R, which denote the set of blocks in the full and the reduced instruction sets, respectively. We define a set of functions to denote the code size and the execution time of a basic block when compiled into the two different instruction sets. Let $s_F(v)$ and $s_R(v)$ denote the code size of a block v compiled into the full and the reduced instruction sets, respectively. Similarly, we denote by $t_F(v)$ and $t_R(v)$ the (worst-case) execution time of block v compiled into the full and the reduced instruction sets, respectively.

Given a path p, the cost of transforming p can be calculated by first summing the code size difference for blocks being transformed, and then adding the mode switch overhead to the sum. Note that transforming the blocks on the path not only causes insertion of new mode switch instructions but also possibly results in removal of certain mode switch instructions that were previously needed. Specifically, when a block is transformed into the full instruction set, we should remove the mode switch instructions that were previously inserted on the edges connecting the block with other blocks that are already in the full instruction set.

To account for the removal of mode switch instructions as well as the insertion of newly introduced mode switch instructions, we define $E^M(p)$ to be the set of edges where mode switch instructions are newly introduced, and $E^m(p)$ to be the set of edges from which existing mode switch instructions are removed. In addition, we let $V(p)$ the set of all the basic blocks on path p. Then, the set of blocks to be transformed on the path p is given by $V(p) \bigcap R$, which contains only those blocks on p that have not yet been transformed. Then the cost $c(p)$ of transforming path p can be computed as follows:

$$c(p) \quad = \quad \sum_{v \in V(p) \cap R} (s_F(v) - s_R(v))$$

$$+ \quad o_s \times \left(|E^M(p)| - |E^m(p)| \right) , \qquad (1)$$

where o_s denotes the total size of instructions required for a single mode switch.

On the other hand, the benefit $b(p)$ of transforming a path can be computed by first summing the execution time difference for each block multiplied by its execution frequency, and then subtracting the mode switch overhead. That is, the benefit associated with transformation of a path p is given by

$$\begin{aligned} b(p) &= \sum_{v \in V(p) \cap R} \left(c_V(v) \times (t_R(v) - t_F(v)) \right) \\ &- o_t \times \left(\sum_{e \in E^M(p)} c_E(e) - \sum_{e \in E^m(p)} c_E(e) \right) \end{aligned} \qquad (2)$$

where o_t denotes the execution time overhead incurred by a single mode switch, while $c_V(v)$ and $c_E(e)$ give the execution counts of block v and edge e, respectively.

Now we define a reward function $r(p)$ for a subpath p to be the ratio of its benefit to its cost. That is, $r(p) = b(p)/c(p)$, which indicates the expected amount of reduction of the WCET for the unit increase in the code size.

Based on this cost-benefit model, we select the blocks to be transformed as follows. First, we enumerate all the acyclic subpaths of the WCEP of the program, and calculate the cost and benefit associated with each of them. Then we apply a simple heuristic that iteratively selects the subpath to be transformed by giving priority to the one with the maximum reward function value (i.e., $r(p)$). The selection is repeated until no more transformation can be done because one or more of the following conditions are met: (1) selection of any of the remaining subpaths would violate the code size limit, (2) no further reduction of the WCET is possible, or (3) all the blocks in the program have already been transformed.

Note that, when a subpath is selected and the blocks on it are transformed, the cost and benefit of other subpaths may change because (1) certain subpaths share the transformed blocks with the selected subpath, and (2) the insertion of mode switch instructions possibly affects the cost and benefit of transforming other subpaths. Therefore, we adjust the cost and benefit of each subpath as we iterate the selection of the subpath to be transformed. Moreover, since the execution time of the WCEP has been reduced, it is no longer guaranteed to have the largest execution time among all the possible execution paths in the program. When the WCEP of the program is changed by the iterative procedure of selective code transformation, we should re-enumerate the acyclic subpaths of the the new WCEP and resume the selection procedure. Therefore, we need a mechanism to update the timing information associated with each of the intermediate nodes in the syntax tree, so that we can detect a possible change of the WCEP. The next section describes the method to handle the update of timing information.

3.2. Augmented Timing Schema

We check whether or not the WCEP of the program is changed by the transformation of a subset of basic blocks, by updating the timing information associated with nodes in the syntax tree. Beginning from the leaf nodes (i.e., basic blocks) whose execution times have been changed by selective code transformation, we propagate the timing information along the tree edges, until we reach the root node. By recording in each syntax tree node the WCET of the program construct that it represents, we can update the changes in the WCET (and thus the WCEP) by re-evaluating the timing formula for only those nodes on the paths from the leaf nodes whose execution times are changed to the root node.

In updating the timing information, our WCET analysis method should be able to handle the execution time overhead of mode switch instructions. Therefore, the basic timing schema [7] is augmented so that the execution times of control flow edges can be accounted for, as well as those of basic blocks. Specifically, we augment the structure of the syntax tree used in the hierarchical WCET analysis, so that each intermediate node has as its child node the set of control flow edges contributing to the WCET of the corresponding program construct. When the execution time of an edge is changed due to insertion or removal of mode switch instructions, this timing information is propagated along the tree edges as well as the execution time changes of basic blocks.

