Timing Analysis of Parallel Software Applications for Multi-core Embedded Systems

Muhammad Waqar Aziz*[1] and Syed Abdul Baqi Shah[2]

1. *Department of Computer Science, CECOS University of IT and Emerging Sciences, Pakistan*
2 *Science and Technology Unit, Umm Al-Qura University, Saudi Arabia*
* **Coressponding Author:** Email: waqar@cecos.edu.pk

Abstract

Real-Time Embedded Systems (RTES) 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 RTES assumes sequential code running on single-core platforms. However, as computation is steadily moving towards using a combination of parallel programming and multi-core hardware, new challenges in timing analysis, and especially in WCET analysis need to be addressed. Towards this direction, this paper presents the Timing Analysis tool for Parallel Embedded Software (TAPES). The proposed tool allows the WCET estimation of parallel applications running on multi-core hardware through a hybrid measurement-based analysis method, that combines the program flow and timing information into an Integer-Linear Programming problem to estimate the WCET. In addition, the TAPES tool allows the measurement of the longest end-to-end execution time by capturing the timing properties of the parallel executing threads using time-stamped execution traces of the program. The applicability of the proposed tool is demonstrated through the timing analysis of an embedded parallel benchmark suite – the ParMiBench. The results showed that the calculated WCET estimates have significantly less over-approximation compared to the measured WCET estimates. The comparison of the calculated and measured WCET estimates showed modest over-estimates.

Keywords: Multi-core Embedded Systems; Worst-Case Execution-Time Analysis; Parallel Embedded Software; Real-Time Systems.

1. Introduction

After hitting a technological dead end in providing further computational speed by increasing clock frequency, attention has been (re)put on parallel programming multi-core architectures as the solution to increase the performance of computation once again. Although the execution time of an individual task can be reduced by decomposing it into parallel executing threads, it poses a number of problems while designing Real-Time Embedded Systems (RTES), for example, in tasks scheduling. Generally, to ensure the correct working of RTES, a schedulability analysis is performed that checks all tasks can meet their deadlines at run-time. This requires the knowledge about the Worst-Case Execution Time (WCET) of the individual tasks. In addition, the WCET analysis is also required to guarantee the behavior of RTES and as an input to system response time computation.

The WCET analysis is normally performed statically or dynamically [28]. The static analysis methods analyze the control-flow paths, without executing the program code, and combine this information with the abstract hardware model to obtain upper bounds. In contrast, dynamic analysis methods execute the program on actual hardware or simulator to measure its execution time [28]. Typically, measurement-based approaches are not considered to produce *safe* (i.e., not under approximated) estimates. Because the results obtained by measuring the execution times constitute a subset of the actual possible executions, where a pathological worst-case could have been missed during testing. Hence, reliable guarantees of observing the worst-case cannot be given. However, it is sufficient for soft RTESs where occasional misses of deadlines are tolerated. For safety-critical systems, where absolute safety of programs is required, static analysis should be performed.

In static analysis, the necessary flow

information, such as loop bounds and infeasible paths, is derived in the *program-flow analysis* [1, 5, 10, 13]. Next, the execution times of the program segments are derived in the *processor-behavior analysis*, by statically modeling the hardware [4, 16, 26]. Finally, the results of the previous steps are combined together in a *calculation* method [11, 15, 21, 23] to obtain the WCET estimation. Unfortunately, these steps work adequately for sequential programs running on single-core architectures, but are challenged in the parallel-computing world. First, parallel applications do not execute as a stream of sequential instructions, so conventional control-flow analyses must be updated to consider the inherent concurrency. Second, hardware no longer has bounded timing behavior due to inter-thread interferences, which makes hardware modeling impossible or extremely hard. Third, WCET calculation techniques are suited for additive sequential models, where the execution times of the segments are added to drive the WCET.

