Worst-Case Execution Time Analysis of Parallel Systems

Andreas Gustavsson
School of Innovation, Design and Engineering
Mälardalen University
Box 883, S-721 23 Västerås, Sweden
Email: andreas.sg.gustavsson@mdh.se

***Abstract*—The problem of finding the Worst-Case Execution Time, WCET, of a program executed on a specific hardware architecture is a very challenging task. A lot of effort has been put into analysing sequential programs executing on single-core hardware. The result is a variety of different methods and tools.**

The author currently works on finding methods for static WCET analysis of parallel software. The emphasis of the work is put on analysing the impact of synchronisation between threads executing on a shared memory architecture. The analysis is done on the software level, so less focus is put on the effects of the actual hardware on which the parallel program executes.

The analysis is based on a small parallel programming language incorporating some fundamental synchronisation primitives; locking and unlocking of shared resources. The programming language is formally defined, which allows the correctness of the analysis to be proven.

I. Introduction & Motivation

The execution time of a program can be a very complex function of the input data, the hardware, and the binary code executing on the hardware. The complexity mainly has two sources: the complex performance characteristics of modern processor architectures, and the sometimes very large number of possible paths through the code. This makes properties like the Worst-Case Execution Time, WCET, of a program very difficult to determine.

Today, parallel hardware, like multicore and MPSoC (Multi-Processor System-on-Chip) processors, host only a few cores. However, it is expected that within a few years' time, a chip could contain hundreds of cores, similar to today's GPUs (Graphical Processing Units). These architectures will be quite unlike current multicore architectures.

The same will hold for the software. Since long, programmers have been able to write programs containing several concurrent units of execution, often referred to as threads. Today, programmers *should* focus on introducing such parallelism in their programs, in order to utilise the parallel hardware in the best way.

Research in WCET analysis must take this trend into account – this is what the author intends to do, with emphasis on analysis of thread synchronisation. Some of the major predictable difficulties of this approach are:

1) Nothing is known about the execution environment; there could e.g. be several programs executing on the same hardware.
2) The WCET is generally assumed to be a constant property of a program.
3) The WCET estimate for a system of threads executing in parallel (on parallel hardware) will be very pessimistic if nothing is known about the scheduling policy used.
4) Introducing parallelism in software generally explodes the number of possible patterns of execution, and thus also the size of the state space to be analysed.

Since safety-critical hard real-time systems are considered, some realistic assumptions about the execution environment could be made. For this kind of systems, a predictable behavior is crucial. So regarding problem 1, it can be assumed that specific system resources, like a parallel processor or some of its cores, are dedicated to specific tasks.

Regarding problems 2 and 3, a WCET analysis could assume that the (synchronising) threads of a considered parallel program are effectively executed in a sequential manner, with the worst-case ordering (assuming that there are no timing anomalies [20] due to parallel execution of the threads). This would mean that the WCET estimate would be a constant property of the program and that it could be safely used in any schedulability analysis. However, the result of the WCET analysis would probably be an extremely pessimistic WCET estimate. This is since some threads would most likely be scheduled to execute in parallel and thus a reduction of the execution time bound could possibly be guaranteed.

This is a problem that has not yet been attacked within the field of WCET analysis. It will be the work of the author to define an analysis for programs containing concurrently executing and synchronising threads. A limitating assumption that the threads of a program are uninterruptedly executed in parallel on individual processor cores, would still expose the fundamental problems of defining such an analysis. Therefore, such an assumption will initially be made in the work of the author. Later on, the assumption might be relaxed by incorporating details on scheduling-introduced behaviors[1] in the

[1]A scheduling method resulting in a behavior not too far from that of the assumed model could be accomplished using some form of 2-level hierarchical scheduling [1], [25]. If the program is assumed to execute alone within a subsystem, the assumed system model basically holds within that subsystem. A possible problem could be that there are not enough cores to allow all threads to execute in parallel. It would probably not be too difficult to adapt the analysis to incorporate this situation, though (given that the properties of the global and subsystem schedulers are known).

analysis; effectively making the analysis scheduler-dependent. The result might be different WCET estimates for the program, depending on the used scheduler.

