OTAWA: An Open Toolbox for Adaptive WCET Analysis

Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat

Institut de Recherche en Informatique de Toulouse
University of Toulouse, France
{ballabri,casse,rochange,sainrat}@irit.fr

Abstract. The analysis of worst-case execution times has become mandatory in the design of hard real-time systems: it is absolutely necessary to know an upper bound of the execution time of each task to determine a task schedule that insures that deadlines will all be met. The OTAWA toolbox presented in this paper has been designed to host algorithms resulting from research in the domain of WCET analysis so that they can be combined to compute tight WCET estimates. It features an abstraction layer that decouples the analyses from the target hardware and from the instruction set architecture, as well as a set of functionalities that facilitate the implementation of new approaches.

Keywords: Real-time, Worst-Case Execution Time, static analysis.

1 Introduction

In hard real-time systems, critical tasks must meet strict deadlines. Missing one such deadline may have dramatic consequences, either in terms of human lives or economical and environmental issues. To avoid this, special attention is paid on determining a safe schedule for tasks.

Real-time task scheduling has been and is still a hot research topic, but it relies on the knowledge of the Worst-Case Execution Time (WCET) of the critical tasks. While this time is generally considered as known, the techniques required to compute it are not straightforward and cannot always support complex target architectures. As a result, research on WCET analysis is also very active.

The general approach to WCET analysis includes three steps: (a) flow analysis mainly consists in identifying (in)feasible paths and bounding loops; (b) low-level analysis aims at determining the global effects of the target architecture on execution times and at deriving the worst-case execution times of code snippets; (c) finally, the results of the flow and low-level analyses are combined to derive the overall WCET. For each of these three steps, several techniques have been investigated in the literature. Besides, several tools have been developed to implement these results. Most of them are still research prototypes but a few are now commercialized.

The design of the OTAWA toolbox presented in this paper started in 2004, at a time where very few tools were publicly available. Our main motivation was

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 35–46, 2010.

to provide an open framework that could be used by researchers to implement their analyses and to combine them to already implemented ones. The expected characteristics of this framework were versatility (ability to support various target hardware configurations as well as various instruction sets) and modularity (support to facilitate the implementation of new analyses). `OTAWA` is not a tool but a *toolbox*: this means that it comes as a C++ library that can be used to develop WCET analysis tools. However, it includes a number of algorithms which make this task really easy (example tools are distributed with the library).

The paper is organized as follows. Section 2 reviews the state of the art on WCET analysis and gives a short overview of existing tools. The `OTAWA` toolbox is presented in Section 3 and Section 4 illustrates its capacities through a short report of how it has been used in several projects. Concluding remarks are given in Section 5.

2 Worst-Case Execution Time Analysis

2.1 Different Approaches to WCET Analysis

The execution time of a program on a given processor depends on the input data and on the initial state of the hardware (mainly on the contents of the memories). To evaluate its worst-case execution time (WCET), it is necessary to consider all the possible input values and all the possible initial hardware states. In most of the cases, this is not feasible: (a) determining input data sets that cover all the possible execution paths is hard, especially with floating point inputs[1]; (b) even if this was possible, the number of paths to explore would be too large and the time required to measure all of them would be prohibitive; (c) it is not always possible to initialize the hardware state prior to measurements so as to investigate all the possible states. Because it is not feasible to measure all the possible paths considering all the possible initial hardware states, it is now commonly admitted that WCET analysis must break down the execution paths into code snippets (generally basic blocks) so that time measurement or estimation is done on these units and the overall WCET is computed from the unit times.

One way to classify approaches to WCET analysis is to consider the way they determine the worst-case execution time of code snippets. Some of them are based on measurements performed preferably on the real target hardware, but possibly on a simulator [4] while other ones use a model of the hardware to compute these times [16][19][27].

