I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1645

4TH INTERNATIONAL WORKSHOP ON WORST-CASE EXECUTION TIME ANALYSIS (WCET2004)

Satellite Event to ECRTS'04, Catania, Italy,

Isabelle Puaut (Workshop Chair)

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

Message from the Workshop Chair

Welcome to the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis. The workshop is a satellite event to the 16th Euromicro Conference on Real-Time Systems (ECRTS 2004). It was held in Catania, Italy, on the 29th of June 2004. This is the fourth event in the series after the successful meetings in Delft (Holland) in 2001, Vienna (Austria) in 2002, and Porto (Portugal) in 2003 (see http://www.ecrts.org/wcet/).

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 provides a relaxed forum to present and discuss new ideas, new research directions and to review current trends in this area. The workshop is based on short presentations that 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, modelling and analysis of processor features
- Calculation methods for WCET
- Strategies to reduce the complexity of WCET analysis
- Timing analysis through measurement
- Probabilistic analysis techniques
- Integration of WCET and schedulability analysis
- Evaluation and case studies
- Tools for timing analysis
- Current practice in industry
- Integration of WCET analysis in development process

The proceedings include the papers presented during the workshop as well as a detailed transcript of the discussions.

If you would like to reference any article included in the WCET2004 proceedings, please note that these proceedings are published as a research report from IRISA, number PI-1645 (see http://www.irisa.fr/bibli/publi/pi/).

I would like to express my congratulations to all participants, authors, members of program committee, external reviewers, session chairs (Raimund Kirner, Stefan M. Petters, Jan Gustafsson), that have made this event a successful one. Special thanks go to Lucia Lo Bello and the local organization team in Catania, as well as Laurent David and Raimund Kirner, for the local arrangements.

Isabelle Puaut
University of Rennes, France

Program committee

- Andreas Ermedahl (Univ. Mälardalen, Sweden)
- Raimund Kirner (TU Vienna, Austria)
- Stefan Petters (Univ. York, UK)
- Isabelle Puaut (University of Rennes / IRISA, France)
- David Whalley (Florida State University, USA)
- Reinhard Wilhelm (Saarland University, Germany)

Table of contents

Session I: Compiler and Runtime Optimizations for WCET Determination

Chair: Raimund Kirner (TU Vienna, Austria)

Presentations

In the first presentation *"Predictable Timing Behavior by using Compiler Controlled Operations"* Vesa Hirvisalo gave a position to use compiler techniques to improve temporal predictability of real-time systems. The motivation is to replace unpredictable hardware and operating system features by more predictable features. As a first approach, caches are replaced by scratchpad memories to improve predictability of memory references.

The second talk given by Hemendra Singh Negi was about *"Simplifying WCET Analysis by Code Transformations"* as a preprocessing step before doing the path analysis to gather information about feasible paths. These code transformations make infeasible paths in programs more obvious and therefore simplify the path analysis. Several examples are given to give an idea how this transformation should look like.

In the third presentation *"Optimizing JVM Object Management Operations to Improve WCET Predictability"* Corrado Santoro presents technical solutions for a Java Virtual Machine to make its timing behavior more predictable. The garbage collection becomes predictable as it is only called directly before allocating memory and frees only as much memory as is needed for the new allocation. Sources of unknown behavior like dynamic classes or dynamic arrays are handled by restricting the programming language respectively by using code annotations.

Discussion

The following discussion centered around the properties of compilers for real-time systems and their possible support for timing analysis.

Christian Ferdinand: I come from the industrial side, working for the company AbsInt. We are providing commercial timing analysis tools. Regarding the point of emitting information by the compiler for timing analysis, all of our customers are using existing compilers. These compilers are typically quite old because for people developing safety-critical applications there is typically no chance to change the compiler. For example in the automotive industry, people are not only using compilers but there is also a high amount of hand-written assembly code in it. So I don't see a chance for the safety-critical area; one cannot change the tool chain and to somehow getting the information from the compiler what optimizations have been performed. For the generation of more predictable code it is the same problem. Depending on the domain it requires many years to define how a safety-critical application is build. For example, there is a design to use ADA or C then it is frozen at the beginning of the project and only ten years later they indeed use C. Ten years is a typical time period for the avionic area when it will come into production. It is not easy to propose anything else then C or ADA. Therefore, I don't see that the industry is going to switch from the current programming paradigms. It will be C for the next 20 years. And still

declining but active there is ADA and that's all. I do not see that the industry is willing to pick up any other coding paradigm for the next years.

Peter Puschner: Coming back to Christian's statement, of course, I see the current situation in industry. And of course, things will change very slowly, I agree. And we shouldn't have any illusions on how things will develop and I think it would be quite difficult to get kinds of *revolutions*. We will rather have to expect *evolutions*, but I think we should not be too pessimistic even if it takes some time. It is the purpose and also the responsibility of research to come up with new solutions and to work out new solutions. It is our responsibility to push the state of the art into a direction. Someday, people cannot deny that technology is there, that certain techniques are state of the art and have to be used. This sometimes happens. If you build a system and the system puts people into problems and later on you find out that the tools that have been used do not reflect the state of the art. After some time of course, everybody will accept it. It will take some time to reflect research results in industrial technology but at some time I think our research results have to be picked up. And so, even it takes time, I would not say that we will not ever see them as strictly as you [Christian Ferdinand] did in your first statement. I would say that would make any research ridiculous and useless. But that is not the case.

Christian Ferdinand: Maybe, that was a little bit too pessimistic. It is not to discourage you in that area. Probably it is a kind of dilemma. As for the most safety-critical applications like fly-by-wire, airbags in cars and all such things, I see people that do have a budget and might be interested in using WCET tools and say that they will not change other tools, so that is the dilemma. But there are huge other areas which are less critical, like telecommunication. They use also C or in some cases Real-Time Java. These areas are much less critical but they are typical in projects where timing is not so critical, i.e. it is more soft real-time. They are not really interested in the worst case but in some worst average case or something like that; things that can you usually do with measurements. But they are not going to pick up the tools, they are not going to invest into them. But this community provides tools that are working for them, they are working for Real-Time Java and maybe some other languages, there is a chance. But currently, these tools are very specially and very expensive, and they are only going to be used for the most critical system designs.

Iain Bate: Listening to the conversation of you, you are both right. The problems we are always have to face in the critical systems domain is legacy. Therefore, we often have to keep the antiquated approaches like frames systems development. And any such approach has to deal with legacy. What we always have to remember is that these are large legacy components which are hard to deal with, using languages like ADA or C; they are conservative coding style approaches. And the other thing to remember from the industrial practice is that timing analysis is only a minor part of what they have to do. And anything that we propose like code transformation has to be weighted off against the bigger problems like functional verification, unit testing, etc.

Guillem Bernat: One extra comment to this special area is that now there is already the awareness that there is a problem of finding the worst-case timing behavior. A few years ago there were these simple chips where worst-case analysis tools where easy to build and the hardware was easy to analyze. Now we hear somewhere in the avionics they were starting using caches, pipelines and very complex processors. There is a lot research to do. People are already committing to using these advanced processors and then they say: *"oh, we have a problem and we do not know how to do in that case"*. As you say, it is changing very slowly, but engineers are starting to use these processors even in very critical systems. The other point on the compilers is that it is true that people buy the compiler and stay with that compiler. But we had some experiences recently where compiler vendors where approaching to us: if our compiler knew how to do the WCET

analysis we can sell it to more customers. This is the first time we got this message of compiler vendors now trying to find an edge on their problems, having more real-time features, support for real-time operating systems, and a lot of support for instrumentation and WCET. To grab the argument from Peter [Puschner] and you [Iain Bate]: things are changing, slowly but they are changing. The good thing is that we must be there when people start asking for this.

Raimund Kirner: I can just express my opinion of how it works in industry, they just pick every solution that is available ready to use, that's all. In a company in that area there is no significant time for research or development budget. But there is also a good chance for applying research on WCET compiler support to industrial practice. For certain new chip products a compiler has to be written from the scratch, for example, for some special signal processors and so on. In such a situation it could be useful to think from the beginning which data structures and mechanism are suitable to support aspects like predictability or timing analysis. And as Guillem [Bernat] said, it helps if compiler vendors are looking for niche markets by supporting timing analysis. Then if other people see that it works quite well it may also convince them to adapt such technologies. It may be still a question of time, but as long as we do not have a solution, industry will not adapt it, for example, for their compilers. It may still take 20 years, but at least we have to do the basic research, because timing analysis is still a very serious problem. And of course, if everyone just says that due to all the legacy systems it is too complicated to do anything, there will be no change. But of course, this will not bring any advantage for the software development process used by them. Luckily, development departments of many companies already think of using better and more structured software development processes. And I think, for new projects, they are also willing to change their whole software development method to achieve more predictability. Therefore, I think there are some realistic possibilities to introduce these technologies.

Peter Puschner: If I am allowed I would like to ask a question, actually more than making a statement or answer to one of these statements we had before. Since we had been talking about transformation, every compiler does some kind of code transformation. If we speak about optimization it does a little bit more than just code transformation, but even translating from C to machine language is some kind of transformation. Since we have some people here from industry and also other people are dealing with industry or having contacts to industry, I would be interested to know what are the limits of transformations that do you think would be allowed in order to generate code for a safety-critical system. I don't know if it is possible to define such a limit, but I think that there have to be some transformations and I know it is easy to certify compilers which do very simple transformations from source code into machine code, but where does this simplicity end when you want to certify a system? Is there any idea or does anybody have an idea how this can be expressed, because I think the second talk[1] pointed out to - at least from a research point of view – a very interesting solution to the problem of WCET analysis. But I heard a comment that it would not be feasible in an industrial setting, so therefore my question.

Iain Bate: I think the answer is easy. What you are allowed to do is driven by cost. If it is cheaper to transform the code and just skip the transformation and do the analysis without the transformation you will be allowed to do it. But you have then go to the customer just to define the transformation. It is a cost tradeoff. There is nothing to stop you doing optimizations and transformations as long as you can show that you do not introduce any additional hazards into the software. And that is the tricky bit, showing it for all conceivable cases to that it might be applied. Normally, the reality is that they just say that you cannot just do it. But I see a lot of work on tools like code generation, you know, tools like SCADE, that successfully brings models into the

[1] H. S. Negi, A. Roychoudhuri and T. Mitra: *Simplifying WCET analysis by code transformations.*

code and compile them down. That is allowed because people did a lot of efforts just to define those transformations which was very expensive the first couple of times they did it. But what they consider is that it was worth doing the transformation. Any compiler just does transformations.

Raimund Kirner: I am not really the expert in this area on certification. So, I'm interested about what are compiler optimizations – let's say code transformations to improve performance – are currently used for certified systems in the avionics. As far as I know they already use code transformations to improve performance. Maybe Iain [Bates] knows some details on this.

Iain Bate: No, they do not use optimizations. Still, they turn off all optimizations in the compiler.

Raimund Kirner: I was asking because I heard from some unofficial sources that people are also thinking about using compiler optimizations in the avionics.

Iain Bate: It depends on the system. There are a lot of systems in the avionics, on an aircraft like the flight control system which is one the most critical system on an aircraft and any form of optimization will not be considered. Then, the various integrity levels down from that, low integrity level, less real-time, less critical systems, then certainly code optimizations would be considered. Though, on those systems they probably do not do WCET analysis, they would use more test-oriented methods. But certainly, on the parts of the system where one is doing static analysis one would not turn the optimizations on. Would you agree?

Christian Ferdinand: If you have a verifier, a tool that can verify that the optimized code is also correct, i.e. corresponds to the source, you can do it. But as long as such tool is not available they will probably switch off all optimizations.

Stefan M. Petters: I heard once the argument, probably not in the highest critical systems area, the most used optimization would be actually "-O1" in terms of – and there comes the argument now - this is the most often used compiler optimization stage and as such is most trusted compared to the no-optimization setting. I cannot confirm that this is really the case but I have been told that.

Guillem Bernat: One thing that we should not forget is that the scope for which WCET applies is quite broad. On one extreme there are the absolute safety-critical systems as the avionics. What they actually say is about you do not trust the compiler. On another scope we have the people in the automotive industry which are very keen using WCET approaches because they start having to share the same chip with other applications and they have budgets. They also have to guarantee timing properties as this is safety-critical but not at the level of the avionics. And now we have the telecom industry where something does not work and this is always the problem. And possibly the timing problems, how do we go to analyze this? My suggestion is that we try to keep the conversation open in the sense that if we talk about WCET, some people always do the absolutely safety-critical, some do the soft real-time systems. Actually, we should try to get a broader view. Some aspects cover all these approaches. For example the real-time application of Java which I do not have seen flying in a plane but doing a lot of other efforts.

Iain Bate: If you look at the avionics standards they define critical systems of possibly causing harm of lives. If you look at some standards like IEC 61508 which is a reference standard for programs and electronic controllers, etc. They define critical systems if it could loss life or economic damage in case of a failure. And therefore, to a certain extent I agree with Guillem [Bernat] in a sense that there are a lot of people at telecommunication, mobile phones manufacturers etc. who are beginning to get more willing to consider execution time analysis

because if the system fails and they have shipped a millions units that it is very costly, and that is a different type of critical system, where economics is driving it. They spend much more efforts into verification, certification, power proofed, etc.

Christian Ferdinand: In avionics everything is regulated, so there is a need for certification. In the car manufacturers OEMs and suppliers there is no such thing, there is no regulation and they typically use all optimizations the compiler provides, even for the safety-critical applications.

Stefan M. Petters: Not yet! They use not yet a standard, but…

Christian Ferdinand: Yes, but even if it involves them the people *insist* that they *can use all optimizations* that are there. And if you look at the USA and the critical systems there, there is a danger for the OEMs there that they will be filed with a huge amount of money if they fail to use state of the art. So, if there are tools that show the absence of errors, so to say in the timing domain the WCET tools, they are going to use them to avoid these damages. So, *we have to have WCET tools for the car manufacturers that also work with the highest level of optimization.*

Raimund Kirner: I want to make an open comment on this because when you think about introducing compiler support for timing analysis, of course, it requires a lot of changes. For really safety-critical systems there is just a very small market compared to other systems. Therefore, it might be also useful to consider the application of timing analysis for domains that do not require absolutely safe bounds. This may also introduce WCET analysis techniques and also compiler support into mass production. If this happens it will also convince people from hard real-time systems domain that compiler support for WCET and predictability would help to improve the overall design flow. This would be a chance we can think about.

Predictable Timing Behavior by using Compiler Controlled Operation

Vesa Hirvisalo and Sami Kiminki
Helsinki University of Technology
Laboratory of Information Processing Science
P.O. Box 5400, FIN-02015 HUT, Finland
Vesa.Hirvisalo@cs.hut.fi, Sami.Kiminki@iki.fi

Abstract

We propose coordinated use of compiler techniques to improve predictability of timing behavior of hard real-time systems, and thus, to tighten their worst-case execution times. We aim at a generic methodology of compiler optimizations that replace the use of unpredictable hardware and operating system features by the use of more predictable features. We call the approach compiler controlled operation, because it is based on using compilers to control operations that are traditionally controlled by hardware or operating systems. As an example of the approach, we overview our work in progress on a small experimental system.

1 Introduction

This paper discusses how compiler techniques can be used to build software systems having predictable timing behavior. Predictability of timing behavior is needed to guarantee temporal correctness of hard real-time systems.

There are several trends that make giving such guarantees increasingly difficult. New applications based on the use of real-time software components are rapidly emerging. To cope with the complexity of the software systems, high-level software development tools are being adapted. Many applications require high performance in addition to predictable timing behavior. Because of the increasing performance requirements and the use of generic purpose hardware components to limit production costs, hardware for real-time systems is becoming complex. These factors increase dynamism of the systems and make their timing behavior hard to predict.

Timing guarantees are given by schedulability analysis that is typically divided into intra-task analysis and inter-task analysis. Intra-task analysis resolves worst-case execution times (WCET) for tasks. Improving execution speed can be useless, even if the improvements yield tight worst-case execution times, unless such tight worst-case execution times can be guaranteed. Inter-task analysis determines whether the tasks can be guaranteed to be scheduled so that they meet their deadlines. Similarly, techniques for faster average-case execution can be useless, or even harmful, when inter-task real-time analysis is considered.

We concentrate on unpredictability caused by modern hardware and typical operating system features. Such features include – but are not limited to – cache memories, interrupts, and context switches. There are two things common to features considered by us. First, they make timing analysis difficult by using features that are not apparent from the application code. Second, more predictable techniques exist for implementing them in special cases. Using such alternative techniques yield tighter worst-case execution times.

Our generic solution is to use compiler techniques to transform the software to use more predictable implementation techniques, when such transformations are possible. We call the approach *compiler controlled operation*, because it is based on using compilers to control operations that are traditionally controlled by hardware or operating systems. Compiler controlled operation is based on static program analysis. Therefore, it is closely related to the use of WCET analysis methods based on static program analysis. In addition to improving timing behavior, compiler controlled operation can be used to other purposes, e.g., energy saving.

The structure of the rest of this paper is the following. In Section 2, we discuss compiler controlled operation in general. In Section 3, we consider briefly some possible realizations for compiler controlled operation. In Section 4, we overview our work in progress on an experimental system that concentrates on using fast on-chip RAM memory (often called scratchpad memory) to implement the operation of classic cache hardware and compiler-time scheduling to partially implement a process abstraction. The last section draws some conclusions and discusses some related work.

2 Compiler controlled operation

Programs are abstract entities, but they are executed in some concrete execution environment that usually has several specific features supporting the execution of programs. Typically, the execution environment includes the various features of the operating system and the whole underlying hardware (including both on-chip and off-chip features). The environment significantly affects the timing behavior of a program.

Compiler controlled operation means that some operations in the execution environment of an application are controlled by the compiler that compiles the application. This requires cooperation between the compiler and the execution environment.

The main task of operating systems is to implement process abstraction. This includes managing processes, scheduling processes, and providing processes with inter-process communication, synchronization and protection. Especially scheduling combined with synchronization causes problems in timing prediction, because of the run-time decisions made by the operating system.

Modern hardware includes speculative features to increase speed and supporting features to increase flexibility. The use of such features often make timing prediction hard. The typical speculative hardware feature that causes problems in timing prediction is the cache memory. The combined use of parallel processes and cache memories can cause severe problems in predictability [11].

Traditionally, the programmers of a system are responsible for using predictable techniques instead of the generic ones (e.g., application-controlled memory management). There are situations, where this can be automated. The use of predictable techniques instead of the generic ones can be implemented as optimizations done by a compiler. Such compiler optimizations consider the whole execution environment instead of the instruction set architecture of the processor. As in traditional compilers, several optimizations can be used in a coordinated way, and they can be used to promote predictability (e.g., tight WCETs) instead of average speed.

In addition to the transformations, analysis required for the transformations can be done by compilers, as well as timing verification. As for many WCET tools, user support may be needed to guide the compiler. However, some tasks can be fully automatized, e.g., the use of a scratchpad memory as a cache.

Considered from the point of view of the application developer, transparency is important regardless whether the analysis can be made fully automatic. Independent of the implementation, the same way of coding and the same interfaces should be used.

3 Various possibilities for compiler controlled operation

Traditionally, compilers are aware of the execution environment only partially. They know the target hardware architecture including target processor and memory layout, but often the operating system semantics and semantics of other applications in the system are completely unknown. Operating system interfacing is typically provided by libraries. Isolation is even a goal in many operating system designs.

In closed systems, there exists less reasons for such isolation, as often all application and operating system semantics are completely available at design time. Providing such information for the compiler reveals many exciting possibilities for optimization. We give some examples on how compiler might exploit extended information on execution environment.

A compiler can automate scratchpad memory usage and allocation. As scratchpad operations are explicit in the program code, they pose no inherent unpredictability. Instead, program code using scratchpad operations can be analyzed with existing WCET tools. Thus, the problems of cache behavior unpredictability can be avoided (see [10] as an introduction to such scratchpad usage).

When timing information of tasks is provided, the compiler can perform scheduling optimizations. For example, compiler might statically schedule and join multiple periodic tasks of same frequency or integer multiple of some base frequency into one task [1]. Further, if the information of hardware interrupt rate boundaries is known, interrupt handling may be transformed to polling by the compiler. Polling and branch prediction may improve WCET guarantees in some cases.

If the compiler is aware of operating system semantics, optimizations to inter-process communications can be performed. Semaphore synchronization may be transformed to statical task rescheduling in some cases, as well as many remote procedure call patterns are transformable to simpler inter-process procedure calls. Such transformations simplify scheduling analysis, and are thus susceptible to promote WCET guarantees.

The transformations need not to concern only application code, but may also be directed to the operating system. A typical task performed manually is the tuning of operating system features, such as the sizes of various buffers, and implementation techniques of features such as inter-process communication primitives. The operating system feature-selection and tuning can be seen as global optimization problem for the compiler.

Furthermore, when high-volume systems are of concern, the compiler could even tune the hardware execution environment. As with the operating system, tuning of hard-

ware execution environment can also be seen as global optimization problem. For example, the amount of scratchpad memory in the system, number of registers, special instructions (such as division, multiplicate-and-accumulate, and scratchpad transfer instructions), can all be seen as parameters to a global optimization problem. Today, hardware features are selected and tuned manually, and the choices may have great effect to the system performance.

4 Work in progress

The PAD system consist of two components: a compiler, padCC, and an operating system, padOS. It is a small experimental system that is designed for the study of platforms for closed embedded control systems that have periodic hard real-time timing requirements. The goal of the system is to promote predictable timing and low energy consumption in high performance applications.

padOS is a small real-time operating system. It supports multitasking, but has no memory protection, because it is designed for closed applications. It implements EDF scheduling and prioritized interrupt handling. padOS supports background tasks and inter-process communication.

padCC is a C compiler. In addition to optimization based on the instruction set architecture of the target processor, padCC supports optimizations based on the execution environment. padCC uses scratchpad memory hardware to implement the operation of classic cache hardware. Instead of using associative memory that is able to handle misses, padCC generates code that uses scratchpad memory instead of of main memory to store and access data. The analysis and code generation closely resembles the register allocation techniques used in optimizing compilers. Thus, all memory transfers are statically known.

padCC does partial compiler-time scheduling. A C program with operating system primitives is considered as a concurrent program that is compiled into a sequential program when possible (see [4] for an introduction to such techniques). The static scheduling is coordinated with the scratchpad memory allocation. Inter-task scratchpad allocation is realized by giving the sequentialized code to the scratchpad memory allocator.

All the transformations described above are optimizations. They are done by padCC, when they are possible. If the scratchpad cannot be allocated for some memory operation or some task cannot be scheduled statically, then that part of the code is left unchanged. As typical for optimizations, these actions are transparent to the user. Their main effect is to improve predictability of the timing of the execution. However, a deep understanding of the timing is needed to tune a program to fully use the features of the PAD system.

The PAD system has its roots in on our previous work on cache performance analysis [8]. The current version is designed for the ARM7TDMI processor [2] in a system that has 8kB of scratchpad memory.

5 Conclusion

The techniques used by us are not unique. The use of simple hardware features to promote predictability is very common in hard real-time systems. Also, using compiler techniques to implement statically-decided operation has been studied. However, we feel that multiple compiler techniques should be used in a coordinated way to promote predictability. In this way, our approach can be seen as an extension to the software synthesis approach [5] (exemplified by the compiler-based static scheduling) with new hardware-related optimizations [3, 14] (exemplified by scratchpad usage).

Our approach is a compromise between approaches toward very predictable systems (see [6, 12]) and the current practice. The building of predictable systems is also related to the corresponding analyses (e.g., [13]). Our research is especially dependent on the development of WCET analysis based on static program analysis and its relation to hardware development (see [7]). Our current practical work is limited. In the future, we aim at an implementation that makes full scale experimentation possible. Because of the successful previous studies on specific techniques (e.g., [9, 10]), we expect good results.

6 Acknowledgments

This work has been supported by Finnish Academy grant 51509. We also thank Kimmo Tuomainen and Juha Tukkinen for their input during early stages of this work.

References

[1] P. Altenbernd. *Timing Analysis, Scheduling, and Allocation of Periodic Hard Real-Time Tasks*. PhD Dissertation, Paderborn university, Department of Mathematics and Computer Science, 1996.

[2] ARM7TDMI Technical Reference Manual, 2001. Document code ARM DDI 0210B, rev 4, www.arm.com.

[3] Keith D. Cooper and Timothy J. Harvey. Compiler-controlled memory. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–11, San Jose, California, USA, 1998.

[4] S.A. Edwards. Compiling Concurrent Languages for Sequential Processors. *ACM Transactions on Design Automation of Electronic Systems (TODEAS)*, 8(2):141–187, 2003.

[5] R. K. Gupta and G. De Micheli. Hardware-software Co-synthesis for Digital Systems. *IEEE Design and Test of Computers*, pages 29–41, September 1993.

[6] J. Gustafsson, B. Lisper, R. Kirner, and P. Puschner. Input-Dependency Analysis for Hard Real-Time Software. In *Proceedings of the IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, October 2003.

[7] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The Influence of Processor Architecture on the Design and Results of WCET Tools. *Proceedings of the IEEE Symposium on Real-Time System (RTSS)*, 91(7):1038–1054, July 2003. Special Issue on Real-Time Systems.

[8] V. Hirvisalo. *Using Static Program Analysis to Compile Fast Cache Simulators*. PhD Dissertation, Helsinki University of Technology, March 2004.

[9] B. Lin. Efficient Compilation of Process-Based Concurrent Programs without Run-Time Scheduling. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 211–217, February 1998.

[10] P. Marwedel, L. Wehmeyer, M. Verma, S. Steinke, and U. Helmig. Fast, Predictable and Low Energy Memory References through Architecture-Aware Compilation. In *In Proceedings of Design Automation Conference Asia and South Pacific (ASPDAC)*, Yokohama, Japan, January 2004.

[11] I. Puaut. Cache Analysis vs Static Cache Locking for Schedulability Analysis in Multitasking Real-Time Systems. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, Vienna, Austria, June 2002.

[12] P. Puschner and A. Burns. Writing Temporally Predictable Code. In *Proceedings of the IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, pages 85–91, January 2002.

[13] J. Schneider. Cache and Pipeline Sensitive Fixed Priority Scheduling for Preemptive Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 195–204, Orlando, Florida, USA, November 2000.

[14] J. Sjödin and C. von Platen. Storage allocation for embedded processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 15–23, Atlanta, Georgia, USA, 2001.

Simplifying WCET Analysis By Code Transformations

Hemendra Singh Negi Abhik Roychoudhury Tulika Mitra
School of Computing, National University of Singapore
{hemendra,abhik,tulika}@comp.nus.edu.sg

ABSTRACT

Determining worst case execution time of a program by static analysis is important for the design of real-time software. WCET analysis at the programming language level requires the detection of the longest path in the program. A tighter bound on the WCET of a program can be achieved by identifying the infeasible paths in the program's control flow, which is a difficult problem. Due to the branches in a program structure, the number of possible paths in the program can grow exponentially. In this paper we present a method to transform the code such that the number of paths in the program could be reduced and hence the search space for the infeasible paths is brought down. This could reduce the complexity of determining infeasible paths in a program and also result in tighter WCET.

1. INTRODUCTION

The design of real-time embedded software requires that a guarantee must be given about the time taken by a program. Especially, in hard real-time systems, where the failure of a program to give results within a required amount of time may have serious consequences, the problem of determining Worst Case Execution Time (WCET) of a program becomes more critical. The WCET of a task is also important for scheduling the tasks in real-time systems. However, it is very difficult to obtain an accurate WCET of a task. Therefore, a tight bound on the WCET by static analysis methods is always desired to achieve better scheduling of tasks.

