ISBN 978-981-11-0008-6
Proceedings of 2016 6th International Workshop on Computer Science and Engineering (WCSE 2016)
Tokyo, Japan, 17-19 June, 2016, pp. 7 - 12
doi: 10.18178/wcse.2016.06.002

Worst Case Execution Time Calculation of Parallel Embedded Real-Time Software

Muhammad Waqar Aziz and Syed Abdul Baqi Shah[+]

Science and Technology Unit, Umm Al-Qura University, Makkah, 21995, Saudi Arabia

Abstract. Embedded Real-Time Software (ERTS) must be verified for their timing correctness where knowledge about the Worst-Case Execution Time (WCET) is the building block of such verification. Traditionally, research on the WCET analysis of ERTS assumes sequential code running on single-core platforms. However, as computation is steadily moving towards using a combination of parallel programming and hardware designs, new challenges in WCET analysis need to be addressed. This work derives safe WCET estimates of parallel ERTS using a hybrid approach that combines the flow and timing information of the parallel software. The timing information is obtained via measurement-based analysis by using time-stamped execution traces. The applicability of the proposed method is demonstrated by calculating the WCET estimates of parallel embedded programs in the ParMiBench benchmark suite. The results showed less pessimism in the computed WCET estimates compared to the measured WCET estimates.

Keywords: embedded real-time software, worst-case execution-time analysis, parallel computing, software testing and analysis.

1. Introduction

A Real-Time System (RTS) is an embedded system, where the system tasks need to be completed within the specified time otherwise catastrophic results may occur. To ensure that the RTS would work correctly, a schedulability analysis is performed during the RTS development, which checks whether or not all tasks can meet their deadlines at runtime. This requires the knowledge about the Worst Case Execution Time (WCET) of individual tasks. In addition, the WCET analysis of RTS is needed in its design, for system understanding and more importantly to guarantee their behavior. WCET analysis can be performed either statically (without executing the program) or dynamically (by executing the program and measuring its execution time) [1]. Typically, measurement-based approaches are not considered to produce *safe* (i.e., not under approximated) estimates. Hence, reliable guarantees of observing the worst-case cannot be given.

In contrast, static timing analysis always provides a *safe* upper bound of the WCET [2]. Moreover, it does not require measuring devices/controlled environment and takes shorter analysis times than measurement-based approaches. The static analysis of a program consists of the following three phases: (1) *program-flow analysis* [3]-[5] by which both necessary and useful execution flow information is derived, such as loop bounds and infeasible paths; (2) *processor-behavior analysis* by which execution times of the program segments are determined using either statically modeling the hardware [6] or making direct runtime measurements on it [7]; and (3) *calculation* [8], where the results of the previous steps are combined to calculate the WCET estimates of the program. The focus of this paper is on the calculation phase, assuming that the flow information is available.

The growing performance requirements of RTSs need computation intensive software applications with high data throughput rates and low latency [9]. To cope with these requirements, multi-core architectures are

[+] Corresponding author. Tel.: +966-58-392-7245.
E-mail address: sashah@uqu.edu.sa.

now increasingly used in embedded real-time domain [10]. In addition, parallel computing can be applied to optimally use the available hardware [11]. Unfortunately, the above mentioned steps of WCET analysis work adequately for sequential programs running on single-core hardware, but are challenged in parallel computing. This demands the evaluation of the existing WCET analysis methods and techniques for parallel embedded programs, so that they can be modified or for proposing new ones. The objective of this work is to investigate a suitable calculation method to derive the WCET estimation of parallel embedded software.

To make parallel software analyzable, the existing state of the art related work proposed the use of either specialized hardware [12] and programming language [2] or focuses on a particular aspect of parallel computing [11]. In contrast, the main contribution of this work is to derive the WCET estimates of parallel embedded software executing on arbitrary multi-core hardware. The WCET estimate of a task is calculated by utilizing its execution flow information at the source code level and the timing information (obtained by measuring the execution times of the program segments). This information is combined into an optimization-based problem, such as Integer-Linear Programming (ILP) [13].