Another approach is to focus on the way the unit times are combined to derive the complete WCET of the task. Some approaches need the program under analysis to be expressed in the form of an Abstract Syntax Tree (AST) [20][10] and compute the WCET using analytical formulae related to the algorithmic-level statements found in the code. These approaches do not fit well with codes that

[1] An exception is the case of programs written under the single-path programming paradigm [24] but this still remains marginal.

have been optimized at compile time. Other solutions consider the Control Flow Graph (CFG) built from the object code and use path-based calculation [15] or formulate the search of the longest execution path as an Integer Linear Program [18].

Besides to these contributions that are at the root of research on WCET analysis, a lot of work has been done to design algorithms able to derive execution times of basic blocks considering various and more and more complex hardware features. These solutions will be shortly reviewed below.

2.2 Analysis of the Behavior of Hardware Mechanisms

WCET estimates are expected to be safe since the determination of a schedule of tasks that insures that hard deadlines will be met is based on them. So far, no way to obtain guaranteed WCET upper bounds without getting into the hardware architecture details has been found. Instead, existing methods compute execution time with cycle-accuracy. As a result, a number of papers have been published to show how to take into account the behavior of specific variants of hardware mechanisms in WCET analysis. In addition, it is often required for WCET estimates to be tight so as to avoid oversizing the hardware to insure schedulability. For this reason, research work also focuses on reducing the WCET overestimation.

Many hardware schemes now present in the processors used in embedded systems are supported by WCET analysis techniques:

- the processor pipeline has been studied for many years from the first scalar architectures [20] to superscalar configurations [21][30] and pipelines featuring dynamic instruction scheduling [16][19][27];
- the analysability of cache memories has also been largely investigated. Several techniques have been proposed to analyze the behavior of instructions caches [1][15], data caches [29][28], and multi-level memory hierarchies [14];
- dynamic branch predictors have retained attention too [9][3][6];
- other mechanisms like the memory row buffer have been considered [5] to model specific hardware.

A recent paper [31] reviews and compares the WCET tools that have been designed by members of the ARTIST European Network of Excellence. Some of them are research prototypes and other ones are commercial tools. Since most of these tools were not publicly available or not mature enough at the time we started research analysis (2004), designing our own software was a necessity. However we always kept in mind the need of making it open so that it can host any research result on WCET analysis.

3 The OTAWA Toolbox

3.1 Objectives

Developments for the OTAWA toolbox started in 2004 with the objective of designing a framework that could integrate various kinds of methods related to WCET

analysis and support a large range of target architectures. Such a framework is expected to:

- capitalize results of research on WCET estimation
- make it possible to compare two different techniques that address the same problem (e.g. instruction cache analysis) though experiments carried out under identical conditions
- support the investigation of new algorithms by allowing experiments that combine them to state-of-the-art techniques. This avoids having to make either overwhelmingly optimistic or pessimistic assumptions for features that are out of the scope of the current work.

To achieve these goals, `OTAWA` exhibits two key features. First, both the target hardware and the code under study are processed through an abstract layer: this way, analyses are designed independently from the hardware and from the instruction set and can then be used for any architecture. Second, the toolbox provides all the facilities to implement new analyses with limited effort

The overall structure of the `OTAWA` toolset is shown in Figure 1. It includes a number of components that provide two classes of facilities: some of them maintain information about the application under analysis and about the target hardware architecture, as detailed in Section 3.2; other ones manipulate and enrich this information towards an estimation of the worst-case execution time (see Section 3.3).

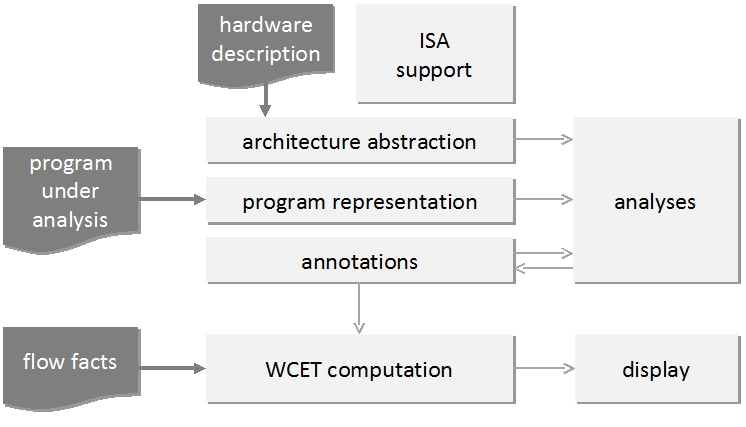

Fig. 1. Overview of the `OTAWA` framework

3.2 Abstract Layers

As explained above, `OTAWA` is organized in independent layers that provide an abstraction of the target hardware and associated Instruction Set Architecture (ISA) as well as a representation of the binary (`.elf`) code under analysis.

