OTAWA, a Framework for Experimenting WCET Computations

Hugues Cassé, Pascal Sainrat

OTAWA, a Framework for Experimenting WCET Computations

H. Cassé, P. Sainrat

IRIT-UPS, 118 route de Narbonne 31062 Toulouse, France.

Abstract: In this article, we present OTAWA, a framework for computing the Worst Case Execution Time of a program. From its design, it provides an extensible and open architecture whose objective is the implementation of existing and future static analyses for WCET computation. Inspired by existing generic tools, it is based on an architecture abstraction layers where hooked annotations store specific analyses information. Computing the WCET is viewed as performing a chain of analyses that use and produce annotations until getting the WCET evaluation. Finally, the efficiency of the framework, in term of development productivity, is evaluated by two case studies that show some pitfalls that we are currently fixing but also the success of the approach.

Keywords: IPET, WCET, framework, real-time, ETS.

1. Introduction

1.1 Motivation

A large class of embedded software needs to proceed in real-time context. As they may run in critical devices like transportation, it must be proved that they meet hard time constraints. One method uses the computation of Worst Case Execution Time, WCET in short, that evaluates the maximum execution time of a program whatever the inputs and checks that real-time constraints are satisfied.

Through the various ways for computing the WCET, the approach based on static analysis allows getting accurate and safe estimations. Usually, the WCET computation by static analysis requires modelling not only the program but also the host architecture. The classical techniques handle well the current embedded programs and the currently used simple processors although the growing size of the software induced by more and more intelligent devices will soon require more computation power and modern processors.

Nevertheless, modern processors features like multiple-issue pipeline, trace caches, branch prediction, simultaneous multithreading, are still not well modelled and stay hard to support in WCET computation. Likely, the growing size and complexity of embedded software may make current techniques inefficient or too slow. These issues require either the improvement of the existing methods, or the development of new approaches.

To experiment methods for processing these issues, we have developed a tool named OTAWA[1]. Unlike other existing tools designed for a specific analysis algorithm, it provides an extensible and open framework with the ability to support many different approaches. We intend to use it as a sandbox where new or improved algorithms may be easily developed.

1.2 Existing Tools

First, we looked for a tool supporting the implementation and the experimentation of new algorithms and we examined existing tools. To fill our needs, the tool had to exhibit extensibility and openness properties.

The open source domain provides very few tools for WCET computation. A very early tool is Cinderella [1] but it seems to be no more developed since several years. More recently, the Heptane tool [2] is delivered by IRISA, France. It provides many neat features like multi-target support, opened sources, modular structure and visualization facilities. However, we have rejected it because it has been developed for the Extended Timing Schema [3] approach. Integrating or developing different methods is difficult. Heptane requires program sources for building the Abstract Syntax Trees and thus prevents the use of optimizations that blurs the match between the binary and the AST. This last issue prevents from processing off-the-shelf components whose sources are rarely available.

From the commercialized domain, we have mainly two tools, Bound-T and aiT. Bound-T [4] is a complete multi-target solution for WCET computation developed by Tidorum Ltd, Finland and supported by the European Space Agency. It is based on the Implicit Path Enumeration Technique [1] method and includes some semi-automatic facilities for computing loop bounds. The instruction model, far away from the machine code, is so original that it should be difficult to support multi-issues processors. Although it provides some facilities for external modules, it seems to be too much closed and specialized.

AbsInt Gmbh, a German corporation, provides a tool called aiT [5] that is used by several device manufacturers in real-time embedded applications. It is based on two main techniques. The abstract interpretation is used for modelling the cache and the pipeline behaviour and uses the PAG generator [6]. On the other side, Integer Linear Programming is used for handling the program control flow. In addition, aiT is delivered with many tools in order to visualize WCET and hardware states wrapped around the Control Flow Graph. Although aiT has been used by several projects, its internal architecture is not well known, possibly due to its commercialization success. Consequently, it seems to be too closed and too specialized for fitting our needs.

[1] OTAWA stands for Open Tool for Adaptative WCET Analysis.

As a result, we have no choice except developing OTAWA [7], a new tool providing the required properties. Yet, this project is also supported by our skills in architecture modelling and by our experience with generic tools. Thus, such a project seemed to us reachable.