Problem 4 will have to be handled and solved by adapting the abstraction level of the analysis. The size of the analysed state space could e.g. be reduced by only considering well-structured parallel programs (written using e.g. OpenMP), which might greatly simplify the analysis. There will, however, be a significant tradeoff between the handling of this problem and the achievable precision in the resulting WCET estimate.

II. WCET Analysis

WCET analysis can be performed either statically or dynamically [33]. In dynamic WCET analysis, measurements of the actual execution time of the software running on the target hardware are performed. This method is not guaranteed to execute the program's worst-case path though, which could e.g. include some error-handling routine. Thus the WCET might be gravely under-estimated; i.e., there might exist paths through the code with considerably worse (longer) execution times than the worst execution time detected by the measurements.

In static WCET analysis, the actual program code and the properties of the target hardware are analysed offline. This method tries to find a tight estimation of the WCET (i.e., an estimation not too far from the actual WCET), but always over-estimates it. Solving the problem of finding the actual WCET, in the general case, is actually comparable to solving the halting-problem (a decision-problem [13]).

Static WCET analysis is normally split into three subtasks: the *low-level analysis*, which attempts to find safe timing estimates for executions of code sequences, the *flow analysis*, which constrains the possible paths through the code, and the *calculation*, where the most time-consuming path is found, using information derived in the first two phases.

III. Related Work

In low-level analysis, most research efforts have been dedicated to analyse the effects of different hardware features, including pipelines [5], [10], [18], [28], [31], caches [16], [18], [31], [32], branch predictors [3], and superscalar CPUs [17], [27].

In flow analysis, most research has been dedicated to *loop bound* analysis. Flow analysis can also identify *infeasible paths*, i.e., paths which are executable according to the program control-flow graph structure, but not feasible when considering the semantics of the program and possible input data values. There are a number of approaches to flow analysis, using e.g., abstract interpretation, symbolic execution, Presburger arithmetics, specialized data flow analyses, and syntactical analysis of parse trees [8], [11], [12], [19], [31].

Three main methods exist for the WCET calculation: The tree-based method [2], [3], [18], originating from Park's *timing schemas* [23]; the *path-based* method [10], [29]; and the *Implicit Path Enumeration Technique* (IPET) [6], [12], [16], [24], [31], where the WCET calculation problem is formulated as an integer linear programming problem, and the set of execution paths is restricted by linear constraints.

As a result, today's WCET analysis tools are able to analyse a large variety of software targeted for execution on single-core processors [33].

Some initial efforts have been put into analysing multi-core architectures and concurrent/parallel software. Staschulat et. al. [30] consider an integrated system- and task-level analysis to estimate memory access times for single-processor tasks running in parallel with tasks of other processors. Their approach requires full information about all tasks running in the system, and it makes quite strong assumptions about the task model.

Yan and Zhang [35], [36] present a static method for analysing a multicore processor with a shared L2 instruction cache. A limitation of this analysis is that the L1 data cache is assumed to be perfect (i.e., all accesses are assumed to be hits – which is generally not the case) and thus does not affect the content of the L2 cache.

Lv et. al. [21] and Wu and Zhang [34] use model-checking of timed-automata to perform WCET analysis. In this approach, a timed automata-model of the system to be analysed is created. Then, specific properties of the model are verified to find a WCET estimate for the analysed system. The achievable tightness of the WCET estimate depends on the level of details in the timed automata-model. Thus, model-checking is not a traditional static WCET analysis method.

Both papers mainly propose methods for reducing the size of the state space by altering the program model without affecting the true WCET of the model. This is a very important aspect when using model-checking overall. If the model is too large and complex, the state space will "explode", which means that the number of possible states is very large and analysing the model becomes infeasible.