ISA Support. This layer hides the details of the target ISA to the rest of the toolset and provides a unique interface to retrieve the information needed for the analyses related to WCET estimation. This way, most of the components of the toolbox can be designed independently of the instruction set. The layer features support for various ISAs implemented through a set of plugins. Available plugins have been generated using our `GLISS` tool [25] that will be presented in Section 3.4 and support the following ISAs: PowerPC, ARM, TriCore, HCS12, Sparc.

Abstraction of the Hardware Architecture. Precise estimation of the worst-case execution time of instruction sequences requires a detailed knowledge of hardware parameters:

- processor: width and length of the pipeline, number of functional units and their latencies, binding of instruction categories to the functional units, specification of the instruction queues (location, capacity)
- caches: capacity, line width, organization (number of sets, number of ways), replacement policy, write policy, etc.
- memories: as part of the French MORE project, we have implemented support for various memory components (DRAM, scratchpad), each exhibiting a specific access latencies and buffering policies.

Support for additional components or new architectural features can be added by the developer with limited effort. From the user side, the hardware parameters can be specified with an XML file. Since investigating formats that could describe any kind of processor was out of the scope of our research, we decided to limit our XML format to the minimum needed to specify standard architectures. Taking into account real-life processors still requires specific developments by the user, but this is made easy by the classes and functionalities available through the `OTAWA` library.

Properties. One key facility in the `OTAWA` toolbox is the possibility of defining *properties* that can be used to annotate any kind of object defined in the library (e.g. an instruction) in a very convenient way.

These annotations are first used to build the CFG (or AST) representation of the program. The set of instructions in the program is retrieved through the ISA support layer, and a Control Flow Graph (CFG) can be built for each function. It consists of basic blocks, i.e. sequences of instructions with single entry and exit points, and of edges that express the possible flow from one basic block to another one. In the case of indirect branch instructions (e.g. implementation of a `switch` statement), it may happen that `OTAWA` requires help from the user who must specify the possible branch targets so that appropriate edges can be included in the CFG. The program under analysis is then represented within `OTAWA` as a set of interconnected CFGs.

The properties are also useful to store the results produced by the different analyses (e.g. the instruction cache analyzer annotates instructions to indicate

whether they will hit or miss in the cache and the pipeline analyzer takes this information into account to determine the instruction fetch latencies).

3.3 Analyses

As explained before, the determination of the WCET of a task involves a number of analyses that must be performed in an appropriate order. In `OTAWA`, such analyses are implemented as *Code Processors*: a code processor is a function that uses available annotations and produces new annotations.

The distributed `OTAWA` library includes a set of analyses. Some are related to flow analysis and produce information that are useful to other analyses. Examples in this category are a CFG builder, a CFG virtualizer that produces a virtual CFG with the functions inlined (this allows call-contextual analyses) or a loop analyzer that determines loop headers and dominance relationships which are used e.g. by the instruction cache analysis. Other code processors provide facilities to use state-of-the-art analysis techniques, like Abstract Interpretation [11]. Finally, a number of analyses compute data that can be combined to determine the final WCET. They include the pass that generates the integer linear program defined by the IPET method [18] but also several analyses that take into account the features of the target hardware. At this state, `OTAWA` provides the code processors needed to analyze:

- instruction caches: the approach is based on Abstract Interpretation and is close to Ferdinand's method [13]. However, our implementation includes the improvements proposed in [2] to reduce complexity while performing an accurate persistence analysis in the case of loop nests.
- pipelines: the method implemented in `OTAWA` to compute the worst-case execution times of basic blocks is original and has been presented in [27]. While the `aiT` tool uses abstract interpretation to build the set of possible processor states in input/output of each basic block, which is likely to be time consuming, our method is based on execution graphs and involves an analytical calculus. This is similar to the approach used in the `Chronos` tool [19] except for we consider a parametric view of the processor state at the entry of each basic block when they consider the worst-case processor state. Experiments have shown that our algorithm provides more accurate results with comparable computation times.
- dynamic branch predictors: we have developed an algorithm to take into account the behavior of a dynamic branch predictor [6]. It formulates the question of determining whether a branch will be always/sometimes/never well predicted as an integer linear program. However, experiments have shown that the complexity of this program is significant.

Handling Analyses Dependencies. When considering the full process of computing the WCET of a task, it appears that the involved analyses are interdependent through a producer-consumer scheme: annotations produced by one analysis are required by other passes. To respect these dependencies, the analyses

must be invoked in an appropriate order as illustrated on the example shown in Figure 2. OTAWA provides a means to automatically handle dependencies: *features* can be used to express the properties that are required or produced by a code processor. For each defined feature, a default code processor must be specified: it will be executed when the feature is required if it has still not been produced. As a result, a simple WCET tool can be designed by only invoking a final WCET computation code processor: default code processors will be automatically called to generate the missing data.

```
Object code loader
Flow facts loader
CFG builder
Loop analyzer
Instruction cache analyzer
Basic block timing analyzer
Structural constraints builder (IPET)
Cache-related constraints builder (IPET)
WCET computation (call to ILP solver)
```

Fig. 2. An example scenario for WCET estimation

Display Facilities. OTAWA provides facilities to dump out program representations and annotations produced by the different analysis. These outputs are as useful to the WCET tool developer, as well as to the real-time application developer. The former can use them to debug or to tune new analyses while the latter can get a better understanding of the program behavior and locate program regions that break the deadlines.

More recently, OTAWA has been integrated in the Eclipse programming environment. The OTAWA plugin allows benefiting from the Eclipse graphical user interface to improve OTAWA use ergonomics and helps in achieving a better integration in the development cycle. The developer can seamlessly develop an application, compile it and get the WCET of some program parts. From the graphical program representation, time-faulty program regions can be also easily identified and fixed. Figures 3 and 4 show how the source code and the CFG of the program under analysis can be displayed and colored to highlight the critical paths (the darker code regions are those that are responsible for the larger part of the total WCET).

Cycle-Level Simulator. In addition to the WCET-related analyses, the OTAWA toolbox also includes a code processor that builds and runs a cycle-level simulator. This simulator is generated on top of the SystemC library and matches the XML description of the target architecture. It makes it possible to observe the execution times related to given input values and then to get an empirical insight into the range of WCET overestimation.