1.3 Outline

After this introduction, the second section explains how successful generic tools, Salto and SUIF, has inspired the design of OTAWA. In the third section, we expose the details of the framework architecture and describe the overall approach of writing analyses in this framework. The fourth part attempts to appraise the success rate of the tool and presents two case studies which allow us to experiment the development inside the OTAWA. Finally, the conclusion gives insight in future developments.

2. Framework Sources

OTAWA development has been started for filling internal needs but the resolution of enlarging the framework application domain was motivated by a significant experience in generic tools from the architecture and compilation domains. Two successful generic tools have guided the design of OTAWA.

2.1 SALTO Low-level Optimizer

SALTO [8] is the ultimate result from research from the previous decade on modelling RISC and superscalar processors and on providing architecture-independent low-level optimizers.

Actually, SALTO is a C++ library providing an abstract representation of the assembler and of its execution by the processor. The specialisation of the abstract code is achieved using a collection of architecture-aware back-ends: several ones are provided in the basic distribution but more may be implement by the users. The back-end contains information about the assembly syntax, the programming model and the execution model of the processor. In addition, a configuration file allows tuning some components of the processor (cache, pipeline capacity, functional unit count and so on).

Such a tool proves that it is possible to support different architectures using an abstract representation and has provided a powerful model for achieving this task. This feature is particularly important in the field of embedded system where a large panel of digital equipment is used.

Nonetheless, several issues prevented a straightforward use of Salto in our tool. First, Salto implementation only supports programs in textual assembly sources while most of embedded program to process are only provided as a binary: an additional pass translating back in textual form would consume time and possibly decrease the reliability of the computation. Another drawback comes from the application field of Salto: it was designed to process low-level optimisations with a partial and rough time analysis while the WCET computation requires an accurate time estimation of the activity of all processor components. As a last resort, Salto does not provide any facility for performing processor simulation, a feature that is used in many WCET computation methods.

In the OTAWA framework, we preferred to develop our own architecture abstraction layer based on the Salto architecture model. This layer is built onto back-ends generated by GLISS [9], a tool providing automatic facilities for describing a programming model, decoding binaries and generating functional simulators.

2.2 SUIF Compiler

The SUIF compiler [10] also brought significant features to OTAWA. It was designed as a modular compilation chain for the experimentation of new optimization algorithms. Formerly using a single back-end generating MIPS assembly, it has been extended to support any processor in MachSUIF [11]. Delivered in the nineties, SUIF is a perfect example of a successful generic and experimental tool.

The first feature borrowed by OTAWA concerns the annotation system. The SUIF annotations are pieces of information that accept an identifier and that are hooked to any entity in the program representation. They are handled by any compilation phase to pass data to subsequent phases. In the SUIF implementation, in order to transmit the program between phases, a standard form is provided in order to serialize and unserialize the program representation and the annotations. This facilitates the addition of new phases that can store or share their own data. Nonetheless, SUIF annotations does not fit well with the transformations of the program representation required by the compilation: some annotations are lost or are moved on inadequate entities. Yet, OTAWA implementation of annotations is not hurt by such a limitation because it processes the program in a read-only fashion.

The next feature adapted to OTAWA concerns the two-level representation of the program. The Low-SUIF representation contains the machine instructions, possibly linked as an expression tree while the High-SUIF represents the program by its control statement as found in the sources. Both representations are living in parallel and are linked together. As a result, the low-level and high-level optimizations may be applied on the same program representation. Thus, the compilation process is equivalent to reduce the high-level representation until getting only a linear Low-SUIF sequence of instructions. The OTAWA adaptation has improved this feature: it supports many different high-level program representations on the same code because it builds them from the annotations.

The last feature borrowed by OTAWA is the possibility of chaining optimization phases. In SUIF, each optimization is an executable program that reads the serialized program representation from the previous phase and produces a new serialized program representation. Using the annotations, optimizations

can pass specific pieces of information along the compilation process. The whole compilation may be performed using either a shell script or an executable launching each phase. Building or modifying compilation chains is easier thanks to this possibility. This way, adding a phase is very simple as it only involves modifying the script.

This feature has been included in OTAWA. For the sake of efficiency, our analysis phases live in the same executable and does not need the serialization of the program representation. Yet, the uniformity of the analysis interface allows a more flexible composition and supports the use of plug-ins.