There is some work on data flow analyses for parallel programs [4], [7], [14], which is of relevance to WCET flow analysis of parallel programs. Constant propagation has also been considered [15]. A survey of analyses for concurrent and parallel programs is found in [26].

IV. Research Questions

The goal of the author's research is to find suitable models and methods for WCET analysis of parallel systems; mainly parallel software executing on some parallel hardware, like multicore and MPSoC architectures. Since the field is quite new, focus is put on basic models taking parallelism into account, mainly for some form of flow analysis and calculation.

While important, there is less focus on the low-level analysis. Instead, emphasis is put on analysing the impact on the WCET from allowing synchronising and communicating threads executing on a shared memory architecture.

Some concrete research questions are:

1) Is model-checking a feasible tool for performing WCET analysis of parallel programs, containing synchronising threads?

2) Is there a feasible static program analysis method to perform WCET analysis of parallel programs, containing synchronising threads?
3) If there is a feasible static program analysis method to perform the WCET analysis, can it be separated into the current static (flow, low-level, and calculation) analysis phases?
4) What is the achievable precision of the chosen method and how does it scale with respect to analysis of real-world parallel systems (i.e., what is the complexity of the method)?

V. Research Results

Using the UPPAAL modeling and verification tool box, the impact of thread communication and synchronisation via shared resources, such as caches and buses, on a multicore architecture has been investigated [9]. In the analysed model, one thread is executed per processor core (i.e., there can be no scheduling or migration of threads) and synchronisation occurs through spin-locks.

Using the verification subsystem of the UPPAAL model-checker, different properties, such as the WCET, of the system-model is found and verified.

A problem with the method used for the case study in [9] is that too many properties of the analysed system are considered at once; e.g., properties of the cache hierarchy are taken into account while analysing the impact of thread synchronisation. Therefore, the used method does not scale very well with the size of the analysed problem. A more suitable abstraction level needs to be found (e.g., only considering the impact of having synchronising threads) in order to make the method scale.

The conclusion is that the used method does not seem to be a very good candidate for analysing parallel systems. Thus, the answer to research question 1 is probably no, at least if using a method similar to that of [9]. However, model-checking might still be a useful tool that could be integrated as part of some other method.

VI. Research Direction

Current and future work includes the definition of a static analysis method for a system of synchronising threads, executing on a shared memory architecture. The analysis will be based on a formally defined parallel programming language called **PLock**. This allows certain properties, such as correctness, of the analysis to be proved with respect to the semantic of **PLock**. However, the analysis (with the necessary extensions added to it) is intended to be general enough to be applicable to real-world languages, such as C/C++, as well.

PLock is an extension to the standard imperative **While**-type of languages, which can be found in some literature on semantics; e.g., [22]. A program written in **PLock** can contain one or more threads that are uninterruptedly executed in parallel on individual processor cores. A statement S is defined as:

$$S ::= \texttt{halt} \mid \texttt{skip} \mid x := a \mid S'; S'' \mid \texttt{if } b \texttt{ then } S \texttt{ fi} \mid \texttt{if } b \texttt{ then } S' \texttt{ else } S'' \texttt{ fi} \mid \texttt{while } b \texttt{ do } S \texttt{ done} \mid \texttt{lock } l \mid \texttt{unlock } l$$

`halt` simply stops the thread. `lock` l makes the issuing thread wait until it is allowed to lock the lock named l (which could happen immediately). `unlock` l immediately unlocks the lock named l. The other statements follow a semantic comparable to the standard C semantics regarding their functional behavior.

An abstraction in the semantic of **PLock** is that *all* statements are defined to require 1 time unit to execute. This is perhaps not very realistic, especially when considering e.g. an assignment ($x := a$) to the variable x occurring in two threads at the same time instance. Taking properties and effects of the hardware, on which the program executes, into account, one of the threads would most probably need to wait for the other to finish. Furthermore, writing to memory is usually a very time-consuming task. The semantic of **PLock** will allow both threads to concurrently write to x without any delays. (Note that this describes a situation with a race condition between the two threads, and that the resulting value of x is non-deterministically chosen between the possible values written to it.) However, the behavior of the memory system (and other shared resources for that matter), like delays occurring when reading, writing, and waiting for exclusive access to it, can be modeled at the program level. This can be done using the `skip`, `lock` l and `unlock` l statements to protect a variable and delaying the access to it.