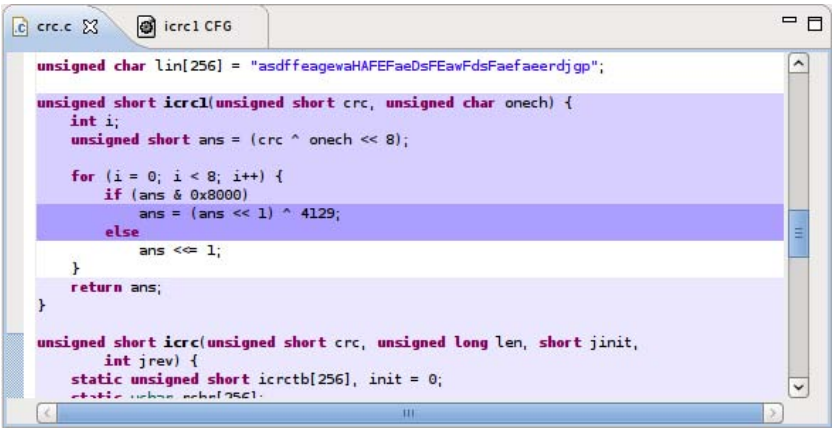

Fig. 3. Colored display of source code

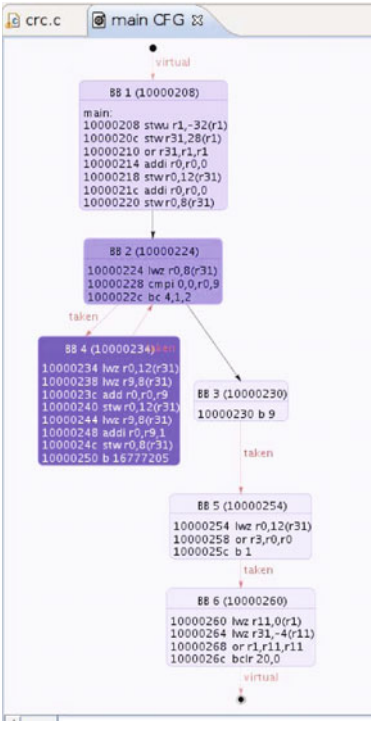

Fig. 4. Colored display of a CFG

3.4 Complementary Tools

In OTAWA, the abstraction of the Instruction Set Architecture is provided by a loader module in charge of loading the object code of the application under study and of building a representation of this code that is independent of the ISA. To design these loader modules, we use instruction set simulators generated by our GLISS tool from ISA specifications expressed in the SimNML language [25]. An instruction set simulator is a library that provides functions to decode, disassemble and emulate instructions (emulation is performed on a logic state of the memory and registers).

Another complementary tool is oRange that is used to determine flow facts and more particularly loop bounds [12]. It works on the C source code of the application and uses flow analysis and abstract interpretation techniques to derive contextual loop bounds (a loop in a function can have different bounds for different calls). Debug information inserted by the compiler in the object code are used by OTAWA to assign source-level loop bounds to the corresponding machine-level instructions. In the future, oRange will be integrated into the OTAWA framework.

Finally, the integer linear programs generated as part of the IPET method [18] are solved by invoking the lp_solve tool [32].

4 Examples of Use

The OTAWA toolbox has been involved in several projects. In this Section, we give insight into how it has been successfully used to fulfill a variety of objectives.

The goals of the MasCotTE project[2] were to investigate the possibility for WCET analysis techniques to consider off-the-shelves processors with different levels of complexity. The basic functionalities of OTAWA coupled to our GLISS tool made it possible to model two processors used in automotive applications: the Freescale 16-bit Star12X and the high-performance Freescale MPC5554. This work has been reported in [8].

In the MERASA project[3], the objective is to design a multicore processor able to execute mixed-critical workloads while offering timing predictability to hard real-time tasks. Two approaches to WCET analysis are considered: measurement-based techniques, with the RapiTime tool and static analysis, with the OTAWA toolset. Abstractions of the MERASA multicore and support for the TriCore ISA have been developed, as well as models for the specific components of the architecture: dynamic instruction scratchpad [22], data scratchpad used for stack data, predictable bus and memory controller [23]. In addition, OTAWA has been used to analyze the WCET of a parallel 3D multigrid solver and used

[2] **Ma**îtris**e** et **Con**trôle des **T**emps d'**E**xécution (*Controlling Execution Times*). This research has been partially funded by the French National Research Agency (ANR).

[3] **M**ulti-Core **E**xecution of Hard **R**eal-Time **A**pplications **S**upporting **A**nalysability. This research is partially funded by the European Community's Seventh Framework Programme under Grant Agreement No. 216415.

as a pilot study for the project. The contribution of this work, reported in [26], is in (a) the analysis of the synchronizations between parallel threads and, (b) the tight evaluation of the synchronization-related waiting times, based on the ability of OTAWA to analyze the WCET of specified partial execution paths.