The features from Salto and SUIF, listed in this section, have driven the design of OTAWA. The result is a framework that should support easy extension and fast development of WCET analyses.

3. Framework Architecture

This section presents the architecture of the OTAWA framework.

2.1 Framework Overview

Figure 1 shows an overview of OTAWA as a layer stack.

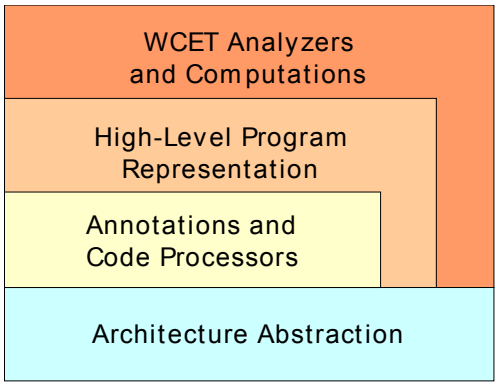

Figure 1: Framework Layers.

The base of the stack is the architecture abstraction layer that provides a generic abstraction for representing programs and hardwares to upper layers.
The second layer provides annotations that may be hooked to the entities of the architecture abstraction. Facilities are given in this layer for modifying the annotations.
In the third layer, the high-level program representations are built using the annotation system. They are viewed by the architecture abstraction layer as usual annotations.
Finally, in the uppermost layer stand programs that implement analyses and computations of the WCET. They use information given by the other layers.
One may notice that the architecture abstraction is accessible by any other layer. Thanks to annotations, it is the actual backbone of the framework. All produced information remains linked to the underlying program code. Retrieving properties of the hardware platform is then easier.

2.1 Architecture Abstraction

The architecture Abstraction represents the foundation for the rest of the framework. As shown in figure 2, one of its components, the loader, is in charge of scanning the program binary and the hardware configuration file in order to produce the process representation. This one contains not only the program representation but also the description of the hardware platform.

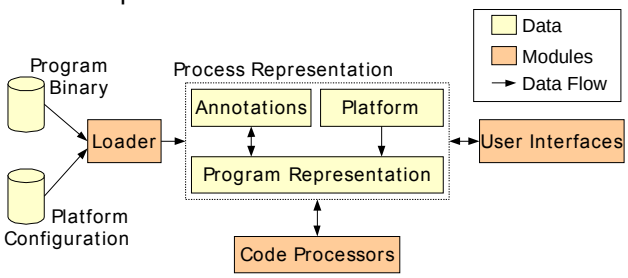

Figure 2: Architecture Abstraction Layer

Actually, the upper layers do not really work with the final representation of the process but, instead, with an abstract interface of entities of the process.
The link between the abstract interface and the actual classes is performed by the loader component. OTAWA retrieves the actual classes by scanning the binary and by selecting a matching plug-in.
As a result, the framework can support any architecture provided an adapted plug-in is available. It should also be emphasized that the code processors in upper levels becomes insensitive to the hardware architecture.
However, information about the hardware can be retrieved in any layer. Indeed, this kind of information may be accessed from the platform description database in an easy and portable way whatever the actual architecture is.

2.2 Annotation System

The annotation system is another component of OTAWA that aims to enlarge the framework usage.

An annotation is a piece of data with an identifier, for retrieving it among other annotations, and a type, providing facilities for serialization, display and so on. An annotation may be hooked to any entity providing annotation list services like architecture abstraction objects or high-level program representations.

Annotations are a very generic feature that fits well with the process of WCET computation. The different analyses that drives to the WCET works by capitalizing more and more information about the behaviour of the program. If an information item is lacking, the analysers assume a default safe value to continue their task.

The analysers, called code processors in OTAWA, are using annotations as such: they explore the architecture abstract representation and pick

annotations provided by previous processors. Their results are stored in the same way using other annotations or modifying existing ones. Consequently, the WCET may be viewed as a special annotation produced by the WCET computation processor.

As an example, Figure 3 shows a chain of code processors that computes the WCET by the Implicit Path Evaluation Technique.

1. Control Flow Graph building.
2. Loop bounds acquisition.
3. Global analysis for instruction cache [12].
4. Trivial analysis of data cache.
5. Basic block timing computation by simulation.
6. Variables assignment on CFG edges and nodes.
7. ILP inequalities building [12].
8. ILP objective function building.
9. ILP system resolution (producing WCET).