In addition, the timing formulas used for calculating the WCET of various program constructs are also modified to incorporate the edge execution times. Specifically, the execution times of control flow edges are included in the timing formula for each program construct. Since there can be more than one edges connecting one program construct to another, the timing formulas take the maximum of these edge execution times in calculating the WCET of two program constructs executed sequentially. The actual augmented formulas are not presented in the paper due to space limitation.

While propagating the timing information updates on the syntax tree, we can determine whether or not the WCEP of the whole program is changed, by concentrating on the conditional statement nodes whose timing formula is re-evaluated. Since the subtree that represents the WCEP of the program can only be different when a conditional statement node selects different child node for its WCET calculation, we can detect any change of the WCEP by checking the conditional statement nodes whose WCETs are re-evaluated.

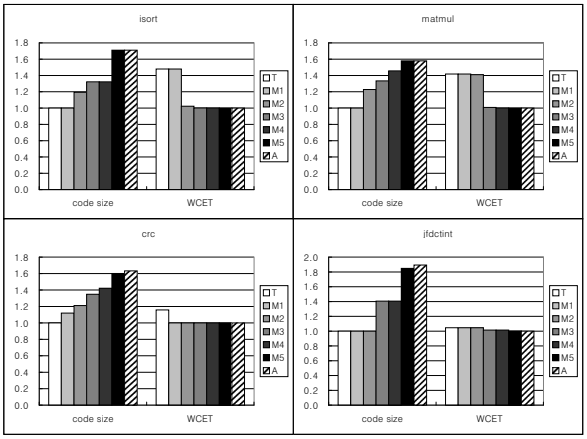

Figure 2. Code size and WCET for four benchmark programs.

4. Results

To demonstrate the validity and effectiveness of our proposed approach, we implemented the proposed technique for our target ARM/Thumb dual instruction set processor, and performed a set of experiments. The benchmark program used for our experiments were derived from applications commonly used in embedded systems software. The *isort* program performs insertion sort on an array of integer, and *matmul* implements matrix multiplication algorithm. The *crc* benchmark computes the cyclic redundancy check code, while *jfdctint* implements the forward DCT (discrete cosine transform) for JPEG image compression algorithm.

Figure 2 shows the results from our experiments. For each of the benchmark programs, we initially set the code size limit equal to the size of the program compiled entirely into Thumb instructions, which is denoted by T in the figure. For comparison purposes, we generated code by transforming all the basic blocks in the program into ARM instructions, which is denoted by A. Then we gradually increased the code size limit so that the available code space budget is 20 %, 40 %, 60 %, 80 %, and 100 % of the difference between code sizes of A and T, and the resulting code for each code size budget is labeled from M_1 to M_5. In the figure, the code size for each program is normalized to that of T, while the WCET is normalized to that of A.

From the figure, we observe that the code sizes of application programs increase as we provide more code size budget, while the WCET is reduced. One interesting observation is that the WCET of each program decreases sharply when the code size exceeds a certain point. After closely examining the resulting code, we found that this sudden decrease of the WCET occurs when a time-critical loop can be transformed into ARM instructions within the given code size budget.

The results indicate that a tradeoff relationship exists between a program's code size and WCET. The amount of achievable reduction in the WCET is different from one benchmark to another, which is dependent on the characteristics of the program. We expect that we can exploit more flexible tradeoff between code size and WCET for larger application programs, whose execution time is spent on a number of different code sections in the program.

5. Conclusions

We have proposed a technique that enables a flexible tradeoff between a program's code size and its WCET, by employing a dual instruction set processor. In the proposed technique, we first compile the whole program into the reduced instruction set, and then selectively transform a subset of basic blocks on the WCEP to reduce the WCET as much as possible within a given code size budget. For the selective code transformation, we defined a path-based cost-benefit model that can accurately estimate the impact of transforming a set of blocks on the code size and the WCET of the entire program. Our technique incorporates a simple greedy selection heuristic to iteratively select the set of blocks to be transformed from the reduced instruction set to the full instruction set. When the algorithm iteratively selects the blocks to be transformed, the program's WCEP may change because the execution time of the previous WCEP is now reduced. To rectify this problem, we augmented the syntax tree and the timing formulas used in the hierarchical WCET analysis, so that we can easily detect a possible change of the WCEP by updating the timing information of each program construct on the syntax tree. The results from our experiments show that there exists a tradeoff relationship between a program's code size and its WCET, and the proposed technique can be effectively used to exploit this tradeoff.

References

[1] T. Ball and J. R. Larus. Effi cient path profi ling. In *Proceedings of the 29th Annual IEEE/ACM Symposium on Microarchitecture*, pages 46–57, Paris, France, 1996.

[2] S. Furber. *ARM System Architecture*. Addison-Wesley, 1996. ISBN 0-201-40352-8.

[3] A. Halambi, A. Shrivastava, P. Biswas, N. Dutt, and A. Nicolau. An effi cient compiler technique for code size reduction using reduced bit-width ISAs. In *Proceedings of the DATE (Design, Automation and Test in Europe)*, Paris, France, March 2002.

[4] A. C. (http://www.arc.com). *The ARCtangent-A5 Processor*.