The MORE project[4] aims at providing a framework for the investigation of code transformations used to improve several criteria like code size, energy consumption or worst-case execution time. This framework is being designed using the OTAWA toolbox and includes evaluation tools (for worst-case execution times or energy consumption), transformation tools (emulators for code compression and data placement in memories, a plugin to control GCC optimizations through the GCC-ICI interface [17]) and an iterative transformation engine that explores the transformation space to determine the best combination of transformations with respect to the requirements. The original usage of OTAWA lies in the design of transformation emulators that use the annotation system. For example, code compression is implemented the following way: profiling data is collected using the cycle-level simulator available in OTAWA; the instructions that should be compressed are determined; then each instruction is annotated with the address where it would be found in the compressed code; finally, the WCET is analyzed considering the addresses in the compressed code (the addresses are used for the instruction cache analysis). This way, it is possible to estimate the impact of compression algorithms without having to generate the real compressed code neither the corresponding code loader for WCET analysis. Further details can be found in [7].

5 Conclusion

Safe scheduling of critical tasks in hard real-time systems requires having knowledge of their Worst-Case Execution Times. WCET analysis has been a research topic for the last fifteen years, covering various domains from flow analysis (typically determining loop bounds and infeasible paths) to low-level analysis (computing the execution times of basic blocks taking into account the architecture of the target hardware). Several research groups have designed their own WCET tool to support their research and all these tools are complementary but redundant to a certain extent. It would probably be fruitful to promote their interoperability and we believe that the OTAWA toolbox presented in this paper exhibits interesting features for this purpose. First of all, it provides an abstract interface to the program to be analyzed and to the target hardware, which make it possible to develop platform-independent analyses. Second, it features a number of facilities that allow fast implementation of new analyses with a powerful means to capitalize their results towards the final WCET computation. It also provides an efficient mechanism to handle the dependencies between analyses.

[4] **M**ulti-criteria **O**ptimizations for **R**eal-time **E**mbedded systems. This research is partially funded by the French National Research Agency (ANR) under Grant Agreement ANR-06-ARFU-002.

OTAWA has been successfully used in several projects and has proved its flexibility and openness, as well as the efficiency of its key features.

The OTAWA toolbox is available under the LGPL license from www.otawa.fr.

References

1. Alt, M., Ferdinand, C., Martin, F., Wilhelm, R.: Cache Behavior Prediction by Abstract Interpretation. In: Static Analysis Symposium (1996)
2. Ballabriga, C., Cassé, H.: Improving the First-Miss Computation in Set-Associative Instruction Caches. In: Euromicro Conf. on Real-Time Systems (2008)
3. Bate, I., Reutemann, R.: Efficient Integration of Bimodal Branch Prediction and Pipeline Analysis. In: IEEE Int'l Conference on Embedded and Real-Time Computing Systems and Applications (2005)
4. Bernat, G., Colin, A., Petters, S.: pWCET a Toolset for automatic Worst-Case Execution Time Analysis of Real-Time Embedded Programs. In: 3rd Intl Workshop on WCET Analysis (2003)
5. Bourgade, R., Ballabriga, C., Cassé, H., Rochange, C., Sainrat, P.: Accurate analysis of memory latencies for WCET estimation. In: Int'l Conference on Real-Time and Network Systems (2008)
6. Burguière, C., Rochange, C.: On the Complexity of Modelling Dynamic Branch Predictors when Computing Worst-Case Execution Times. In: ERCIM/DECOS Workshop on Dependable Embedded Systems (2007)
7. Cassé, H., Heydemann, K., Ozaktas, H., Ponroy, J., Rochange, C., Zendra, O.: A Framework to Experiment Optimizations for Real-Time and Embedded Software. In: Int'l Conf. on Embedded Real Time Software and Systems (2010)
8. Cassé, H., Sainrat, P., Ballabriga, C., De Michiel, M.: Experimentation of WCET Computation on Both Ends of Automotive Processor Range. In: Workshop on Critical Automotive Applications: Robustness and Safety (2010)
9. Colin, A., Puaut, I.: Worst Case Execution Time Analysis for a Processor with Branch Prediction. Real-Time Systems Journal 18(2) (2000)
10. Colin, A., Puaut, I.: A Modular and Retargetable Framework for Tree-based WCET Analysis. In: Euromicro Conference on Real-Time Systems (2001)
11. Cousot, P., Cousot, R.: Abstract Interpretation - A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: ACM Symp. on Principles of Programming Languages (1977)
12. De Michiel, M., Bonenfant, A., Cassé, H., Sainrat, P.: Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In: IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (2008)
13. Ferdinand, C., Martin, F., Wilhelm, R.: Applying compiler techniques to cache behavior prediction. In: ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems (1997)
14. Hardy, D., Puaut, I.: WCET analysis of multi-level non-inclusive set-associative instruction caches. In: IEEE Real-Time Systems Symposium (2008)
15. Healy, C., Arnold, R., Mueller, F., Whalley, D., Harmon, M.: Bounding pipeline and instruction cache performance. IEEE Transactions on Computers 48(1) (1999)
16. Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R.: The Influence of Processor Architecture on the Design and the Results of WCET Tools. Proceedings of the IEEE 91(7) (2003)