Figure 3: WCET Computation Chain using IPET Method

The resulting system supports a flexible composition of processors. For example, if we want a rough WCET approximation, some processors may be either fully removed, or replaced by simpler ones.

It should also be noted that the independence, induced by the annotations between the processors, releases constraints on the processor chain. It does not need to be linear as in our example. A single processor may be replaced by a whole sub-chain, some parts may be conditional or some results may be obtained in different ways, possibly from external sources.

As a last point, the chaining of code processors is performed thanks to a unique interface. This uniqueness may be used to plug external code processors and the implementation of user interfaces around the OTAWA framework.

As a result, the annotations are really the glue that maintains the other components sets. Consequently, the high-level program representations are built onto them.

2.3 High-level Program Representations

OTAWA provides a set of high-level program representations including:

- Control Flow Graph (CFG),
- Abstract Syntax Tree (AST),
- Context Tree (CT).

They are built by dedicated code processors and tied to the architecture abstraction entities by annotations. As the annotation system provides some extensibility properties, it is not a matter to add new program representations.

Code processors that process these representations may work in two modes: either they may call the matching processor, or they may fail if the required representation is not available. In any case, this avoids to rebuild many times the same representation.

A specific facility provided by OTAWA is the virtual CFG. A virtual CFG is the copy of an actual CFG that is then modified according the needs of code processors. For example, a call edge may be replaced by the CFG of the called function for achieving inlining. In the same way, the Basic Blocks (BB) may be split according to rules of l-blocks of some instruction cache management algorithm [1, 13]. The virtual nodes, representing a sub-CFG, allows us to implement efficiently a CT [13] or scope trees [14].

2.4 WCET Computation

The top layer performs the main task of OTAWA, that is, applying some static analyses in order to compute the WCET. Of course, this layer benefits from the facilities provided by the components previously mentioned.

The framework is not dedicated to some kind of WCET computation method. It provides a set of tools required by usual methods like ETS, IPET or any algorithm based on a static analysis of the program. At some point, the framework is even not especially tailored for WCET computation. It may be used in any case where a program in binary code needs to be scanned.

Currently, OTAWA supports a full chain for IPET computation and the base analyses for ETS. Although we have no plan to implement other methods, there is actually no limitation preventing from adding other algorithms like, for example, the abstract interpretation approach.

The framework has been designed and developed for two years and it provides now enough facilities for computing WCET and for being broadly used.

2.5 Current State

OTAWA is a library of C++ classes, some plug-ins including a PowerPC loader and an ILP engine based on lp_solve [16] and some utility programs. It is also delivered with a PowerPC simulator.

We have planned to develop soon a loader for ARM architecture and some plug-ins for supporting other ILP engine.

From the WCET point of view, two approaches have been developped, IPET and ETS, but other approaches could be implemented using OTAWA as, for example, [5, 16].

The IPET approach includes the following code processors:

- automatic generation of CFG from binary,
- basic block timing analysis by simulation,
- generation of CFG-based ILP constraints,
- support for first-level direct-mapped instruction cache [13, 17],

- minimal support for data cache,
- CFG timing analysis using Δ method [18].

The ETS provides very few processors:

- automatic generation of an AST from a C source,
- ETS block timing analysis by simulation,
- basic timing schema evaluation,
- support for first-level instruction cache (direct-mapped or set-associative with LRU replacement policy).

Both methods share a flow fact loader and a flow fact description file format. Although the flow information is bound to regular loops, we plan to improve this component.

To validate the OTAWA architecture, we have performed two case studies.

4. Case Studies

Two case studies have been performed for checking the ability to extend and to develop new analyses in OTAWA. The work has been achieved by two developers who were not proficient with OTAWA within a delay of three months.
This survey aimed at answering two questions. Firstly, from a qualitative viewpoint, we wanted to check if the OTAWA Application Programming Interface is good enough for developing quickly new analyses. In the second, we wanted to evaluate the rough efficiency of the framework in order to know if OTAWA is fast enough for real WCET computation or for use in an experimental context.