To overcome the disadvantages of both static and dynamic analyses, this work investigates a hybrid solution for the timing analysis of parallel embedded applications running on multi-core architectures. A hybrid approach combines the elements of static and dynamic analyses [19]. It has the same steps as static analysis, except that the processor-behavior analysis is replaced by direct run-time measurements on the hardware. In this work, we propose the Timing Analysis tool for Parallel Embedded Software (TAPES) that consumes a parallel program as input and produces its *calculated* and *measured* WCET estimates. To calculate the WCET, a novel program-flow analysis method is proposed for parallel applications that identifies the execution behavior of their sub-threads at the source code level. The obtained flow information is then combined with the execution times of the program segments to calculate the WCET estimates (hence termed as *calculated WCET*). In addition, TAPES also allows the measurement of the longest end-to-end execution time (i.e., the *measured WCET*) of the parallel program to capture its low-level timing behavior. The traces are obtained by executing the parallel application using Gem5 architecture simulator [2].

The layout of the paper is as follows: The next section details the preliminaries required to understand the rest of the paper. Section 3 provides the details of the hybrid measurement-based method used to calculate the WCET of parallel programs, followed by execution-time measurement details in Section 4. The effectiveness of TAPES is demonstrated via timing analysis of embedded parallel benchmark suite – *ParMiBench* [17], in Section 5. The related work is briefly surveyed in Section 6, while the paper is concluded in Section 7.

2. Preliminaries

2.1 Task/ Execution Model

This work deals with computing the WCET estimates of a parallel program composed of synchronizing threads that execute simultaneously on multiple cores of a single device. Thus, it should not be confused with massively parallel systems involving several programs running on a grid of computing devices. Hence, the issues related to massive parallelism, such as communication costs due to networking of different nodes, are not dealt with. While most of the existing work (e.g., [22, 24]) assume a time-predictable shared-memory multi-core architecture, we consider the problem of finding the WCET for arbitrary multi-core hardware. It was, therefore, expected that the execution times in the case of hardware with no analyzability characteristics tend to vary tremendously because of unbounded interferences between the executing threads. Nevertheless, this work is scalable enough to be applied to both analyzable and off-the-shelf hardware.

2.2 Scope

This work focuses on parallel applications developed using POSIX threads, which is the widely used industry standard for developing parallel programs. The POSIX standard provides the explicit characterization of thread control (creation and join) and synchronization through mutexes and barriers. The control of parallel program is handled implicitly by the POSIX thread APIs (Pthread). As a hybrid measurement based analysis is performed in this work that replaces the processor modeling with measurement of the execution times of the program segments. Hence, the micro-architecture analyses, such as contention effects caused by the parallel hardware are out of the scope of this work.

3. Hybrid Measurement-Based Analysis

The proposed TAPES tool uses the hybrid measurement-based analysis method to calculate the WCET estimates of parallel applications. The details of the hybrid method are provided below, which include the description of a novel program flow analysis method for parallel applications.

3.1 Program Flow Analysis

Contrary to sequential programs, the flow analysis of a parallel application is responsible to investigate the dependency among threads and the program concurrency. The proposed flow analysis method allows loop analysis and identifies the threads basic information, its dependencies and the program concurrency. To achieve this, the following steps are defined for the flow analysis of parallel embedded applications, as shown in Figure 1.

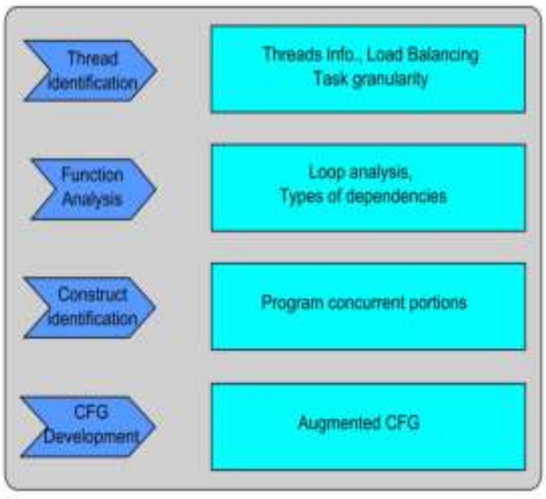


Figure 1: Steps of the proposed program flow analysis method for parallel applications