17. Huang, Y., Peng, L., Wu, C., Kashnikov, Y., Renneke, J., Fursin, G.: Transforming GCC into a research-friendly environment - plugins for optimization tuning and reordering, function cloning and program instrumentation. In: 2nd Int'l Workshop on GCC Research Opportunities (2010)
18. Li, Y.-T., Malik, S.: Performance Analysis of Embedded Software using Implicit Path Enumeration. In: Workshop on Languages, Compilers, and Tools for Real-time Systems (1995)
19. Li, X., Roychoudhury, A., Mitra, T.: Modeling out-of-order processors for WCET analysis. Real-Time Systems Journal 34(3) (2006)
20. Lim, S.-S., Bae, Y.H., Jang, G.T., Rhee, B.-D., Min, S.L., Park, C.Y., Shin, H., Park, K., Kim, C.S.: An accurate worst case timing analysis technique for RISC processors. In: IEEE Real-Time Systems Symposium (1994)
21. Lim, S.-S., Kim, J., Min, S.-L.: A worst case timing analysis technique for optimized programs. In: International Conference on Real-Time Computing Systems and Applications (1998)
22. Metzlaff, S., Uhrig, S., Mische, J., Ungerer, T.: Predictable Dynamic Instruction Scratchpad for Simultaneous Multithreaded Processors. In: MEDEA Workshop (2008)
23. Paolieri, M., Quiones, E., Cazorla, F., Bernat, G., Valero, M.: Hardware Support for WCET Analysis of HRT Multicore Systems. In: Int'l Symposium on Computer Architecture (2009)
24. Puschner, P., Burns, A.: Writing temporally predictable code. In: 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (2002)
25. Ratsiambahotra, T., Cassé, H., Sainrat, P.: A Versatile Generator of Instruction Set Simulators and Disassemblers. In: Int'l Symp. on Performance Evaluation of Computer and Telecommunication Systems (2009)
26. Rochange, C., Bonenfant, A., Sainrat, P., Gerdes, M., Wolf, J., Ungerer, T., Petrov, Z., Mikulu, F.: WCET Analysis of a Parallel 3G Multigrid Solver Executed on the MERASA Multi-core. In: Int'l Workshop on Worst-Case Execution Time Analysis (2010)
27. Rochange, C., Sainrat, P.: A Context-Parameterized Model for Static Analysis of Execution Times. Transactions on High-Performance Embedded Architectures and Compilers 2(3) (2007)
28. Sen, R., Srikant, Y.N.: WCET estimation for executables in the presence of data caches. In: 7th International Conference on Embedded Software (2007)
29. Staschulat, J., Ernst, R.: Worst case timing analysis of input dependent data cache behavior. In: Euromicro Conference on Real-Time Systems (2006)
30. Theiling, H., Ferdinand, C.: Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In: IEEE Real-Time Systems Symposium (1998)
31. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström: The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. ACM Transactions on Embedded Computing Systems 7(3) (2008)
32. Open source (Mixed-Integer) Linear Programming system, `http://lpsolve.sourceforge.net/5.5/`