4.1 First Case Study

The goal of this study is the comparison, inside the OTAWA framework, of two methods for handling direct-mapped instruction cache with the IPET approach.
The first method, described by Li and Malik [17], models the cache by a projection of the CFG according to each cache line. This Cache Conflict Graph (CCG) is then processed to generate new constraints included in the ILP system and to modify the maximized function representing the WCET. This method is very heavy because it adds a lot of new constraints and variables to the ILP system. As ILP solvers have most of the time an exponential complexity, the overall computation time grows quickly while a lot of memory is required for building the CCG. Yet, this method is well integrated within the IPET approach making the cache sensitive to any flow fact information of the program without additional work.
The second method (CAT) has been adapted from the work of Healy and al. [13] to IPET. For each cache line, a Data Flow Analysis is performed on the CFG in order to compute the state of the cache line at the entry and at the exit of each basic block. The context tree of the program, composed of function environments and loops, is then built and used for assigning a category at each point of the program possibly inducing an instruction memory access. These categories may be always-miss, always-hit, first-miss or first-hit and shows if an instruction causes a cache miss or a cache hit according to the iterations of the container loop. Using the loop bounds, the hit count and miss count of each memory access are computed and injected in the ILP system as constants added to already existing constraints. This method is lighter than the previous one but it seems to be less accurate than CCG as it can only benefit from loop count flow facts. Even worse, the original algorithm performs some simplification making some categories assignment too pessimistic.

The actual question of the study was not to compare the accuracy of both methods, CCG is almost ever better than CAT, but to check if the overhead induced by CCG is worth compared to the results of the CAT method. OTAWA is well-suited to perform such an experiment because both algorithms are developed and evaluated in the same environment. The comparison is relatively straight and fair with very few side effects coming from the implementation.
The evaluation has been performed on the SNU-RT benchmark from Seoul National University and the results are represented in Figure 4. The clear bars shows the improvement of the WCET estimation by CCG relative to the CAT method while the dark ones represent the increase in computation time. Both are in percent. Most of dark bars going out of the graph are not fully represented.

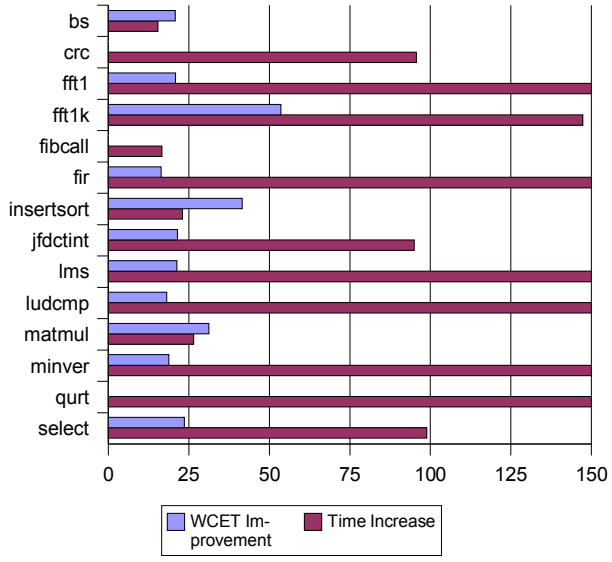

Figure 4: CCG / CAT Comparison

With an average accuracy improvement of only 20%, the CCG method is much slower than the CAT method with an average computation time increase of 160%. Choosing between these algorithms will rely on the use of the WCET: CCG best fits the need of an accurate WCET while CAT brings fast computation.

4.2 Second Case Study

The goal of the second case study was to show that OTAWA can accept other program representations and WCET computation methods.

The ETS [10] has been chosen because it is based on AST, a program representation very distant from the CFG used in IPET. For building the AST, we have developed a specific loader processor that uses an AST description file from the Heptane tool [2].

Then, we have implemented different processors in order to compute the WCET according to the ETS method:

- evaluation of AST blocks execution time using a microprocessor simulator,
- application of ETS basic rules on the AST for computing WCET,
- analysis of instruction cache behaviour for direct-mapped and set-associative policies based on categorization,
- use of instruction cache information in the WCET estimation.

The development has been completed and is fully working although it suffers from the usual limitations of the ETS method : the C source must be available and it only support statement without complex conditions, that is, without short-circuit logic operators and function calls.