2. Task Model and Scope

This work focuses on the timing analysis of individual tasks, which are composed of parallel executing threads. The main task creates (and later joins) child threads one or more times. This means that the task under analysis consists of both sequential (without threads) and parallel (threaded) portions. The threads could wait at the barriers (for synchronization) during their execution or for a lock acquisition operation before accessing a shared program segment (critical section). However, this work considers the case, where all the threads execute the same code. Due to this consideration, the synchronization related stall can be ignored [11], as the synchronization time is negligible. As the work is focused on data parallelism, where the input data is equally divided among the threads, there is no code segment to protect from simultaneous access of threads.

Since measurement is used instead of static modeling of the hardware, the proposed method can be classified as hybrid measurement-based analysis. A hybrid approach combines the elements of static and dynamic analyses [14], i.e., it has the same steps as static analysis, except that the processor-behavior analysis is replaced by direct run-time measurements on the hardware. This makes the micro-architecture analyses (e.g., cache analysis) and other contention effects caused by parallel hardware out of the scope of this work. It is worth mentioning here that this work is contrary to the system-level analysis, where multiple tasks are executed in parallel. Thus, the issues related to massive parallelism, such as communication costs due to networking of different nodes, are not considered.

3. The Worst-Case Execution Time Calculation

This work derives the WCET estimate of a program, using the traditional ILP-based formulation [13] that combines the flow and executing time information of the program. However, it follows a measurement-based analysis approach to determine the execution times of program segments, instead of performing processor-behavior analysis statically. To derive the WCET estimate (Z) of a program, the ILP problem is formulated as the following objective function that seeks to maximize:

$$Z = \sum_{i=0}^{n} c_i * x_i \tag{1}$$

where c_i is the cost a basic block of a program in terms of execution time and x_i is its count, i.e., the number of times this basic block is executed. In simple words, the program execution time is calculated as the sum of the products of the execution counts and times (1) of its constituent basic blocks. The method of calculating the WCET estimates is depicted with the help of Fig. 1. The execution time of the basic blocks was calculated using the time-stamped execution traces of the task. To this end, two identified instrumentation point (*ipoint*) instructions were inserted at the start and end of each basic block. There were no restrictions placed on deploying instrumentation and hence probe-free tracing was supported. While delimiting a basic block, these *ipoint* instructions cause the target to produce a timestamp (called a timing trace) upon execution. The execution times of the basic blocks were computed by identifying the start and end of a basic block. This was

achieved by parsing the required traces from the bulk of information logged in the trace file. An algorithm was developed to automatically compute the execution time of the basic block from the parsed traces. Instead of executing the program with random input data several times and then selecting the worst execution time, the worst-case input data was generated using evolutionary testing [6].

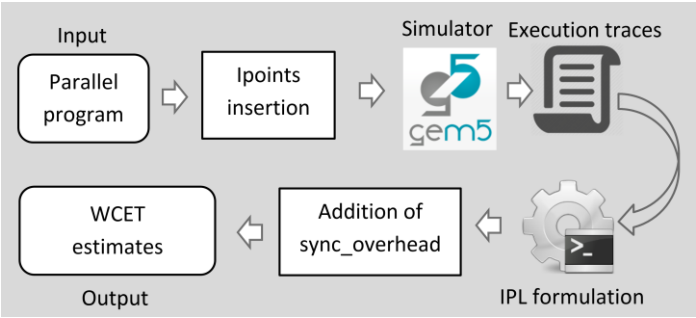

Fig. 1: Method for calculating the WCET estimates of parallel embedded real-time software.