[5] K. Kissel. MIPS16: High-density MIPS for the embedded market. Technical report, Silicon Graphics MIPS Group, 1997.

[6] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–694, 1995.

[7] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 72–81, December 1990.

Aspect-Level WCET Analyzer: a Tool for Automated WCET Analysis of a Real-Time Software Composed Using Aspect and Components*

A. Tešanović*, J. Hansson*, D. Nyström†, C. Norström†, and P. Uhlin*

*Linköping University
Dept. of Computer Science
Linköping, Sweden
{alete,jorha}@ida.liu.se

†Mälardalen University
Dept. of Computer Engineering
Västerås, Sweden
{dag.nystrom,christer.norstrom}@mdh.se

Abstract

Increasing complexity in development of real-time systems requires the integration of new software engineering techniques, such as aspect-oriented and component-based software development, with real-time system development. Since software technology for building real-time systems has to support timeliness, methods and tools for analyzing temporal behavior of the software composed out of components and aspects are needed. We contribute by providing a tool that enables automated worst-case execution time analysis of different configurations of aspects and components.

1 Introduction

Increasing complexity in development of real-time systems accompanied by the demand for enabling their configurability requires the integration of aspect-oriented and component-based software development with real-time system development. We have developed an approach to aspectual component-based real-time system development (ACCORD) [8] that integrates the two software engineering techniques, aspect-oriented and component-based software development, into real-time system development. ACCORD introduces a real-time component model (RTCOM) that provides explicit support for aspect weaving, while enforcing information hiding, i.e., it preserves basic ideas from component-based and aspect-oriented software development.

Since software technology for building real-time systems has to support timeliness [5], methods and tools for analyzing temporal behavior of the software composed out of components and aspects are needed. It is well-known that the worst-case execution time (WCET) is of primary importance for timing analysis of real-time systems.

We contribute by providing a tool that enables automated WCET analysis of different configurations of aspects and components. The tool is based on aspect-level WCET analysis [7]. The main goal of a tool for aspect-level WCET analysis is determining the WCETs of different real-time system configurations consisting of aspects and components before any actual aspect weaving (system configuration) is performed, and, hence, help the designer of a configurable real-time system to choose the system configuration fitting the WCET needs of the underlying real-time environment without paying the price of aspect weaving for each individual candidate configuration. If necessary, i.e., if very precise WCET estimates are needed, the tool for aspect-level WCET analysis can be followed by further analysis of the resulting weaved code using a more specialized WCET tool (e.g., that performs both low level and high level WCET analysis).

The paper is organized as follows. Section 2 gives the background information about RTCOM. The main constituents of automated aspect-level WCET analysis, including aspect-level WCET specifications and the aspect-level WCET analyzer, are described in section 3. Finally, in section 4 we discuss limitations and benefits of the current implementation of the tool.

2 Background

RTCOM consists of three parts: (i) functional part, which can be modified by aspect weaving, (ii) run-time part describing the run-time behavior, e.g., WCETs, of components and aspects, and (iii) composition part describing the composition rules of components and aspects. Detailed description of RTCOM can be found in [8], and here we present a brief overview of its functional part which represents the actual code of the component.

To enable efficient temporal analysis of components weaved with aspects and facilitate structured aspect weaving, while preserving information hiding, RTCOM assumes the following for the functional part (i.e., the actual code) of the component.

- Each component provides a set of mechanisms, which are basic and fixed parts of the component infrastructure. Mechanisms can be viewed as fine-granule methods or functions.
- Each component provides a set of operations to other components and/or to the system. The implementation

*This work is supported by ARTES (A network for Real-Time and graduate Education in Sweden).

of the operations determines the initial behavior of the component, i.e., the policy framework. Operations can be viewed as coarse-granule methods or functions. Operations are implemented using the underlying mechanisms, which are fixed parts of the component.

Existing aspect languages can be used for implementing aspects and integrating (weaving) them into the functional part of RTCOM. Aspect weaving is done by the aspect weaver corresponding to the aspect language [3]. In an aspect language each aspect declaration consists of advices and pointcuts. A *pointcut* consists of one or more join points, and is described by a pointcut expression. A *join point* refers to a point in the component code where aspects should be weaved, e.g., a method, or a type (struct or union). An *advice* is a declaration used to specify the code that should run when the join points, specified by a pointcut expression, are reached. Different kinds of advices can be declared, such as: (i) *before advice*, which is executed before the join point, (ii) *after advice*, which is executed immediately after the join point, and (iii) *around advice*, which is executed in place of the join point.

Each aspect, as prescribed by RTCOM, is implemented using a number of mechanisms of a component and represent a (new) component policy. Implementation of an aspect is not limited only to mechanisms of one component as the same aspect can influence several components, and, thus, can be implemented using the mechanisms from a number of components. Weaving of aspects into the code of a component does not change the implementation of mechanisms, only the implementation of operations within the component. Hence, aspect weaving changes the policy of the component by changing one or more operations, and changing the number of mechanisms used by a particular operation.