4.2 Usability Evaluation

Both case studies have been implemented by developers non-skilled with OTAWA. Each work has taken three months during which OTAWA has proved to have a fast learning curve. We guess it comes certainly from the small size of the OTAWA core and Application Programming Interface. Once, the annotation, the loader and the processor core have been understood, the user has only to learn a small set of classes dedicated to its own domain. A lot of work is hidden behind the architecture abstraction layer and in the existing processors.

For getting an estimation of the work done in each case study, we have measured the produced sources using SLOC count (Single Line Of Code), a comment-insensitive source line unit. The resulting code is composed of several well-commented C++ files, including headers and sources. As the first case study counts 1494 SLOC and the second 1054 SLOC, we have a really small code size. It seems OTAWA has reached an important objective, that is, an improved productivity in the development of analyses. Of course, it will need further experience for more accurate results.

Yet, this experimentation has shown lacks in the API of the framework. As a minimal API was an important requirement during the OTAWA design, both case studies had to develop possibly redundant facilities that should be embedded in the framework core. For example, both instruction cache analyses in the first case study requires a Data Flow Analysis : if the processors share the same DFA engine provided by OTAWA, their development load would be reduced and the overall performances would be improved by a single shared optimized implementation.

Additionally, the experimentation has shown that new features, used in case studies, should be included in OTAWA. For example, the concept of l-block, a partition of basic blocks according cache blocks bounds, is used in each instruction cache processors. So it is for the cache modelling data structures that are used by the three algorithms of the two case studies. This last issue show also that a generalization work, in the sense of object programming, should be done for better managing the model of architecture features and improving the re-usability of the models. For emphasizing this, the cache replacement policy modelization is used in the same way whatever the algorithms used in both case studies.

Finally, the case studies have shown that some modules of the current OTAWA implementation need improvement in order to be really efficient. This is the case of the flow fact loader that, as in many other tools, only supports regular loops. Some recent WCET papers contain some solutions that should be implemented in OTAWA soon.

4.3 Efficiency Evaluation

As the existing OTAWA code processors and the case studies just implement well-known algorithms, it is not meaningful to appraise the framework according the accuracy of the computed WCET. The key point of this paper is rather to evaluate OTAWA as a tool for the development of experimental algorithms.

A better efficiency criterion is the computation time because the development of new algorithms may require fast testing and incremental approach. Although the extensibility and openness facilities of OTAWA are key features, they have also a time cost as the computation speed may be a blocking hurdle. If the resulting analyses are too slow, the framework may be felt as unusable and will be left out.

Figure 6, on the next page, shows WCET computation times of the IPET method using the CCG algorithm to take into account an instruction cache. Along an exponential scale, the clear bars show the total time in millisecond measured on an Intel PIV 1.4GHz 512Mb. Most times are under one second except the fft1 benchmark which produces a peak at five seconds: although these benchmarks are relatively small, these results are promising. The dark bar, the time taken by the ILP solver, shows it consumes the bigger part of the computation time. This comes mainly from the CCG algorithm that generates a lot of new variables and constraints for the ILP system. Therefore new performances gain will require an improved or a new ILP engine.