- Thread Identification is related to finding the thread basic information, such as the number of threads created and the number of times these threads are created. This information is useful in thread scheduling and their mapping to cores. This step also includes analyzing the load balancing, i.e., how the work is distributed among the threads, as improper load balancing increases the discrepancy of execution time between sub-tasks. Moreover, the task granularity is also analyzed in terms of its decomposition among threads. The fine-grained task decomposition would result in increased synchronization and communication overhead and vice verse. In such case, the time spend in synchronization and communication should be analyzed and added to the overall WCET.
- Function Analysis allows the in depth analysis of the function passed to the thread. In a parallel program this is usually the code that is executing in parallel and thus has the major effect on the timing estimates. Inside the function, all the loops are analyzed, as embedded programs spend most of their execution time in loops and the one which seems to take more iterations is selected based on programmers observation. The selected loop is further divided into different segments as per the definition of a basic block [14]. The identified program segments are then instrumented with counters to count the number of times they are executed (termed as their *count*). This execution count information is required later in the calculation phase to compute the WCET estimates. Furthermore, different types of dependencies, if exist, are highlighted, such as data dependency or dependency

of one part of the program on the other.

- Construct Identification deals with the identification of concurrency present in the parallel code. This is achieved by identifying the synchronization constructs, e.g., barriers, locks, condition variables, and joins. These construct also provide the synchronization related information, such as, the location where synchronization is occurring, threads sharing the sync variables, path on which a thread holds lock(s) and so on. This information is used in the next step to augment a Control Flow Graph (CFG).
- CFG Development is concerned with the development of a conventional CFG of the parallel application. CFG is a data structure that defines the set of all possible execution paths of a task [28]. A CFG can be constructed automatically, e.g., through a compiler. Although conditional path analysis is not the aim of this research, the constructed CFG of the whole program can be used to determine all the conditional paths (as a CFG also includes all conditional paths). Consequently, a separate conditional path analysis is not considered. Further, as all edges and nodes of the CFG are taken in account to cover all possible paths, any state machine coverage analysis is not required. The thread's information, collected in previous steps, is then augmented to the developed CFG to represent thread synchronization and dependencies. In this way, the flow related information of the parallel application can be accounted for within the WCET computation process.

3.2 Execution Time Measurement

The execution time (*cost*) of the identified program segments is obtained using a tracing mechanism that extracts timestamps from an instrumented parallel program. To achieve this, the program segments (basic blocks) are first enclosed with the instrumentation points (*ipoints*) that are inserted using m5ops utility provided by Gem5 simulator. m5ops provides timestamps without affecting the actual execution time, thus producing no instrumentation cost. While delimiting a basic block, these *ipoint* instructions cause the target to produce a timestamp upon execution. However, when this instrumented program is executed, it produces a bulk of information, which is given to the trace parser to extract the generated timestamps and hence compute the execution cost. This cost information is combined with the execution count information in the calculation phase to compute the WCET estimates.

3.3 WCET calculation

The WCET of a parallel program is derived by combining the flow constraints and executing times of program segments into an Integer-Linear Programming problem – that is obtained from Implicit Path Enumeration Techniques (IPET) [14]. The flow constraints include the execution counts, which represent the number of times each basic block is executed. Whereas, the cost of executing the basic blocks is obtained through instrumentation, as explained in Section 3.2. To derive the execution time estimate of each thread, the ILP problem is formulated as the following objective function, which is taken from [14]:

$$Z = \sum_{i=1}^{N} c_i * x_i \qquad (1)$$

where c_i is the cost a basic block in terms of execution time and x_i is the number of times this basic block is executed. The vaiable x is bounded by loop iterations, which cannot be infinite. Infinite loops would never allow a RTES to meet its timing constraints and thus would lead to its failure An upper bound is determined by maximizing the sum of the products of the execution counts and times.