Consider a simple example of an ordinary linked list implemented based on RTCOM. The mechanisms needed are the ones for the manipulation of nodes in the list, i.e., `createNode`, `deleteNode`, `getNextNode`, `linkNode`, and `unlinkNode`. Operations implementing the policy framework, e.g., `listInsert`, `listRemove`, `listFindFirst`, define the behavior of the component, and are implemented using the underlying mechanisms. In this example, `listInsert` uses the mechanisms `createNode` and `linkNode` to create and link a new node into the list in first-in-first-out (FIFO) order. Hence, the policy framework is FIFO.

Assume that we want to change the policy of the component from FIFO to priority-based ordering of the nodes. Then, this can be achieved by weaving an appropriate aspect. Figure 1 shows the `listPriority` aspect, which consists of one pointcut `listInsertCall`, identifying `listInsert` as a join point in the component code (lines 2-3). When this join point is reached, the code in the `before` advice `listInsertCall` is executed. Hence, the aspect `listPriority` intercepts the operation (a method or a function call to) `listInsert`, and before the code in `listInsert` is executed, the advice, using the component mechanisms (`getNextNode`), determines the position of the node based on its priority (lines 5-14).

```
aspect listPriority{
1:
2: pointcut listInsertCall(List_Operands * op)=
3:   call("void listInsert(List_Operands*)")&&args(op);
4:
5: advice listInsertCall(op):
6:   void before(List_Operands * op){
7:     while
8:        the node position is not determined
9:      do
10:       node = getNextNode(node);
11:       /* determine position of op->node based
12:       on its priority and the priority of the
13:       node in the list*/
14:    }
15: }
```

Figure 1. The `listPriority` aspect

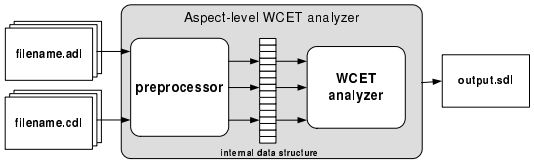

Figure 2. The overview of the automated aspect-level WCET analysis

3 Aspect-Level WCET Analysis

Figure 2 presents an overview of the automated aspect-level WCET analysis and its main constituents, including: (i) the input files that are aspect-level WCET specifications of aspects and components, (ii) the aspect-level WCET analyzer that consists of the preprocessor and the WCET analyzer, and (iii) the output files that represent the result of aspect-level WCET analysis.

The following sections give the description of aspect-level WCET specifications and the internals of the aspect-level WCET analyzer.

3.1 Aspect-Level WCET Specifications

Aspect-level WCET specifications are inputs to the aspect-level WCET analyzer, and they are defined by the run-time part of RTCOM [8].

Based on the description of the functional part of RTCOM (see section 2), the following can be observed: (i) aspect weaving does not change temporal behavior, i.e., WCETs, of mechanisms, and (ii) aspect weaving changes temporal behavior, i.e., WCETs, of operations, by changing the number of mechanisms that an operation uses. Therefore, if the WCETs of mechanisms are known and fixed, and the WCETs of the policy framework and aspects are given as a function of the mechanisms used, then the WCET of a component weaved with aspect(s) can be computed by calculating the impact of aspect weaving to WCETs of operations within the component (in terms of mechanism usage).

Thus, aspect-level WCET specifications that are fed into the tool consist of WCET specifications of the components, i.e., policy framework, and the aspects as a function of mechanism used. In the representation of WCET specifications we utilize the notion of symbolic WCET analy-

```
policy(noOfElements){
  operation{
    name listInsert;
    uses{
      createNode 1;
      linkNode   1;
    }
    intWcet 1ms;
  }
  operation{
    name listRemove;
    uses{
      getNextNode noOfElements;
      unlinkNode  1;
      deleteNode   1;
    }
    intWcet 4ms;
  }
....
}
```

```
mechanisms{
  mechanism{
    name createNode;
    wcet  5ms;
  }
  mechanism{
    name linkNode;
    wcet  4ms;
  }
  mechanism{
    name getNextNode;
    wcet  2ms;
  }

....
}
```

Figure 3. The WCET specification of the policy framework

```
aspect listPriority(noOfElements){
  advice{
    name listInsertCall;
    type before;
    changes{
      name listInsert;
        uses{
          getNextNode noOfElements;
        }
    }
    intWcet 4ms+0.4*noOfElements;
  }
....
}
```

Figure 4. The WCET specification of the `listPriority` aspect

sis [1]. Hence, we assume that the WCETs of the mechanisms are known and can be specified by symbolic expressions. Furthermore, the policy framework WCET specification consists of the mechanism usage of each operation in the framework, and the internal WCET specification. Similarly, the WCET specification of an aspect describes: (i) the type of each advice within the aspect, (ii) operations an advice modifies, (iii) the usage of mechanism by the advice while modifying the operations, and (iv) the internal WCET of the advice. The internal WCET of the operation/advice is the WCET of the code in the operation/advice excluding the WCETs of the mechanism calls. We assume that the internal WCETs are known and can be expressed in a form of a symbolic expression.

Figure 3 presents an instance of a WCET specification for the policy framework of the linked list component. Each operation in the framework is named and its internal WCET (`intWcet`) with the number of times it uses a particular mechanism are declared (see figure 3). The WCET specification for the aspect `listPriority` that changes the policy framework is shown in figure 4. Since the maximum number of elements in the linked list can vary, the WCET specifications are parameterized with the `noOfElements` parameter.