The problem of determining the WCET of a program by static analysis methods has to be solved at the following two levels [12]: (1) Programming language level, to discover the longest path from the start to the end of the program [7] and (2) Micro-architectural level, to take into account the effect of features such as pipeline, cache and branch prediction [6, 5]. The determination of WCET at the programming language level involves the detection of infeasible paths in the program and then use that information to give a tight bound on the execution time of the task ([3, 11]). In this paper, we only consider the programming language level analysis of the WCET. We will first describe the types of infeasible paths along with some techniques on how to detect them. We then present our idea to reduce their numbers and get a better estimation of the WCET of a task.

The knowledge about infeasible paths in a program can be used to give a tighter bound on the WCET. There could be infeasible paths because of the correlation between branches. For example, in Figure 1(A), <3,4,5,6> is an infeasible path because if the outcome of branch at line number 3 is true

```
1 for(i:= 0; i<limit; i++)
2 {
3      if ( i < 3 )
4              S1;
5      if ( i > 3 )
6              S2;
7 }
        (A)
```

```
1 sumeven := 0;
2 for (j:=0; j<=limit; j++)
3 {
4    if (j % 2 == 0) then
5       sumeven = sumeven + j;
6 }
              (B)
```

Figure 1: Infeasible paths due to branch correlation

then the outcome of branch at line number 5 can not be true. Detection of such types of infeasible paths has been studied in [2, 3]. Another type of infeasible paths which can be present in a program are ones that span over multiple iterations of a loop. For example consider the code to calculate the sum of even numbers, as shown in Figure 1(B). If the path <3,4,5,6> is taken in some iteration of the loop then it is not possible to take it again in the next iteration of the loop.

Detection of infeasible paths in a program is an important but difficult problem. A technique to detect and use infeasible path information is presented in [3]. We briefly describe their technique here to motivate how it could be benefited by our code transformation approach. In [3], the authors have used an effect based technique to determine the infeasible paths in a program and used this information for calculating the WCET of a loop. They first determine how a conditional branch can be effected by an assignment to a variable and/or the outcome of another conditional branch. The conditional branch could have one of the three types of effects: *unknown, fall-through* or *jump*. The effects on the conditional branches by the assignment of a variable are then exploited while traversing the basic blocks in every path of the program to determine whether the path is feasible or not.

Timing prediction of loops via control flow (as in [3]) poses a lot of problems for timing analyzer. A lot of space is required to represent all the paths, unavailability of which might abort the timing analyzer. Moreover, a large number of paths will result in a significant increase of the execution time of the timing analyzer. Therefore, a method which can reduce the number of paths, will be very useful. We present our approach as a pre-processing step to reduce the number of paths and hence reduce the complexity and time taken by the timing analyzer.

2. OUR PROPOSED TECHNIQUE

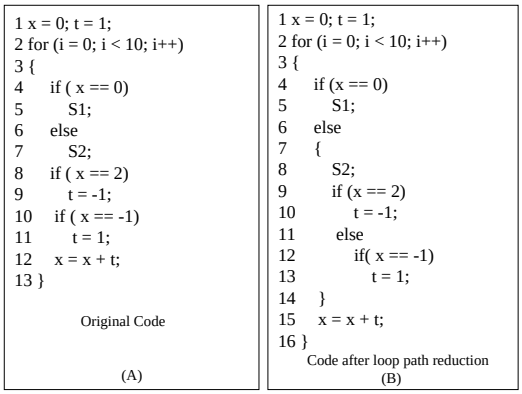

Figure 2: Example code to illustrate our technique

We observe that the detection of infeasible paths is inherently exponential in terms of the number of branch constraints. Hence, we try to develop a strategy to identify which branch conditions can be removed from consideration during the detection of infeasible paths such that the complexity of the detection algorithm could be reduced and at the same time a tighter bound on the WCET could be provided. We also try to optimize the code such that the number of paths in the code can be reduced. We try to exploit the constraints generated at branch conditions to optimize the code. In this section we will illustrate our technique with the help of an example and also show how the WCET analysis as per [3] can be benefited by it.

Reducing number of loop paths. Consider the piece of code shown in Figure 2(A). The values of x in the Figure 2(A) seen at line number 4 are in the form of a simple harmonic motion around the value 0. The sequence of values seen for x at line number 4 are (0,1,2,1,0,-1)*. '*' represents zero or more repetitions. The control flow graph for the code in Figure 2(A) is shown in Figure 3(A). From Figure 3(A), it is apparent that there are 3 branch conditions and 8 paths in each iteration of the loop. The various possible paths for each iteration in terms of basic blocks executed are given below.

```
a : 2 3 4 6 7 8 9 10 11      b : 2 3 4 6 7 8 10 11
c : 2 3 4 6 8 9 10 11        d : 2 3 4 6 8 10 11
e : 2 3 5 6 7 8 9 10 11      f : 2 3 5 6 7 8 10 11
g : 2 3 5 6 8 9 10 11        h : 2 3 5 6 8 10 11
```

However, it could be observed from the branch constraints that the results of branch conditions at block 3 and 6 could never be true simultaneously. Therefore block 4 can never be executed together with block 7. Moreover, both (true/false) paths from block 3 reaches block 6 and 8 where block 6 is a conditional statement and blocks between 6 and 8 could only be executed along with the false path from block 3. Also the constraint variable (x) of block 6 does not get assigned along the true path from block 3. Therefore, blocks 6 and 7 could be moved in the false path from block 3. Figure 3(B) shows the result of such a transformation.

Due to the transformation, the number of paths in the loop gets reduced to 6 from the initial number 8. Using the similar observation for conditional branches at blocks 3 and 8, the code can be optimized as shown in Figure 3(C), reducing the number of paths to 5. And finally the code can be modified to as shown in Figure 3(D), reducing the number of paths to 4.

The WCET analysis on the basis of the technique given in [3] will involve the following steps: determining the effect of assignments on the three branch conditions and then using this information to determine the infeasible sequence of paths. The technique will be greatly benefited by the optimization as the number of paths are decreased and so is the complexity of the technique which traverse over the paths to determine feasibility of paths and also the sequence of paths which is infeasible in consecutive iterations.

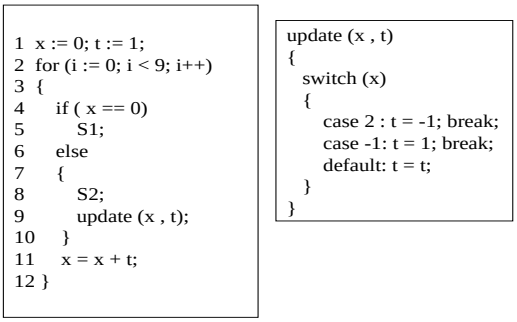

Figure 4: Example code after path length equalization

Equalizing path lengths. The optimization given in the previous section will transform the original example code into an optimized code as shown in Figure 2(B). We now try to deduce a transformation for this code to further simplify the WCET analysis. For our purpose, we propose a new type of block in the CFG along with basic blocks. The new block will be called as *functional block* which will represent a function. The various paths inside such a functional block will not be considered in the WCET analysis. We will see later in this section that a safe WCET bound can still be reached even though the number of paths considered for WCET are reduced without actually removing such paths.

We can identify the following paths, in each iteration of loop, from Figure 3(D).

```
a : 2 3 4 10 11              b : 2 3 5 6 8 10 11
c : 2 3 5 6 8 9 10 11        d : 2 3 5 6 7 10 11
```

The execution of loop will result in the following sequence of taken paths (abdbac)*. It is apparent that *aa, bb, cc, dd* along with *ad, abc, bdc* and many more, are infeasible sequences of paths that could be taken in consecutive iterations. Determining such infeasible sequences of paths with techniques as in [3] will be quite complex and computationally expensive. However, we propose the following code transformation to simplify things. The code in Figure 2(B) can be modified to the code as in Figure 4. The CFG for the modified code is shown in Figure 5

The combining of the blocks in path from 6 to 10 into *update* function and writing the *update* function in the way shown in Figure 4 could be very fruitful. Every call of the *update* function will take a constant amount of time due to the structure of the *update* function, hence the time taken

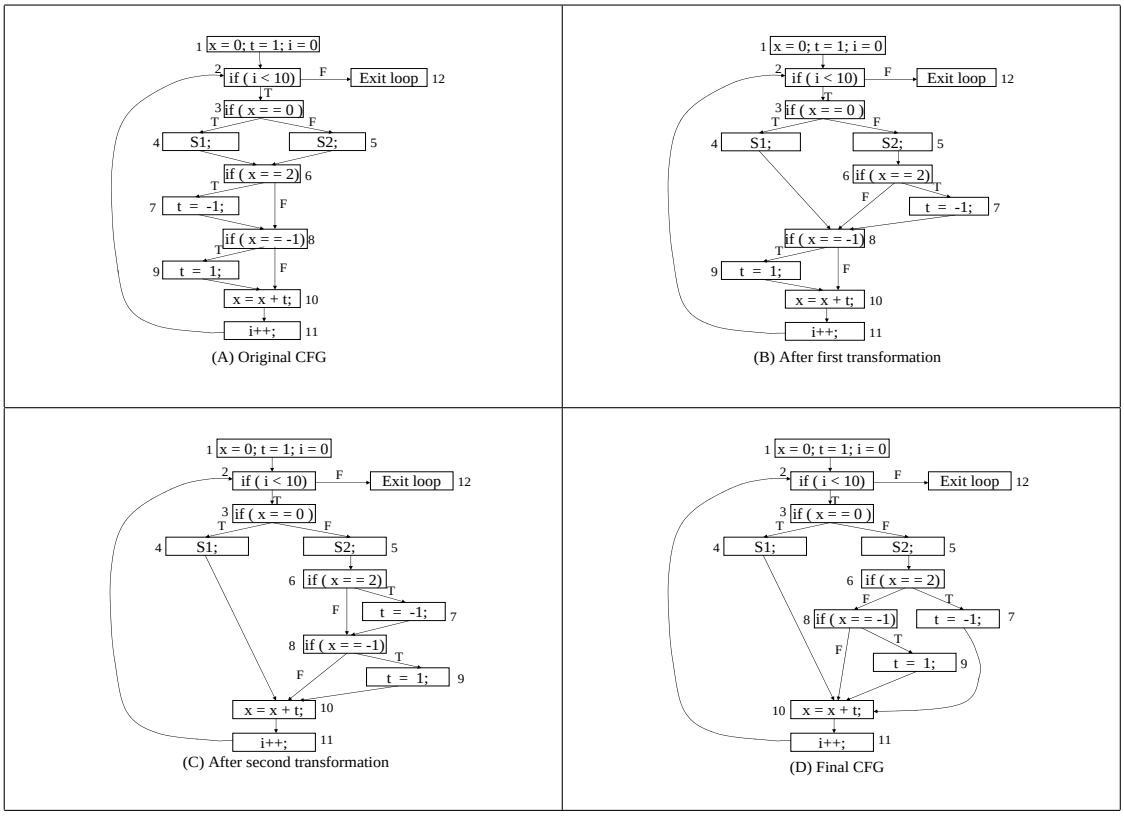

Figure 3: Reduction of number of loop paths in Control Flow Graph

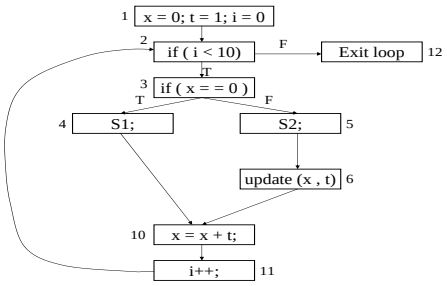

Figure 5: Control Flow Graph after path equalization

to execute block 6 in Figure 5 will always be the same, irrespective of the path taken within the function. The block 6 is a *functional* block in Figure 5, and a constant time can be assigned to it just like basic blocks.

The transformation of code will result in the following two possible paths (Figure 5) in each iteration of loop, that should be considered by the analyzer to detect infeasible sequences of paths taken in consecutive iterations.

```
a : 2 3 4 10 11          b : 2 3 5 6 10 11
```

The execution of loop will result in the following sequence of taken paths (abbbab)*, from which it is easy to identify that the infeasible sequences of paths are *aa, bbbb, abba, ababa*. The transformation results in reducing the search space for possible infeasible paths, to a great extent. Therefore the complexity of infeasible path detection as per the technique in [3] is greatly reduced and will result in a tight and safe bound on WCET. Even though there exists other infeasible paths when the paths inside the update functions are considered, such infeasible paths can be ignored in WCET analysis as every call to update function takes constant amount of time.

3. CONCLUSION

Detection of infeasible paths in a program is important for WCET analysis. However, it is difficult to detect all the infeasible paths in a program and moreover the search space for infeasible paths could grow exponentially in terms of number of branches in the program. Our technique can not only reduce the number of paths in the program by optimization but could also consolidate a group of paths into one path as far as WCET analysis is concerned. Thus we reduce the complexity of infeasible path detection while still maintaining the safe bound on WCET.

4. DISCUSSION & FUTURE WORK

Mueller and Whalley in [8] have also exploited the idea of restructuring the control flow and replicating code. However, they have used it for compiler optimization via avoiding conditional branches. Previously, Puschner in [9, 10]

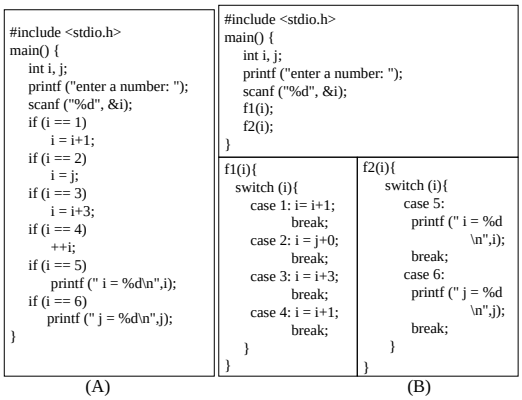

Figure 6: Example Code: Toy6

have also given a code transformation based approach to reduce the complexity of WCET analysis. The author has proposed a single path paradigm for programs so that there could only be a single path in a program hence making WCET determination simple. Such a transformation will have to trade a lot of performance with predictability. On the other hand, with our proposed technique, the WCET analysis complexity could be reduced to a large extent without much trade off in performance. Another work by Al-Yaqoubi ([4, 1]) also describes a technique to simplify the control flow of complex loops by partitioning the control flow into sections that are limited to a predefined number of paths. Each section is then treated by the timing analyzer as a loop that iterates only once. Using the same example **Toy6** as in [1] (shown in Figure 6(A)), we see that our transformation (shown in Figure 6(B)) can reduce the number of paths in Toy6 from 64 to 1, without much increase in the code length and still giving a tight prediction for time using timing analyzer as in [3]. Function `f1` in Figure 6(B) can be assigned a constant amount of time (equal to any single case of the switch statement), similarly function `f2` can also be assigned a constant amount of time and both `f1, f2` are treated as functional block while calculating WCET. Hence, our approach can reduce the complexity of control flow much better than that in [4], without trading of much in terms of code length and tightness of estimation.

Note that, our technique requires a modification in the actual code in order to reduce the complexity, which in case of some hard real-time systems might not be allowed. It should also be noted that our technique is not a timing analysis technique. It could be used as a **preprocessing** step to other infeasible path detection and timing analysis techniques such as [3, 2]. Our technique could reduce the complexity of other techniques and provide tighter bounds on WCET. Other techniques need to be modified in order to handle the functional blocks. However, at the present stage we do not have a concrete technique to determine the potential regions in the code which could be worked upon for transformation. For example, a certain type of *if* structures in the program can be optimized for reducing the paths as in the given example in this paper and also a group of basic blocks can be converted into a functional block by transforming *if* statements into a *switch* statement inside the new function. In our future work, we plan to come up with efficient methods to automatically determine potential regions for transformation.

5. REFERENCES

[1] N. Al-Yaqoubi. Reducing timing analysis complexity by partitioning control flow. Master's thesis, Florida State University, Tallahassee, FL, 1997.

[2] R. Bodik, R. Gupta, and M. Lou Soffa. Refining data flow information using infeasible paths. In *ESEC/SIGSOFT FSE*, 1997.

[3] C.A. Healy and D.B. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Transactions on Software Engineering*, 28(8), 2002.

[4] L. Ko, N. Al-Yaqoubi, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M. G. Harmon. Timing constraint specification and analysis. In *Software Practice and Experience*, pages 77–98, January 1999.

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

[6] Y. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 1995.

[7] Y-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM Design Automation Conf. (DAC)*, 1995.

[8] F. Mueller and D. B. Whalley. Avoiding conditional branches via code replication. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 55–66, June 1995.

[9] Peter Puschner. Transforming execution-time boundable code into temporally predictable code. In *Proceedings of IFIP World Computer Congress, Stream on Distributed and Parallel Embedded Systems*, pages 163–172, 2002.

[10] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91, January 2002.

[11] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, pages 132–140, 2001.

[12] Reinhard Wilhelm. Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone. In *VMCAI 2004*, volume 2937 of *LNCS*, pages 309–322, 2004.

Optimizing JVM Object Operations to Improve WCET Predictability

Angelo Corsaro[1], Corrado Santoro[2]

[1]Washington University
Department of Computer Science and Engineering
1 Brookings Drive, BOX 1045, St. Louis, 63130
Missouri, USA
EMail: corsaro@cse.wustl.edu

[2]University of Catania
Dept. of Computer Science and
Telecommunication Engineering
Viale A. Doria, 6 - 95125 - Catania, Italy
EMail: csanto@diit.unict.it

Abstract

This paper describes the optimizations introduced in Juice*, a J2ME virtual machine for embedded systems. These optimizations are designed to make possible the determination of the WCET of the JVM bytecodes related to object and array management. The solution proposed, which is based on subdividing the heap in a set of* chunks *of fixed size, allows to execute those bytecodes either in a constant time or in a linear time with an upper bound that can be determined.*

1 Introduction

In real-time systems, determination of the worst-case execution time (WCET) plays a fundamental role in task feasibility analysis and scheduling. Frameworks for WCET analysis [1] are based on determining the expected execution time of each instruction of the given task. In a real-time Java environment, this implies to obtain the WCET of each Java bytecode. Such an analysis could be hard for those bytecodes that need to manipulate Java heap or access the structure of involved objects, class/interface hierarchy, etc. In such a context, this paper describes the optimizations introduced in *Juice* [3], a J2ME virtual machine designed by the authors to be run upon NUXI [5], a light executive for Intel-based embedded systems[1]. Juice uses a heap management technique and an object layout that facilitate object allocation, object's attributes access and garbage collection. The employed technique allows to perform these operations in a predictable time. The paper focuses on object allocation/deallocation and attribute reading/writing, showing how these operation can advantage of the proposed heap management technique.

[1]NUXI can be downloaded at `http://nuxi.iit.unict.it`

2 Heap Management

Operations related to heap management are those executed when an object has to be allocated or collected. In general, the time required to perform the creation of a new object depends on the size of the object that, in turn, depends on the amount of attributes declared in the object's class and in its ancestors. The operations required for object allocation can be summarized as: (*i*) determine the number and the type of the attributes, in order to compute the size of the memory area to allocate in the heap, and (*ii*) find a *contiguous* area of free memory, in the heap, where to allocate the created object.

The former operation could imply to navigate class hierarchy in order to find all the attributes the object possesses; indeed, number of attributes can be computed at class loading time, thus storing object size in a field of the structure representing the class in memory. The latter operation instead implies to scan the heap until a piece of free memory, whose size is greater than or equal to the requested amount, is found. This operation requires, in general, a time dependent on the size of the heap and of the object [4, 6]. This means that the WCET of such an operation cannot be exactly computed, but only upper bounded with a limit that depends on heap size.

To overcome the problems above, we propose a technique that, by borrowing some principles from Unix-style file system handling, provides an efficient algorithm to allocate any object in a time that depends only on the size of the allocated object, a parameter that can be exactly estimated with a static bytecode analysis. Our solution subdivides the entire heap in

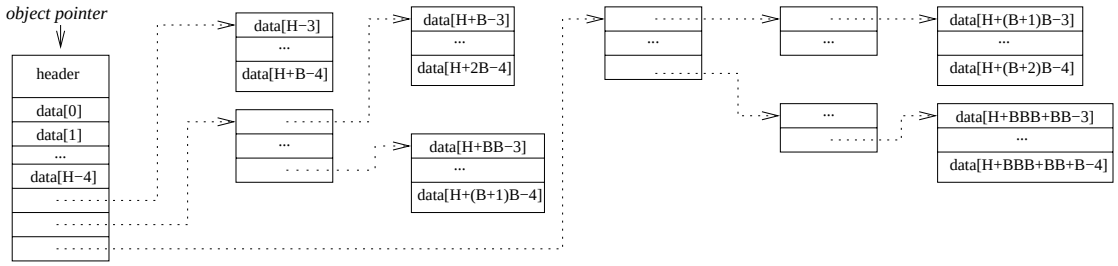

Figure 1. Object Data Allocation Policy

a sequence of *chunks* of a fixed size CS. All chunks are organized in a linked list, started by a pointer FC representing the *free list* of chunks (the first four bytes of each chunk represent the pointer to the next free chunk). Allocating a chunk means to pick it from FC, moving the latter to the next free chunk; while releasing a chunk implies to place it at the head of the free list, thus updating FC accordingly.

Using such a memory layout, allocating an object, given its size S, implies to pick a number of free chunks equal to $NC = \lceil \frac{S}{CS} \rceil$, operation that does not require to walk the heap and whose duration can be predictable. Similarly, releasing an object implies to return allocated chunks to the free list.

With such an allocation policy, the main issue is that the allocated chunks could not be contiguous and object's data could be spread over different chunks; a technique to suitably link those chunks together is thus needed, and it also must take into account that object's data access has to be fast and predictable.

2.1 Object Allocation Policy in Juice

In Juice, a Java object is composed of a **header**, which contains information such as the pointer to the corresponding class, the object's monitor, etc., and a **data**, which contains the array of object's fields. If the object is an array, **data** contains the array elements. We make the following assumptions: (i) chunks are double-word (32-bit) aligned, thus CS is a multiple of 4. We call $B = \frac{CS}{4}$ the number of d-words of a chunk[2]; (ii) the chunk size is greater than the size of **header**, i.e. $size(header) < CS$; and (iii) **header** is structured in such a way as to be double-word aligned, we call $H = \frac{CS - size(header)}{4}$ the number of d-words left in a chunk after the object header. When an object is small, i.e. $size(header) + size(data) \leq CS$, a single memory chunk is enough; otherwise, the first part of the chunk is filled with **header** while **data** is placed in the remaining chunk part and in other chunks linked using a hierarchical structure of forwarding pointers as depicted in Figure 1. In particular, d-words from 0 to $H-4$ after object header store the corresponding elements of object's data, while d-words from $H-3$ to $H-1$ are used as single-, double- and triple-indirection links to other chunks, each one containing B data elements. Therefore, as detailed in Figure 1, d-word $H-3$ is a pointer to a chunk containing elements from $H-3$ to $H+B-4$, d-word $H-2$ points to a chunk containing *pointers* to chunks containing *elements*, etc.

[2]This choice is due to the fact that most of the JVM types are 4-bytes long (integer, floats, object and array pointers, etc.).

3 WCET for Object Operations

3.1 Object Allocation

In traditional heap management techniques, the time required to allocated a new object depends on the sizes of both the heap and the object to allocate. In the proposed approach we need to pick a number of chunks, from the free list, equal to:

$$1 + \left\lceil \frac{n - H + 3}{B} \right\rceil + \lfloor log_B(n - H + 3) \rfloor + \left\lfloor \frac{n - H + 3}{BB} \right\rfloor \quad (1)$$

where n is the number of object fields or array elements. This number depends only on object size and, if B is a power of 2, it can be easily calculated using bit-shift and *if* instructions.

Using the relation above, the WCET of the `new` bytecode can be exactly computed. The only exception is the use of the `Class.forName()` construct to load and instantiate a new object; in this case, the type of the object—and thus its size—is unknown until runtime and the WCET cannot be exactly computed: only an upper bound can be determined by assuming a reasonable maximum number of attributes that an object could have (in Juice, we assumed that an object cannot have more than 255 attributes).

3.2 Array Allocation

Java treats arrays as objects: an array of elements of type "T" is treated as an object of class "[T" ("array of T"). For this reason, the structure of an array, in Juice, is the same of an object, given that it has no attributes and the **data** part is used to represent array elements. Allocating an array, given that its size is known, implies to perform the same operations done for object allocation, and thus the calculation of the WCET is subject to the same formula 1. However, if the array size is known only at runtime (and this could happen very often), a different approach is needed. Indeed, this is a common problem of WCET computation in presence of dynamic arrays, and should be solved with other well-known techniques, such as by determining an upper bound, using annotation, etc. [1].

3.3 Reading/Writing Attributes

Reading and Writing object attributes is performed, in Java, by means of the bytecodes `getfield`/`getstatic` and `putfield`/`putstatic`.

Since Juice is a virtual machine for embedded system, Java classes are intended to be "ROM'ized". To this aim, Juice adopts an ahead-of-time pre-link and resolution process that, together with transforming classes into a ROMable representation, replaces each `get-`/`putfield` attribute with the "quick" version. Attribute index is thus referred to the array stored in the object. Given that attributes can be spread over different chunks, the access could require to navigate the chain of pointers. The code of such an operation is reported in Figure 2 for the `getfield` bytecode: as it can be seen, it is fast and its WCET can be exactly determined.

3.4 Juice Heap Layout and Garbage Collection

One of the main known issues that impede the use of Java in (hard) real-time environments is the presence of the garbage collector. The instants in which the GC is activated and the duration of its execution cannot be predicted, and thus any WCET/schedulability analysis is, in general, impossible. Such problems are overcome by the *Real-Time Specification for Java (RTSJ)* [2] with the introduction of *scoped memory*.

In our approach, the use of memory chunks greatly simplifies garbage collection, independently of the particular algorithm that is then used (reference-counting, three-color-marking, etc.). In fact, chunks are fixed-sized and free chunks are organized in a linked list, therefore *no compacting process is needed*. Collecting

```
dword getfield_quick (HOBJECT p, int index)
{
  dword * p1, * p2, * p3;
  if (index < (H - 3)) return p->data[index];
  index -= (H - 3);
  if (index < B) {
    // follow index at H - 3
    p1 = (dword *)p->data[H - 3];
    return p1[index];
  }
  index -= B;
  if (index < B*B) {
    // follow index at H - 2
    p2 = (dword *)p->data[H - 2];
    p1 = (dword *)p2[index / B];
    return p1[index % B];
  }
  // follow index at H - 1
  index -= B*B;
  int i0 = index / (B*B);
  int i1 = (index % (B*B)) / B;
  int i2 = index % B;
  p3 = (dword *)p->data[H - 1];
  p2 = (dword *)p3[i0];
  p1 = (dword *)p2[i1];
  return p1[i2];
}
```

Figure 2. Juice's `getfield` code fragment

an object no longer used implies to return the associated chunks to the free list, one-by-one, operation that can be performed also *incrementally*, n chunks per time. Using such a characteristic, the garbage collector of Juice[3] is designed to perform a *known number* of operations each time it is invoked. More specifically, the cost to pay when an object occupying n chunks has to be allocated is to ask the garbage collection to free, at most, n unreferenced chunks. With such an approach, execution of the GC is always tied to object allocation (i.e. when new free memory could be needed) and its duration can be predicted.

4 Known Issues

The heap management policy presented in this paper, even if it guarantees good allocation performances and predictability in WCET determination, suffers of two main problems: *limited number of fields/array elements* and *memory fragmentation*.

4.1 Limited `data` elements

As shown in Figure 1, the maximum number of elements the **data** part can refer is $H+BBB+BB+B-4$. In Juice, where we chose $B = 32$ (and thus $H = 24$), this limit is equal to 33844. It is enough for object's

[3] In the current implementation, the GC is based on a simple reference-counting

attribute, but it could be a problem for array allocation. A possible solution could be to increase B, but, as we will see in the following, this choice provokes an increment of memory internal fragmentation. The solution adopted in Juice is to flag large arrays with a bit in the **header**, and add another level of indirection in the **data** array. This implies an upper bound equal to $H+BBBB+BBB+BB+B-5$, i.e. 1082419 elements when $B = 32$. This new upper bound can now be considered enough for embedded applications. However, the introduction of such a variation implies an additional cost in accessing array elements, which is useless when the limit of 33844 elements is not overcome by the application: thus, in Juice, a command-line flag is used to activate the "large array" option.