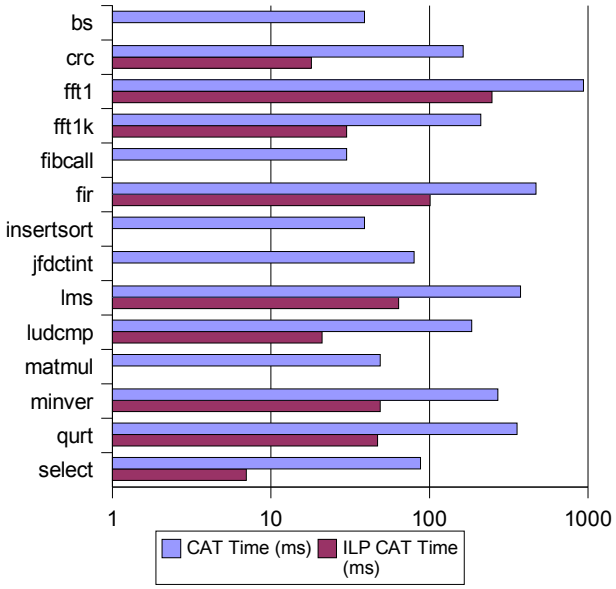

Figure 6: WCET Computation Time by CAT

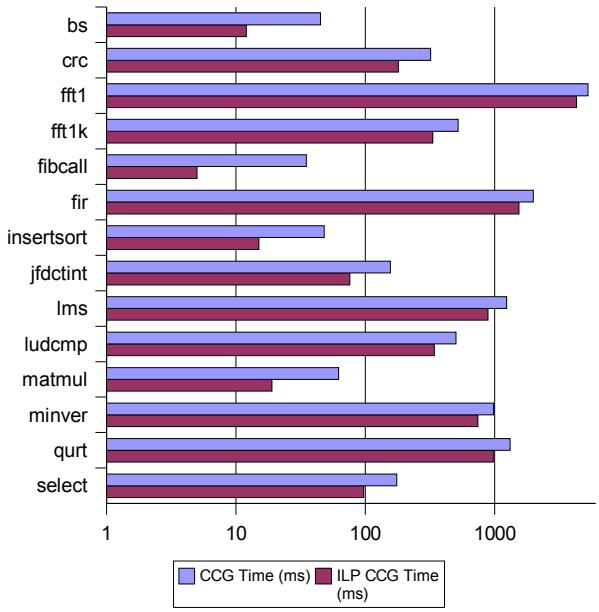

Figure 7: WCET Computation Time by CCG

Figure 7, shows the computation times for the IPET method with the CAT instruction cache management algorithm. This figure uses the same conventions as Figure 6. One may observe that, in accordance with CCG / CAT comparison, the measured times are a magnitude below the CCG times. Even if the graph is not of great help for understanding this, the ILP solver takes much less time: in average, the ILP solver consumes 55% of the time for the CCG method compared to 10% for the CAT method.

Hopefully, the computation times in both cases seems to be tractable within an experimental context. Most time consuming tasks are induced by a piece of code, the ILP solver, external to the framework that may be possibly replaced by a faster implementation. Although OTAWA has privileged extensibility and openness in its internal architecture, the obtained performances are promising and the framework is usable in the domain of WCET computation. We strongly believe that this result may be extended to other WCET methods. From a pragmatic viewpoint, the power devoted to the management of internal structures of OTAWA will not degrade too much the intrinsic performances of the ported WCET computation methods.

6. Conclusion

This paper presents OTAWA, an open framework dedicated to the WCET computation by static analysis in an experimental context. Its design has borrowed some powerful features to existing successful generic tools like Salto or SUIF to achieve easy extensibility and unconstrained openness goals. As a result, we have a tool using an abstract architecture layer for representing the program while miscellaneous analyses are performed using so-called code processors that, organized in chains, store and use annotations on the abstract architecture layer. Although the framework is not specialized to a WCET computation method, a particular processor chain can implement a specific method. Currently, OTAWA provides chains for the IPET and ETS methods. Additionally, the framework provides some facilities often used in WCET computation like DFA, program high-level representations, flow fact loader and so on.
To experiment the tool, two case studies have been performed. In the first one, we have added instruction cache management to the IPET chain according to two algorithms, CCG and CAT, in order to compare their performances. In the second case study, we have implemented a relatively full computation chain for the ETS method. The implementation of both case studies has been successful and has demonstrated the usability of the framework for this task in spite of some remaining minor pitfalls that are easy to fix in future versions of OTAWA. Another interesting measure in experimental context concerns the computation time. Despite the weight of features ensuring extensibility like annotations, the measured times are creditable and make the framework usable in an experimental context.

Although OTAWA has reached a milestone in its development, the overall tool architecture is not yet fully validated. The abstract architecture layer has only be tested with the PowerPC processor family but we hope to implement quickly a plug-in for the ARM processors. So are the plug-in for ILP engine used in the IPET method: we plan to include a plug-in for the GLPK library.
Some place remains also for improvement. Our flow fact loader is rather rough. Although it is not specialized for a WCET computation method, it only supports regular loops bounds flow facts while there are solutions for representing more complex loop bounds or non-loop flow facts. Moreover, the match between flow facts and the program is very low-level:

it should be handy to provide support for annotations in source but this requires a better integration with the compiler. In spite of some interesting attempts to solve the problem, tracking flow fact annotations in optimized compilation remains practically a hard task. OTAWA needs also to improve existing code processors or to add new ones. For example, the support for data cache is still very crude: each load or store instruction is considered as a cache miss. Improvement may be performed according to different levels including local data access, array access and alias analysis.