Aspect-level WCET specifications are currently implemented such that an aspect-level WCET specification of a component is contained in a file that has the extension *cdl* (component description language), while aspect-level WCET specifications of aspects have the extension *adl* (aspect description language). Our tool for aspect-level WCET analysis outputs a file with an extension *sdl* (system description language) that contains all the operations of the components in the configuration of the real-time system under analysis, and their respective resulting WCETs.

3.2 The Aspect WCET Analyzer

The aspect-level WCET analyzer consists of two parts: the preprocessor and the WCET analyzer. The preprocessing step is needed to extract the WCET information contained in the input files in a form usable by the WCET analyzer. Hence, the preprocessor takes aspect-level WCET specifications of aspects and components of a real-time system configuration as an input. It analyzes the WCET specifications given and produces data structures that store: (i) values of internal WCETs for operations and advices, (ii) values of WCETs of mechanisms, (iii) parameters existing in the symbolic expressions of operations, mechanisms, and advices, and (iv) dependency information, e.g., the mechanisms used by an operation, and advices modifying an operation. These data structures are internal to the aspect-level WCET analyzer and they are coupling the preprocessing and the analyzing part of the tool (see figure 2). The preprocessor is implemented using Bison [2] and Flex [4].

Since internal WCETs in the aspect-level WCET specifications are symbolic expressions, the values of these need to be determined, and the fist step is to obtain the values of parameters in the expressions. This is done by the aspect-level WCET analyzer in the step before invoking the WCET analyzer. The global function `checkParameters()` of the aspect-level WCET analyzer checks the data structures created in the preprocessing step detecting the parameters of operations, mechanisms, and advices (used in symbolic expressions), and prompts the human user for their values. The resulting parameterized data structures are then used by the WCET analyzer as an input to calculate the WCETs of all operations within the real-time system configuration under development.

The WCET analyzer performs the actual aspect-level WCET analysis. It does so based on the resulting parameterized data structures obtained in the preprocessing step of the analysis, and the algorithm for aspect-level WCET we developed previously [6, 7]. The algorithm provides a set of rules that define how to compute a new WCET of an operation weaved with aspects, depending on the type of an advice in the aspect. For example, for the advice of the type before modifying an operation, the new WCET of the operation would be computed using the value of an old WCET (i.e., WCET of an operation without aspects), and augmenting that value with the WCET of the before advice. This rule reflects the fact that the code of the before advice would, after aspect weaving, be inserted before the code of the operation. Similar rules exist for the advices of types after and around. Following the example of the linked list component, we can compute the WCET of the operation `listInsert` modified with an advice `listInsertCall` of the type before as follows.

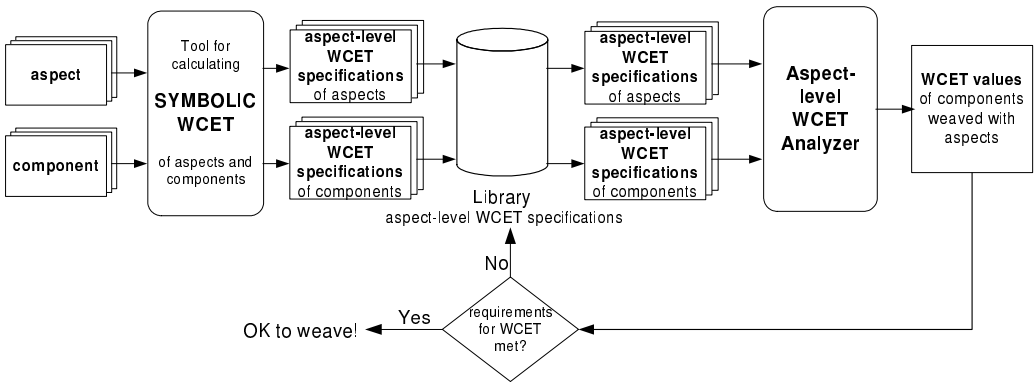

Figure 5. An overview of the aspect-level WCET analysis lifecycle

(aspectualized)listInsertWcet
= listInsertWcet(without aspects) + (before)listInsertCallWcet
= 14 + 2.4*noOfElements

where

listInsert(without aspects)
= intWcet + Σ mechanism*usage
= 1 + createNodeWcet*1 + linkNodeWcet*1
= 1 + 5*1 + 4*1 = 10

(before)listInsertWcet
= intWcet+ Σ mechanism*usage
= 4 + 0.4*noOfElements+ getNextNodeWcet*noOfElements
= 4+2.4*noOfElements

4 Limitations and Benefits

Ideally, the complete process of the aspect-level WCET analysis should have a lifecycle as presented in figure 5. The process starts with the implementation files of components and aspects, which are fed into a tool that performs the symbolic WCET analysis on the code, i.e., computes symbolic expressions for WCETs, and extracts these into aspect-level WCET specifications. These specifications are stored in a library and are used by the aspect-level WCET analyzer. Based on the output of the aspect-level WCET analyzer, i.e., computed values of the WCETs of a real-time system configuration consisting of components and aspects, we can determine the configuration eligibility for use in the underlying real-time environment with respect to WCET constraints of the environment. If a given configuration does not fulfill the requirements with respect to the WCET, the designer can choose another configuration, i.e., another set of aspect-level WCET specifications, until the WCET requirements are met, and the actual weaving can be performed.