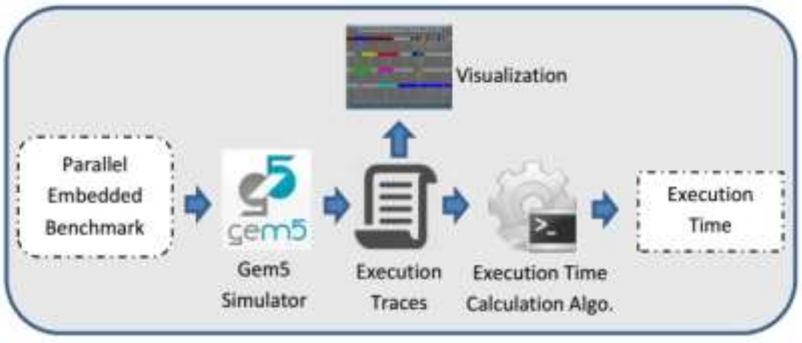


Figure 2: Internal working of the developed measurement-based analysis tool

4. Dynamic Analysis

TAPES also provides the facility to perform dynamic analysis of parallel applications, by executing and measuring the execution time. TAPES allows capturing the execution time of a parallel application, so that its execution can be analyzed from different aspects. Furthermore, TAPES can provide information regarding start and end times of threads, scheduling of threads, the CPU time taken by each thread, end-to-end time of each thread and the program itself. The starting and ending points of thread execution are detected, as depicted in Figure 2, by reading the execution traces generated by the simulator. In this way, the execution times of individual threads, as well as the entire application can easily be calculated. To automate this process, an algorithm is developed that calculates the thread execution time from the obtained traces.

5. Experimental Evaluation

5.1 Execution Platform

Like sequential programs, the measurement of parallel applications can be performed on the given hardware or on a simulator [28]. Simulation is one of the standard timing analysis technique that is used to estimate the execution time of tasks [28]. As this work is related to finding the WCET of arbitrary multi-core hardware, we do not assume a time predictable multi-core hardware (e.g., MERASA [27]). Instead, the widely used computer architecture simulator (Gem5) is used to simulate off-the-shelf multi-core hardware. One can argue on the use of the simulator for performance analysis instead of real hardware platform. Gem5 was selected as it is a cycle-accurate simulator that gives a cycle accurate model of the actual real-time embedded hardware [2]. Therefore, it provides the real-time behavior and almost the same impact as a real hardware platform. Moreover, Gem5 is a modular platform that provides full-system simulation to execute a program in the operating system environment, which is also our research interest. Additionally, Gem5 supports several commercial Instruction Set Architectures (just as ARM, ALPHA, x86, SPARC, PowerPC and MIPS), CPU types, cache levels, memory and other components, which make it more powerful than other similar simulators, such as SimpleScalar [3].

Initially, we did performed our experiments on Raspberry Pi. However, it did not provide any mechanism of getting traces, which could be used to obtain the exact clock cycle of the executed thread or its part, as needed in this work. Such facilities are only provided by the specialized equipment and tools provided by commercial companies, such as RapiTime by Rapita Systems [30]. Noticeably, these commercial products are very expensive and still do not provide all facilities needed for our experiment, such as support of simultaneous multithreading. Furthermore, during the process of design space exploration for real-time embedded hardware systems, it is not possible to use a fix hardware platform. For these obvious reasons, Gem5 proved itself as a perfect choice for our experimentation. The configuration of Gem5 used in this experiment included four cores of ARM detailed architecture (ARMv7-A ISA based) with default size of L2 cache (2MB), 256 MB of memory and ARM embedded Linux as the guest operating system contained in a disk image. The experimental stages of our framework are depicted in Figure 3.