4.2 Memory Fragmentation

The proposed approach provokes both external and internal fragmentation. The former is due to the fact that an object could be spread over non-contiguous chunks: this does not fit the working scheme of a CPU cache and thus can lead to performance reduction. However, we remind that, in general, the use of caching could be a problem for (hard) real-time systems, since caches may introduce large jitters in CPU opcode executions thus affecting WCET calculation.

Internal fragmentation, derived from the unused space left in chunks, is instead more important, since it implies a reduction of the amount of available memory. For this reason, the value of B should be chosen in such a way as to find a good compromise between the allowed *maximum number* of object's attributes and array elements, which is $M = H+BBB+BB+B-4$, and the degree of internal fragmentation. Figure 3 reports the trend of M and the amount of wasted memory, due to internal fragmentation, with respect to a value of B ranging from 16 to 128 d-words. The amount of wasted memory is measured considering 10, 50, 100 and 500 allocated objects with no attributes, thus producing the maximum fragmentation. As Figure 3 reports, the wasted memory with 500 objects, choosing $B = 32$ as in Juice, is approximatively 42 KBytes, a not-so-high cost to be payed for the use of fixed-sized memory chunks.

5 Conclusions

This paper described the heap management and object allocation techniques employed in the Juice virtual machine. As it has been shown, the proposed approach was studied in order to have operations for object allocation and access not only optimized but, above all, with a predictable execution time, making possible the determination of WCET.

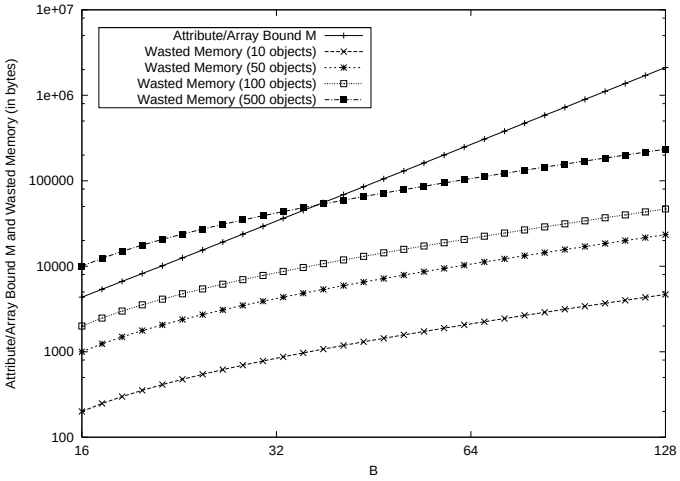

Figure 3. Wasted Memory due to Internal Fragmentation

References

[1] I. Bate, G. Bernat, and P. Puschner. Java Virtual-Machine Support for Portable Worst-Case Execution-Time Analysis. In *Proc. 5th IEEE ISORC 2002*, pages 83–90, Apr. 2002.

[2] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[3] A. Corsaro and C. Santoro. A C++ Native Interface for Interpreted JVMs. In *1st Intl. JTRES Workshop (JTRES'03)*. LNCS 2889, Springer, 2003.

[4] S. M. Donahue, M. P. Hampton, M. Deters, J. Nye, R. Cytron, and K. Kavi. Storage allocation for real-time, embedded systems. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software: Proceedings of the First International Workshop*, pages 131–147. Springer Verlag, 2001.

[5] C. Santoro. *An Operating System in a Nutshell*. Internal Report, Dept. of Computer Engineering and Telecommunication, UniCT, Italy, 2002.

[6] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.

Session 2: Low Level Analysis

Session chair: Stefan M. Petters (University of York, UK)

Presentations

A discussion of the influence of scratchpad memories on the WCET analysis has been subject of the first presentation given by Lars Wehmeyer. The authors came to the conclusion, that the WCET can benefit considerably from scratchpad memories, with no extra effort on the analysis.

Jürgen Stohr presented a method to control the influence of PCI DMA transfers on the WCET. Focus of this work is a PC style architecture and how to make that more suitable for real-time applications.

Synergetic effects of cache related preemption delays were presented by Jan Staschulat. This centers around multiple executions of the same straight line code as it is used in the automotive industry and how that can be analyzed with much better results than previous simplifying approaches.

The last talk by Oleg Parshin described a method to analyze instruction caches for individual components and how the results for those components can than be integrated into an overall WCET approach when the components are composed to form systems.

Discussion

The following discussion centered on the feasibility of componentization of real-time systems and the impact on WCET analysis.

Peter Puschner: (To Oleg Parshin) In this analysis you say you analyze all procedures and this means even the first step of the analysis is for a concrete memory architecture. So is there a step which abstracts away from the concrete layout and size of cache?

Oleg Parshin: The cache line size needs to be known.

Peter Puschner: Did you try to store the actual memory references in some more abstract way? For example a procedure is used in several different contexts with different memory layouts and by storing the information in a more abstract way, you could reuse information gathered in previous analysis steps.

Oleg Parshin: We used the AbsInt tool, hence had no influence.

Stefan M. Petters: But technically it should be possible to have two analysis steps, use abstract information about memory regions used and then come up with the final result in a computational stage instantiating the analysis results.

Oleg Parshin: You need to know concrete size of the cache.

Stefan M. Petters: That's when you abstract away from the relative location and just say, even if you fix the cache size you can just say, within the module I'm doing the about that and when I use it in some context I need to know where it's actually located, but that should be more troublesome.
On a more general scale we have heard about scratchpad memories twice today. Does anybody know about the industrial uptake of those, compared to locked caches, which is the obvious alternative?

Christian Ferdinand: In most recent developments you have the choice: You have some amount of memory which may be either used as scratchpad memory, or cache and this may be configured at boot up, or even split it some this and some that way.

Stefan M. Petters: But is it really used?

Christian Ferdinand: In automotive it is truly used, because they start of with a design with an expected life span (in terms of delivery) of 6 years, but after 3 years they get so many requests for changes, that whatever it possible with that piece of hardware will be done. But not automatically, as they currently don't know how to use this.

Iain Bate: Where you use abstract interpretation in aiT style tool for cache analysis, one thing we have found working with industry industrial use of poll status true/false for runtime exceptions they use abstract interpretation. In practical code they get a lot of *mays*, which need either manual inspection or a safe estimate. While this is a very different problem, when you are using this actually for cache analysis, what percentage of memory accesses do you get as *mays*, because that can be seen as wasted resource and one would try to cut that down.

Christian Ferdinand: The problem is this is a really good question that comes up all the time The problem is that really depends on the application. One can write application code, where this prediction would be extremely poor….

Iain Bate: I'm talking about well written code.

Christian Ferdinand: For example, we as a company only provide the tools, but have not much insight into the application, but there is one example from Airbus, where they applied it on what they call real-time benchmark, which is basically an old version of the fly-by-wire code of the 340. This is generated with the SAO Scade tool mainly and we had a look at the Coldfire 5307 with 8 KB. One problem is no-one really knows what the real WCET is, but what they did is a lot of measurements and they also used a legacy method based on measurements and a lot of argumentation what could be the WCET. Our result was between the results of the legacy method and the measurements. They believe they had about 20% overestimation with the legacy method and if we assume this correct, we get about 10% overestimation.

Iain Bate: I'm not talking about the overestimation and the overall execution time analysis results. It's actually how many memory references you get as mays. Say you have got a million memory references, what percentage?

Christian Ferdinand: I do not know precisely where the overestimation comes from. Could be pipeline analysis or non-precision of the cache analysis. It's a factor of 8 between a cache hit and a cache miss. If you assume all the 10% overestimation going into cache analysis, you come up with 1% or 2 % of wrongly predicted mays, so it's very precise, it's 99% precision in that specific case.

Stefan M. Petters (to Jan Staschulat): Setting aside future computational problem, how do you think you can get that to scale as it is now, in terms of looking for a branch. The examples you had were not too big and with the direct mapped caches or two way set associative caches you have been on the lower end of things. So how would you think you can manage the bigger problems?

Jan Staschulat: The high complexity is due to the analysis of multiple preemptions, because several cache states have to be analyzed in the control flow graph. To simplify one can abstract from specific cache states and merge several cache states. This will reduce the analysis complexity but also lower the precision. So the question is: *"how much precision can you get and how much effort you are willing to pay"?*

Peter Puschner (to Jan Staschulat): Is there any first others? Have you ever measured the time you needed to solve the problem?

Jan Staschulat: They are long, indeed. Several hours.

Peter Puschner: Even for your small examples?

Jan Staschulat: Yes. We are thinking about a different implementation. The analysis of several preemptions has to be simplified. Instead of considering all paths of a preempting process, several cache states should be merged leading to a reduced number of states. The underlying concept of data flow algorithm considers than less states and the analysis complexity is reduced.
But what is really necessary is the analysis of multiple process activations, because this is the loop of straight line programs of automotive applications. Since a cache is not useful for linear programs, a second program execution can be seen as a loop, and consequently cache lines are reused.

Alexander von Buelow (to Jan Staschulat): Does you approach also work for interrupts or are they not considered?

Jan Staschulat: At the moment interrupts are not considered. Interrupts can happen any time, as often as you wish. You have to assume it occurs n times in a given time period, and the user has to specify this information somehow. I don't know how to consider interrupts in this analysis yet. You might want to consider jitter also.

Stefan M. Petters (to Jan Staschulat): In your data regarding your multiple preemptions, you just compared your results to previous approaches, but didn't actually get to the point what the real simulation results were in context of an arbitrary number of runs of a program with a specification of the number of preemptions.

Jan Staschulat: I didn't show the simulation results in this table, but I ran some experiments with the ARM Developer Studio and I instrumented the debugger to stop the preempted process at a preemption point and run the preempting process and then let the preempted process continue. I did this for all combination of n given multiple preemption points. The comparison showed that our CRPD analysis was close to the simulated results. Of course this verification by simulation works only for small examples.

Jan Gustafsson (to Lars Wehmeyer): I have another question to this scratchpad idea. It's actually two questions: The first one is: *"how does your method scales with larger applications"?* and the

other question is: *"your analysis method is part of the compiler?"*. What are your opportunities to get this into a real compiler? Any thoughts about this or contacts? Because obviously this has to be done inside the compiler to work. I think the scratchpad idea is really great, but how is the future?

Lars Wehmeyer: Regarding the first part I guess your question aims at the direction of ILP and complexity issues. Right now we're using the simplex ILP solver and we are formulating our models accordingly as an ILP and then giving that to the solver. But we have been discussing this problem with other groups at our University, especially theoretical computer science, and we have come to the conclusion that what we describe are actually knapsack problems and they belong to the group of packaging problems. Usually you can find a linear time approximation and you can also find a polynomial time approximation. So there is the possibility by trading in some of the precision of the final solution for the execution time. Using these approach and thoughts for scaling it to larger benchmarks and larger applications should not be a big problem. So you have to find out how far you are of the optimal solution and how much computation for finding an allocation of objects to the memory are we willing to accept in order to get closer to a certain precision.
And the second question, right now the approach is integrated into our research compiler, but I guess, in order to really integrate it into any other compiler, all you would need is the information about the analysis that was performed, you would need either a static analysis or profiling and I think with that information and an energy model which models the environment and you know the energy required to access the different types of memory and you know about execution frequencies it should be actually possible to integrate it more or less into any compiler. Of course you would need to find some way of interfacing, but actually it is just a way of telling the linker where to place the different objects in memory. That's all you need to place a certain memory object onto the scratchpad. You just tell the linker, put this to that address and that address is mapped to scratchpad memory. So I guess, by cleverly interfacing knowledge you have within the compiler and having analysis regarding the runtime behavior of your application, externally solving the problem and then back annotating the information you get from solving the problem into your compiler, maybe it could even be done by solving the optimization problem and then using a linker script so it's not that deep within the compiler.

Iain Bate: I just wonder, how many of those working on scratchpad memory have had a look at the work SoC people have done on optimizing performance and power characteristics? People like Frank Vahid or Tony Givargis are still on this work. It's because they have done an awful amount of work in their domain. They have not combined it with execution time analysis, very much like testing systems in order to see what the worst case performance is but they have done a lot of work in how to configure your memories, caches and scratchpads for the best effect. I wonder whether anybody has looked at that communities work.

No one!

Peter Puschner: I would like to make a comment on this work of modularized or reusable WCET analysis: I think this is really an interesting piece of work and I would like to encourage you to push this further, because I think this is really one of the open questions on WCET analysis: How to reuse information for timing analysis? You mentioned the software development process and what we all do is, we compile modules and we don't compile all modules again and again, but we compile only the modified ones and link them with the others. I think this should also be true for WCET analysis. If we analyze a piece of code, we should be able to store this information in a way that can be used later on and make the analysis easier. So you don't want to do the whole complex analysis again and again, but rather do the complex analysis tasks only once and then

store the information in a way, so we can reuse our results cheaply, so we don't have to go through all the analysis and all the modules, functions and libraries again and again. And I think this is one step in that direction.

Stefan M. Petters: Another step in that direction, which is upcoming work looking at real-time components on the vendor side. A real-time operating system, where you don't have the actual hardware it's running on, so you want to provide as much information in terms of timing as you can and then instantiate that on a particular piece of hardware or a particular environment and I think that is certainly worthwhile.

Guillem Bernat: Can we have a feel of what is the impact of these recompilations of the WCET? From your experiments, what is the difference in the execution time of a module when you execute it somewhere else? Is it for example, a 1% or a 50% difference?

Oleg Parshin: I can't tell you exact numbers.

Guillem Bernat: When you compile a module and recompile it you expect the functional behavior to be the same. The answer to Peter is: You can do that if you assume that if you add some small module somewhere else does not change that much and therefore you are sort off in the same dimension, then you can, in all other cases you have to do a full analysis end to end. The question is the ability to do it component-wise is only true if the components have a meaning and they don't really depend too much on the context they are running in.

Peter Puschner: Yes, in some way. Of course, if you design a system it depends on the interfaces. In this way, the degree in which you can decompose the problem depends on the interference of the different parts and of course, if the interference is too large, it might not be worthwhile, but then the question is whether the solution is the right solution. We can't recompile and reanalyze all this. If this takes an hour, you would not want to do it every time you change a module. If it really takes an hour, then the solution is probably not the right solution: Isn't the architecture wrong?

Jan Gustafsson: One way of dealing with this could be to analyze parts, whether they are context sensitive or not and if they are not then you don't have to do this all over again. That could be a part of the analysis.

Stefan M. Petters: It's actually a question whether you want to go down the route and separate the context non sensitive from the context sensitive stuff analyze one and wait for the other to happen or whether you split the analysis rather than the application into bits you know and things you can't know at this stage and try to get the second stage as small and simple as possible, because you don't want to have engineer XYZ sitting somewhere and trying to figure out what you initially were trying to do.

Iain Bate (to Guillem Bernat): I've got a question for Guillem Bernat: the WCET analysis work presumably subject to regression testing and trying to carry all those many measurements as possible forward for a task once you do a change to that task that is important. Have you thought how you might handle that?

Guillem Bernat: Well, that's exactly why I asked the question. In measurement based analysis we assume that the impact on the execution time once you make changes to the code are actually quite small. So you can reuse a lot of measurements on your code, unless there are some pathological cases. The issue is, the approach we are actually using is you test your components

and carry that forward, but once you have your system finally assembled, you do a full analysis of the whole system once, assuming there is no more changes on the code.

Iain Bate: Because what happens in the functional testing world, they make unit tests only on the units which have changed which is isolated from the rest of the system and then they do very limited progressively less integration tests for the number of changes, so they do some scenarios of functional testing, so in the end this assumes that the changes don't ripple through too much.

Guillem Bernat: You don't even need to recompile the code, you just relink it with a slightly different link map and then all the addresses change and then there is allegedly nothing you can do. I think the point here is – going back to my point of different systems – in a absolute safety critical system, in order to get meaningful values, you will completely redo an end-to-end analysis, any time you introduce a small change. If you have a system with less criticality, like the automotive industry, then they accept there is an error on that value, but it's more or less in that value and the error is not an order of magnitude. From my experience they know that small changes in the code won't double the execution time. You get an extra couple of cache misses but that's what it costs.

Peter Puschner: I'm a bit surprised to hear this, because my guess would be: If you do some static analysis and also analysis of interferences as proposed, and you change a module, you would be able to see the effect of this change on the interference, because I see now I would use this and that cacheline too, which I didn't have before and therefore I see there is now an interference with module X. But if you measure, how would you be able to see this interference, because you don't go into the detail?

Guillem Bernat: You don't know where it is, but you observe the effect and that's the philosophy change, the change of mentality. A large project with a couple of 10.000 lines of code, the number of memory accesses made is in the order of millions, hundred millions. The people who design caches are very clever and they have been making for a long time something, which guesses the impact of the memory accesses. So yes, you change a module, hence something that was a cache miss becomes a cache hit, but the miss happens now somewhere else. Statistically speaking after a very large number of memory accesses this averages out. So we say: a good cache mechanism hits about 9x% would you expect that a change in the program makes your cache miss 50%?

Peter Puschner: But that takes us away from hard real-time, right?

Guillem Bernat: That's what I'm saying. There is an absolute guarantee somewhere else and as soon as you move over edge of the absolute safety critical then you are at this gate.

Iain Bate: An observation from a Federal Aviation Authority perspective. They are very concerned about the use of caches, the multimode sort of things like pipeline effects, because of these sort of problems and therefore a lot of work looks into what the issues are. A lot of systems are just avoiding caches wherever possible turning them off, etc. Some of them are using them, but they haven't got a grip on what to do. Things like CAS20, which is the guidance document, still funding more and more work in this area on how to handle this problem and not just from the execution time analysis perspective.

Peter Puschner: Just to clarify that: You don't observe the task you have modified in isolation, but again you are looking at the whole system. So you don't just make a local analysis after a

change, but you have to go back to your global analysis of the whole system to get out the knowledge of all the interferences, which is a more complex problem.

Guillem Bernat: Let me take a step back. We are doing measurement based WCET analysis as opposed to static analysis. Measurement based analysis does not address the issue of what happened to that particular cache line, but you just measure the execution time and if it happens to be longer, then you can assume that maybe there was an additional cache miss or something. Now when you move away from characterizing each individual memory access that's when the large numbers are on your side. That's the first piece. Secondly, when you do testing or when you analyze a small component, there are two steps to manage this. One is to determine how long it takes for a component to execute in isolation and the second step is in the high level analysis to put all things together. But those are completely separate processes. This is not linear, but N log (N) with the size of the program and is independent on the testing. Even if you do it manually, putting together all the information would not take longer then the time to compile your code. At least it's in the same order of magnitude. Deriving the information of each module is what takes time, but that is linear with the complexity of the module, so there is no computational explosion, but depends on how you do your testing. You could do 4 weeks of testing and would get much better data, but you want to separate the issues of deriving information on the module and how to put that information together. So that's where one comes up with the capability issue. In fact, one can even deal with the problem of a whole subsystem being not developed. Because you say "I assume that module X is going to take that much and that's a design estimate" and then can reason whether one is 300 times over budget or within budget. The key point here is, if you move away from the absolute, absolute guarantee that you get the absolute, absolute worst case life is much easier. Besides no safety critical system had that information.

Peter Puschner: And probably not as safe.

Guillem Bernat: All safety critical systems have the built in assumptions, that a chip will fail, data corrupted etc.

Iain Bate: The issue is showing that this is not a cause of failure, one can accept one part of a redundant system to fail, but you don't want all of them fail at the same time. Since in those systems, the redundant parts are mostly working in locked step, your assumption of having statistics on your side does not hold, because they are supposed to behave identical.

Peter Puschner: I think you are right that people assume that systems or parts of systems fail, but what they usually do is they try to avoid to introduce any sources of failures themselves. In order to reduce complexity, and one strategy to do that is to separate the system into components and try to make them as independent as possible, to keep the interfaces and interferences really small. What you are trying to do is, to develop one component and try to verify correctness including temporal correctness and then rely on what you got. If you then compose these components, you want to be sure they still work. You want to have that for timing not only for functional aspects. In the moment I start to neglect the interfaces and interferences, if you just measure, you avoid the view on the interference on purpose. You just neglect that.

Guillem Bernat: Not really I think that's a different thing. You measure module A and measure module B and then you put A and B together considering the worst possible interference and that's the kind of thing we do. The probabilistic analysis framework allows you to reason: "*What is the worst possible combination of this and this?"* Then you can actually ask whether this combination is feasible or whether you can reduce the pessimism.

Peter Puschner: If you want to argue about it. It has to be simple, otherwise it's too complex. You take two tasks and can hardly do it. If you just measure it's still a probabilistic statement, but you can't prove anything.

Stefan M. Petters: In the end, I think what drives all of it is cost. Assuming one can pour 200 million dollars into each individual plane to make it 100% save or one can be 100% - 10^{-20} sure and save that money, I see people being willing to go down the route of saving that money.

Guillem Bernat: We had an argument with someone saying: *"Well, worst case is too pessimistic for us! We want our testing, we don't believe it.*" Testing is too optimistic, the absolute worst case is too pessimistic and they want to have something in between. If it is good enough, if it is less likely to fail than any other component of the system, then that's good enough for them.

Peter Puschner: Let me refer to a different area where quite the opposite of what you claimed happens and that's when you look at communication protocols. For cars, where, for example, CAN seemed to work quite well for a very long time and now companies are getting into trouble and they start to see that what they have to use and they are all now getting more or less into the time triggered business, because they realize that building on probabilities gets them into trouble when complexity reaches a certain level. So now they go down and say: *"We can't do that anymore. We have to have systems, which are decomposable, where we have an isolated view on each of the components and where we reduce the interference between components really to a minimum"*.

Stefan M. Petters: But they have still a probabilistic argument for a component failure.

Peter Puschner: Of course, there are always hardware failures. But what they try to avoid is to build in probabilistic failures themselves, because then it's difficult to argue about dependencies of things happening.

Stefan M. Petters: But there's still the question of who has to foot the bill if something goes wrong.

Iain Bate: An interesting thing is, how big a guarantee do you want? And what that comes down to is cost of human life. When I have done enough assessment for a system in, let's say, avionics and then it comes down to issues, like how many people are on the aircraft, which countries is it flying over, because there are unfortunately certain differences, in what a nationals life is worth, and in follow up what is the cost of litigation in court. What they do is certainly a trade off between the costs of safety, because they come to the point where they are happy to write off these costs of litigation if they loose an aircraft. The question is: *"how far are we willing to go?"*, because there is no such thing as absolute guaranties. There are always probabilities, even in static analysis it's still probabilities. For example your analysis tool aiT. You are relying on the compiler your compiling your tool with, do you have formal proof of the compiler? No. Do you have formal proof of the processor model? No. You have to rely on the documentation and observations. So it all comes down to probabilities of cost of life.

Stefan M. Petters: But it's more complicated like that. Then you have authorities like the FAA, which just want to make sure, they can't be sued and don't have to foot the bill to pay for the development. If you have those agencies involved it's a bit more tricky.

Iain Bate: You can't sue the FAA. They don't hold responsibility for life, that's the thing of the aircraft companies and operators. The FAA has no legal responsibility. In the military world it's different though, because the MoD may be sued.

Guillem Bernat: In the end the question is: "How much is enough?" I can do a very rough testing on a simulator and that tells me that this code executes for 1000 cycles my deadline is a million cycles, that's good enough. You would consider that enough. The tools are not reliable, but nevertheless you are sure enough it does not blow up. If you need something which costs a lot of money and time, which does not improve that statement, then people just don't use it. If it's clearly schedulable, that's OK. Even if it's clearly un-schedulable, that's also OK. In the cases at the borderline is when you make the distinction between the application domains, whether simple end-to-end measurements are OK or whether you need a very elaborate method.

Influence of Onchip Scratchpad Memories on WCET prediction*

Lars Wehmeyer, Peter Marwedel
Embedded Systems Group, CS Dept., University of Dortmund, Germany
{Lars.Wehmeyer, Peter.Marwedel}@udo.edu

August 31, 2004

Abstract

In contrast to standard PCs and many high-performance computer systems, systems that have to meet real-time requirements usually do not feature caches, since caches primarily improve the average case performance, whereas their impact on WCET is generally hard to predict. Especially in embedded systems, scratchpad memories have become popular. Since these small, fast memories can be controlled by the programmer or the compiler, their behavior is perfectly predictable. In this paper, we study for the first time the impact of scratchpad memories on worst case execution time (WCET) prediction. Our results indicate that scratchpads can significantly improve WCET at no extra analysis cost.

1 Introduction

For the currently available technologies, there is an increasing speed gap between processor speeds and memory speeds. Caches are being used in order to bridge that gap, especially in PC-like systems. However, the approach used in such systems has some disadvantages for embedded systems:

1. Caches are known to be one of the main contributors to the total energy consumption of systems [1], and
2. Caches are typically designed to improve the average case access time.

Analysis techniques to determine their contribution to the worst case execution time are complicated and, for some replacement policies, just missing. Scratch pad memories (sometimes also known as tightly coupled memories) are small memories mapped into the address space of a system. They are used whenever an address is within the address range assigned to that memory. Scratch pad memories are more energy efficient than main memories (since they are smaller) but also more efficient than caches (since only the required information is read from or written into the scratchpad memory). Scratchpads are currently being used by designers in a very ad-hoc fashion, and a comprehensive methodology of how to use them is, surprisingly, still missing.

Earlier work proposed compile-time algorithms for mapping hot spots of applications to scratchpad memories. This work was mainly motivated by the resulting energy savings, much of which result from a reduction of the average access time. However, the algorithms also have an extremely beneficial impact on worst case execution time estimation: it is fully predictable which memory will be used for a certain memory access. Hence, scratchpad memories provide 100% predictability concerning the timing of memory references. This predictability is explored in the current paper. In this work, we combine views from three different perspectives: an architectural view on scratchpad-based memory structures, a compiler view on how to map hot spots to these memories and a real-time system view on the resulting WCET. To the best of our knowledge, it is the first paper that provides a detailed analysis of the impact on the WCET of optimized mappings of applications to scratchpad memories.

2 Related Work

Many architectural features are included in modern microprocessors in order to meet the customers' demand for high average case performance. Especially in embedded systems having to meet real-time constraints, this is in general not very helpful, since the inclusion of pipelines, caches and branch prediction units makes it more difficult to predict a guaranteed upper bound for worst case exeution time [2]. Complicated analysis tools have been developed and are in use to shed light on the effect of these architectural features on WCET (see [3] for an overview). The difficulty lies in the fact that e.g. for caches, the hardware detects at runtime whether a memory access results in a cache hit or miss. In order to predict this behavior during WCET analysis, the worst case behavior of the cache has to be determined for the considered application. Several analysis methods have been proposed for instruction caches [4, 5] as well as for data caches [6]. The aforementioned publications solely deal with inclusion of caches in WCET estimation, which shows the considerable analysis effort required to predict cache behavior.

aiT [7] is a software tool that can help developers of safety-critical applications to verify that their programs will always meet the specified deadlines. This is done by determining an upper bound for the worst case execution time of the application. aiT guarantees the generated WCET results to be safe, which is generally infeasible using simulation techniques alone. Also, aiT abolishes the need to perform time consum-

*This work has been sponsored in part by EU-project ARTIST2

ing simulation runs in order to determine typical performance values. The aiT WCET analyzer has been designed according to the requirements of Airbus France for validating the timing behaviour of critical avionics software.