In other words, **PLock** is a modeling language excluding unnecessary details and incorporating a simplified parallel timing model. This makes **PLock** powerful enough to model a realistic behavior of a parallel system, but simple enough to let the WCET analysis focus on the important aspects; like the fundamental problems of analysing the effects resulting from thread synchronisation-introduced behaviors.

Acknowledgment

The author would like to thank his supervisors, Björn Lisper, Andreas Ermedahl and Jan Gustafsson, for their valuable advices and support.

The author would also like to thank the Swedish Research Council (VR) for funding his work through the project 2008-4650, Worst-Case Execution Time Analysis of Parallel Systems.

References

[1] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. Bril, "Towards hierarchical scheduling on top of vxworks," in *Proc. of the 4^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT08)*, 2008, pp. 63–72.

[2] A. Colin and G. Bernat, "Scope-tree: a program representation for symbolic worst-case execution time analysis," in *Proc. 14^{th} Euromicro Conference of Real-Time Systems, (ECRTS'02)*, Vienna, Jun. 2002, pp. 50–59.

[3] A. Colin and I. Puaut, "Worst case execution time analysis for a processor with branch prediction," *Journal of Real-Time Systems*, vol. 18, no. 2/3, pp. 249–274, May 2000.

[4] M. B. Dwyer and L. A. Clarke, "Data flow analysis for verifying properties of concurrent programs," in *Proc. ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, Dec. 1994, pp. 62–75.

[5] J. Engblom, "Processor pipelines and static worst-case execution time analysis," Ph.D. dissertation, Uppsala University, Dept. of Information Technology, Uppsala, Sweden, Apr. 2002, ISBN 91-554-5228-0.

[6] A. Ermedahl, "A modular tool architecture for worst-case execution time analysis," Ph.D. dissertation, Uppsala University, Dept. of Information Technology, Uppsala University, Sweden, Jun. 2003.

[7] D. Grunwald and H. Srinivasan, "Data flow equations for explicitly parallel programs," in *Proc. Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993, pp. 159–168.

[8] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution," in *Proc. 27^{th} IEEE Real-Time Systems Symposium (RTSS'06)*, Dec. 2006.

[9] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson, "Towards WCET analysis of multicore architectures using UPPAAL," in *Proc. 10^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, B. Lisper, Ed. Brussels, Belgium: OCG, Jul. 2010, pp. 103–113.

[10] C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon, "Bounding pipeline and instruction cache performance," *IEEE Transactions on Computers*, vol. 48, no. 1, pp. 53–70, Jan. 1999.

[11] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen, "Supporting timing analysis by automatic bounding of loop iterations," *Journal of Real-Time Systems*, vol. 18, no. 2-3, pp. 129–156, May 2000.

[12] N. Holsti and S. Saarinen, "Status of the Bound-T WCET tool," in *Proc. 2^{nd} International Workshop on Worst-Case Execution Time Analysis (WCET'2002)*, 2002.

[13] S. C. Kleene, *Introduction to Metamathematics*. North-Holland Publishing Co, Jan. 1980.

[14] J. Knoop, B. Steffen, and J. Vollmer, "Parallelism for free: Efficient and optimal bitvector analyses for parallel programs," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 3, pp. 268–299, May 1996.

[15] J. Lee, S. P. Midkiff, and D. A. Padua, "A constant propagation algorithm for explicitly parallel programs," *International Journal of Parallel Programming*, vol. 26, no. 5, pp. 563–589, Oct. 1998.

[16] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'95)*, La Jolla, CA, Jun. 1995.