5.2 Benchmark Suite

This research required a publicly-available software application which should both be embedded and parallel. To fulfill these requirements, *ParMiBench* suite was selected for experimentation in this work. ParMiBench is an open source parallel version of a subset of MiBench benchmark suite [9] – many of whose benchmarks appear to be suitable candidates for WCET analysis [6]. ParMiBench is a set of embedded parallel benchmarks that is actually designed to evaluate the performance of embedded multi-core systems. The benchmark is implemented using C language and POSIX threads to achieve parallelism and it supports Unix/Linux based platforms [17]. The Gem5 simulator was used to execute the *ParMiBench* benchmark and measure the execution time of its threads. *ParMiBench* suite includes benchmarks from various domains of the embedded applications, such as control and automation, networks, offices, and security. These benchmarks are *Susan* (for image processing), *BasicMath* (for mathematical operations), *StringSearch* (for string searching), *Dijkstra* (to find the shortest path) and *Sha* (for data partitioning). We have used the standard *gcc* compiler without using any compiler specific optimization, so that the impact of compilation tools is minimized. Nevertheless, the role of compilation tools on parallelizing the user programs is recognized as out of scope of this work.

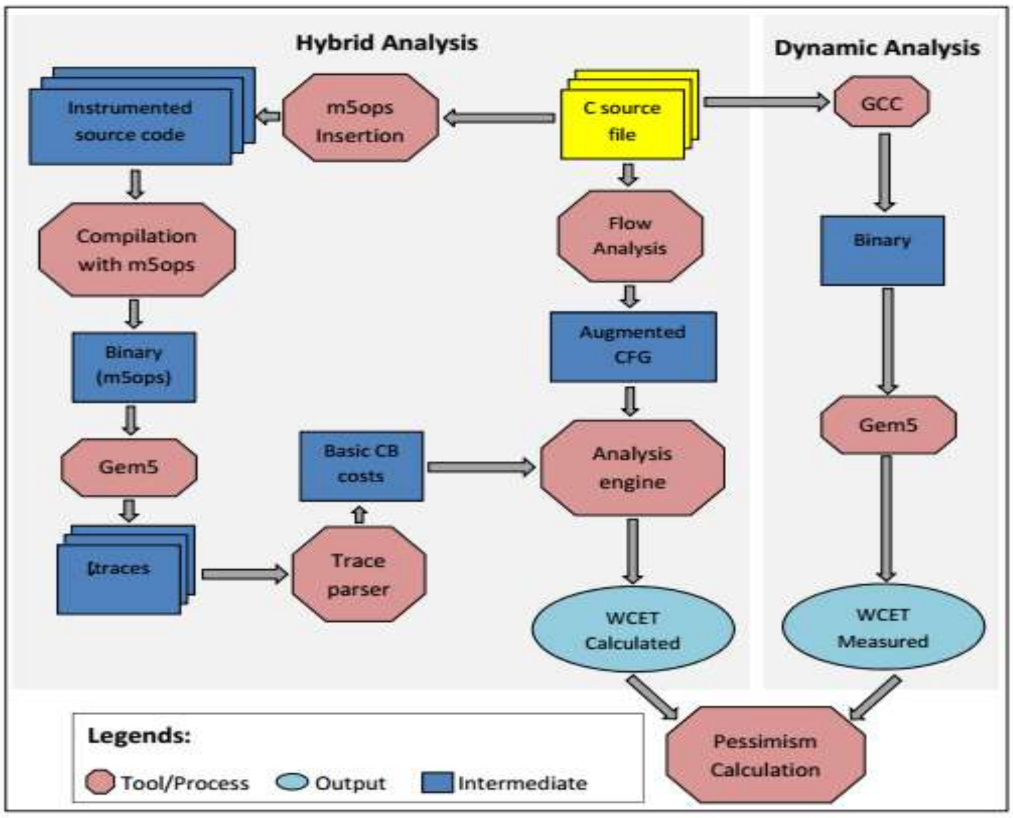
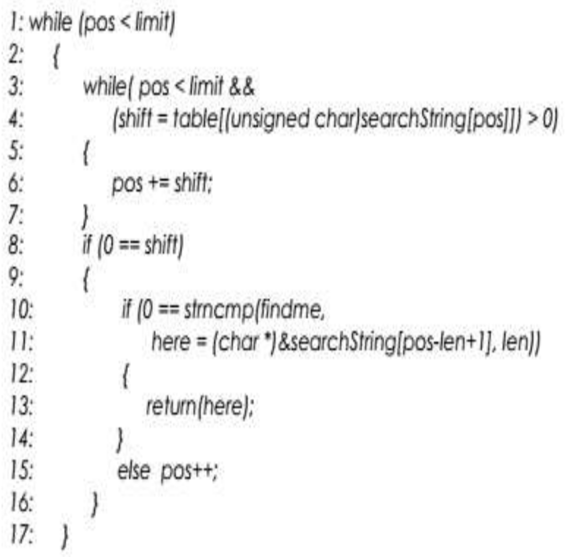