Scratchpad memories are being used as an alternative to caches due to their performance and their reduced energy consumption [8]. Scratchpad memories do not have a hardware to control their contents at runtime. Therefore, the assignment of memory objects to the different memories has to be handled either by the programmer or, in an automated process, by the compiler, who can analyze memory access patterns and distribute objects accordingly. The scratchpad can either retain the assigned memory objects throughout the running time of the application (static case), or the contents of the scratchpad may change at runtime (dynamic case). Allocation techniques to statically allocate data to the scratchpad were introduced e.g. in [9], whereas [10, 11] presented a dynamic approach for data and instructions, respectively. Further work concerning the utilization of scratchpad memories was conducted by [12, 13]. Both static and dynamic scratchpad usage are under full control of the compiler or the programmer, making the methods inherently predictable at compile time. In this paper, we will concentrate on the static allocation technique.

For the work presented in [14], the goal of the static allocation of both instructions and data to the scratchpad memory is energy saving. Therefore, an instruction level energy model for the used processor, an ARM7TDMI [15], was developed [16] and used in the *encc* compiler. The compiler determines the execution counts of functions and basic blocks and the number of accesses to variables in order to compute the most promising objects to be assigned to the limited scratchpad space. The actual optimization problem, which is similar to the well-known knapsack problem, is solved using an ILP solver. Then, the chosen memory objects are placed on the scratchpad, making control flow and address corrections where necessary.

3 Workflow

To determine a scratchpad memory's impact on WCET, we used the workflow shown in figure 1: The *encc* compiler generates an executable program which makes use of the available scratchpad. The memory objects allocated to the scratchpad memory are chosen according to the following algorithm (for details, please refer to [14]) which selects those elements with the highest benefit with respect to energy. In order to do this, all memory objects are weighted according to their execution or access frequency (for functions or data elements, respectively). The size of the objects is also considered, allowing the optimzation problem to be formulated as an integer programming problem as follows:

$$Maximize \sum_i m(i) * E(F_i) + \sum_j m(j) * E(Data_j)$$

subject to

$$\sum_i m(i) * S(F_i) + \sum_j m(j) * S(Data_j) <= SPsize$$

where $m(x)$ is a binary decision variable having the value $'1'$ if the corresponding object is allocated to the scratchpad and $E(x)$ is the benefit in energy consumption if object x is stored on the scratchpad instead of main memory. $S(x)$, the size of object x, is used in the constraints to ensure that the scratchpad capacity is not exceeded. In this form, the optimization problem can be solved using a commercial ILP solver [17]. The *encc* compiler then uses the solver's results to allocate the chosen objects to the scratchpad memory. The scratchpad size is varied in powers of two from 0 to a total of 4096 bytes in our experiments.

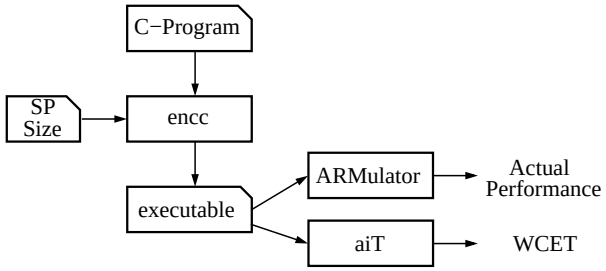

Figure 1: Workflow

The generated executable with the optimal set of objects allocated to the scratchpad is then fed through ARM's instruction set simulator *ARMulator* to obtain the number of actually executed cycles for the given input data set. Apart from this, the executable is analyzed using aiT [7] to determine an upper bound for the WCET (commonly called WCET) of the executable.

aiT supports the specification of memory regions with different attributes. The only relevant attribute for this work is the number of wait states that occur during memory accesses. According to the values measured for our ARM7 evaluation board, we assumed three waitstates for all main memory accesses and one wait state for scratchpad accesses.

To enable aiT to analyze the executable with memory objects allocated to the scratchpad, some annotations concerning instructions that use PC-relative addressing are required. These annotations ensure that the correct addresses will always be assumed during aiT's analysis. In order to keep the manual annotation overhead low, we decided to allow only the allocation of complete functions (i.e. not basic blocks and multi basic blocks as described in [14]) and data elements onto the scratchpad. This restriction can be easily overcome with a slightly increased annotation effort. The used toolchain supports this annotation process.

4 Results

The benchmarks used to explore the impact of a scratchpad on WCET are given in table 1. They comprise two speech encoding and decoding algorithms from the mediabench benchmark suite [18]. The programs were compiled with varying scratchpad sizes, as described in the previous section. The execution time is expected to decrease (along with the energy consumption) when the scratchpad capacity is increased. The effect of larger scratchpad size on average case performance and on WCET can be seen in figures 2 to 4.

The G.721 benchmark takes a little more than 2 million cycles to complete with our used input data on a system with only one main memory, whereas aiT estimates the WCET for

Name	Description
adpcm	Speech encoding and decoding using Adaptive Diff. Pulse Code Modulation
G.721	Speech encoding and decoding, reference implementation of the CCITT
Multi_Sort	Combination of sorting algorithms

Table 1: Benchmarks

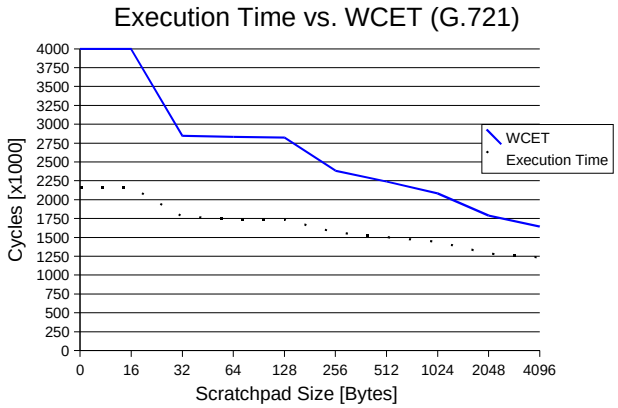

Figure 2: Results for G.721 benchmark

the worst case input to be about 4 million cycles. Since it is in general not possible to determine the worst case input data set for an arbitrary application, using a simulation approach is not feasible to determine a guaranteed upper bound for WCET. As can be seen in figure 2, increasing the scratchpad capacity not only improves the average execution time, but also has a strong positive effect on the WCET estimate. Where average case execution time is reduced to about 1,250,000 cycles for a 4k scratchpad, corresponding to a reduction of 43%, WCET reduces down to 1,650,000 cycles, which means a reduction of 58% compared to the inital case with no scratchpad. Thus, the effect on WCET is even greater than the effect on average case execution time.

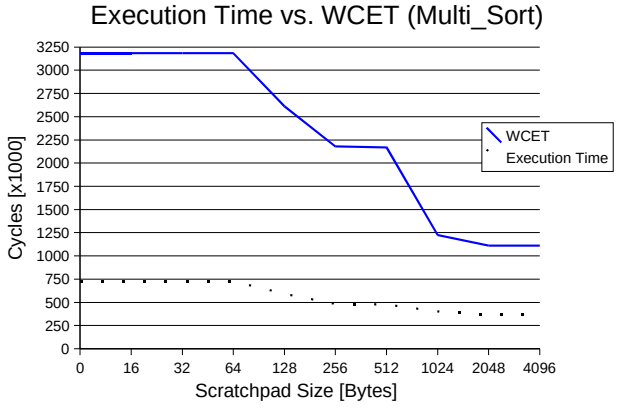

Figure 3: Results for Multi_Sort benchmark

For the Multi_Sort benchmark, similar results can be observed. By only changing the scratchpad capacity and using our compile-time algorithm to solve the problem of allocating an optimal set of memory objects to the scratchpad memory, we find that the WCET decreases by about 65%, whereas the actual execution time for our used average input data only decreases by about 50%. Without further requirements concerning WCET analysis (as e.g. required if a cache was used in the system), the use of a scratchpad memory thus shows a positive impact on WCET.

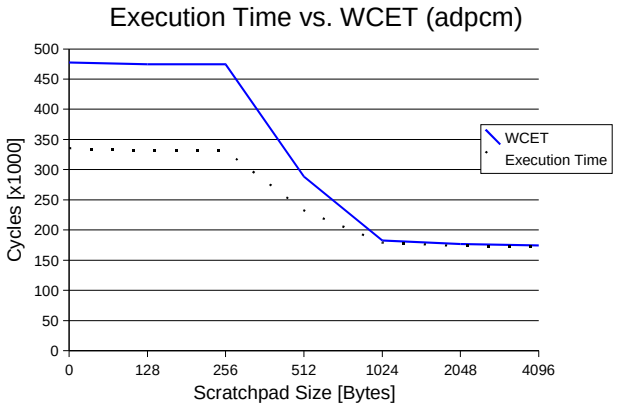

Figure 4: Results for adpcm benchmark

For the adpcm benchmark, even the initial WCET values are very close to the actual execution times. This seems to be due to the fact that all execution paths within this benchmark are very similar to the critical path. Despite this good initial WCET estimate, using a scratchpad can still improve the WCET prediction: If an onchip memory of more than 512 bytes is used, the difference between actual performance and WCET becomes negligible. The reductions in average case execution and WCET estimate reduce by 49% and 63%, respectively, highlighting the fact that WCET benefits strongly from use of a scratchpad memory.

One reason for the positive effect of scratchpad memories is possibly due to worst case assumptions concerning pipeline stalls. In the case of a three stage pipeline (as in the ARM7TDMI used in our experiments), a pipeline stall will require three instructions to be fetched from memory before the next result is generated by the CPU. If the latency for a single memory access is three cycles, then nine additional memory cycles will be required to completely re-fill the pipeline (assuming, as on our evaluation board, memory chips that do not support accelerated burst transfers). If, on the other hand, the used memory has a latency of only one cycle (as is the case for a scratchpad memory), then the pipeline can be filled with only three additional memory cycles. aiT has to assume all possible pipeline stalls to be able to guarantee that the predicted WCET result is always safe. The fact that the overhead for these pipeline stalls can be reduced by using a scratchpad memory explains the good results concerning WCET.

5 Summary and Future Work

In this work we show for the first time that using scratchpad memories in real-time systems is beneficial for WCET estimation. Using a known algorithm to allocate memory objects (both instructions and data) to the scratchpad memory, and a commercially available WCET analysis tool, we have shown that the decrease of the WCET caused by scratchpad memories is even larger than the decrease of the average case execution time. This is possible without any modification in the used timing analysis tool. Many performance enhancing architectural

modifications (e.g. caches) make WCET estimation a difficult task. If scratchpads are being used, the user only needs to know the latency cycles of the used memories.

This work shows an additional advantage of scratchpad based architectures beyond previously published results (which investigated average case execution time and energy consumption). All this is feasible with a decreased complexity of WCET tools.

In the future, we will consider how scratchpad memories compare to cache models that are being supported in some WCET analyzers today. This comparison is not really fair, since caches require extensive support and careful analysis in WCET analysis, whereas scratchpad memories can be integrated at no extra analysis costs. However, caches are being used in many systems today to improve the average performance and therefore have a practical significance.

Apart from using the energy-aware allocation algorithm from [14], we will also consider employing a similar technique which primarily takes into account the memory objects that lie on a program's critical path. By reinforcing the selection of these memory objects instead of those memory objects that consume most energy, the positive effect on WCET should become even more obvious.

6 Acknowledgement

The authors would like to thank "AbsInt" Angewandte Informatik GmbH for their support concerning WCET analysis using the aiT framework.

7 Results of discussion

This section briefly highlights some of the topics raised and discussed after the presentation of this work, only including those points that are directly linked with this contribution.

Q: What are the reasons for the overestmation with small or no scratchpad memories? Could it be that some annotations are missing?

A: The WCET estimates were validated assuming only main memory. The annotations are thus assumed to be correct.

Q: Are scratchpad memories being used in industry?

A: The TriCore has an onchip memory that is configurable as cache or as scratchpad. It is being used e.g. in the automotive industry. Optimal exploitation is unsolved, however.

Q: Concerning AbsInt's aiT tool for cache analysis mentioned in future work, how much performance is lost in the prediction compared to execution?

A: For an example from industry (RT benchmark) using 8k unified cache, aiT was compared to the used legacy method and a simulation. It was found to be inbetween the two with about 10% error rate.

Q: How does the approach scale to bigger programs?

A: The used knapsack is actually a packaging problem, for which polynomial approxmiation schemes exist, so running time of the memory allocator is not really an issue.

Q: How about integrating the approach into a real compiler?

A: Required information could be exported from the internal data structures and solved externally using an ILP solver. Distribution among memories could then be achieved by back-annotation to the program or by a linker script. The technique could thus probably be implemented outside a compiler.

In addition to the immediate discussion following the presentations, some discussions with other participants have lead to further insights into possible reasons for the high overestimation for small scratchpad memory sizes. Experiments are still being performed to validate the improved results.

References

[1] Milind B. Kamble and Kanad Ghose. Analytical Energy Dissipation Models for Low-Power Caches. In *Proc. International Symposium on Low Power Electronics and Des ign*, pages 143–148. ACM/IEEE, August 1997.

[2] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proceedings of the IEEE*, 91(7), July 2003.

[3] Peter Puschner and Alan Burns. A Review of Worst-Case Execution-Time Analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[4] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 380–387, November 1995.

[5] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

[6] Thomas Lundqvist. A WCET Analysis Method for Pipelined Microprocessors with Cache Memories. Technical report, Dept. of Computer Engineering, Chalmers University of Technology, June 2002.

[7] AbsInt Angewandte Informatik GmbH. aiT: Worst Case Execution Time Analyzers. http://www.absint.com/ait, 2004.

[8] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. In *10th Int. Symp. on Hardware/Software Codesign (CODES)*, May 2002.

[9] P. R. Panda, N. D. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip*. Kluwer Academic Publishers, 1999.

[10] M.Kandemir, J.Ramanujam, M.J.Irwin, N.Vijaykrishnanand I.Kadayif, and A.Parikh. Dynamic Management of Scratch-Pad Memory Space. In *Proceedings of the 2001 ACM Design Automation Conference*. DAC, June 2001.

[11] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. *Int. Symp. on System Synthesis (ISSS)*, pages 213–218, 2002.

[12] J.Ph. Diguet, S. Wuytack, F. Catthoor, and H. De Man. Formalized Methodology for Data Reuse Exploration in Hierarchical Memory Mappings. In *ISLPED 1997 Monterey CA*. ACM, August 1997.

[13] P.R.Panda, F.Catthoor, N.D.Dutt, K.Danckaert, E.Brockmeyer, C.Kulkarni, A.Vandercapelle, and P.G.Kjeldsberg. Data and memory optimization techniques for embedded systems. pages 149–206, April 2001.

[14] S. Steinke, L.Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. *Design, Automation and Test in Europe (DATE)*, pages 409–417, 2002.

[15] ARM Ltd. ARM7TDMI Technical Reference Manual. http://www.arm.com/pdfs/DDI0210B_7TDMI_R4.pdf, 2004.

[16] S. Steinke, M. Knauer, L. Wehmeyer, , and P. Marwedel. An Accurate and Fine Grain Instruction-Level Energy Model Supporting Optimizations. In *Proceedings of the International Workshop - Power and Timing Modeling, Optimization and Simulation, Yverdon-les-bains, Switzerland*, September 2001.

[17] ILOG. CPLEX. http://www.ilog.com/products/cplex.

[18] Stephen Brown. MediaBench Home. http://cares.icsl.ucla.edu/MediaBench, 2004.

Controlling the Influence of PCI DMA Transfers on Worst Case Execution Times of Real–Time Software *

Jürgen Stohr Alexander von Bülow Georg Färber
Institute for Real–Time Computer Systems
Prof. Dr.–Ing. Georg Färber
Technische Universität München, Germany
{**Juergen.Stohr,Alexander.Buelow,Georg.Faerber**}**@rcs.ei.tum.de**

Abstract

The PCI Local Bus is used in all general purpose computer systems. Peripheral devices connected to this bus may perform transactions autonomously. If a processor accesses the main memory or performs an I/O instruction, the execution time of these operations depends on the working load of the PCI bus and of the communication protocols being used by the chip set. In this paper the influence of the PCI Local Bus on real–time software is demonstrated. A method is presented reducing these impacts of the PCI Local Bus on the execution time of real–time software. Thus accesses to PCI peripherals from real–time tasks behave more deterministically.

1. Introduction

When determining the worst–case execution time of real–time software, all the underlying hardware has to be taken into account. Major reasons for varying execution times of software are the caches and the TLBs as the costs of a cache miss are immense. If there is a miss the execution time of loading a cache line from the main memory into the processor depends on the transactions being performed by the PCI peripherals. In addition, if a real-time task wants to perform I/O and therefore has to access a peripheral device directly, the chip set also should not be busy if deterministic computation times are favored.

In most cases the behavior of the chip set is transparent to software. It interconnects the processors, the main memory and the peripheral devices. An essential part of the chip set of general purpose computers is the PCI Local Bus which is used to connect the peripheral devices to the host. As these peripheral devices can be programmed to perform accesses to the main memory autonomously by processing DMA transfers, there is a steady influence on the execution time of software.

In this paper the impacts of the PCI Local Bus on the worst–case execution time of real–time software are examined. A method is presented which can be used to postpone the DMA transfers of PCI devices in order to get more deterministic execution times of real–time software. This method can be used if a general purpose and a real–time operating system are running in parallel on the same machine and real–time and non real–time devices are connected to PCI.

This paper is organized as follows. Section 2 gives a survey of the PCI Local Bus being used in real–time systems. In section 3 some important facts of PCI are explained. Our method making the PCI Local Bus behaving more deterministically is described in section 4. The advantages of this method are demonstrated in section 5. The paper is summarized in section 6.

2. Related Work

In real–time systems, the PCI Local Bus is used to interconnect the various components. Examples are the RAPID [2] [6] and the SARA [3] project. However, only a few publications deal with PCI: In [4] PCI connects a reconfigurable computing device to its hosts processor. In this paper, some performance measurements concerning PCI are done. Baumgartl and Härtig discuss in [1] PCI busmastering DMA. In contrast to the conclusion of this paper, they noticed that to their knowledge it is impossible to give timing guarantees for DMA operations involving the PCI bus. In [7] the impact of PCI–Bus load on real–time applications is evaluated. A slowdown factor is defined to describe these impacts formally.

When designing and validating a computer system for hard real–time usage the worst case execution time (WCET) of software has to be known. There are many components of the system that have an influence on the WCET, for example the architecture of the CPU and the design of the caches. The chip set and its buses and communication protocols affect the exe-

*The work presented in this paper is supported by the *Deutsche Forschungsgemeinschaft* as part of a research programme on "Real-time with Commercial Off-the-Shelf Multiprocessor Systems" under Grant Fa 109/15-1.

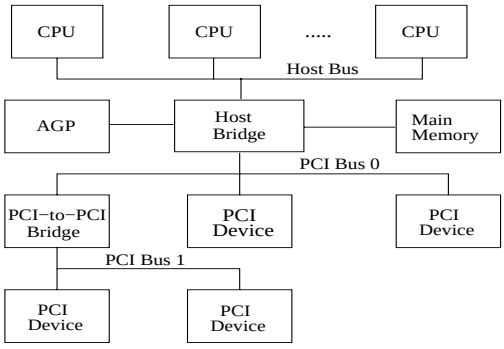

Figure 1: System Architecture

cution time, too. There exist two different approaches when evaluating the WCET: the modelling and the measurement based approach. The pros and cons of each approach are discussed in [5]. As the specifications of processor and chip set are often not available for public use, we have chosen the measurement based approach to start our work with.

3. PCI Local Bus

In state of the art computer systems, the Host Bridge interconnects the CPUs, the Advanced Graphics Adapter (AGP), the main memory and the PCI peripherals. Figure 1 illustrates a simplified system architecture: The Host Bus connects the CPUs to the Host Bridge and the PCI Local Buses are used to connect the peripheral devices to the Host Bridge.

3.1. Functionality

The PCI Local Bus is a multi–master bus. Every device is allowed to become initiator. The initiator, or bus master, initiates a transfer. The target, or slave, is the device currently addressed by the initiator for the purpose of performing a data transfer. PCI devices can access the bus autonomously without the aid of a CPU. In order to avoid collisions, each initiator has to request the bus from the PCI bus arbiter before performing any transfers.

Usually the arbiter is integrated into the PCI chip set; specifically, it is typically integrated into the Host Bridge. As the PCI specification does not define the scheme to be used by the PCI bus arbiter, the order the devices are accessing the bus depends on the chip set being used. The PCI specification only states that the arbiter is required to implement a fairness algorithm to avoid deadlocks.

As the Host Bridge acts to the PCI bus as a PCI device, it has to request the PCI bus from the arbiter like any other device before becoming initiator. If a peripheral device wants to access the main memory the Host Bridge is the target.

A PCI–to–PCI bridge provides a bridge from one PCI bus to another. It works as a traffic coordinator between the two buses. It monitors the transactions that are initiated on the two PCI buses and decides whether or not to pass the transaction through the opposite PCI bus.

3.2. Burst Transfers

When performing a transfer, a peripheral device first has to request for bus ownership as mentioned above. The transfer itself is consisting of a single address phase followed by two or more data phases. The start address and the transaction type are issued during the address phase. The target device latches the start address into its address counter which is incremented from one data phase to the next one. This kind of data transfer is called a *DMA burst transfer*.

If a bus master acquires ownership of the PCI bus, it initiates a transaction. The two most important types of transactions are listed below:

memory transactions A device accesses the memory of another PCI device. If the target addresses reside in the main memory, the Host Bridge serves as target.

I/O transactions These transactions are used to access the command and status registers of the PCI devices. Normally, I/O transactions are generated by the Host Bridge.

4. Impact of the PCI–Bus

If a PCI peripheral device performs a burst transfer accessing the main memory, the execution of real–time software can be delayed in two ways:

- If the CPU executing a real–time task has to access the main memory, these accesses are affected by parallel transfers of peripheral devices. This happens if a peripheral device initiates a DMA burst transfer accessing the main memory just before the access to the main memory of a CPU is initiated.
- A real–time task wants to perform I/O, e.g. writing data to a hard disk, and therefore has to access a peripheral device. If there is a high workload on the PCI bus, the I/O transactions may be delayed. This latency depends on the arbitration scheme being used by the bus arbiter.

As the policy of the PCI arbiter depends on the chip set, it cannot be said in which order the peripheral devices are allowed to access the bus. If some devices have got pending requests, the arbiter chooses one of them. If the access to the bus is granted to a certain device, it is allowed to perform its transfers for a certain span of time, which is specified by the *Latency Timer*.

Master Enable Bit The execution time of real–time software when accessing a PCI peripheral varies depending on the workload on the PCI bus. But there is a way to configure the

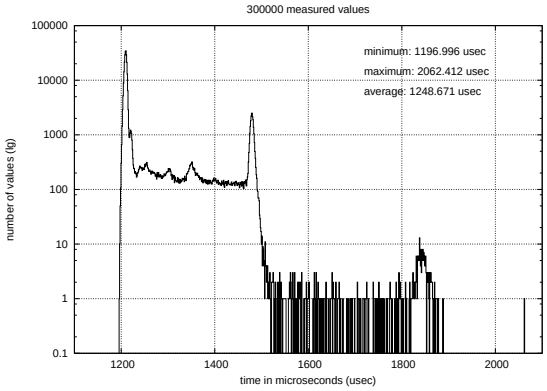

Figure 2: Accesses of a CPU to main memory with concurrent PCI activity

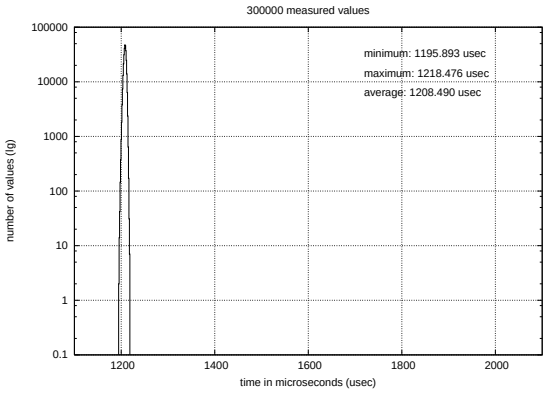

Figure 3: Accesses of a CPU to main memory without concurrent PCI activity

PCI Local Bus to behave more deterministically: Each peripheral device can be prevented from becoming initiator by clearing the *Master Enable Bit* which is defined in the *PCI Command Register*. If this bit is cleared on every peripheral device which may become initiator during certain critical sections, the PCI Local Bus can be used by the Host Bridge exclusively. Thus a real–time task performing transactions across the PCI bus cannot be delayed.

The time needed to clear every Master Enable Bit depends on the number of peripheral devices connected to the bus, the activity of each device and of the arbitration algorithm. Before performing the I/O transactions needed to clear the Master Enable Bit, the Host Bridge has to arbitrate for the bus. This latency depends on the peripherals able to perform transactions. If the number of devices being able to become initiator decreases, the time needed to clear the Master Enable Bit of a remaining peripheral decreases, too.

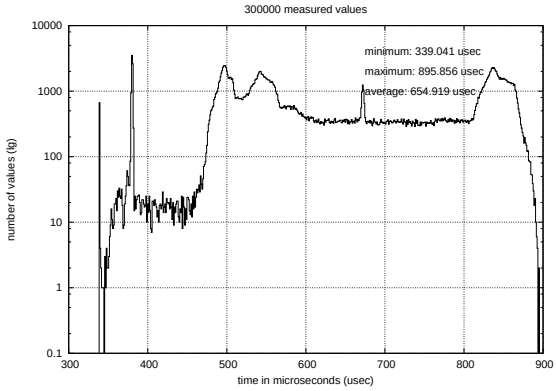

Figure 4: Accesses of a CPU to a PCI device with concurrent PCI activity

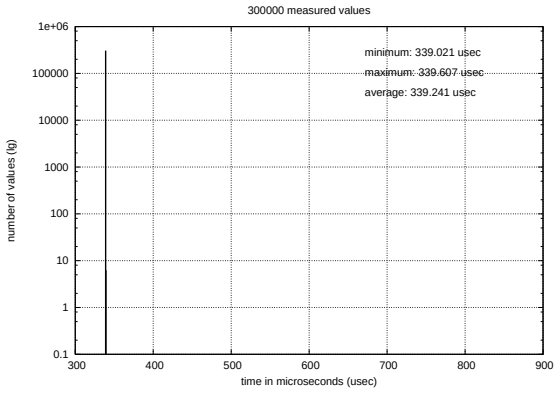

Figure 5: Accesses of a CPU to a PCI device without concurrent PCI activity

PCI–to–PCI bridges If PCI–to–PCI bridges are used, all peripherals not needed in real–time context should be put behind this bridge if possible. Thus only the Master Enable Bit of the bridge has to be cleared. Then the bridge ignores all memory and I/O transactions detected on the secondary side. In order to minimize the time needed to clear the Master Enable Bit of each relevant PCI device, a few guidelines can be stated:

- The devices causing the heaviest workload should be disabled first.
- All non-RT relevant devices should be grouped behind a bridge, thus only the bridge has to be disabled.
- The devices needed in RT context should be plugged into the PCI bus nearest to the Host Bridge.
- Only the devices able to initiate DMA burst transfers should be disabled. The Master Enable Bits of periph-

eral devices which are not used by the operating system or which only act as slaves need not to be cleared.

5. Results

We have performed some measurements in order to study the effects when the Master Enable Bits are cleared:

Accesses to Main Memory These measurements have been taken on a Dual PII machine using the Intel 440FX chip set. On one CPU a real–time task was started which accessed the main memory performing 2048 read and write accesses. Attention has been payed not to cache the memory accesses. The second CPU was kept in an idle loop not affecting the measurement. The results of this measurement are illustrated in figure 2 and 3. If there is PCI activity in parallel, the execution time to perform the 2048 accesses varies by 72%; if there is not any PCI activity the execution times are varying by 2%.