Figure 5 also illustrates limitations of current automated aspect-level WCET analysis. The tool that computes WCETs in the form of symbolic expressions and extracts these to aspect-level WCET specifications should be an adaptation of the tool for symbolic WCET analysis to the aspect-level WCET analysis. The current implementation of the aspect-level WCET analyzer works only with aspect-level WCET specifications.

The current implementation, although given the limitations, provides benefits over traditional WCET analysis performed on weaved code since it enables calculations on WCET specifications, not on actual components and aspects. This way we reduce the overhead of performing the weaving and then WCET analysis for each potential configuration of aspects and components. Additionally, aspect-level WCET analysis can be generalized beyond symbolic WCET analysis if another approach (or a tool) for WCET analysis is used for determining the internal WCETs of operations, mechanisms, and advices.

References

[1] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *Proceedings of the 25th IFAC Workshop on Real-Time Programming*, Palma, Spain, May 2000.

[2] C. Donnely and R. Stallman. *Bison: The YACC-Compatible Parser Generator*, 2002. Available at: http://www.gnu.org/manual/bison-1.25/bison.html.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[4] V. Paxson. *Flex: A fast scanner generator*, 2002. Available at: http://www.gnu.org/manual/flex-2.5.4/flex.html.

[5] P. Puschner and A. Burns. A review of worst-case execution-time analysis (editorial). *Real-Time Systems*, 18(2/3):115–128, May 2000.

[6] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Integrating symbolic worst-case execution time analysis into aspect-oriented software development. OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development, November 2002.

[7] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Aspect-level worst-case execution time analysis of real-time systems compositioned using aspects and components. In *Proceedings of the 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming (WRTP'03)*, Poland, May 2003. Elsevier.

[8] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Towards aspectual component-based real-time systems development. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA'03)*. Springer-Verlag, February 2003.

Fully Automatic, Parametric Worst-Case Execution Time Analysis

Björn Lisper
Dept. of Computer Science and Engineering, Mälardalen University
P.O. Box 883, SE-721 23 Västerås, SWEDEN
bjorn.lisper@mdh.se

Abstract

Worst-Case Execution Time (WCET) analysis means to compute a safe upper bound to the execution time of a piece of code. Parametric *WCET analysis yields symbolic upper bounds: expressions that may contain parameters. These parameters may represent, for instance, values of input parameters to the program, or maximal iteration counts for loops. We describe a technique for fully automatic parametric WCET analysis, which is based on known mathematical methods: an abstract interpretation to calculate parametric constraints on program flow, a symbolic method to count integer points in polyhedra, and a symbolic ILP technique to solve the subsequent IPET calculation of WCET bound. The technique is capable of handling unstructured code, and it can find upper bounds to loop iteration counts automatically.*

1 Introduction

Parametric (or symbolic) WCET analysis derives a formula for the execution time, expressed in parameters of the program, rather than just a single number. The parameters can be either external, or internal like a symbolic upper bound to a loop count. A parametric WCET formula contains much more information than just a single WCET estimate, and it can be used for applications like online scheduling of tasks where parameters are unknown until runtime, or to find which parts of a code that has the strongest influence on the WCET. Thus, it is also potentially much more useful.

Previous approaches to parametric WCET have been based on the program timing schema model [4], or paths [1, 2]. These methods need manual annotations for constraints on loop counters and infeasible paths. An iterative method to compute parametric WCET bounds for simple loops has also been suggested [10].

Our method is potentially much more powerful than previous approaches. It can find loop bounds and infeasible path constraints automatically, and is capable of using such complex constraints in the calculation phase. Here we give a short account for the method followed by an explanatory example. A more detailed description is found in [8].

2 The Method

The analysis consists of a symbolic flow analysis, a symbolic summation, and a symbolic IPET calculation. See Fig. 1. The analysis uses a control flow graph model for programs, which means it works also for unstructured codes with jumps.

In the flow analysis, upper bounds for execution counts are derived in the form of symbolic expressions. If the program terminates, then the actual execution count for a program point equals the number of different states encountered in that point. Our method derives an upper approximation to the set of possible states in each program point, and then calculates the number of points in this set approximation. This yields an upper bound to the actual number of states and thus, under the assumption that the program terminates, the execution count. The set approximation may be parameterized: the counting is then performed symbolically. Program variables that affect the program flow, but do not change during the execution, are classified as parameters since varying their values will give rise to more states than are actually traversed during a single execution.

Set approximations can also be used to limit the number of states for which certain program paths can be taken. This can be used to find infeasible paths. It is also useful when analyzing program flows for pipelined processors, where different execution paths must be explored for possible pipeline overlap effects.