The other key flow information, used in (1) to compute the WCET estimate, is the execution counts. The execution counts represent the number of times each basic block of the task is executed. The execution counts were acquired by instrumenting the basic block with counters, during the program-flow-analysis phase. Generally, there are a few portions of a program which execute more than once, e.g., functions and loops. Most of the other program segments execute once and thus have no major impact (as their execution count is one). In addition, the function passed to threads in a parallel program usually executes many times (i.e., each thread executes the same function in parallel). This is equal to the number of threads created, which can be determined by inspecting the program code. It is, however, important to stress that this work does not attempt to perform the program-flow analysis. It is rather assumed that the program flow-analysis has been performed already and its information, such as the identification of the number of threads, basic blocks and their execution counts were already available.

4. Experiment, Results and Discussion

In this work, the Gem5 architecture simulator [15] was used to execute the parallel program and measure its execution times. Gem5 was selected as it is open-source, cycle-accurate architecture simulator that provides a tracing mechanism to extract timestamp execution information from an instrumented parallel program. To derive the WCET estimates, *ParMiBench* benchmark suite [16] is used as example parallel embedded software. It is a parallel version of a subset of *MiBench* benchmark suite [20]. *ParMiBench* is open source and it offers a collection of parallel embedded benchmarks that are representative of diverse domains of the embedded applications, such as control and automation, networks, offices, and security. This suite is designed to evaluate the performance of embedded multicore systems where many of these benchmarks are the suitable candidates for WCET analysis [21]. The details of these benchmarks are provided in Table 1.

First, the WCET estimates of a task were calculated (the *calculated* WCET). Then, dynamic analysis was performed, to evaluate the *tightness* of the calculated estimates, by executing the same task and measuring its execution time. To this end, the longest end-to-end execution time of the task were measured (hence termed as the measured WCET). The *tightness* of the analysis was computed in percentage of pessimism in the calculated WCET with respect to the measured WCET. The results are shown in Table 2, where the unit of time is the number of processor clock cycles. From Table 2, it can be observed that the calculated estimates bound the measured estimates. Moreover, the derived WCET estimates are safer than the conventional end-to-end measurement approaches, due to the use of hybrid measurement-based analysis. The use of the worst-case input data further ensures that the program executed for its worst-case time or close to it. This thus eliminates the need of partitioning the input data among threads and measuring the execution time of each thread. This fact was further confirmed when the calculated WCET estimates were compared with the estimates generated via the worst-case input data. In this case, the pessimism of only 0.6% was observed, which clearly emphasized the correctness of the calculated WCET estimates. This also showed that the

method used to calculate the estimates was least pessimistic.

Table 1. Details of the benchmarks in the *ParMiBench* suite

Benchmark	***Descriptioin***	***Parameters***
Susan	Image processing: Performs smoothing, edge detection and finds corners	Input image and operation type flag (e.g. -s, -e and -c)
BasicMath	Math operations: Solves Cubic Equations, finds Integer Square Roots, finds Square Roots of Long type numbers, Performs Degree To Radian Conversion, Perform Radian To Degree Conversion.	Math operation type, input dataset and number of workers.
StringSearch	Performs String Search Operations using: 1. Pratt-Boyer-Moore String Search. 2. Case-sensitive Boyer-Moore-Horspool String Search. 3. Case-Insensitive Boyer-Moore-Horspool String Search. 4. Boyer-Moore-Horspool (Case-insensitive with accented character translation) String Search.	Input text file, input strings file, Algorithm type and workers.
Dijkstra	Gets the shortest paths from node 0 to node N, with two CPU options: Each CPU with its own queue All CPUs share a common queue	Input file and number of nodes
Sha	The parallel 'sha' perform digest computation of file and place the digest in separate output file by employing input data partition strategy.	Input file and number of workers