Accesses to PCI Devices In the following measurements the time needed to access a PCI peripheral from a real–time task are examined. These measurements have been performed on a Dual Athlon machine using the AMD 760MPX chip set. The real–time task performed 1024 read accesses to a PCI peripheral using memory–mapped I/O, whereby always the same memory–mapped address was used in order to avoid burst transfers. The other processor was kept in an idle loop during each measurement. The results of this measurements are shown in figure 4 and 5. If the 1024 accesses are done with PCI activity in parallel, the execution time varies by 164%; if the Master Enable Bits are cleared, the execution time is nearly constant.

Time needed to enter the deterministic state We have also measured the time needed to clear the Master Enable Bit of the relevant PCI device — a PCI–to–PCI bridge — of the Dual Athlon. In parallel to each measurement the host was busy, performing network and disk I/O heavily. The time needed to clear the Master Enable Bit varies between 1.41 microseconds and 5.94 microseconds.

6. Conclusion

As the PCI peripherals are allowed to initiate transactions autonomously, there is an influence on the execution time of real–time software due to DMA burst transfers. These transfers affect the time needed to access the main memory from a CPU leading to varying execution times of software. On the other side, if a real–time task wants to perform I/O, the access time depends on the current workload on the PCI Local Bus.

In this paper a method is presented which makes accesses from a CPU across the PCI Local Bus behaving more deterministically. When switching to real–time context the bus may be configured by clearing the Master Enable Bit of relevant devices. Thus only the Host Bridge is allowed to become initiator and all peripherals are only acting as slaves.

This method is useful for real–time systems accomplishing a lot of I/O. When estimating the WCET of specific code performing I/O, the time needed to enter the deterministic state has to be considered. The measurement based approach is useful when determining the WCET of a given system. However, hard– and software has to be configured in a way which leads to deterministic execution times.

References

[1] Robert Baumgartl and Hermann Härtig. Efficient Communication Mechanisms for DSP-based Multimedia Accelerators. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT'97)*, San Diego, September 1997.

[2] Franz Fischer, Thomas Kolloch, Annette Muth, and Georg Färber. A configurable target architecture for rapid prototyping high performance control systems. In Hamid R. Arabnia et al., editors, *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, volume 3, pages 1382–1390, Las Vegas, Nevada, USA, June 30 – July 3 1997.

[3] L. Lindh, T. Klevin, and J. Furunas. Scalable Architectures for Real-Time Applications - SARA. In *Swedish National Real–Time Conference SNART'99*, Linköping, Sweden, August 1999.

[4] Laurent Moll and Mark Shand. Systems Performance Measurement on PCI Pamette. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 125–133, Los Alamitos, CA, 1997. IEEE Computer Society Press.

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

[6] Stefan M. Petters, Annette Muth, Thomas Kolloch, Thomas Hopfner, Franz Fischer, and Georg Färber. The REAR framework for emulation and analysis of embedded hard real–time systems. In *Proc. of the 10th IEEE Int. Workshop on Rapid Systems Prototyping (RSP'99)*, pages 100–107, Clearwater, Florida, June 16–18 1999. IEEE Computer Society Press.

[7] S. Schönberg. Impact of PCI-Bus Load on Applications in a PC Architecture. In *Proceedings of 24th IEEE International Real–Time Systems Symposium*, Cancun, Mexico, 2003.

[8] Tom Shanley and Don Anderson. *PCI System Architecture*. Addison–Wesley Publishing Company, 3 edition, 1995.

Synergetic effects in cache related preemption delays

Jan Staschulat, Rolf Ernst
Technical University of Braunschweig
Institute of Computer and Communication Network Engineering (IDA)
D-38106 Braunschweig, Germany
{staschulat,ernst}@ida.ing.tu-bs.de

Abstract

Cache prediction for preemptive scheduling is an open issue despite its practical importance. First analysis approaches use simplified models for cache behaviour or they assume simplified preemption and execution scenarios that seriously impact analysis precision. We present an analysis approach for m-way associative caches which considers multiple executions of processes and preemption scenarios for static priority periodic scheduling. The results of our experiments show that caches introduce a strong and complex timing dependency between process executions that are not appropriately captured in the simplified models.

1. Introduction and motivation

Caches are needed to increase processor performance but they are hard to use in real-time systems because of their complex behaviour. While it is already difficult to determine cache behaviour for a single process, it becomes really complicated if preemptive process scheduling is included. Preemptive process scheduling means that process execution can be interrupted by higher priority processes. In this case, cache improvements can be strongly degraded by frequent exchange of cache blocks.

There are several approaches to make caches more predictable and efficient. One approach is to partition the cache sets and to reserve these partitions for individual processes. This has been investigated in [8]. The advantage is that cache lines do not have to be reloaded after interrupts and between consecutive executions of the same process. Also, cache behaviour becomes (partly) orthogonal for processes and therefore more predictable. In [4] process layout techniques are suggested which aim at minimising the inter-process interference in the instruction cache. Another approach [10] is to lock frequently used cache lines. While cache partition and lock strategies are certainly a very useful add-on to improve cache predictability and efficiency, they do not solve the general cache behaviour problem which is critical for larger systems of processes.

Simplified approaches extend the known RMA with fixed context switch costs [1], while recent approaches use data flow analysis of the preempted and preempting process to bound the number of replaced cache blocks [7] [9]. However, these approaches model only a single process activation assuming an empty cache at process start neglecting that cache blocks (CB) might be available for later executions. Pre-runtime scheduling heuristics which take the effects of process switching on processor cache into account have been presented in [6]. However, only non-preemptive scheduling based on earliest deadline first strategies is considered which is much easier than the preemptive case. If a process is preempted several times the total number of cache blocks replaced drops, because a cache block of preempted process can only be replaced once. Such preemption scenarios are not considered in classical CRPD analysis.

This paper is organized as follows. Section 2 reviews related work. In Section 3 we present a new analysis approach to determine the cache related preemption delay (CRPD) for m-way associative instruction caches which considers multiple executions of processes as well as preemption scenarios. We show the results in Section 4, before we conclude in Section 5.

2. Related work

The data flow analysis by [7] determines the CRPD when a process τ_1 preempts process τ_0 by intersecting the number of useful CBs of τ_0 and with the number of used CBs of τ_1 assuming an empty cache at process start. Then, a complex analysis follows analysing all possible combina-

tions of preemptions. The number of preemptions is determined by integer linear programming (ILP) and process phasing based on worst case and best case response time (BCRT) of processes. However, the BCRT analysis is a complicated problem where only approximative solutions have been proposed for the general case [5]. Multiple process executions is not considered and multiple preemptions are simplified by multiplying the maximum CRPD cost by the number of preemptions. For direct mapped instruction caches [9] refine the data flow analysis of [7] by modeling the cache content as a *state* instead of a set. All possible cache states of the preempting and preempted process are intersected to find the maximum CRPD. Preemption scenarios are not considered and an empty cache is assumed at process start.

Current approaches either model only the number and cost of preemptions for a single process execution or the cache effects of multiple process execution without preemptions, but none models both. Only the combination provides sufficient accuracy as we will see in our experiments.

3. Refined approach

Our CRPD analysis considers multiple activations of processes and preemption scenarios. A preemption scenario consists of one preempted process τ_i and of a process τ_j which preempts τ_i during a single execution n times. We represent the process by its control flow graph (CFG), where each node is a basic block and assume a preemption point at each node.

To reduce the exponential combinations of n preemption points (CFG nodes) we use a branch and bound algorithm. It first determines the n most expensive preemption points by analyzing the cost of a single preemption at each node. Then it continues to compare the cost of the combinations of two nodes until the cost C_n for a preemption scenario with n nodes is found. The algorithms bounds at a combination if the current cost plus the cost for future preemptions is smaller than C_n. For sequential code this is straight forward, but for n preemtions within loop body this modeling would lead to unrolling the loop. Therefore we abstract from preemption points of different loop iterations by estimating the preemption point with the maximum cost of the loop body and multiplying it by n. This estimation is exact if $I \geq n$, where I is the maximum loop iterations and conservative if $I < n$. The cost C_n of a preemption scenario is calculated by considering the useful CBs of the preempted process τ_i and the used CBs of the preempting process τ_j. For the analysis of each preemption cost a data flow model is needed.

We base our analysis for m-way associative caches on the cache state analysis of [9]. But define a cache state for each cache set with m blocks. A reaching cache state RCS_B at a basic block B of a process is the set of possible cache states when B is reached via any incoming program path. The live cache states at a basic block B, denoted LCS_B, are the possible first memory reference to CBs via any outgoing program path from B. A least fixed point data flow algorithm computes the values of these sets. The cache behavior including replacement strategy (e.g. LRU) is simulated by preloading the cache state RCS of the predessor node and executing the instruction sequence of basic block B by cache simulation. The resulting cache state represents the RCS of basic block B. The intersection of RCS and LCS is the set of useful CBs at basic block B. The used CBs of preempting process τ_j is given by RCS_{end}, assuming end is the last basic block of τ_j. Finally, the CRPD at basic block B is computed by the intersection of the used cache blocks of τ_j and the useful CBs of τ_i.

3.1. Preemption scenarios

We extend this general modelling for preemtion scenarios as follows. The cost of the first preemption at B_k is calculated by the data flow algorithm of [9]. For a second preemption we insert after B_k n nodes in the CFG of τ_i, if the preempting process τ_j finishes with n different RCSs. Figure 1 shows part of the CFG of τ_i with four basic blocks. To model a preemption at node B_3, and assuming three RCSs at τ_j last node, we insert three preemption nodes P_1, P_2 and P_3. Now the iterative data flow analysis is applied again and the RCS of all other nodes are recalculated. This models the fact that useful CBs might be overwritten by a preemption and thus cannot be replaced again. After recalculating the RCSs the CRPD is recalculated at the

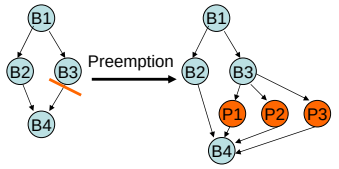

Figure 1. Modeling of a preemption by τ_j with three nodes P_1, P_2 P_3 in CFG of preempted process τ_i

next preemption point. This procedure is applied for every preemption point.

The complexity of this accurate modelling of n preemptions for a single process execution is exponential with the number of RCS states of the preempting task, because the n RCS states are propagated in the CFG of the preempted process and increase the number of cache states.

3.2. Multiple process execution

Many automotive control applications consist of sequential code without loops. A cache will not speed up a single execution because of the high cache miss penalty. A cache architecture is only well desigend if the cache is large enough that CBs of a single process can be reused in later executions. For simplicity we assume that no processes run between two activations of a process τ_i. We model a warm cache cache like in Subsection 3.1 by inserting n nodes in the CFG *before* the first node for n different RCS_{end} sets of process τ_i.

However, it is important to consider the processes which run between two activations. In this paper we assume the conservative approximation that all higher priority and lower priority proceses of the system execute. We model the execution of these intermediate execution of $\tau_i, \tau_{j_1}, \cdots, \tau_{j_k}, \tau_i$ as a sequence of process executions. The cache states of RCS_{end} of the preceding process τ_{j_m} are inserted as start nodes in the CFG of process $\tau_{j_{m+1}}$. The last process is τ_i again. This assumes that the CRPD at the second activation of process τ_i is independent of the order and the frequency of intermediate processes which is shown in [11].

4. Experiments

We select six different benchmarks: a square root calculation `sqrt` [9], array calculation with loops `dac` [12], two sequential programs of add instructions `lin`, `lin2` and two linear programs `nsich` and `statm`, part of a car window lift control generated by STAtechart Real-time-Code generator STARC [3]. The memory size ranges from 94 Byte till 872 Byte. We use the ARM developer studio for processor simulation and DINERO for cache simulation. All benchmarks are compiled for ARM946 assembly language with fixed four byte instruction width. Given the CFG generated from C code by [12], a tool for worst case execution time analysis for single processes, and the ARM memory map file our analyzer computes the CRPD.

4.1. Multiple process execution

Table 1 shows the response time for different cache architectures and benchmarks. A 2-way associative 1024 Byte cache with 8 Byte block size is denoted as 1024-8-2. With the ARM simulator we determine the core execution time of the processes and the instruction trace. t_{resp}^{negi} denotes the response time according to [9] assuming an empty cache at process start and t_{resp}^{ana} the response time calculated by our approach. In our experiments we assumed one clock cycle for a cache hit and a cache miss penalty of 20 clock cycles. The

Benchmark	Cache-C.	t_{resp}^{negi}	t_{resp}^{ana}	t_{resp}^{sim}	P_l[%]
dac/linear	256-8-1	1193	1041	1041	15
dac/linear	512-8-1	1193	1041	1041	15
dac/linear	1024-8-2	1041	813	813	28
sqrt/linear	512-8-1	2119	1549	1492	42
sqrt/linear	1024-8-2	1929	1131	1131	70
sqrt/linear	2048-8-2	1929	1131	1131	70
linear2/nsich	1024-8-1	3336	3070	3032	10
linear2/nsich	2048-8-1	4269	2690	2690	58
linear2/nsich	2048-8-2	4269	2690	2690	58
statm/nsich	512-8-1	4174	4174	4174	0
statm/nsich	1024-8-1	4174	3585	3547	18
statm/nsich	2048-8-1	4174	2274	2274	83

Table 1. Response time in clock cycles for a preemption during second activation for several benchmarks and cache sizes

results show that the response time is pessimistically overestimated by [9]'s approach. The last column shows the performance loss $P_l = \frac{t_{resp}^{negi} - t_{resp}^{sim}}{t_{resp}^{sim}}$, which could be gained with a more accurate analysis. For example, the performance loss in case of a 1KB and 2KB cache for `sqrt/linear` is 70% and for `statm/nsich` even 83% for the 2KB cache. We see that the current approach is less accurate for relevant larger caches.

The results for our refined analysis is in most cases exact to the simulated response time, the maximum error is 4% in case of `sqrt/linear` for direct mapped 512 Byte instruction cache.

4.2. Preemption scenarios

Now we consider multiple preemptions with an empty and preloaded cache. Table 2 presents the preemption cost of five preemptions for four task sets. The results show that for an empty cache our analysis does not improve the accuracy for benchmarks with or without loops. The reason is that the most expensive preemption points are inside

LP/HP Task	Cache Config.	Empty Cache		Preload. Cache	
		Negi	Ana	Negi	Ana
dac/lin	512-8-1	52	52	52	43
dac/lin	1024-8-1	52	52	52	40
dac/lin	1048-8-1	12	12	12	12
sqrt/lin	512-8-1	182	182	182	169
sqrt/lin	1024-8-1	47	47	47	1
sqrt/lin	2048-8-1	42	42	42	0
nsich/lin2	512-8-1	104	104	104	83
nsich/lin2	1024-8-1	104	104	104	74
nsich/lin2	2048-8-1	99	99	99	0
statm/lin2	512-8-1	125	125	125	107
statm/lin2	1024-8-1	125	125	125	97
statm/lin2	2048-8-2	120	120	120	0

Table 2. Comparison of total number of cache misses of Negi et al. and our approach for 5 preemptions with empty and preloaded cache for given lower priority (LP) and higher priority (HP) tasks.

a loop body. However, in the case of multiple activations with preloaded cache our analysis approach yield more accurate results. For a larger 2 KB cache the preemption cost is even zero for all preemptions in benchmark `nsich/linear2` and `statm/linear2`, in contrast to 99 and 120 cache misses in Negi et al.'s approach.

The performance of our analysis ranged from several minutes till several hours. The reason for the long running time is the exponential number of states that are propagated after inserting a preemption node.

5. Conclusion

In this paper we have extended the approach of [9] to consider multiple process activations and preemption scenarios for m-way associative instruction caches. The results with a realistic processor architecture show that cache effects lead to process interdependencies which can easily outweigh individual process execution times. Such cases are not covered by the classical performance analysis approaches which are based on individual process execution times plus independent blocking times (e.g. [2]). However, the complexity of the proposed analyis is exponential with the number of cache states of the preempting process. Future research is necessary to develop less complex algorithms.

Applications with loops behave better also in other approaches. However, for automotive control applications linear code is very important (e.g. Matlab generated code). Here current approaches result high overestimations. On the other hand, cache parameters have a significant influence on process interdependence. We can therefore conclude that cache design should receive maximum attention in embedded system design, use process systems as benchmarks rather than individual processes to consider multiple process activation and that new models and approaches are needed for performance analysis of systems with caches.

References

[1] J. V. Busquets-Mataix and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *IEEE Real-Time Technology and Applications Symposium*, pages 204–212, June 1996.

[2] G. Buttazzo. *Hard Real-Time Computing Systems*. Norwell, MA: Kluwer, 1997.

[3] C-Lab. Wcet benchmarks. http://www.c-lab.de/home/de/download.html.

[4] A. Datta, S. Choudhury, A. Basu, H. Tomiyama, and N. Dutt. Satisfying timing constraints of preemptive real-time tasks through task layout technique. In *Proceedings of 14th IEEE VLSI Design*, pages 97–102, January 2001.

[5] W. Henderson, D. Kendall, and A. Robson. Improving the accuracy of scheduling analysis applied to distributed systems computing minimal response times and reducing jitter. *Real-Time Systems*, 20(1):5–25, 2001.

[6] D. Kästner and S. Thesing. Cache sensitive preruntime scheduling. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 131–145. Springer, 1998.

[7] C.-G. Lee, K. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on software engineering*, 27(9):805–826, November 2001.

[8] F. Mueller. Compiler support for software-based cache partitioning. In *Workshop on Languages, Compilers, and Tools for Real-Time Systems*, La Jolla, June 1995.

[9] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS'03*, Newport Beach, California, USA, October 1-3 2003.

[10] I. Puaut and D. Decotigny. Low-complexity algorihtms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, 2002.

[11] J. Staschulat and R. Ernst. Crpd independence for multiple process execution. Technical report, IDA, TU Braunschweig, March 2004.

[12] F. Wolf. *Behavioral Intervals in Embedded Software*. Kluwer Academic Publishers, 2002.

Component-wise Instruction-Cache Behavior Prediction
– Extended Abstract –

Oleg Parshin* Abdur Rakib† Stephan Thesing* Reinhard Wilhelm*

Abstract

The precise determination of worst-case execution times (WCETs) for programs is mostly being performed on linked executables, since all needed information and all machine parameters influencing cache performance are available to the analysis. This paper describes how to perform a component-wise prediction of the instruction-cache behavior guaranteeing conservative results compared to an analysis of a linked executable. This proves the correctness of the method based on a previous proof of correctness of the analysis of linked executables. The analysis is described for a general A-way set associative cache. The assumptions are that the replacement strategy is LRU and inter-module call relationship is acyclic.

1. Introduction

So far, WCET-determination methods mostly work on fully linked executables, since in this case all needed machine-level information about code allocation is fixed and available. This paper presents a method for component-wise analysis of the instruction-cache behavior, thus supporting incremental program development. This method uses the notion of *cache-equivalence* of memory allocations to express that one allocation of a module in the memory will display exactly the same cache behavior as the equivalent one. This equivalence is exploited to influence the linker, which can choose between several equivalent allocations when placing a module into the executable. The overall picture is the following:

1. A set of modules making up the real-time program is given. Cyclic calling dependencies are assumed to exist only inside modules, i.e., the inter-module call relationship graph is acyclic.
2. A bottom-up module-wise analysis computes a sound approximation to the cache contents at all program points of all modules taking into account safe upper approximations of the cache damages due to external function calls. The results of the module-wise analysis are combined conservatively with respect to an (in general more precise) analysis of a linked executable.

2. Cache memory architectures

A cache can be characterized by three major parameters:

cache size s is the total size of the cache, i.e. the number of bytes it may contain.

line size l **(also called block size)** is the number of contiguous bytes that are transferred from memory on a cache miss. The cache can hold at most $n = s/l$ blocks.

associativity A is the number of cache locations where a particular memory block can reside. The cache contains $\eta = n/A$ *sets*.

If a block can reside in exactly A locations, then the cache is called *A-way associative*. If a block can reside in any cache location ($A = n$), then the cache is called *fully associative*. If a block can reside in exactly one location ($A = 1$), then it is called *direct mapped*. Thus, fully associative and direct mapped cahes are special cases of the A-way cache.

In the case of an associative cache, a cache line has to be selected for replacement when the cache is full and the processor requests further data. This is done according to the *replacement strategy*. Common strategies are *LRU* (Least Recently Used), *FIFO* (First In First Out), and *random*.

In this paper we consider A-way set associative cache with LRU replacement strategy. Detailed formal description of the cache semantics can be found in [1].

*Rearch reported here was supported by the transregional research center AVACS (Automatic Verification and Analysis of Complex Systems) of the DFG (Deutsche Forschungsgemeinschaft). These authors are with FR Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, Germany, {oleg, thesing, wilhelm}@cs.uni-sb.de

†Supported by the IMPRS (International Max-Planck Research School for Computer Science). This author is with Max-Planck Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, hossain@mpi-sb.mpg.de

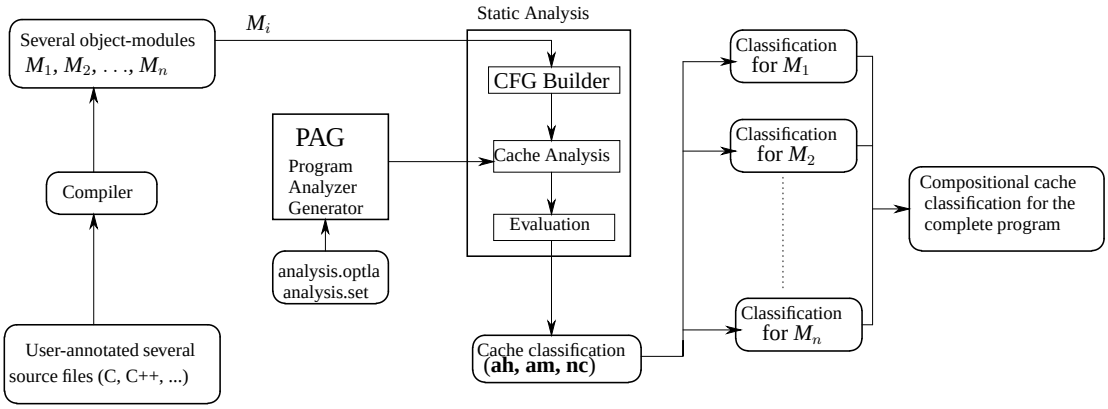

Figure 1. The structure of analysis framework.

3. The analysis framework

As input to our analysis framework (c.f. Figure 1) we have a set of object modules that are yet to be linked to form an executable. We also assume, that the user provides additional information, such as number of loop iterations, upper bound for recursions etc. A parser reads the object modules and reconstructs the control flow graph for each module [9]. Nodes of the control flow graph represent *basic blocks*. For each basic block it is known which memory blocks it references.

Analysis results are computed using Abstract Interpretation [2]. The *collecting semantics* of a module is safely approximated using *abstract cache states* [11].

We distinguish two kinds of analyses. The *must analysis* determines a set of memory blocks that are definitely in the cache at a given program point whenever execution reaches this point. The *may analysis* determines set of memory blocks that may be in the cache at given program point. The complement of may analysis is used to determine which blocks are definitely not in the cache.

The analyses are used to classify the memory references into *always hit*, *always miss*, or *non-classified* [1].

Termination of analyses is guaranteed [3].

4. Notion of equivalence

Our aim is to ensure that the results of cache behavior analysis obtained at module level can be combined in a conservative way with respect to the results of the analysis of a linked executable. The relative and absolute address spaces of a module before and after linking, respectively, will in general be different. Consequently, the cache behavior of a module before and after linking may be different.

We perform cache analysis for each module using available relative address information and a fixed mapping of relative addresses of an object module to cache sets.

This section is concerned with the conditions under which the different allocations of modules will display equivalent cache behavior (in the sense of number of cache hits/misses, not exact content of the cache).

4.1. Equivalent memory allocation with respect to the fixed set mapping

Suppose a set of modules $M_1, M_2, \ldots, M_p$ forms a program. Each module consists of the set $\{m_0^i, \ldots, m_{k_i}^i\}$ of memory blocks (each memory block has the size of a cache line). A linker combines these modules in the sequence $M_1, M_2, \ldots, M_p$. We assume that all object modules are created with base address 0. According to this assumption, in each module M_i the block m_0^i is mapped to the first set of the cache.

A question arises when we consider a linked executable, whether the absolute address in the executable, which corresponds to a relative address inside the module, will be mapped to the same set. Since linkers only shuffle segments of object modules but do not rearrange their internals, all the internal memory addresses become offsets from the new base address of the modules.

Since the number of the set to which block m_i is mapped is determined as $(i \mod \eta)$ [3], two memory blocks are mapped to the same set, if the difference q between their addresses is a multiple of $(\eta \cdot l)$, because each block has the size of l bytes.

In order to preserve a fixed mapping of addresses for module M_1, the executable has to be created with such a base address q. The base addresses of the modules $M_2, \ldots, M_p$ depend on the sum of the sizes of previous modules. To preserve the fixed mapping for these modules, some *wasted space* between them has to be added, such that the base address of each module will be a multiple of $(\eta \cdot l)$.

4.2. Conservative cache-behavior analysis

In order to combine the results of module-wise analysis conservatively with respect to the analysis of a linked executable, we choose a placement of modules according to Section 4.1, i.e., the absolute base address of each module should be a multiple of $(\eta \cdot l)$.

5. Proposed analysis method

As input for the analysis we have a directed acyclic inter-module dependency graph where vertices represent modules and edges represent call relations between modules. Our analysis is based on a bottom-up approach, starting from modules which are not dependent on any other module in this graph (i.e., with $outdegree = 0$). At each stage (for each module) the produced results are kept in a special data structure so that analysis results of a module will be available later to the modules, which have calls to this module.

5.1. Module-wise cache analysis

We analyze cache behavior separately for each procedure using the framework from Section 3 in the two following call contexts: (i) *local call* – a call between two procedures in the same module; and (ii) *external call* – a call between two procedures in the different modules. In the last case we combine at the return point the analysis result of the caller with the analysis result of the callee.

During the analysis of a procedure we assume that cache is initially empty for the must analysis and everything may be in the cache with age 0 for the may analysis.

The analysis for modules, which have calls to other modules (i.e, with $outdegree \neq 0$) uses the precomputed information of its called modules, whenever needed at the calling points. Since the cache contents of the calling module will be changed according to the called module's cache information, we have to consider the cache damages due to calls to the external procedures. In the following subsection we describe how to handle such cache effects during calls between modules.

5.2. Cache damage analysis

The aim of the cache damage analysis is to provide the correct information about the bounds of replacements in a particular set, i.e., to determine an upper-bound of the number of replacements occurring in a particular set for the must analysis, and a lower-bound for the may analysis.

Let us consider must analysis. The elements of each set of the caller's cache are age by the upper bound of the number of replacements in the same set of the callee's cache, and the elements of this set in the callee's cache retain their age during the cache damage update (cf. formal cache semantics in [1]).

If before the call some memory block m is in the caller's cache set f with age x, and the upper bound of replacements in this set due to the call is a, then this block will survive in the cache after the call if $x + a \leqslant A - 1$.

A procedure may be also called from inside a loop of another procedure. If some block m is in the callee's cache at the return point with the age x, then it will be in the caller's cache after return. Since the procedure is called inside a loop, this block may survive in the cache during all following iterations, and be in the caller's cache with the age y before the call. If $y + a \leqslant A - 1$ then this block will survive in the caller's cache during the call with the age $z = y + a$. Hence, there may exist multiple copies of the block m in the same set with different ages. In order to avoid such a situation we flush all callee's memory blocks which are in the caller's cache before the call.

5.3. Properties of the method

The following steps are followed during the analysis of a module: (i) construct the control flow graph for each procedure, (ii) identify the local and the external calls, (iii) analyze all possible paths considering local and external calls of a procedure, (iv) update the cache information according to the call contexts using cache-damage analysis result, and (v) store the cache analysis information for each procedure in the corresponding data structure.

The analysis result of the complete program is the composition of the analysis results of all the modules. We have the following properties of the method.