Sets of states in program points can be approximated from above by classical abstract interpretation [5]. Abstract interpretation is a framework that covers many possible ways to approximate sets of states. One abstract interpretation of particular interest is Halbwach's *polyhedral abstract interpretation* [6], which computes polyhedra as set approximations. Each program variable that may affect the program flow corresponds to a dimension in the polyhe-

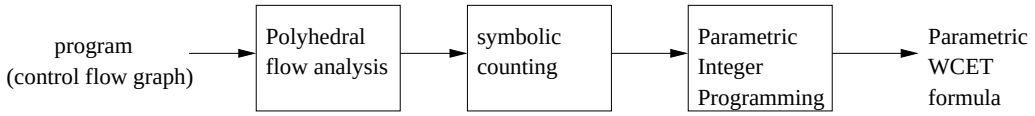

Figure 1. Structure of the method.

dral space. For instance, the set of states in a nested loop with loop indices i, j and upper (parametric) loop limits m, n will be bounded by a four-dimensional polyhedron in (i, j, m, n)-space.

Polyhedra are convex approximations. They describe linear loop index dependencies in nested loops, such as triangular loops, well but will overapproximate index sets for loops with non-unit strides. Other parametric set approximations are certainly possible, and will then provide different tradeoffs between precision and speed. Investigating these tradeoffs is an interesting topic of future research.

The next step is to count the numbers of points in polyhedra. Two techniques are known: through successive projection using known formulae for sums of powers of integers [9], and using *Ehrhart Polynomials* [3]. Both methods can compute parametric results. In our loop example above we would count points with respect to i and j, and return a sum that is parametric in m and n.

The final phase is the IPET calculation. It is done by *Parametric Integer Programming* (PIP) [7], which is a parametric extension of integer linear programming. This algorithm finds the optimum of a linear objective function over the parameterized set

$$\{\, \vec{x} \mid \vec{x} \geq \vec{0}, A\vec{x} + B\vec{z} + \vec{c} \geq \vec{0}, \vec{x} \text{ integral} \,\}$$

where $\vec{z}$ is a vector of parameters.

The parametric sums derived by the flow analysis are typically nonlinear in the parameters. However, each such sum can be replaced by a new symbolic parameter in the symbolic IPET calculation. The constraints will then become linear in these new parameters: PIP can compute an optimum expressed in them, and a subsequent substitution with the original sums followed by a simplification will yield the optimum expressed in the original parameters of the program. However, the new parameters often have direct interpretations, such as upper bounds to execution counts in program points, and can thus be interesting in their own right. An option is to leave them in the final answer. The resulting formula will then provide information how sensitive the WCET is for changes in loop counts and similar, which is interesting when tuning the code for best WCET.

The procedure outlined above is a fully automatic method for parametric WCET analysis that goes all the way from flow analysis to final WCET calculation. As far as we know, no other parametric method achieves this. The parametric calculation generalizes conventional IPET and can deal with advanced architectural features such as pipelining and caches in the same way.

3 An Example

Consider the CFG in Fig. 2. We assume each node n_i in the CFG has an execution time t_i. Each arc i has an execution count x_i. We first analyze it in order to extract upper bounds for the execution counts for all statements. It suffices to analyze the program w.r.t. the possible values of i and n, since they are the only variables affecting the program flow. We assume that B1 and B2 are basic block that update neither i nor n. There is one polyhedron S^i for each program point i, however $S^6 = S^4$ and $S^7 = S^5$ since B1 and B2 touch neither i nor n. We must also have $S^8 = S^3$. The system is solved by *fixed-point iteration* for the simplified system, which converges in nine iterations. The result is shown in Table 1. ($\top$ stands for the universal set: $S^0 = \top$ thus means that we allow any starting state in the analysis.)

We now calculate the number of points in the polyhedra. In the CFG in Fig. 2, the conditions depend on i and n only. n is never updated, and is thus considered a parameter. For each node, the number of elements in the abstract state on the preceding edge provides an upper bound on the execution count. The execution count for node n_0 is trivially one. By the method in [9], we obtain:

$$\begin{aligned}
\left|S^2\right| &= \infty \\
\left|S^3\right| &= \left|S^8\right| = \mathit{if}\ \ n > 0\ \ \mathit{then}\ \ n\ \ \mathit{else}\ \ 0 \\
\left|S^4\right| &= \mathit{if}\ \ n \geq 11\ \ \mathit{then}\ \ n - 11\ \ \mathit{else}\ \ 0 \\
\left|S^5\right| &= \mathit{if}\ \ n \geq 1\ \ \mathit{then} \\
&\qquad \mathit{if}\ \ n \geq 10\ \ \mathit{then}\ \ 10\ \ \mathit{else}\ \ n \\
&\quad \mathit{else}\ \ 0
\end{aligned}$$

This yields bounds $x_i \leq \left|S^i\right|$, where x_i is execution count for basic block n_i.