It is claimed that calculating the WCET estimates using the *ipoints*, as mentioned in this paper, also incorporates the stall times related to synchronization and access of the critical sections. Therefore, there is no need to calculate them separately. The *ipoints* were inserted using *m5ops* utility provided by Gem5. The *m5ops* inserts timestamps without affecting the actual execution time; as a result, there is no instrumentation cost. One can argue that the calculated WCET estimates do not provide an absolute *safe* bound, due to the use of a hybrid approach involving measurement-based analysis. However, in contrast to conventional end-to-end testing, the hybrid measurement-based analysis uses static analysis information about the code and hardware: flow analysis results and minimal hardware information can be used in measurement-based analysis to guide the testing process for better WCET estimations. Moreover, finding an absolute *safe* bound on the execution time is not required for most embedded RTSs, which are soft in their majority. Calculating the WCET estimates using other IPET-based calculation technique, such as constraint-logic programming has the problem of NP-completeness. This requires the investigation of the use of suitable search heuristics to improve the scalability of the WCET calculation technique.

Table 2. WCET Estimates of Some Benchmarks from *ParMiBench* Benchmark Suite

Benchmarks	***Measured***	***Calculated***	***Pessimism***
Stringsearch (5 char long string input)	10702583000	11836753000	10.60 %
Stringsearch (5 char long string input)	12886961500	14990755000	16.325 %
Dijkstra	8847226500	1048455500	18.51 %

5. Related Work

In the literature, there have been few contributions towards analysis of parallel programs, as the existing WCET-analysis research mostly focuses on sequential programs running on single-core architectures. Rochange et al. [17] for the first time highlight the problem of analyzing the timing behavior of non-sequential software on a multi-core architecture. They report manual analysis of a parallel application, which determines the synchronization and communication between its executing threads. The experiences in evaluating the WCET of parallel application, in the MARASA project, are reported in [17]. This study recommends determining the parallelism and synchronization in parallel code for its WCET analysis. However, the described process is completely guided by the user and is specific to the estimation of WCET of one component of a parallel application. Although some work has applied IPET to estimate WCET of parallel programs [18], yet they have not considered the synchronization stalls. Oppositely, other researches [11] have

considered synchronization stalls, but do not apply IPET.

An approach for automatic timing analysis of parallel applications [11] shows how to compute the synchronization-related stall of individual threads. The WCET of the parallel program is determined by computing the WCET of the main thread and adding to it the worst-case stall of child threads at synchronizations. However, the approach relies on user-provided annotations to identify the synchronization patterns. The worst-case response time of parallel applications running on multi-core platforms is computed in [18]. Instead of proposing a new technique, the approach extends only the control flow analysis phase of an existing WCET estimation method. Similarly, the traditional ILP based analysis of sequential program is extended to incorporate the overhead of monitoring [19]. The method calculates the maximum monitoring stall and adds it to the WCET calculated using popular methods.

6. Conclusion

In addition to the usual functional verification, embedded real-time software (ERTS) also requires temporal verification to ensure conformance with timing constraints. Such verification requires knowledge about the Worst Case Execution Time (WCET) of the running tasks in the RTES. Therefore, calculating the WCET of ERTS has a vital value in the design and verification of these systems. Previous work in this area, mostly deals with sequential code running on a single core processor. In this work, the WCET estimates of parallel programs running on multi-core embedded platforms are computed. A hybrid measurement-based analysis approach is used to determine the execution times of program segments, instead of statically modeling the micro-architecture. The IPET-based formulation used takes into account the execution and timing information to yield WCET estimates. In the future, it is planned to apply the proposed method in real-life industrial-scale applications and to determine the causes and localization of undesired timing delays.

7. References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, et al. “The worst-case execution-time problem – overview of methods and survey of tools,” ACM Transactions on Embedded Computing Systems (TECS), 7(3):36, 2008.

[2] A. Gustavsson, “Static Timing Analysis of Parallel Software Using Abstract Execution,” Malardalen University, Licentiate Thesis, 2014.