Termination of the analysis. Termination of the may and must analysis is guaranteed. Cache damage analysis terminates, since the domain is finite, update functions are monotone, and the join functions are monotone, associative and commutative (cf. full version of this paper [4]).

The results of the module-wise analysis are conservative. Our analysis is based on a bottom-up approach and during the analysis of each procedure we take safe initial approximations according to Section 5.1. Therefore, we can conclude the following theorem:

Theorem 1 *The results of component-wise cache behavior prediction are conservative to an analysis results of a linked executable, assuming the equivalent module placement according to Section 4.1.*

For the sake of space, we omit the proof of the theorem. The proof can be found in the full version of the paper [4] .

Maximum wasted memory space. As we have seen

in Section 4.1, some memory space is wasted in order to preserve the equivalent memory allocation w.r.t. the fixed set mapping. The wasted memory space in the worst case is $(\eta \cdot l - 1) \cdot (p - 1)$.

6. Related work

Most of the research on precise cache-behavior prediction is being performed on fully linked executables.

The work [5] propose a compositional instruction-cache behavior prediction. Their goal is to decrease the analysis effort by splitting the analysis into several phases, a module-level analysis, preprocessing calls, and a compositional analysis using this information. The motivation comes from the claim that only small programs can be analyzed by the traditional methods. However, as shown in [10] these methods realized in commercially available tools are in routine use in the aeronautics and also in the automotive industry. Their method needs the availability of all modules, while ours analyzes modules as they are compiled and combines the analysis results in a conservative way. A research group at the Laboratory of Embedded Systems Innovation and Technology (LIT) described in [6] a framework, PERF, which works with the object code generated by the integrated tools in order to determine execution-time limit estimations for functions that compose a real-time system. Their cache behavior prediction method is based on the extended timing schemata proposed by [7, 8].

7. Conclusions and future work

We have presented a technique for predicting the cache behavior for A-way set associative instruction caches component-wise. Given a set of object code-modules, a parser reads the object code-modules and reconstructs the control flow. The cache analysis technique works in a bottom-up way starting from minimal modules of the module dependency graph. The analysis computes a sound approximation to the cache contents at all program points of all modules taking safe upper approximations of the cache damages of called external functions into account. The analysis results can be combined in a conservative way with respect to an analysis of a fully linked executable.

Our current research direction includes component-wise data cache behavior prediction. Data cache analysis is more difficult than instruction cache analysis, because the effective data address may change when an instruction referencing data is executed repeatedly. We will implement a tool to estimate the worst-case execution time of a real-time system, where the system is given as a set of object code modules.

References

[1] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *Proceedings of the Third International Symposium on Static Analysis*, pages 52–66. Springer-Verlag, 1996.

[2] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

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

[4] Oleg Parshin, Abdur Rakib, Stephan Thesing, and Reinhard Wilhelm. Component-wise Instruction-Cache Behavior Prediction. Accepted for 2nd International Symposium on Automated Technology for Verifi cation and Analysis, 2004.

[5] Kaustubh S. Patil. Compositional Static Cache Analysis Using Module-Level Abstraction. Master's thesis, North Carolina State University, 2003.

[6] Douglas Renaux, João Góes, and Robson Linhares. WCET Estimation from Object Code Implemented in the PERF Environment. In *2^{nd} International Workshop on Worst-Case Execution Time Analysis (Satellite Event to ECRTS'02)*, pages 28–35, Technical University of Vienna, Austria, June 18, 2002.

[7] Lim S-S et al. An Accurate Worst-Case Timing Analysis for RISC Processors. *In IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.

[8] Min S.L et al. An Accurate Instruction Cache Analysis Technique for Real-time Systems. *In Proceedings of the Workshop on Architectures for Real-time Applications*, April 1994.

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

[10] Stephan Thesing et. al. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software Systems. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 625–632. IEEE Computer Society, June 2003.

[11] Reinhard Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, Venice, Italy, January 2004.

Session 3: WCET calculation methods

Session chair: Jan Gustafsson (University of Mälardalen, Sweden)

Presentations

The first paper *"A Distributed WCET Computation Scheme for Smart Card Operating Systems"*, by N. Aissa et. al. presented a new method to calculate the WCET on a security protected and very constrained device (a smart card).

The second paper "*Inspection of industrial code for syntactical loop analysis"* by C. Sandberg described methods for analysis of "real" code and some preliminary results.

In the third presentation *"A New Timing Schema for WCET Analysis"* by S. Petters et al., a new technique for handling path-based and low-level analysis.

The forth paper, *"Petri Net Level WCET Analysis"* by F. Stappert described an approach for WCET calculation based on high level Petri Nets.

The last paper in the session, *"Measurement-Based Worst-Case Execution Time Analysis using Automatic Test-Data Generation"* by R. Kirner et al. described a hybrid approach to calculate the WCET partly based on static analysis, partly on measurements.

Discussion

The discussion that ended the session was mainly concerned with the development in the WCET research area since the WCET workshop started in 2001 and the future of the workshop.

Jan Gustafsson: I see two main trends in the WCET area. The first is academic; this area wants to grow and the workshop would like to extend be longer, have more papers etc. So the first question is: *how do you look upon this workshop and the future of it?*

The other trend is to target the industry and to get our methods known and used there. Since the WCET workshop started, at least three tools have been developed for the industrial market. We all hope that these companies will succeed, since this is of benefit for us. We also hope that new companies and tools will emerge. This is because there is a synergy between these tools and the research. We have seen one paper today (the effect of scratchpad memories on WCET prediction[2]) where the aiT tool has been used to support research, and we can expect more papers in the future.

In my department, the aiT tool is used to study the use of static WCET analysis in industrial settings. The results can be used as input to our research.

[2] Influence on Onchip Scratchpad Memories on WCET, by L. Wehmeyer, P. Marwedel, University of Dortmund, Germany

So the other question is obvious: *how do we find our way out to reality*? Can we push even harder? Maybe this workshop can help, too. Maybe we can start an "WCET Interest Group" which is active also between workshops.

One important issue is the one about benchmarks. A common WCET benchmark allows tools and methods to be compared. There is a lot of work to be done here.

Guillem Bernat: There are many questions! What we would like to suggest is that we get a repository where we can store all these codes. For example we could have code with many nested loops.

Friedhelm Stappert: We already have such a repository[3]. We volunteer to have it extended.

Jan Gustafsson: A problem here is that some of the interesting code is not accepted for spreading by the companies. And we want real code. So how do we get it?

Peter Puschner: I don't know if this is a solution, but I think even the patterns are interesting. So, if you find some patterns that you think are interesting, you may create "dummy" code with maybe not the same semantics as the real code, but with similar patterns, say control structure or other property, like dependence on pointers.

Jan Gustafsson: Well, it has to be semantically correct so you can run it and measure it.

Peter Puschner: Well, it should be able to do a simple version with a correct semantics but still with the interesting structure.

Iain Bate: Thanks for the offer! We already have some samples that we used before. We are looking at some software that we might have there on the benchmark and which is available stuff. What we don't want is too many examples. If you look at other benchmarks in other areas you see that they are small. We don't want dozens of examples but rather get it down to 3 or 4 examples. Otherwise, the different groups will look at different subsets of the examples.

A second issue is benchmarks for hardware. It would be good to choose 2 or 3 different processors, maybe one with a simple pipeline, another one that is a little bit more complex, and the last one at the "bleeding edge". This is because it makes no point to have common software and then compile them to and analyze them in different targets.

Jan Gustafsson: So you mean that you should force people to use the same codes, and not select the examples that works just for them?

Iain Bate: Yes. And, for the software, we could have the similar selection as for hardware, that is a simple one, in intermediate on and a complex one.

Raimund Kirner: I have a comment on the type of codes we want to collect. A nice thing with the codes that Christer studied is that "this is code from reality". Also, if you have a nested loop with some dependencies inside which show some difficulties, which poses problems for the WCET analysis, you might argue that this is a sorting algorithm and these problems might not be interesting for other areas of practice. So, we should keep in mind what is the typical application and avoid studying code patterns that are not relevant in practice.

[3] http://www.c-lab.de/home/en/download.html#wcet

Also there are a lot of code examples in books, and maybe we can copy them for a use like this.

Jan Staschulat: My observation is that the WCET community is always a number of years behind the latest developments in processor architecture. I have seen this with the pipelines, then the caches, and now we have the scratchpad memory. First, a new hardware feature is introduced and then a couple of years later an analysis by some research team is proposed to cope with the new complexity. Instead, shouldn't the WCET community propose how a processor should look like to be fast and predictable?

Unidentifier speaker: Right now the processors are going in a direction with more complex features like out-of-order execution and similar. So on the contrary – it is the WCET community has to follow. I think that it is not possible to force the industry to produce processors that are fast and predictable. It is probably more useful to try to overcome the problems with really unpredictable processors.

Jan Gustafsson: In our project in Sweden we have deliberately concentrated on simple processors for embedded systems. These are today typically without caches and even pipelines. These embedded systems also often have hard real-time requirements. In the other end with the really complex processors it may be better with statistical WCET methods, but then you will not have the full safety. This is my simple view, but if you can come up with something better, many should buy that.
I don't think that we can steer the hardware community. It is too much money involved. But we can pick the processors that suit our methods and the customers can do that, too.

Tullio Vardanega: It really seems to me that there are two views. The first is *speed-oriented* software, whether it is hardware or software. The other is *timing-aware* processing and coding.
I can see that there are real academic challenges to take any sort of jumbled code and to derive information from that. But it would be much more valuable to tell the users that there a number of coding styles or idioms that are timing-aware and much more amenable to timing analysis, no matter what the processor is. I see no effort indicating to people that there are idioms that are wrong, and it's no good to squeeze and push our tools that they can understand this rubbish!

Jan Gustafsson: Have you seen rubbish? Are there coding styles which are no good?

Tullio Vardanega: I have seen lots and lots of coding styles which are speed-oriented and which are totally wrong. One can tell (or smell) that what was behind that code was speed and nothing else.
Everybody have a responsibility, when you are a programmer or designer, to solve the timing problem. It is no good to delegate it to a magic tool – to write rubbish and to send it to a tool to solve it. I don't believe in magic.

Peter Puschner: Exactly as you say we have to look for the priorities. Speed is often number one, and what we would like to add is predictability. But really it depends on the area we are looking at. If we are looking at hard real-time systems that need to be dependable, we want predictable timing. Then predictability is the number one issue, but speed if of course still important.
If we want predictability we should design our systems with this in mind. The whole architecture, both hardware and software, should be laid out towards that goal.

We are doing some work in these issues and my feeling is that even if you aim at predictability, the speed, in terms of short worst-case timing, will come as an add-on.

Tullio Vardanega: People are implicitly presenting the notion that real-time programming is complex. For example going from Java to Real-time Java adds on a lot of classes. But when you are programming against time, you take a number of constraints (or even short-cuts) with you and you are more disciplined. It the cost of something, of course.
It is a service to the community if these things are told up-front, and not are discovered at the end of the slope.
The cost you pay is a certain limitation of freedom. I would really like this community of cultivated people to say this to the programmers and don't just say "Give us anything, we will process anything".

Peter Puschner: Just a picture of the user: if you work with a nail you use a hammer, and if you work with a screw you need a screwdriver, but not the other way around. And it's the same here. Use the right tools!

Iain Bate: The only counter-argument to what I am hearing is that you already know all this. For example a lot of the industrial examples are simple hardware and software. But from the academic perspective I assume that this is less interesting. It is useful to look at the more nasty control flow graph for example. It is more of a challenge.

Jan Gustafsson: I agree with Tullio but I don't think that the word is spread all around.

Guillem Bernat: People are aware of what is possible and what is not possible. Moreover, in a real setting there is a lot of priorities, the software, the processor, the development process, etc. To make radical changes, for example: do not use this processor, may be impossible since it may have been selected by other criteria.
So if we work for this aim it is good, but we have only limited possibilities in our position. Sometimes we will have to live with the fact that other criteria completely steers the situation.

Iain Bate: I would like to bring up the second question that Jan brought up namely: how we raise the profile of what we are doing, and what is the next step for a workshop like this?
Our workshop has a good form, and is interesting, but obviously from an academic perspective what we need is to move towards longer papers, more stringently reviewed, longer presentations etc. That is, moving towards a conference, and in my view, have proceedings that are more academically credible, like an IEEE publication with an ISBN number and all of these good things.

Guillem Bernat: We have been discussing this before with Peter Puschner. We are very happy to see all these new faces and you are all very welcome. I hope we all share the same view, i.e., move towards IEEE proceedings etc.
Have many of you have come only to the workshop and not the full ECRTS? (Many raised hands) So, there is no longer a great risk of this workshop would die if it was organized on its own and not as a satellite, as the situation was a number of years ago.
I would like to go to a longer workshop with longer papers etc., but the issue is would we get enough contributions for that?

Peter Puschner: One important point is the flavor of the workshop. The original idea when we started the series was to have room and time for discussion, and not to make this "yet another conference" with one paper presented after the other with almost no discussion, and people submit papers only to get a publication. The flavor shouldn't change a lot so we should avoid moving to these of these "standard conferences".

But, it is definitely a good idea to think about for example publication that goes beyond just a technical report.

Jan Gustafsson: Another thing that would be good is to continue these discussions between users and developers. This workshop would become a meeting point with prepared talks and prepared discussions. This is not in conflict with higher academic goals. Maybe this requires two days.

Peter Puschner: It is also this point of having something to present. How many of you wouldn't have the possibility to come without having a paper presented here? (Some raised hands)

Guillem Bernat: Well if your paper does not show up in the proceedings it does not always give the proper founding for travel. So this calls for a proper proceedings which you can reference and so on.

Iain Bate: Maybe a solution is to have a two day workshop with not many more papers but longer and more detailed papers, while still maintaining the discussions, which can be very useful. To come for 4 or 5 days, the whole week is gone. Two days is just as easy.

Björn Lisper: Here is another suggestion. We can keep the present format of the workshop, with short papers, but then invite authors will have to submit a longer version afterwards. These papers should be reviewed and then be published as post-proceedings. Then we would get proceedings that are peer reviewed and on par with ordinary conference proceedings.

Guillem Bernat: Well, this would not solve the problem that some people do not get funding if they do not have a paper accepted.

Iain Bate: It is more than that. You want to align to a journal or something like that. In the UK for example it is important where you publish and a workshop like this is not of the same high value as a journal.

Guillem Bernat: I would like to go back to the benchmark issue. We can have something like a competition between PhD students who can go to this workshop. They can try different methods on different problems and in that way find open problems, using real-life pieces of code.

Jan Gustafsson: This is a nice suggestion; we can give this community problems to solve ad we can then discuss the solutions. This should be on free will – you can write about this or something else if you like.

Guillem Bernat: Yes, you can just register if you're interested. There is an interesting Matlab programming competition where they set up a problem and then they get a score for their solutions. It is interesting to see how people can really compete to really squeeze smart solutions out of this.

Jan Gustafsson: So, there is still work to do for the organizing committee. So finalizing this workshop, I would like to give the word to Isabelle [Puaut] for some final words.

Isabelle Puaut: I would like to thank everybody for coming, a special thanks to the session chairs, and the organizing committee. And don't forget to come to the restaurant tonight!

A Distributed WCET Computation Scheme for Smart Card Operating Systems

Nadia Bel Hadj Aissa, Christophe Rippert, Damien Deville, Gilles Grimaud

IRCICA/LIFL, Univ. Lille 1, UMR CNRS 8022
INRIA Futurs, POPS research group*

{Nadia.Bel-Hadj-Aissa,Christophe.Rippert,Damien.Deville,Gilles.Grimaud}@lifl.fr

Abstract

Computing WCET in a resource-constrained device such as a smart card in a safe manner raises some difficulties. Indeed, most of the classical algorithms for computing WCET do not address resource-limitation or security issues. In this article, we propose to distribute the computation process between the off-card part running on a powerful workstation and the on-card part specific to the hardware included in the smart card. We also guarantee the safety of our computation process by inserting assertions in the generated code and preventing information leaks from the card to the outside.

1 Introduction

Smart card operating systems have to face very hard constraints in terms of available memory space and computing power. Nonetheless, the specifications of most smart card platforms impose strict deadlines for communications between the card and the terminal to which it is connected. This advocates the real time paradigm to guarantee response times and thus introduces the need for computation of WCET on these very constrained devices. Besides, smart card operating systems have very strict security requirements which must be taken into account by all parts of the operating system, including the WCET computation algorithm. Unfortunately, most of the classical algorithms for computing WCET do not address resource-limitation or security issues. We propose in this paper a novel scheme for safely computing WCET on a very constrained device such as a smart card.

*This work is partially supported by grants from the CPER Nord-Pas-de-Calais TACT LOMC C21, the FP6 Integrated Project INSPIRED, the French Ministry of Education and Research (ACI Sécurité Informatique SPOPS), and Gemplus Research Labs.

We first present the CAMILLE operating system for smart cards, and then describe the main issues when computing WCET on very constrained devices. We then detail the architecture we propose to compute WCET in the CAMILLE operating system and illustrate it on an example of a simple embedded algorithm. We conclude by presenting the future work we plan to conduct.

2 The CAMILLE architecture

CAMILLE [1] is an extensible operating system designed for resource-limited devices, such as smart cards for instance. It is based on the exokernel architecture [2] and advocates the same principle of not imposing any abstractions in the kernel, which is only in charge of demultiplexing resources. CAMILLE provides secure access to the various hardware and software resources manipulated by the system (*e.g.* the processor, memory pages, native code blocks, etc) and enables applications to directly manage those resources in a flexible way.

System components and applications can be written in a variety of languages (including Java, C, etc). The source code is translated in a dedicated intermediate language called FAÇADE [3] by appropriate tools. Using an intermediate language enhances the portability of the various components is a way similar to Java bytecode. To guarantee the efficiency of the system and the applications, the FAÇADE code is translated into native code using an embedded compiler. This compiler converts FAÇADE programs when they are loaded in the device, and performs machine-dependent optimizations to exploit fully the underlying hardware. FAÇADE is an object-oriented language including only five instructions: `jump`, `jumpif`, `jumplist`, `return`, and `invoke` which can be easily type-checked due to its simplicity.

Thus, CAMILLE architecture is divided in two parts. The off-card part is in charge on compiling the application or system components into FAÇADE and compute the proof

of their type-correctness which is included in the generated binary [4]. The on-card part loads this binary, checks the proof, and then translate the FAÇADE program into native code using the embedded compiler. Thus, CAMILLE takes advantage of the computing power and memory space available on the workstation on which runs the off-card part to perform costly operations. The WCET computation scheme we propose is based on a similar distribution between off-card and on-card parts, as detailed below.

3 Distributing the WCET Computation process

WCET can be computed either statically or dynamically. However, the WCET computed by a dynamic analysis can be less than the real execution time of the code [5], which is not compatible with hard real time constraint. Thus, we focus on a static analysis which is more suitable in our context. Since static analysis usually result in a pessimistic estimate [6, 7], one of our goal shall be to reduce the degree of pessimism as much as possible. Classical techniques for computing WCET include the tree-based [6], the path-based and the Implicit Path Enumeration Technique [8] algorithms.

At the source code level, the tree-based method uses a combination of an abstract syntax tree with a timing schema approach [9]. It works on the source code of the program to extract both its logical structure and the annotations introduced by the programmer. In CAMILLE, the on-card part of the system does not have access to the source code of a dynamically loaded extension, but only to FAÇADE binary code and its proof. As FAÇADE is a low-level language close to assembly, no high-level instructions like `loop` or `if-then-else` are included in the FAÇADE instruction set. Thus, FAÇADE code does not include any way to guess the high-level structure of the program, which means that the tree-based technique cannot be used in our context.

The IPET algorithm generates a set of constraints from the Control Flow Graph of the program. The WCET estimate is then generated by maximizing the sum of the products of the execution counts and execution times of the basic blocks forming the CFG. Constraint solving or Integer Linear Programming can be used to solve this maximization problem. Obviously, a simplex algorithm for instance is much to costly in terms of memory and CPU resources to be executed on a smart card. Since the costs of the basic blocks are unknown off-card, the whole algorithm must be executed in the smart card, which is not realistic for complex programs.

The path-based technique can be assimilated to the classical problem of finding the longest path in a graph, which can be solved for instance by a Dijkstra algorithm [10]. The path-based analysis searches the most costly path in the CFG. Considering the memory space necessary to compute WCET for complex programs with many possible paths, and the heavy computations it implies, it is not possible to use this technique as is in a smart card. Thus, we propose to distribute the computation of the WCET between the off-card and the on-card parts of CAMILLE, so that the most costly operations are done off-card. The path search algorithm cannot be applied off-card as it requires the knowledge of the cost of each path to select the most costly one. Indeed, only the card knows about the worst case timing behavior because it depends closely on the target architecture. Moreover, the off-card part cannot quantify the execution time of each FAÇADE instruction which are handled differently by the on-card backend according to the compilation context (*i.e.* optimizations). Exporting relevant information from the card would make it possible to compute WCET in the off-card part. In fact, exporting a profile containing the exact code generated by the embedded compiler and the cycle number corresponding to each native instruction would allow the computation to be finalized off-card. Unfortunately, carrying out such sensitive information from a secure area as the smart card to the outside is reproved by smart card manufacturers in order to prevent both technology leaks and potential timing attacks [11] against the cryptographic protocols implemented in the card for instance.

In the next section, we show how we propose to distribute the WCET computation process by simplifying the CFG outside of the card before sending it to the on-card part of the system.

4 Implementation in CAMILLE

The WCET computational process has to be split up into two phases. In a first step, in the off-card part of CAMILLE, a weighted control-flow graph must be figured out as shown in Figure 1. Each node in the graph represents a basic block (*i.e.* a sequential piece of code without any jumps or labels: labels start a block, and jumps end a block). Iterations are represented by edges labelled with the upper bound of the loop.

Then, a parser flattens the control-flow graph obtained into a tree. This eases the computation of the WCET by the on-card part of the system, since searching the most costly path is less resource-demanding in a tree than in a cyclic graph. Conditional statements are represented by separate branches in the tree. Loops are replaced by a tag on the node representing the execution count of the block. In the case of nested loops, the inner loop is tagged by the product of its execution count and the outer loop one, as illustrated in Figure 1, where BB_4 will be executed $n_4 \times n_5$ times.

Once the tagged-tree is built, it is sent to the card within the binary containing the FAÇADE code and the proof. The embedded compiler is responsible for searching the most

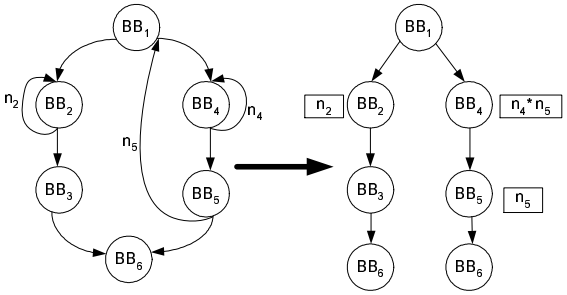

Figure 1. Tranformation of the CFG to tagged-tree.

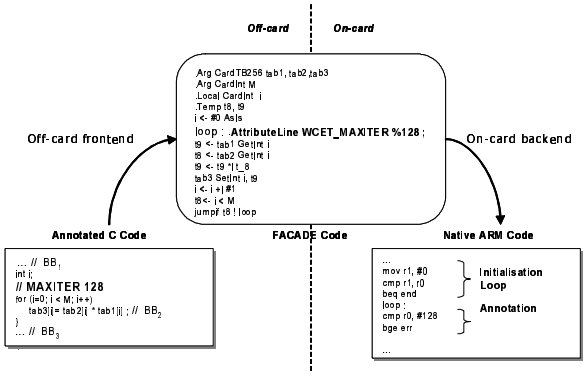

Figure 2. An example of annotation translation.

costly branch in the tagged-tree. As it decodes the FAÇADE instructions one by one, the embedded compiler has to translate the basic blocks of FAÇADE code into native ones.

Once the translation phase is finished, each basic block is assigned an execution time which corresponds to the sum of the number of cycles that will be consumed by each instruction. Then, the tagged-tree is used to compute the WCET of the program by starting with the root node and summing up the execution times of each basic block belonging to the same branch. The formula used to compute the WCET of the right branch is therefore:

$$WCET(Right\ branch) = WCET(BB_1) + n_4 \times n_5 \times WCET(BB_4) + n_4 \times WCET(BB_5) + WCET(BB_6)$$

The WCET of the respective branches are then compared and the global WCET value is sorted out. If the deadline of the program can be met, the code can be executed, otherwise an error message is sent to the off-card part.

To compute the n_i used in the formula presented above, we use annotations inserted in the source code either by the programmer or by code analysis tools. Figure 2 illustrates the CAMILLE compilation scheme of an annotated C code. The off-card part should be extended with a static flow analysis tool capable of translating the assertions inserted by the programmer in the C code to FAÇADE annotations. Figure 2 shows an example of such a programmer-inserted annotation, represented by the C comment `// MAXITER 128` which declares that the following multiplication loop will iterate 128 times. This C comment is simply translated by the static flow analysis tool into the FAÇADE annotation `.AttributeLine WCET_MAXITER %128;`.

While decoding FAÇADE instructions, if it reaches an annotation, the embedded compiler needs to verify it. For instance, if the off-card part claims that a loop will not iterate more than 128 times, the embedded compiler has to explicitly insert code to exit the loop when the loop has iterated 128 times.

The assertion `cmp r0, #128; bge err` compares the register `r0` which stores the value of variable `M` with the declared number of iterations (128). If `r0` is greater than 128, the program exits the loop and branches to an error label.

5 Conclusion and future work

We presented in this paper the scheme we propose to safely compute WCET in a resource-constrained operating system. By distributing the computation between the off-card part running on a powerful workstation and the on-card part specific to the hardware included in the smart card, we are able to circumvent the very strict memory and CPU limitation of the device. We guarantee the safety of our scheme by inserting assertions in the generated code to validate the annotations sent by the off-card part, and by preventing information leaks from the card to the outside. Finally, we show that our scheme can be easily implemented in a secure smart card operating system as CAMILLE. We are now working on an extended architecture which would permit to safely export hardware information outside of the card. This would allow using the IPET technique to compute the WCET off-card without risking to comprise the security of the embedded system.

References

[1] D. Deville, A. Galland, G. Grimaud, and S. Jean. Smart Card Operating Systems: Past, Present and Future. In *Proceedings of the 5th NORDU/USENIX Conference*, February 2003.

[2] D. R. Engler. *The Exokernel Operating System Architecture*. PhD thesis, Massachusetts Institute of Technology, 1998.

[3] G. Grimaud, J.-L. Lanet, and J.-J. Vandewalle. FAÇADE: A Typed Intermediate Language Dedicated to Smart Cards. In *Software Engineering — ESEC/FSE*, number 1687, pages 476–493. Springer-Verlag, 1999.

[4] G. C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[5] J. Engblom, A. Ermedahl, M. Sjdin, J. Gustafsson, and H. Hansson. Worst-Case Execution-Time Analysis for Embedded Real-Time Systems. *Software Tools for Technology Transfer, special issue on ASTEC*, 2001.

[6] A. Colin and I. Puaut. A Modular and Retargetable Framework for Tree-Based WCET Analysis. In *13th Euromicro Conference on Real-Time Systems*, June 2001.

[7] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Real-Time Systems*, 2:249–274, 2000.

[8] Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software using Implicit Path Enumeration. In *Workshop on Languages, Compilers & Tools for Real-Time Systems*, pages 88–98, 1995.

[9] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Journal of Real-Time Systems*, 1(2):159–176, September 1989.

[10] E.W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[11] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO '96*, volume 1109 of *LNCS*, pages 104–113. SV, 1996.

Inspection of industrial code for syntactical loop analysis

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

Abstract

Flow analysis can be used in WCET analysis to, e.g., determine loop bounds and infeasible paths. Such information can be used by low level analysis and for actual WCET calculation. An efficient flow analysis method is syntactical analysis. This method identifies certain predefined syntactical constructs. It is not reasonable to believe that we will be able to use syntactical analysis to identify all conceivable constructs. Therefore we need to learn which to prioritize.