Figure 3: Overview of the experimental framework used in TAPES for WCET estimates



```
1: while (pos < limit)
2:   {
3:       while( pos < limit &&
4:           (shift = table[(unsigned char)searchString[pos]]) > 0)
5:       {
6:           pos += shift;
7:       }
8:       if (0 == shift)
9:       {
10:          if (0 == strncmp(findme,
11:              here = (char *)&searchString[pos-len+1], len))
12:          {
13:             return(here);
14:          }
15:          else  pos++;
16:      }
17:  }
```

Figure 4: Identified code segment

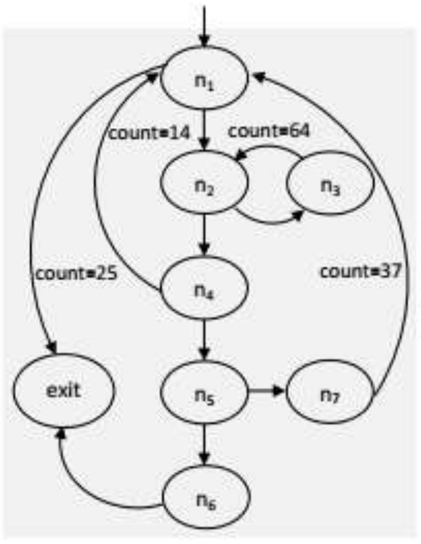


Figure 5: CFG of the identified code segment

5.3 Findings

The following facts were revealed when the proposed program flow-analysis method was applied to the ParMiBench suit. While reporting our findings, usually all the benchmarks of the suite are discussed, but in some cases the explanation is focused on the StringSearch benchmark for simplicity. StringSearch benchmark is related to searching a string from a text file.

- Thread Identification: In ParMiBench the number of threads is fixed in some benchmarks (e.g., Dijkstra and Susan), whereas in others the user is provided the option to enter. In some benchmarks, (e.g., StringSearch) threads are created only once as compared to other benchmarks (e.g., Susan), where threads are created twice or more. In ParMiBench static load balancing is used, i.e., the work is equally distributed among the threads. Instead of partitioning the program logic, the input data are divided in such a way that threads work independently. In addition, coarse-grained task decomposition has been used in most of the cases. In StringSearch benchmark, the sub-tasks such as task decomposition, data partitioning and distribution of work among workers are performed once and sequentially for all types of inputs. Thus, they have no major effect on the execution time.
- Functional Analysis: Taking the example of StringSearch benchmark, the difference lies in the search method used, as far as the timing analysis is concerned. For instance, in the Pratt-Boyer-Moore search algorithm the while loop, shown in Figure 4, was identified as the potential code segment that would consume more time. This loop was divided into two basic blocks: the code used for shifting the characters (inner while loop – line 3-7) and the code used for comparison (the *if* statement – line 8-16).
- Construct identification: To identify the concurrency in the parallel programs, we used mutex and condition variables present in POSIX thread library. The mutex variables provide locking and unlocking mechanism for mutual exclusion of critical section. We used the mutex variables *pthread_mutex_lock* and *pthread_mutex_unlock* variables for the identification of critical section in the code. The condition variables are used for the identification of thread synchronization. We used *pthread_cond_ wait*, *thread_cond_signal* and *pthread_*

cond_broadcast variables for detecting the waiting and signaling to synchronize threads.