[17] S. Lim, J. Han, J. Kim, and S. L. Min, "A worst case timing analysis technique for multiple-issue machines," in *Proc. 19^{th} IEEE Real-Time Systems Symposium (RTSS'98)*, Dec. 1998.

[18] S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki, "An accurate worst-case timing analysis for RISC processors," *IEEE Transactions on Software Engineering*, vol. 21, no. 7, pp. 593–604, Jul 1995.

[19] T. Lundqvist, "A WCET analysis method for pipelined microprocessors with cache memories," Ph.D. dissertation, Chalmers University of Technology, Göteborg, Sweden, Jun. 2002.

[20] T. Lundqvist and P. Stenström, "Timing Anomalies in Dynamically Scheduled Microprocessors," Chalmers University of Technology, Tech. Rep. 99-5, Apr 1999.

[21] M. Lv, N. Guan, W. yi, Q. Deng, and G. Yu, "Efficient instruction cache analysis with model checking," in *Proc. 16^{th} IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10), Work-in-Progress Session*, Apr. 2010, pp. 33–36.

[22] H. R. Nielson and F. Nielson, *Semantics with Applications – A Formal Introduction*. John Wiley & Sons (1992), Jul. 1999.

[23] C. Y. Park and A. C. Shaw, "Experiments with a program timing tool based on a source-level timing schema," *IEEE Computer*, vol. 24, no. 5, pp. 48–57, 1991.

[24] P. P. Puschner and A. V. Schedl, "Computing maximum task execution times – a graph-based approach," *Journal of Real-Time Systems*, vol. 13, no. 1, pp. 67–91, Jul. 1997.

[25] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau, "Evolving real-time systems using hierarchical scheduling and concurrency analysis," in *Proc. 24^{th} IEEE Real-Time Systems Symposium (RTSS'03)*, 2003, pp. 25–36.

[26] M. Rinard, "Analysis of multithreaded programs," in *Proc. 8th Static Analysis Symposium*, ser. Vol. 2621 of *Lecture Notes in Comput. Sci.*, P. Cousot, Ed. Paris, France: Springer-Verlag, Jul. 2001, pp. 1–19.

[27] J. Schneider and C. Ferdinand, "Pipeline behaviour prediction for superscalar processors by abstract interpretation," in *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*. ACM Press, May 1999.

[28] F. Stappert and P. Altenbernd, "Complete worst-case execution time analysis of straight-line hard real-time programs," *Journal of Systems Architecture*, vol. 46, no. 4, pp. 339–355, 2000.

[29] F. Stappert, A. Ermedahl, and J. Engblom, "Efficient longest executable path search for programs with complex flows and pipeline effects," in *Proc. 4^{th} International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES'01)*, Nov. 2001.

[30] J. Staschulat, S. Schliecker, M. Ivers, and R. Ernst, "Analysis of memory latencies in multi-processor systems," in *Proc. 5^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2005)*, R. Wilhelm, Ed., Palma de Mallorca, Jul. 2005.

[31] S. Thesing, "Safe and precise WCET determination by abstract interpretation of pipeline models," Ph.D. dissertation, Saarland University, 2004.

[32] R. White, F. Müller, C. Healy, D. Whalley, and M. Harmon, "Timing Analysis for Data Caches and Set-Associative Caches," in *Proc. 3^{rd} IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, Jun. 1997, pp. 192–202.

[33] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution time problem — overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[34] L. Wu and W. Zhang, "Bounding worst-case execution time for multicore processors through model checking," in *Proc. 16^{th} IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10), Work-in-Progress Session*, Apr. 2010, pp. 17–20.

[35] J. Yan and W. Zhang, "WCET analysis for multi-core processors with shared instruction caches," in *Proc. 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, St. Louis, MO, Apr. 2008.

[36] ——, "Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches," in *Proc. 15^{th} International Conference on Real-Time Computing Systems and Applications (RTCSA'09)*, Aug. 2009, pp. 455–463.