S^2 is overapproximated in the analysis. Therefore, there is no upper bound for x_2. This may seem awkward. However, there are *structural flow constraints* on the execution counts in addition to the upper bounds: for any node in the CFG, the sum of the execution counts for the input arcs must

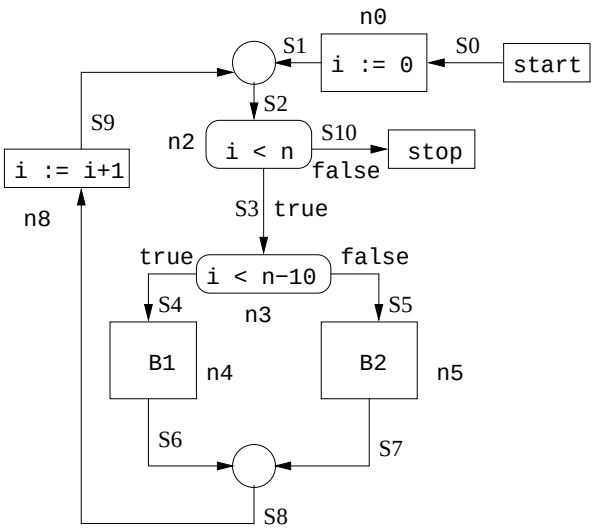

Figure 2. A simple flowchart program.

S^0	S^1	S^2	S^3	S^4	S^5	S^9	S^{10}
$\top$	$i = 0$	$i \geq 0$	$i \geq 0$ $i \leq n - 1$	$i \geq 0$ $i \leq n - 11$	$i \geq 0$ $n - 10 \leq i \leq n - 1$	$i \geq 1$ $i \leq n$	$i \geq 0$ $i \geq n$

Table 1. Result of abstract interpretation for example flowchart.

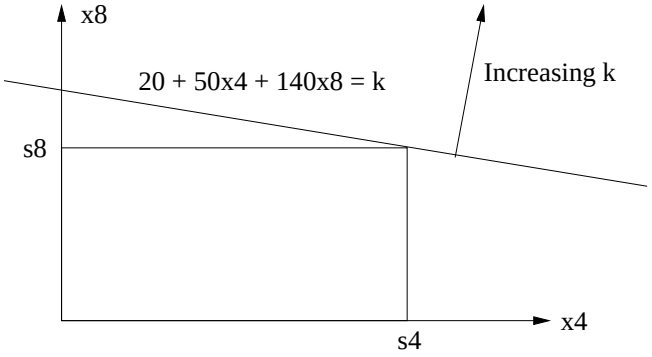

Figure 3. Maximizing the objective function in the IPET example.

equal the corresponding sum for the output arcs. These constraints will ensure finiteness of x_2, see below.

We finally perform a parametric IPET calculation. The WCET estimate is $\max(\sum_{i=0,2,3,4,5,8} x_i t_i)$. The execution count bounds derived from the abstract interpretation yield constraints $x_i \leq s_i, \quad i = 1, \ldots, 10$, where s_i is a symbolic parameter for $\left|S^i\right|$. We also have non-negativity constraints $x_i \geq 0, \quad i = 1, \ldots, 10$.

The structural flow constraints of the CFG can be used to reduce the number of variables down to two. Selecting x_4 and x_8 as basis yields the reduced problem $\max(t_0 + t_2 + x_4(t_4 - t_5) + x_8(t_2 + t_3 + t_5 + t_8))$ under the constraints $0 \leq x_4 \leq s_4$, $0 \leq x_8 \leq s_8$. Let us assume computation times $t_0 = 10$, $t_2 = 10$, $t_3 = 20$, $t_4 = 150$, $t_5 = 100$, and $t_8 = 10$. We then obtain the WCET estimate $50s_4 + 140s_8 + 20$ for $x_4 = s_4$, $x_8 = s_8$. See Fig. 3. With s_4, s_8 as the functions of n given by the polyhedral abstract interpretation we obtain (after simplification):

$$\begin{aligned} n \geq 11: &\quad 190n - 530 \\ 0 < n \leq 10: &\quad 140n + 20 \\ \text{otherwise}: &\quad 20 \end{aligned}$$

4 Conclusions and Further Research

We have described and exemplified a fully automatic method for parametric WCET calculation, that can deal with complex flow constraints and advanced architectural processor features. Future work involves a full implementation in order to evaluate the method with respect to accuracy and practical time complexity.

References

[1] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *Proc. 25th Workshop on Real-Time Programming*, Palma, Spain, May 2000.

[2] R. Chapman. Worst-case timing analysis via finding longest paths in SPARK Ada basic-path graphs. Technical Report YCS246, The British Aerospace Dependable Computing System Centre, Dept. of Computer Science, Univ. York, Oct. 1994.

[3] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *Proc. International Conference on Supercomputing*, pages 278–285, Philadelphia, PA, 1996. ACM.

[4] A. Colin and G. Bernat. Scope-tree: a program representation for symbolic worst-case execution time analysis. In *Proc. 14th Euromicro Conference on Real-Time Systems*, Vienna, June 2002.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[6] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, pages 84–97, 1978.

[7] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, Sept. 1988.

[8] B. Lisper. Fully automatic, parametric worst-case execution time analysis. MRTC report, Dept. of Computer Science and Engineering, Mälardalen University, Apr. 2003. `http://www.mrtc.mdh.se/publ.php3?id=0531`.

[9] W. Pugh. Counting solutions to Presburger Formulas: How and why. In *Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 121–134, Orlando, FL, June 1994. ACM.

[10] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric Timing Analysis. In J. Fenwick and C. Norris, editors, *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'2001)*, pages 88–93, Snowbird, Utah, June 2001.

ISSN 1404-3041
ISRN MDH-MRTC-116/2003-1-SE