[3] I. Bate and D. Kazakov, “New directions in worst-case execution time analysis,” In IEEE Congress on Evolutionary Computation, (CEC 2008), pp. 3545–3552, 2008.

[4] J. Gustafsson and A. Ermedahl, “Merging techniques for faster derivation of WCET flow information using abstract execution,” In Proceedings of the 8th International Workshop on Worst-Case Execution Time (WCET) Analysis, pp. 79–89, July 2008.

[5] D. Kebbal, “Automatic flow analysis using symbolic execution and path enumeration,” In International Conference on Parallel Processing Workshops. ICPP 2006, 8 pp. - 404, 2006.

[6] M. W. Aziz, S. A. B. Shah, “Test-Data Generation for Testing Parallel Real-Time Systems,” Testing Software and Systems, Volume 9447 of the series Lecture Notes in Computer Science, pp. 211-223. 2015.

[7] A. Marref and A. Betts, “Accurate measurement-based WCET analysis in the absence of source and binary code,” In 14th IEEE International Symposium on Object/Component/Service- Oriented Real-Time Distributed Computing (ISORC), 2011, pages 127–135, 2011.

[8] A. Marref, and G. Bernat, “Predicated worst-case execution-time analysis,” Springer Berlin Heidelberg, 2009.

[9] T. Ungerer, et. al. “Experiences and Results of Parallelisation of Industrial Hard Real-time Applications for the parMERASA Multi-core,” In 3rd Workshop on High-performance and Real-time Embedded Systems (HiRES 2015), Amsterdam, the Netherlands, January 2015.

[10] A. Pyka, et. al., “WCET analysis of parallel benchmarks using on-demand coherent cache,” In 3rd Workshop on High-performance and Real-time Embedded Systems (HiRES 2015), Amsterdam, the Netherlands, 2015.

[11] H. Ozaktas , C. Rochange, and P. Sainrat, “Automatic WCET Analysis of Real-Time Parallel Applications,” in 13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013), pp. 11-20, 2013.

[12] T. Ungerer, et. al., “Merasa: Multicore execution of hard real-time applications supporting analyzability,” IEEE Micro (5) (2010) 66–75.

[13] Y-T. S. Li, and S. Malik, “Performance analysis of embedded software using implicit path enumeration,” In ACM SIGPLAN Notices, 30(11):88-98, 1995.

[14] B. Lisper, A. Ermedahl, D. Schreiner, J. Knoop, P. Gliwa, “Practical experiences of applying source-level WCET flow analysis on industrial code,” Leveraging Applications of Formal Methods, Verification, and Validation, 449–463, 2010.

[15] A. Basu, J. Hestness, D. Hower, et al. “The gem5 simulator”. ACM SIGARCH Computer Architecture News, 39(2):1–7, 2011

[16] S. M. Z. Iqbal, Y. Liang, and H. Grahn, “ParMiBench – An Open-Source Benchmark for Embedded Multiprocessor Systems,” IEEE computer achitecture letters, 9(2) July-December 2010.

[17] C. Rochange, A. Bonenfant, S. Pascal Sainrat, et. al. “WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-Core.” In 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010), pp. 90-100. 2010.

[18] D. Potop-Butucaru, and I. Puaut, “Integrated Worst-Case Response Time Evaluation of Multicore Non-Preemptive Applications,” 2013.

[19] D. Lo, and G. E. Suh, “Worst-case execution time analysis for parallel run-time monitoring, ” In 49th ACM/EDAC/IEEE Design Automation Conference (DAC 2012), pp. 421-429, 2012.

[20] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, Mibench: A free, commercially representative embedded benchmark suite, in: IEEE International Workshop on Workload Characterization, IEEE, 2001, pp. 3–14.

[21] J. Gustafsson, A. Betts, A. Ermedahl, B. Lisper, The mälardalen wcet benchmarks: Past, present and future, in: International Workshop on Worst-Case Execution Time (WCET) Analysis, Vol. 15, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.