This paper suggests methods for inspection of industrial code to record properties of code that will give hints how to design a syntactical analysis. We describe the code properties to analyze, and present preliminary results for some industrial real-time code.

1 Introduction

We aim to develop methods for automatic estimation of WCET. *Automatic* in this context means that manual annotations of loops and infeasible paths should be avoided. The task for an automatic flow analyzer in this sense is to calculate "flow facts" such as loop bounds, infeasible paths etc. [4]. This can not in general be done without interaction with the user. If, e.g., a loop bound is dependent on input data these data has to be supplied by the user. The user may also need to supply which parts of the code that should be analyzed. If, e.g., infinite loops are part of the entire program the responsibility is on the user to exclude these.

There are several methods that can be used for flow analysis, [9, 5], one being abstract interpretation and another syntactical analysis.

Syntactical analysis will identify syntactical constructs where we can calculate the number of iterations by looking at the loop syntax (in particular there is no need to calculate the result of each iteration). The initial values of loop variables, loop increment and loop termination conditions can be translated to recurrence equations, which can be solved. The advantage of this method is that it is efficient, with a complexity that is linear to the size of the program. In best case we succeed in bounding all loops in the analyzed code. In case it finds loops which does not match any known pattern, the method can still be used as a "filter" to reduce the amount of work needed by other methods.

If we for example use syntactical analysis in combination with abstract interpretation, e.g., as described by Gustafsson, [7], the remaining loops may be possible to bound by the abstract interpretation. Specially, there is a possibility to syntactically find final values of all variables assigned in the loop body. In such cases the entire loop can be collapsed and replaced by a piece of sequential code, making life easier for other analysis methods. Even in cases when only a few occasional loops can be bounded by the syntactical analysis, this can still benefit in a combination with other methods since the tightest of the loop bounds found by the different methods can be chosen.

The syntactical method can be thought of as a data base of predefined patterns (syntactical constructs) and we can check for each loop if it matches any of the patterns in the database. Since there is a huge number of possibilities to write loops we can not expect the syntactical analysis to be able to recognize all of them. However, we want to fill this database with patterns that matches as many loops as possible when analyzing real-time code. Therefore we need to get an idea of what loop constructs that are common in real-time systems and that are fairly simple to identify without too much of computational efforts.

In cases when loop patterns are not suitable to be analyzed by the syntactical method some alternative methods need to be used. The results from this inspection might also be useful to find the requirements on such methods.

2 Related work

In comparison between specInt95 and some code for embedded systems, performed by Engblom [3], he concluded that using code from desktop applications as base for testing tools for embedded systems may be dangerous due to significant differences in programming style for these categories of programs.

An investigation of a large set of industrial code, carried out by Engblom [2] shows, e.g.,

- Recursive functions may be expected but are not common.
- Use of function pointers can be expected, but is in most programs rare or even absent.
- Deeply nested loops at a global level are quite common.
- Unstructured loops may occur.
- Multiple entries to the program may occur.
- Functions from other modules may be invoked.
- Multiple loop exits are quite common (almost 1 of 3 of those in the investigated programs).
- High decision nest complexity is not common.

In [11] we see that the use of a WCET tool in practice may lead to unexpected problems. Some of the problems encountered, related to loops, were:

- Sentinels in arrays are sometimes used as termination conditions in loops.
- Loop termination conditions may depend on input.
- Loop counters may be calculated using complex arithmetics.
- Complex arithmetics is sometimes used in loop termination conditions.

Another related investigation, [1], shows that for a certain operating system the program constructs were quite simple. No nested loops, unstructured code or recursion were found. Some function pointers were used.

Our conclusion is that a careful investigation of industrial code for real-time systems is needed to prepare a WCET analysis tool to handle code that exists in reality. For a syntactical analysis we specially need to get more knowledge about loop details.

3 The inspection

We will use an existing flow analyzing tool for automatic inspection, *SWEET* (SWEdish Execution Time tool, [8]). Most of the methods that needs to be implemented for this investigation are quite basic, and can probably be re-used when implementing the actual syntactical analysis, at least for the simple cases. This tool takes intermediate code, produced by a C compiler, as input.

Code needs to be inspected in a variety of views to get useful measurements. Some of the properties measured by Engblom can benefit the design of syntactical analysis. However we need to get a more detailed view, specifically about loops. The following inspection items will be explained in more detail in the subsequent sections: the overall program structure, the loop nesting and details about individual loops. Some of these items have been covered in previous work. They, however, need to be done also for the code used for this more detailed investigation, intended to aim the design of syntactical analysis.

3.1 Program structure

The program structure can be analyzed by examining the call graph. The depth of a call graph can give hints about how much calculation power that is needed for analyzing the code. Extending the call graph with information about the presence of loops will give more information. Comparing a call graph in DAG form with one in tree form will show to what extent functions are reused. Calculating loop bounds in functions called from different sites in a context sensitive manner will cost more in calculation efforts, but may give better (tighter) loop bounds. Recursive functions should be recorded separately. These may be hard to bound syntactically, but simple cases may be possible to handle in case an interprocedural analysis is performed.

3.2 Loop nesting

The nesting level of loops can be counted both locally (per function) and globally (per task or program). It is important to bound deeply nested loops, since they will have the highest influence on the final WCET. Also other analyzing methods (e.g., abstract interpretation) may suffer from high computational load when analyzing these.

3.3 Loop conditions

Reducible loops have a single entry point (we only consider reducible loops since we assume that irreducible loops are already replaced by their multiple reducible loops counterpart [12]).

There may be an arbitrary number of exits from loops. Those with no exits (infinite loops) may occur in real-time systems. We cannot find the loop bound of such a loop syntactically. However, although we can't analyze them we need to count them to conclude how big portions of a program that needs to be excluded from the analysis.

Loops that contain more than one exit branch have been shown to be analyzable syntactically [10], but are in general harder to calculate the bounds of, both in terms of computational efforts and implementation issues. Thus the number of loop exits is of interest as well as the number of targets of these exit branches.

For each termination condition there will be one or more variables involved (otherwise the loop is either equivalent to an infinite loop or with a non-looping construct). Each such variable needs to be carefully investigated. The following properties of the variable may be of interest:

- The initial value.
- The variable update in the loop.

Initial values

We are mostly interested in the initial value at the loop termination condition in which it is used (do loops may be handled a bit different, since the initial execution of the loop body might affect the initial value of some variables). The initial value can either be deduced from a constant, or depend on a variable that is updated in an outer loop or depend on an input value to the code.

Initialization from constants. If the initial value only depends on constants this will simplify the analysis, and makes it more likely that we can find a loop bound. The constant value can be found as an assignment from an expression containing only operands that are constants or that are other variables that recursively depend on constants. It is of interest to record the locations of the constants in terms of function nesting level. If all the constants are not present in the same function as were they are used, a more advanced syntactical analysis is needed (e.g., a global analysis).

Initialization from variables updated in an outer loop. There are certain kinds of nested loops where the loop bound of the inner loop can be calculated even if the number of iterations depend on an outer loop induction variable [8]. Therefore these class of initializations are of a certain interest. The following properties needs to be recorded:

- The loop nesting levels between the use and initialization. For example "triangular loops" can be recognized by a syntactical analysis, and it may be possible to find loop bounds in case there is a loop nesting level of one between the two loops.
- The properties of the source to the initialization are of interest. Gerlek et.al., [6], makes a classification of induction variables that may be useful as basis. The problem of finding the bound for a loop can be expected to vary based on these.
- Is the assignment of the initial value done from an expression containing more than one induction variables in outer loops? If so, it will be a harder case to handle.

Initialization from input values. The initial value can in some cases be deduced from some input value. Input to a real-time system may in general occur in various

ways. One would be that the code just use some global variable that appears uninitialized to the analyzer (e.g., the code reads from a labelled input port).

Different analyzable units (e.g., tasks) may need to communicate to each other, and this data will appear as inputs. The actual input data can in such case be deduced to some global entity, e.g., global variables. These variables may occur as initialized to the analyzer.

In case of an analysis local to functions, also the function arguments are input.

There is a special problem in identifying input values. The analyzer need to distinguish between those global variables that are input to the analyzed code and those that are rather constants or induction variables in the context of the use in a loop condition. This can normally not be done by looking at a part of the system code in isolation. In general it can be hard to perform an analysis on a complete system because of the interaction of an operating system.

In our inspection all definitions that depends on global variables will be recorded as inputs. When doing the syntactical analysis the user may supply information about which are input variables and the value of these. Optionally a certain pass just to identify intertask communication might be developed. Finding loop bounds if the initial values depend on input is much the same problem as finding it for constant initial values.

Updating of variables in the loop body

One or more of the variables in a loop termination condition must change their value in the loop, or the loop will never terminate using that condition. A simple form is a loop counter, i.e. a variable which value will contain the current iteration number while executing the loop. Also other forms of updates are of interest. If the value of the variable forms a series which we can identify, there is a good chance that we can calculate the loop bound. The following properties will be recorded:

- Is the update self-referencing, i.e. is the right hand side an expression that includes the target of the assignment (directly or indirectly) within the loop.
- The operator(s) applied in update(s). If only linear updates are involved a calculation of the loop bounds can be expected to be less complicated.
- The other operand(s). This may be constant (directly or indirectly) or an input dependent variable. In both cases it will probably be easier to find loop bounds than for induction variable dependent initialization values.
- Do the other operand alters its value in the loop? In case it does, it is probably harder to find a resulting loop bound (e.g., if an increment is altered in a conditional statement).
- Is the update statement conditional? If so, it is in general not possible to find the wanted series.

3.4 Branch conditions

Branch conditions are also of interest to the syntactical analysis. Infeasible paths can be found if the reaching definition for the involved variables are calculated in a context sensitive manner. The values of variables involved in conditions therefore needs to be recorded in the same manner as initial values of loop conditions.

3.5 Arrays

In case array elements are used in place for simple variables in loop termination conditions this imposes difficulties to the syntactical analysis. Such language elements can take many different syntactical forms. Below are listed some that will be recorded in the first round.

- *Initial value.* A variable in a loop termination condition is an array element (or its value depends on an array element), and this variable is loop invariant. For certain sub-cases we might be able to find loop bounds.
- *Update.* A variable in a loop termination condition is updated using an expression containing an array element (or a variable which value depends on an array element). The updated variable is an induction variable. The index variable might be an induction variable in an outer loop.
- *The "sentinel problem".* There is some loop termination condition that compares an array element with some other value. The array index is an induction variable in the current loop. This category can be tricky to handle. But since we know that strings as well as sometimes pointer arrays are often traversed this way, it is important to know the number of occurrences of this kind. Maybe we can calculate the loop bounds for some sub-cases.
- *Other uses of array values.*

3.6 Pointers

The use of pointers in loop termination conditions imposes problems for the syntactical analysis. Pointers that points to a distinct variable in a certain context might give us some hope. We should distinguish these uses of pointers from other.

4 Preliminary results

Code from three insdustrial systems (in total more than 80k lines of source code) has been inspected concerning the properties shown in the tables below. Inspection has been done in a context sensitive manner, meaning that each call site of a function has been counted rather than the function definition. The reason was that we wanted to be able to detect interprocedural dependencies.

The maximum call tree depth per root function (e.g., task) was 7. There were 392 loops with a single exit while there were 166 with 2, 58 with 3, and 12 with more that 3 exits. We found that 592 loops had a single target of the loop exit, which might still give them a hope to be syntactically bound, while 36 had multiple exits. No recursive function was found. Three infinite loops and three uses of function pointers were found in one of the systems.

In table 1 we can see that the global nesting depth is quite big. The difference between the local and global nesting means that functions containing loops are often called within loops. The rightmost column is a count of the nodes in a scope tree (a call tree where each function call and each loop instance has a node, as discussed in 3.1) where only the loop scopes are counted.

Table 2 shows the re-use of functions. Note that only functions containing loops has been considered.

The used method for performing the inspection in a context sensitive manner resulted in some of the root-functions (tasks) growing quite large. As a result of this,

Table 1: *The number of loops with a certain nesting level.*

Level	Global count	Local count	Scope count
0	68927	574	0
1	51262	53	384
2	8985	1	1771
3	250		9036
4	106		14919
5	32		28467
6			36138
7			26834
8			10893
9			1056
10			64

Table 2: *Re-use of functions.*

Nr of calls per function	Nr of functions
1	148
2	44
3	26
4	15
5-9	27
10-14	15
15-19	9
20-30	8
30-40	10
40-50	5
>50	74

the analysis could not be completed in some cases. The analysis of loop variables could only be carried out for 290 loops. Of these, 138 were found to be dependent on dereferenced pointers and was not further investigated.

In the remaining loops there were 175 variables involved in termination conditions, of these 154 were updated in the loop (table 3), and 3 were updated in different conditional branches of the loop. The operations of the update has not been investigated in detail, but the operands are based on constants only. Note that because of the context sensitive analysis 'loop' and 'variable' should in this paragraph be read as and *instance* of a loop and *instances* of a variable respectively.

In most cases the initial value could be found as a constant in the same function, however in 14 cases in the calling function and in 5 cases the value was obtained from a called function. The value were never found in a cross call-tree node.

The initial values were in 5 cases defined based on variables updated in the enclosing loop (table 4).

Table 3: *Definition from program constant.*

Distance to defining function	Update operand	Initial value
-1	0	5
0	154	151
1	0	14

Table 4: *Definition from induction variable.*

Distance to loop	Update operand	Initial value
0	0	5

5 Conclusions and Future work

The big nesting level indicates that there might be lots of dependencies between loops, as well as a needing to search for loop variable definition in outer functions. However, the subset of loop variables that we were able to investigate does not confirm this. Surprisingly only a few loops actually depend on outer loops and should be simple to bound. Instead, there are problem with dereferenced pointers in lot of cases (48% of the analysed loops). A conclusion is that to be able to bound all loops a flow analysis need to handle pointers in a powerful way. A manual inspection was made on one of the systems to get some hints about the loops with pointers (this was a medium sized system, containing 45 of the 138 dereferenced pointers). It appeared that all of them were of interprocedural type (function parameters or global variables).

To make the results more valuable further inspections need to be done. It is necessary to handle functions in a context sensitive manner without linking them all together. It is also necessary to make a more detailed investigation of the dereferenced pointers.

To increase the usefulness of the inspection more code will need to be collected.

6 Acknowledgments

We would like to thank ESAB and CC Systems for making their code available for our work. This work is performed within the Advanced Software Technology (ASTEC) competence center, supported by the Swedish National Board for Industrial and Technical Development (NUTEK).

References

[CP99] A. Colin and I. Puaut. Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System. Technical Report No 1277, IRISA, November 1999.

[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.

[Eng99a] J. Engblom. Static Properties of Embedded Real-Time Programs, and Their Implications for Worst-Case Execution Time Analysis. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*. IEEE Computer Society Press, June 1999.

[Eng99b] J. Engblom. Why SpecInt95 Should Not Be Used to Benchmark Embedded Systems Tools. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*. IEEE Computer Society Press, May 1999.

[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] J. Gustafsson, B. Lisper, N. Bernmudo, C. Sandberg, and L. Sjöberg. A Prototype Tool for Flow Analysis of C Programs. In *WCET 2002 Workshop Wienna*, pages 9–12, aug 2002.

[GSW95] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.

[Gus00] J. Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Department of Computer Systems, Information Technology, Uppsala University, May 2000.

[HSRW98] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding Loop Iterations for Timing Analysis. In *Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, June 1998.

[RSE+03] M. Rodriguez, N. Silva, J. Estives, L. Henriques, D. Costa, N. Holsti, and K. Hjortnaes. Challenges in Calculating the WCET of a Complex On-board Satellite Application. In *WCET 2003 Workshop Porto*, 2003.

[San03] C. Sandberg. Elimination of Unstructured Loops in Flow Analysis. In *WCET 2003 Workshop Porto*, 2003.

A New Timing Schema for WCET Analysis

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

Abstract

The timing schemas proposed in various approaches for Worst Case Execution Time (WCET) estimation lack the ability of handling more path-based low level analysis and non-structured code. As it is extremely efficient in the computational stage, we propose a technique to handle these cases, while still retaining most of the efficiency of the syntax tree timing schema. This is achieved by changing the rules for the construction of the computational tree. As a result, the analysis becomes more path aware without affecting the safety of the approach.

1 Motivation

Tree-based calculation methods for WCET analysis were first proposed by Park and Shaw [1], based on a syntax tree representation of the program. A *timing schema* is attributed to certain high-level language constructs, which is essentially a formula for computing the upper bound of their execution time. Bernat et al. [2, 3] extended this approach to incorporate Execution Time Profiles (ETP) instead of integer values. Obtaining ETPs requires a tracing mechanism whereby the data are collected, but there are some related drawbacks. Some tracing mechanisms (e.g. via the NEXUS interface cf. [4]) do not provide compatible traces, so a path-based approach would be much more suitable, but it must cope with computational complexity.

Alternatively, instrumentation points (ipoints) can be manually or automatically placed into the code to generate a trace. There is generally no restriction on where ipoints can be placed in the code, thus basic blocks could have several ipoints, whilst others have none at all. Here we consider that each basic block contains one ipoint at most, without loss of generality. Nonetheless, this still implies that inaccuate placement exists. Inaccuracy in this context means an ipoint that is not placed at the very beginning of a basic block. If this happens for the first basic block of alternative paths, unnecessary overestimations occur.

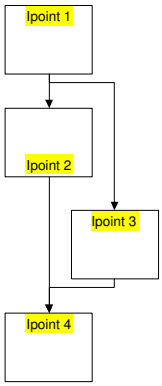

Figure 1. Ipoint Placement Example

For illustration purposes, the assumption of only the longest observed execution time being used for computation is made. Figure 1 provides a simple example of an `if-then-else` code. With the previous approach the block containing *ipoint 1* has two execution times, with only the longer being used. Additionally the block containing *ipoint 2* appears shorter and as a result is less likely to

contribute to the overall WCET in the computational stage. The assertion of potential overestimations applies to both probability distributions and integer values. Furthermore, unstructured code needs better support, as provided in the previous approaches. Unstructured code may arise from a couple of sources, ranging from deliberately or automatically set gotos in mission critical software (cf. e.g. [5]), multiple loop control conditions (e.g. ”if within range”), compiler optimisations or Ada exceptions.

2 Program Representation

To solve the problems described in the previous section an elemental change to the presentation is proposed. In a first move, the unit of computation is no longer basic blocks, but rather the transition from one ipoint to another ipoint. This corresponds to moving the weight of computation from the nodes of a control flow graph to the edges. As such the program representation and the timing schema are changed. In relation to the *extended syntax tree* which is used in [2] to generate the computation formulas, a computational tree is produced, which will be named CTR throughout this paper. Similar to the previous approach four constructs have to be considered.

2.1 Node

A leaf node in the CTR represent a transition from one ipoint to another ipoint. The content may either be an integer number or an ETP.

2.2 Sequence

A sequence of nodes is combined the same way as with the previous approaches. Dependent on the basic setup some sort of convolution or a simple addition may be used.

2.3 Alternatives

The representation of an alternative path now contains the transition from an external ipoint into the alternative path and the transition out of the alternative path to an external ipoint. As a result, a set of alternatives has to share the same starting and the same finishing ipoint (cf. Fig. 2 and Fig. 3).

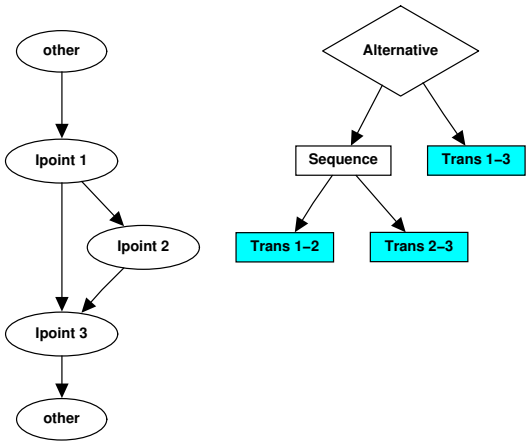

Figure 2. Optional Code

In the case of a simple `if-then` construct omitting the `else` part as depicted in Figure 2, the alternatives contain a single transition and a sequence of transitions respectively. An `if-then-else` construct has sequences of transitions in both alternatives. The two or more alternative parts – more alternatives may be the result of switch statements – are evaluated using the `max` operator as defined in the respective approaches.

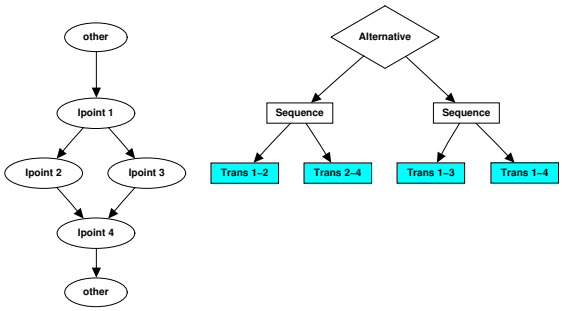

Figure 3. Alternative

2.4 Loop

A loop in this context consists of three parts. A loop entry, a loop iteration and a loop exit. Figure 4 provides an example of a loop and its representation. The basic requirement of these three blocks is that the end ipoint of the entry, the start ipoint of the exit and the start and end ipoint of the iteration have to be identical. The simple case of a loop is when the loop head contains an ipoint and this ipoint is part of every path leading into the loop, out of the loop and during any loop iteration. The loop head is defined as any code from the start of the loop to the point where an exit condition is met. In the computational stage, a loop is treated as a sequence of the entry node, N iterations of the iteration node and the exit node. For the iteration node, the previously proposed unrolling of iterations may take place.

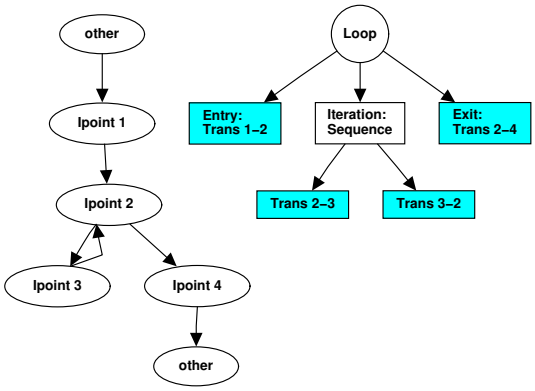

Figure 4. Loop

In the case of the loop head having no ipoint (which fulfils the the criteria in being part of any path as described above) an ipoint of the body that has to be part of any iteration may be chosen as the common ipoint. As the combination of the entry and the exit node contains already one iteration of the loop, two measures are necessary. On one hand the number of iterations N has to be adjusted for the computational stage. On the other hand, an alternative to the loop has to be created, which represents the fact that the loop may not iterate at all. Figure 5 provides the CTR for a graph similar to the one in Figure 4, where *Ipoint 2* is assumed to be in the loop body. However, this transformation will usually be straightforward, as the transition between *Ipoint 1* and *Ipoint 4* should already be visible in the control flow or ipoint graph.

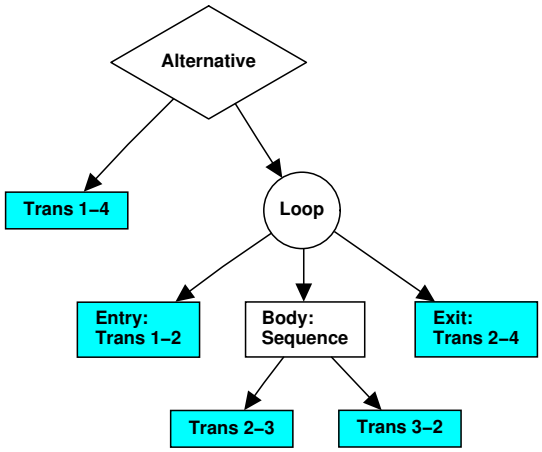

Figure 5. Loop with no Ipoint in Head

2.5 The Timing Schema

The timing schema provided uses the abstract operators $\otimes$ and $\odot$. In the calculation, these revert to a simple addition and multiplication in case of integer numbers, and a convolution and power operator in ETP-based approaches.

- $C_{i,j}$ describes the WCET of the transition from ipoint i to ipoint j which may be a measured leaf node or a
- Sequence (i, j, k): $C_{i,k} = C_{i,j} \otimes C_{j,k}$
- Alternative (i and j outside the alternative and $C_{i,j}^{1..o}$) representing the o alternative paths):
  $C_{i,j} = \max(C_{i,j}^{1}, C_{j,l}^{2}, \ldots, C_{i,k}^{o})$
- Loop (i and k outside then loop and j inside the loop):
  $C_{i,k} = C_{i,j} \otimes (C_{j,j} \odot n) \otimes C_{j,k}$

Within the loop expression n is equal to the number of loop iterations if the common point j lies within the loop head, or number of iterations minus one if the common point j lies within the loop body.

3 Discussion

The actual schema is not provided in this paper in terms of mathematical equations due to space restrictions, but the schema may be derived straight forward from the tree representation. The question arises, how the proposed approach supports the different aspects described in the motivation. The Nexus interface is supported by the loose definition of the contents of the alternatives. As only start and end ipoints are defined, the rigid construction rules are resolved. An enforced ipoint, by introducing an additional branch instruction, where deemed necessary, is comparably straight forward. The overhead of "inaccurately" set ipoints is removed by the transitional description of the nodes. This results overall in a tighter bound on the real values. Possibly most striking is the support of some non-structured code. Especially loops with more than one exit condition are a common problem. This is resolved as the entry and exit nodes of a loop may contain several paths leading to or from the common node. Code like exceptions may be expressed, but may lead to prohibitive execution times of the analysis, as the analysis moves towards a fully path-based approach with every exception considered.

The freedom of expression is possibly the biggest drawback of the approach. On the one hand, it is hard to "read" the CTR and associate its nodes with real code constructs, which is much more straight forward with a syntax tree representation. On the other hand there is more than one CTR solution for almost any real world program. In the extreme, a fully path-based CTR is possible with a set of alternatives at the top level, each representing a possible walk on the control flow graph. The move from a computationally feasible to a computationally infeasible analysis is easily done.

4 Conclusion

In this paper we have proposed a powerful and flexble program representation and a timing schema, which may be applied to any timing schema based approach. The only requirement is the concept of transitional computational cost being associated with the leaf nodes of the CTR. This schema may be applied independently of the low level analysis used to derive the values, without loss of safety of the results, while supporting tighter WCET bounds. Additionally it works equally for integer values or ETP-based approaches.

Future work in this area should focus on experiments to establish the actual gain, which may be expected by using this schema compared to the previously proposed schemas.

References

[1] C. Park and A. Shaw, "Experiments with a program timing tool based on source–level timing schema," *IEEE Transactions on Computers*, vol. 24, pp. 48–57, May 1991.

[2] 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.

[3] 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.

[4] S. M. Petters, "Comparison of trace generation methods for measurement based WCET analysis," in *3nd Intl. Workshop on Worst Case Execution Time Analysis*, (Porto, Portugal), July 1 2003.

[5] J. Engblom, "Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis," in *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium (RTAS '99)*, (Vancouver, Canada), IEEE, June 1999.

Petri Net Level WCET Analysis

Friedhelm Stappert
C-LAB
Fürstenallee 11, 33102 Paderborn
Germany
friedhelm.stappert@c-lab.de

Abstract

We present an approach for Worst-Case Execution Time (WCET) Analysis of embedded system software that is generated from Petri net specifications. The main characteristic of the approach is that standard Petri net analysis methods are utilized in order to automatically derive additional flow information for WCET analysis. Furthermore, the approach presented in this paper clearly separates the analysis of model behavior from the other WCET analysis phases. The method is compared with similar approaches for WCET analysis on the model level. Furthermore, an application example is presented.