- CFG Development: In this step, the CFG of the parallel application was developed (an example is shown in Figure 5). The CFG is augmented with the count information produced in the functional analysis step.

5.4 Comparison and Observations

In the scarcity of research studies on timing analysis of parallel programs running on multi-core architectures, it is very difficult to compare TAPES with the existing approaches. Firstly, TAPES allows hybrid analysis whereas the existing studies are based on either static or dynamic analysis. Secondly, each of these studies consider a particular aspect of parallelism different from others (details are provided in the related work section). Thirdly, each study has calculated the WCET estimates of different software applications. Lastly, the resources (e.g., the modified WCET estimation tool) for re-doing the experiments are not available. To evaluate the pessimism of the proposed hybrid solution the calculated WCET estimates are compared with the measured ones. This is not only due to the above mentioned reasons but is also in accordance with the general trend found in the research studies on timing analysis of parallel programs [8, 22, 24].

Table 1 shows the measured WCET of different benchmarks obtained by executing them, the WCET estimates of these benchmarks calculated using the hybrid analysis and the percentage of pessimism in the calculated WCET estimates with respect to the measured estimates. Note that the units of time, in Table 1, is the number of processor clock cycles. From the table, it can be observed that the calculated estimates bound the measured estimates. This is precisely the benefit of using the hybrid analysis: the WCET estimates are computed using the measured execution times of program segments and flow information, instead of relying on the worst-case input to trigger long end-to-end execution times.

However, other real life software applications would be needed to test the scalability of this analysis. One major obstacle in this regard is the unavailability of such software applications which are embedded, real-time and parallel using Pthreads standards. Since, such real life applications are not yet available, scalability testing of this analysis cannot be performed and left as future work.

6. Related Work

In the literature, the WCET analysis is performed through static analysis [24], model checking [7, 29] and measurement-based approaches [21], as depicted in Figure 6. However, there have been few contributions towards timing analysis of parallel programs, as the existing WCET-analysis research mostly focuses on sequential programs running on single-core architectures. Rochange et al. [25] for the first time highlight the problem of analyzing the timing behavior of nonsequential software on a multi-core architecture. They report a manual analysis of a parallel application, which determines the synchronization and communication among its executing threads. In the following, we shall exclusively review WCET analysis of

Table 1: The measured WCET estimates along with the WCET estimates calculated using the hybrid analysis.

Benchmark	WCET Measured	WCET Calculated	Pessimism
String search	12886961500	14991728000	16.33 %
Susan	10702583000	11836753000	10.60 %
Dijkstra	4357948800	5376740000	23.38 %

parallel programs, which is the topic of this research.

The worst-case reponse time(WCRT), of parallel applications running on multi-core platforms, is computed in [24]. Instead of proposing a new technique, the approach extends a state-of-the-art WCET estimation method to estimate the WCRT of parallel applications. In this regard, only the control flow analysis phase is modified: the classical control flow analysis runs (separately) on the functions/tasks that are put in parallel, and then new edges are added between basic blocks to model synchronization / communications among tasks. One new edge is added for each inter-task communication along with duration to model the message transmission time for communications. The hardware-level analysis runs unmodified on each function/task taken in isolation, as if it was not communicating with the other tasks executed on the other cores. Similarly, the WCET computation step is applied unmodified, as new constraints are automatically added in the WCET calculation equations (by introducing new edges) and communication delays are automatically taken into account.