As a last word, OTAWA is currently used in our team for the development of new methods for supporting features of modern processors like multi-issue units, branch prediction or symmetric multi-threading. We are also exploring the feasibility of an incremental / adaptative approach to WCET computation. We hope that this framework will help to speed up the implementation of these new algorithms and, in turn, this work will represent a real proof of the extensibility and the openness of OTAWA.

7. Acknowledgement

We especially want to thanks M. Benoit and M. Tawk for the work performed to implement the case studies.

8. References

[1] Y.-T. S. Li, S. Malik, "Performance analysis of embedded software using implicit path enumeration", Workshop on languages, compilers, and tools for real-time systems, 88-98, 1995.

[2] A. Colin, I. Puaut, "*A modular & retargetable framework for tree-based WCET analysis*", Proc. of ECRTS, 37-44, 2001.

[3] S.-S. Lim, Y.H. Bae; G.T. Jang; B.-D. Rhee, S.L. Min; Chang Yun Park, H. Shin, K. Park, C.S. Kim, "*An accurate worst case timing analysis technique for RISC processors*", Real-Time Systems Symposium, 97 - 108 , Dec. 1994.

[4] N. Holsti, S. Saarinen, "*Status of the Bound-T tool*", 2nd International Workshop on Worst-Case Execution Time Analysis (WCET'2002), June 2002.

[5] C. Ferdinand, F. Martin, R. Wilhelm, "*Applying compiler techniques to cache behavior prediction*", ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems, 37-46, Jun. 1997.

[6] M. Alt, F. Martin, "*Generation of efficient interprocedural analyzers with PAG*", Proceedings of SAS'95, Static Analysis Symposium, LNCS 983, pages 33-50, Sep. 1995.

[7] H. Cassé, C. Rochange, P. Sainrat, "*An Open Framework for WCET Analysis*", IEEE Real-Time Systems Symposium - WIP session, pp 13-16, Lisbonne, Dec. 2004.

[8] R. Rohou, F. Bodin, A. Seznec, G. Le Fol, F. Charot, F. Raimbault, "*Salto : system for assembly-language transformation and optimization*", INRIA RR-2980, Sep. 1996.

[9] "*GLISS*", http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=54.

[10] R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K. Tijang, S.-W. Liao, C.-W. Tseng, M.W. Hall, M.S. Lam, J.L. Hennessy, J.L, "*SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers*", ACM SIGPLAN Notices, 29(12):31-37, Dec. 1994.

[11] G. Holloway, C. Young, "*The flow analysis transformation libraries of machine suif*", Second suif compiler workshop, Aug. 1977.

[12] Y.-T. S. Li, S. Malik, A. Wolfe, "*Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches*", Proc. of ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-time Systems, pp 47-55, June 1997.

[13] C.A. Healy, R.D. Arnold, F. Mueller, S.B. Whalley, M.G. Harmon, "*Bounding Pipeline and Instruction Cache Performance*", IEEE Transactions on Computers, 48, 1, 53-70, 1999.

[14] J. Engblom, A. Ermedahl, M. Sjoedin, J. Gustafsson, H. Hansson, "*Worst-Case Execution-Time Analysis for Embedded Real-Time Systems*", Journal of Software Tools for Technology Transfer, 2001.

[15] "*lp_solve*", http://groups.yahoo.com/group/lp_solve/

[16] C.A. Healy, R.D. Arnold, F. Mueller, D.B. Whalley, M.G. Harmon, "*Bounding pipeline and instruction cache performance*", IEEE Trans. Computers, 48 (1):53-70, Nov. 1999.

[17] Y.-T.S. Li, S. Malik, A. Wolfe, "*Efficient microarchitecture modelling and path analysis for real-time software*", Proceedings of the 16th IEEE Real-Time Systems Symposium, 254-263, Dec. 1995.

[18] J. Engblom, A. Ermedahl, "*Pipeline timing analysis using a trace-driven simulator*", Proc. Of 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99), IEEE Computer Society Press, Dec. 1999.