1. Introduction and Related Work

A safe and precise WCET analysis must always take into account possible program flow, like loop iterations and dependencies between if-statements. This information is usually derived from the source- or object-code of a program. However, source code is often generated from higher-level specifications like StateCharts or Petri nets. In such cases, it is possible to derive additional flow information by analysing the possible behavior of the according model. Using this additional information, a more precise WCET estimation is achieved than by just analysing the generated source code alone.

To generate a WCET estimate, we usually consider a program to be processed through the phases of *program flow analysis*, *low level analysis* and *calculation*. Most WCET research groups make a similar division notationally, but sometimes integrate two or more of the phases into a single algorithm.

The program flow analysis phase determines possible program flows, and provides information about which functions get called, how many times loops iterate, if there are dependencies between `if`-statements, etc. The information can be obtained by *manual annotations* (integrated in the programming language [15] or provided separately [3, 9, 16]). The flow information can also be derived using *automatic flow analysis* methods [5, 11, 12, 17]. Most approaches for automatic flow analysis are based on the source- or object code of a program. In contrast, this paper presents an approach based on Petri nets, thereby extending the program flow analysis phase from the source-code level to the model level.

The low-level analysis phase determines the execution time for each atomic unit of flow (e.g. an instruction or a basic block), given the architecture and features of the target system. Low-level analysis takes into account performance enhancing features like caches, branch predictors and pipelines.

In the calculation phase a program WCET estimate is computed, combining the information derived in the program flow and low-level analysis phases.

In recent work, WCET analysis on the model level has been considered for the case of StateCharts [8, 7] and Matlab/Simulink models [14]. Previous work also includes analysis on the algorithm level for the case of an MPEG decoder [1]. In the latter, knowledge about the algorithm performed by the given code – namely decoding an MPEG stream – and its possible input is exploited in order to achieve better results for the estimation of the WCET of the code. The WCET analysis of Matlab/Simulink models presented by Kirner et al. basically works by generating wcetC [13], a special form of C with additional annotations suitable for WCET analysis. These annotations include e.g. loop bounds that can be easily derived from the Matlab/Simulink specification. For each block of the Matlab/Simulink model, the generated wcetC code is analysed by an existing WCET analysis tool. The results of the analysis – namely the calculated worst-case execution times of single blocks and tasks – are then propagated back into the high-level representation of the model in the Mat-

lab/Simulink environment. Thus, the main contribution of the approach is to integrate a WCET tool in the overall environment of Matlab/Simulink using special annotations in the generated code in order to provide information about loop bounds and to propagate the calculated WCET values back to the according parts in the modelling environment. The analysis does however not derive additional information about the behaviour of the model itself.

The approach presented by Erpenbach in [8] works on StateCharts [10]. It is similar to the concept described above in that it also uses the code generated for single states of the StateChart model as basis for the WCET analysis on the model level. In addition, information is compiled about the maximum number of state transitions that can occur before the system becomes stable again after the triggering of an external event. The WCET of each possible state transition is derived by analysing the corresponding generated source code in isolation, using an existing WCET analyser. All possible sequences of state transitions – i.e. from the triggering of an external event to a stable state – are represented by an extended control flow graph. The final WCET is calculated by finding the longest path in this graph.

A key observation for the two approaches is that they mix the low-level analysis, i.e. the analysis of concrete execution times, with the analysis on the model level, by back-annotating these times into the corresponding model parts. However, since the low-level WCET analysis is performed on pieces of the generated code in isolation, the results of the overall WCET calculation are less precise since global timing effects reaching across the borders of these pieces cannot be considered.

The main advantage of the approach presented here is that it clearly separates the analysis of model behavior from the other WCET analysis phases. The model analysis does not use or produce information about execution *times*, but instead delivers information on the worst-case execution *count* of certain parts of the model. The analysis of concrete execution times is the task of a subsequent low-level analysis. Thus, the method presented here is independent from the implementation of the other WCET analysis phases.

2. Petri Net Analysis

The purpose of Petri net WCET analysis is to find the longest possible execution time a given Petri net needs to go from a defined start- to a defined end-marking. The overall architecture of our Petri net based WCET analysis is depicted in Figure 1. The analysis – divided into the two phases *reachability analysis* and *behavioral analysis* – compiles additional information about the behavior of the net, which is handed over to an existing WCET tool [4] in form of special annotations in the generated code, using the *Flow Facts* language introduced in [3].

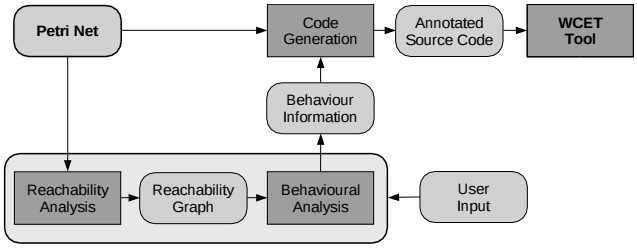

Figure 1. Petri Net Analysis Architecture

A detailed description of the reachability analysis and behavioral analysis is given in [18]. In this paper, we restrict ourselves to the description of the flow facts generated from the results of the Petri net analysis. The analysis derives the following information about the given Petri net:

- The worst-case number of steps the net can make until the defined end-state is reached
- For each transition, an upper bound on how often it will fire at most during all steps
- For each step, the set of transitions that could fire at that point in time

Here, a *step* is assumed to be the firing of exactly one transition.

Note that the Petri net analysis does not make any assumptions about the source-code generated from the Petri net to be analysed. Particularly, no information about execution times of single transitions is needed. Dealing with timing and source-code is done much more efficiently and precisely by the subsequently employed WCET tool. The only assumption that is made about the implementation of the Petri net execution is that it performs one transition firing per step, which is a common execution paradigm for Petri nets. Furthermore, a clear mapping between the transitions of the Petri net and their corresponding source code has to be ensured. This is achieved by integrating the source-code annotation into the code generation process as shown in Figure 1.

2.1 Example

Figure 2 shows a simple Petri net, which is a small part of a large net modelling the behavior of a Khepera minirobot. The net receives its input by means of the two input places $param$ and $nextState$ of transition In. Then, depending on the values, one of the transitions $Reset, Active, Active1, Active2, Active3$ computes the new output values, which are then returned via the

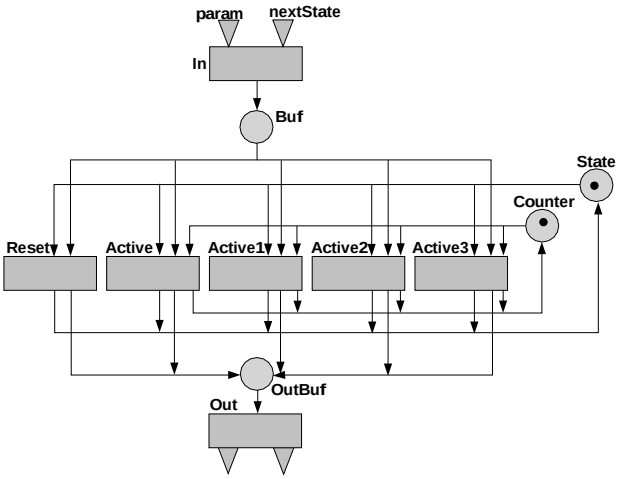

Figure 2. Example Petri Net

```
while net is alive
  get enabled transitions
  if T1 is enabled
    fire T1
  if T2 is enabled
    fire T2
  ...
end while
```

Figure 3. Petri Net Code

output places of transition *Out*. The code generated for the execution of the Petri net might look like as sketched in Figure 3.

The reachability graph resulting from the given Petri net is shown in Figure 4. As the first result, the Petri net analysis returns that the net will take at most three steps until the end-state is reached. Consequently, we know that the `while` loop in Figure 3 will take at most three iterations.

As the next result, we can see that each transition of the Petri net can fire at most once during the whole execution. Therefore, a flow fact of the form $\texttt{Loop:[]:}x_{\texttt{T}} \leq 1$ is generated for each transition $T \in \{In, Reset, Active, Active1, Active2, Active3, Out\}$, stating that the corresponding basic block in the generated code will be executed at most once during all iterations (denoted by the '`[]`' brackets) of the `while` loop (named `Loop`). For a detailed specification of the Flow Fact language we refer to [6].

Furthermore, the Petri net analysis derives detailed information about which transitions may fire in each step. As can be seen in Figure 4, only transition *In* can fire in the first step, transitions *Reset*, *Active*, *Active*1, *Active*2, *Active*3 in the second step, and only transition *Out* can fire in the third step. This is reflected by the following generated flow facts:

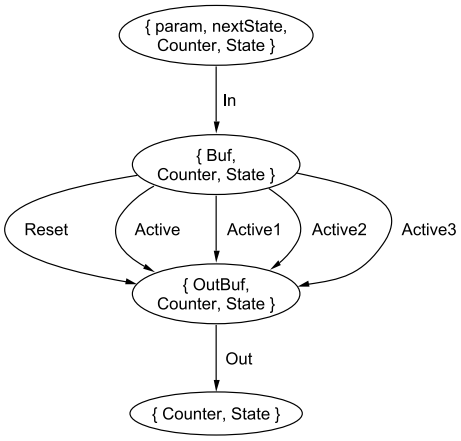

Figure 4. Reachability Graph

- $\texttt{Loop:<1>:}x_{\texttt{In}} = 1$
- $\texttt{Loop:<1>:}x_{\texttt{Reset}} + x_{\texttt{Active}} + x_{\texttt{Active1}} + x_{\texttt{Active2}} + x_{\texttt{Active3}} + x_{\texttt{Out}} = 0$
- $\texttt{Loop:<2>:}x_{\texttt{Reset}} + x_{\texttt{Active}} + x_{\texttt{Active1}} + x_{\texttt{Active2}} + x_{\texttt{Active3}} = 1$
- $\texttt{Loop:<2>:}x_{\texttt{In}} + x_{\texttt{Out}} = 0$
- $\texttt{Loop:<3>:}x_{\texttt{Out}} = 1$
- $\texttt{Loop:<3>:}x_{\texttt{Reset}} + x_{\texttt{Active}} + x_{\texttt{Active1}} + x_{\texttt{Active2}} + x_{\texttt{Active3}} + x_{\texttt{In}} = 0$

The generated code was analysed with our WCET tool prototype [4], assuming a NEC V850E as target processor [2]. The tool performs the low-level analysis and calculation as described in Section 1. The transition names in the above flow facts were manually replaced with the names of the according basic blocks in the generated code. This mapping can currently not be done automatically. However, as shown in Figure 1, annotating the source code is integrated in the code generation process. Therefore, in the final implementation the mapping will also take place without user interaction.

The results, together with the actual WCET of the code are shown in Figure 5. First, the code was analysed without consideration of the generated flow facts (column named 'no facts'). Only the mandatory upper bound for the number of loop iterations was given, since otherwise a WCET analysis would not be possible. Without flow facts, the actual WCET (rightmost column) was overestimated by about 100% (2425 versus 1177 cycles). When taking into account the flow facts (column named 'with facts'), the WCET estimation was 1744 cycles, which is significantly closer to the actual WCET. This improvement is due to the fact that the calculation phase can make less pessimistic assumptions about the possible execution paths of the code. The results of the low-level analysis are not affected.

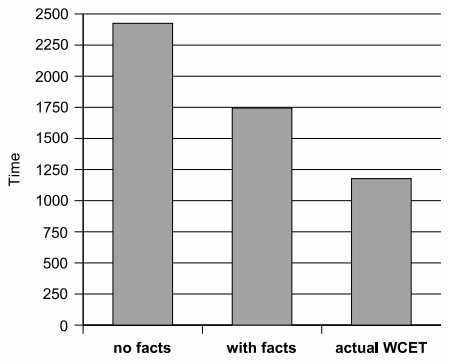

Figure 5. Analysis Results

3. Conclusion

In this paper, a WCET analysis for Petri nets was presented. Independently of the source-code generated for a given Petri net, the analysis compiles information about the behavior of the net. The gathered information is then converted to a set of flow facts, which are exploited by a subsequently employed source-code based WCET tool. Using this additional information, a more precise WCET estimation is achieved than by just analysing the generated source code alone. Information from the model (i.e. Petri net) level is therefore not lost on the next lower level of the generated source-code.

From the example in Section 2.1, it can be seen that the flow facts derived by the Petri net analysis lead to a much tighter prediction of the WCET, even for the relatively simple net presented in the example.

A basic feature of the presented approach is the strict separation from other WCET analysis phases, thereby consequently following our overall approach for a modular WCET tool architecture [4, 6]. This modular approach makes it easy to e.g. replace an algorithm for a certain analysis phase with a more efficient one.

Although the presented approach is based on Petri nets, the basic proposal of separating the model analysis from the other WCET analysis phases could also be applied for other modelling paradigms, like e.g. StateCharts or Matlab/Simulink models.

References

[1] Peter Altenbernd, Lars O. Burchard, and Friedhelm Stappert. Worst-Case Execution Times Analysis of MPEG-2 Decoding. In *Proc. 6th International EUROMICRO Conference on Real-Time Systems*, Stockholm, June 2000.

[2] NEC Corporation. *V850E/MS1 32/16-bit Single Chip Microcontroller: Architecture*, 3^{rd} edition, January 1999. Document no. U12197EJ3V0UM00.

[3] J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proc. 21^{th} IEEE Real-Time Systems Symposium (RTSS'00)*, November 2000.

[4] Jakob Engblom, Andreas Ermedahl, and Friedhelm Stappert. A Worst-Case Execution-Time Analysis Tool Prototype for Embedded Real-Time Systems. In Paul Pettersson and Sergio Yovine, editors, *Workshop on Real-Time Tools*, Aalborg, Denmark, August 2001. Published as Technical Report No. 2001-014, Department of Information Technology, Uppsala University, Uppsala, Sweden.

[5] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *Proc. Euro-Par'97 Parallel Processing, LNCS 1300*, pages 1298–1307. Springer Verlag, August 1997.

[6] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Sweden, June 2003.

[7] E. Erpenbach, F. Stappert, and J. Stroop. Compilation and Timing Analysis of Statecharts Models for Embedded Systems. In *The Second International Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99)*, Washington, D.C, Oct. 1999.

[8] Edwin Erpenbach. *Compilation, Worst-Case Execution Times and Schedulability Analysis of Statecharts Models*. PhD thesis, University of Paderborn, Germany, 2000.

[9] C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1978.

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

[12] N. Holsti, T. Långbacka, and S. Saarinen. Worst-Case Execution-Time Analysis for Digital Signal Processors. In *Proceedings of the EUSIPCO 2000 Conference (X European Signal Processing Conference)*, September 2000.

[13] Raimund Kirner. The programming language wcetc. Technical Report 2/2002, Technische Universitt Wien, Institut fr Technische Informatik, 2002.

[14] Raimund Kirner, Roland Lang, Gerald Freiberger, and Peter Puschner. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. In *EUROMICRO Conference on Real-Time Systems*, 2002.

[15] Raimund Kirner and Peter Puschner. Transformation of Path Information for WCET Analysis during Compilation. In *Proc. 13^{th} Euromicro Conference of Real-Time Systems, (ECRTS'01)*, June 2001.

[16] C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5(1):31–62, March 1993.

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

[18] Friedhelm Stappert. *From Low-Level to Model-Based and Constructive Worst-Case Execution Time Analysis*. PhD thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, University of Paderborn, 2004. C-LAB Publication, Vol. 17, Shaker Verlag, ISBN 3-8322-2637-0.

Measurement-Based Worst-Case Execution Time Analysis using Automatic Test-Data Generation *

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

Abstract

Traditional worst-case execution time (WCET) analysis methods based on static program analysis require a precise timing model of a target processor. The construction of such a timing model is expensive and time consuming.

In this paper we present a hybrid WCET analysis framework using runtime measurements together with static program analysis. The novel aspect of this framework is that it uses automatic generation of test data to derive the instruction timing of code sequences. Program paths are decomposed into subpaths to make execution-time analysis based on runtime measurements feasible.

1 Introduction

The current situation on WCET analysis is not satisfying, because widely used industrial-strength WCET analysis tools are still missing [7].

One challenge of WCET analysis is the variable instruction timing of processors. Complex processors have performance increasing features like caches or pipelines that maintain an internal state that depends on the execution history. Precisely modeling these features is problematic as on the one side it becomes quite complex and on the other side, exact information of the previous instruction stream cannot be calculated in general. A further problem is that the vendor's documentation of a processor's instruction timing is often a very rough approximation of reality. Problems arising on WCET analysis using static hardware modeling are described in [5].

A further problem of static WCET analysis is that it is very time consuming to model features of complex processors and furthermore, it has to be done for each processor for which WCET analysis is required. The alternative is to use measurement-based WCET analysis. However, simply performing exhaustive end-to-end measurements is not feasible for real-size programs. Therefore, measurement-based WCET analysis is used in combination with static analysis techniques. Approaches to hybrid WCET analysis do already exist [4, 1] but research in this area is just at the beginning. The path analysis problem of static WCET analysis is currently shifted to the problem of generating test data for measurement-based approaches. The current approaches require the user to provide test data or simply use random testing. In addition to runtime measurements, Ernst and Ye propose to switch back to traditional static WCET analysis techniques in case that the test data provided by the user did not cover all program blocks [4].

In this paper we present a measurement-based WCET analysis framework with automatic generation of test data. The problem of automatically generating the test data is tackled by standard program analysis techniques like *model checking* [2, 9] or *constraint-based analysis* [10]. The approach is based on decomposition of program paths into subpaths of program segments. A static WCET calculation method is used after the instruction timing of subpaths of program segments has been assessed by runtime measurements.

The paper is structured as follows: Section 2 gives a discussion about demands from industry for the use of WCET analysis tools. The measurement-based WCET

*This work has been supported by the FIT-IT research project "Model-Based Development of distributed Embedded Control Systems (MoDECS)".

analysis framework is described in Section 3. Section 4 discusses technical aspects of the framework. Finally, Section 5 concludes this paper.

2 Requirements for an Industrial-Strength WCET Analysis Tool

Before describing our new WCET analysis method, we give a motivation for its development by describing the industrial needs on a WCET analysis method. Previously proposed WCET analysis methods often only demonstrate several analysis capabilities without showing their applicability in an industrial environment. To be more precise, the following list gives demands for a WCET analysis tool raised by people working in industry. This list also contains aspects regarding the use of modeling tools like MATLAB/Simulink, as they are increasingly used in industrial software development.

1. The tool must work with minimal user interaction. In particular, it cannot be expected that users of the tool provide manual code annotations about possible and impossible execution paths of the code. For example, when using a modeling tool like MATLAB/Simulink, the WCET analysis tool must be able to extract this information by analyzing the code generated by the code generator of MATLAB/Simulink.

2. The method must integrate into the development tool chain of customers without modification of tools from the tool chain (e.g., components of MATLAB/Simulink, code generator, C compiler).

   However, it may be possible to use the tool chain in a restrictive manner to enable the application of a certain WCET analysis method. For example, the available application development features of a modeling tool like MATLAB/Simulink may be restricted or certain compiler optimizations may be deactivated.

3. The method must be easily adaptable to new releases of software components of the tool chain. Expensive adaptations of the WCET method to new releases of software components have to be avoided.

   The situation that a development tool of the tool chain explicitly supports a specific WCET analysis methods is currently very rare. For example, it can be possible that a compiler provides certain support to perform WCET analysis [6]. But such tools are typically in a prototype state without commercial support. Therefore, the best current strategy for developing a WCET analysis tool is to adapt to existing COTS software development tools.

4. The WCET analysis method must be easy to retarget to different hardware settings, i.e., the implementation or configuration effort must be small enough for an economic useability of the WCET analysis method. Depending on the concrete WCET analysis method, there are in principle two different possibilities for retargetability. First, it can be required to order further implementation effort from the WCET tool provider. Second, it may be possible that the tool is flexible enough so that the customer can adapt the tool by himself. The latter approach is applicable for adequate measurement-based WCET analysis methods.

The adaption of a WCET analysis method to new hardware configurations can be kept easy when the WCET analysis method is based on measurements on the real hardware. Because in this case the WCET analysis method does not have to provide a so-called *exec-time model*, which describes the execution times for given code sequences. In measurement-based approaches the exec-time model is substituted by measurements on the real target hardware. There exist also measurement-based WCET analysis approaches that use hardware simulation instead of measurements on the real hardware [3, 4]. Such approaches rely on the existence of a cycle-accurate hardware simulator which is often not available.

In the following section a new WCET analysis method is presented that is able to fulfill the requirements from industry as given above. This WCET analysis method will be applied to program code automatically generated from MATLAB/Simulink models because there is additional information available about the structure of the generated code.

3 The WCET Analysis Framework

A new WCET analysis approach is needed to fulfill the requirements from industry listed in Section 2. Traditional methods based only on static code analysis are not flexible enough to retarget them with reasonable effort to new target processors. Though often used in practice, end-to-end runtime measurements are not an alternative, due to the exploding number of possible execution paths in real-size programs.

The WCET analysis method we describe in this paper is a hybrid approach of static and dynamic analysis methods. The dynamic part is performed by run-

time measurements on the real hardware platform. If available for the particular platform, the measurements could be also performed by a cycle-accurate simulator.

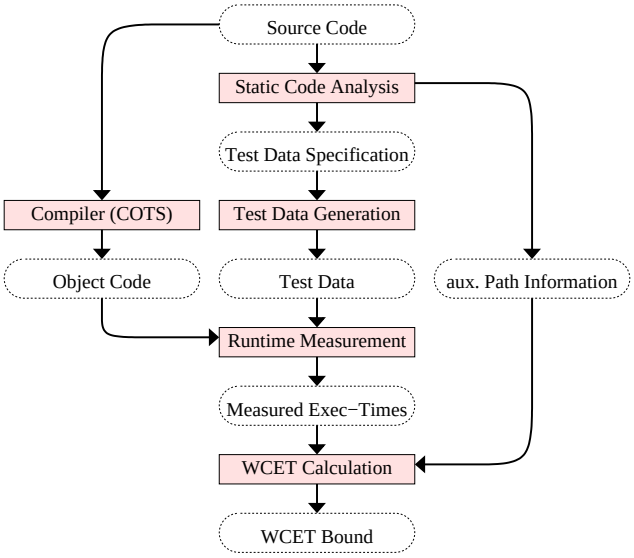

Figure 1. Components of the WCET Analysis Framework

The basic components of our WCET analysis framework are shown in Figure 1. The framework takes the program as input in both the source code and the object representation. The translation from source code to object code is done by a COTS compiler.

The *static code analysis* phase analyzes the source code with the goal to derive information about which test data should be generated for the runtime measurement and to derive path information that will be used by the final WCET calculation step:

Runtime Measurement is used to derive the instruction timing of paths through program segments. A *program segment* is a subgraph of a program's control flow graph with a unique start node such that only the start node has incoming edges from external nodes and all outgoing edges lead to the same external node.

The measurement of a specific execution path is enforced by generating input data that enforce the execution of this path. A coverage criterion has to be defined that describes the required runtime measurements. As already discussed, exhaustive execution path measurements of programs are not possible for real-size programs.

Having defined a coverage criterion, semantic code analysis is used to calculate the required test data. This analysis does not have to be implemented from scratch. Instead, the idea is to transform the program into a formal description of its program semantics that can be directly used by an existing analysis tool to generate the needed test data. The concrete analysis technique for test data generation has to be selected after evaluating its scalability regarding program size. Typical techniques that are interesting for this task are *model checking* [2, 9] or *constraint-based analysis* [10]. *Program slicing* [11] can be used to reduce the semantic models of the program by selecting only those parts of the code that influence the execution of a certain execution paths. Stepwise test data calculation can be used to further reduce the number of required test data. The idea is to calculate which further code locations will be also executed once input data for a specific code location have been selected. The measurements are done using a highly retargetable measurement framework.

To keep the test suite small, a hybrid approach consisting of static and dynamic WCET analysis is used. The *WCET Calculation* stage uses the execution time of each feasible path through program segments together with additional path information to calculate the WCET bound. The relevant path information includes iteration bounds for each loop, also called *loop bounds*. Depending on the code complexity, such loop bounds may be calculated automatically. If a loop bound cannot be calculated automatically, additional information has to be provided by the user. When analyzing code automatically generated from modeling tools like MATLAB/Simulink, additional knowledge about the structure of the code is known. As a result, most of the loop bounds in the generated code are typically hard coded and therefore can be derived automatically.

The *WCET Calculation* based on implicit path enumeration is done after performing the runtime measurements of the program segments [8].

The challenges of this WCET analysis framework are the automatic generation of test data and the extraction of control flow information from the program code. Both tasks cannot be done fully automatically for arbitrary program code. Therefore, user annotations respective restrictions on the code structure have to be used.

3.1 Decomposition of Execution Traces

To keep the number of required test data for runtime measurements within a feasible quantity, it is necessary to decompose the program paths into smaller parts and combine the obtained results to get the overall WCET bound. The choice of the right length of program subpaths for runtime measurement is based on a trade-off between complexity and precision. Complexity is given

by the number of required runtime measurements. In case of complex processors having an internal state that influences the execution time of instructions, measurement precision is better when measuring longer subpaths of the program.

One important aspect for the decomposition of execution traces is the demanded coverage criteria for the measurements. The coverage criteria will be defined at the level of *program segments*. For using this framework to obtain safe WCET bounds on a given hardware platform it has to analyzed what is the possible overestimation for particular coverage criteria.

4 Discussion

The design criteria of the measurement-based WCET analysis framework described in Section 3 are motivated by the requirements for an industrial-strength WCET analysis tool as summarized in Section 2. The decision of performing the program analysis at source code level is due to the requirement of high retargetability of the framework to new hardware platforms. However, for certain application domains it may be more important to have the analysis done after the code compilation at object code level. For example, it could be required to verify the path coverage calculated for source code level at object code level in case of critical code optimizations done by the compiler. In this case, the concept of the measurement-based WCET analysis using on automatic generation of test data is the same, but the implementation would be more hardware-dependent as it is also required to have a parser for the object code.

Technical realizations like inserting instrumentation code to measure the execution time of program segments are not discussed in this paper.

5 Summary and Conclusion

This paper describes a novel WCET analysis framework based on runtime measurements. The requirements for the framework are high portability to new target processors and an easy integration into COTS software development tool chains. We described a hybrid approach using static and dynamic timing analysis techniques. The central idea is to decompose the program paths into smaller subpaths and use formal methods to automatically derive the required test data to measure the execution time of the subpaths. Programs are structured into program segments to decompose program paths into smaller subpaths. After measuring the execution time of subpaths, a static WCET calculation is used to obtain the WCET bound.

Future work will focus on the assessment and selection of concrete formal program analysis techniques to generate the test data.

References

[1] G. Bernat, A. Colin, and S. M. Petters. WCET analysis of probabilistic hard real-time systems. In *Proc. 23rd Real-Time Systems Symposium*, pages 279–288, Austin, Texas, USA, Dec. 2002.

[2] A. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proceedings of the International Conference on Software Engineering*, Edinburgh, Scotland, UK, 2004.

[3] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications*, Hong Kong, Dec. 1999.

[4] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Proc. International Conference on Computer-Aided Design (ICCAD '97)*, San Jose, USA, 1997.

[5] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and results of wcet tools. *Proceedings of the IEEE*, 91(7):1038–1054, Jul. 2003.

[6] R. Kirner and P. Puschner. Timing analysis of optimised code. In *Proc. 8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, pages 100–105, Guadalajara, Mexico, Jan. 2003.

[7] 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, June 2002. Department of Computer Science, University of York.

[8] P. Puschner and A. V. Schedl. Computing Maximum Task Execution Times – A Graph-Based Approach. *The Journal of Real-Time Systems*, 13:67–91, 1997.

[9] S. Rayadurgam and M. P. E. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. 8th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '01)*, Washington DC, USA, Apr. 2001.

[10] N. T. Sy and Y. Deville. Consistency techniques for interprocedural test data generation. In *Proc. Joint 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE03)*, Helsinki, Finland, 2003.

[11] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.