A method for determining the impact of parallel monitoring on WCET is defined in [20]. The method calculates the maximum monitoring stall (time) and add it to the WCET calculated using popular methods. The traditional ILP based analysis for sequential program is extended to incorporate the overhead of monitoring. A FIFO behavior is added between the main and the monitoring cores. This non-linear FIFO behavior is modeled as a MILP problem to produce worst-case monitoring stalls. These can be incorporated into traditional IPET methods for WCET estimation.

An approach for automatic timing analysis of parallel applications is presented in [22] that show how to compute the synchronization-related stall time 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 time of child threads at synchronizations. The stall time of child threads is computed based on two types of synchronization patterns (critical sections and progress synchronizations). To ease the identification of these patterns, an annotation format is also designed. However, the approach relies on user-provided annotations to identify the synchronization patterns.

In the existing work, path based analysis methods used for WCET calculation, are classified as tree-based, path-based and IPET-based [28] (see Figure 6). In tree-based calculation, the WCET bound is calculated by bottom-up traversal of an abstract syntax tree of the program. Although tree-based calculation is very fast, they return loose WCET estimates [21]. Path-based calculation derives WCET considering all paths of the program, requiring the examination of an exponential number of paths [21]. In IPET-based calculation, the program flow information and execution times are transformed into an ILP problem where the longest execution is maximized [28]. The WCET analysis of parallel code can also be performed using the model-checking techniques [7] (using the model-checking on the timed automata system). The structure of the program is represented by the timed automata and then the UPPAAL modeling and verification tool is used to perform WCET analysis. The WCET analysis is performed by running the model-checker (to verify system properties) using UPPAAL automatically. However, the drawbacks of this method are reported by the authors in an ongoing research work [8] on analysis of threads synchronization.

Research on WCET estimation on multi-core architectures has mainly focused on the predictability of accesses to shared resources. For example, the predictability features and the timing variance of single and multi-core processors are analyzed in [12]. The paper provides an overview of the hardware features leading to predictability problems. Some examples of multi-core

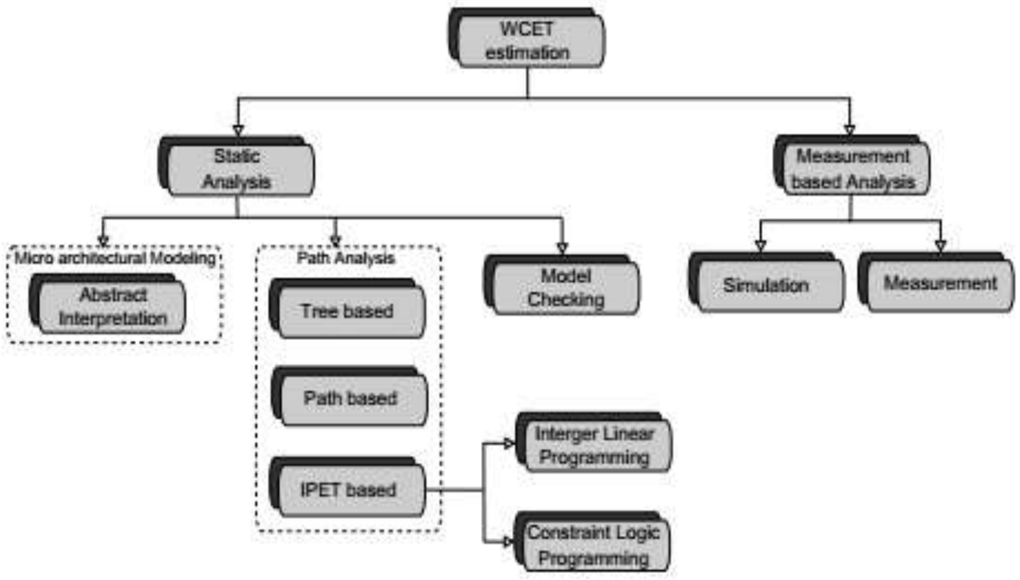


Figure 6: Classification of Worst-Case Execution Time estimation methods

configurations are also shown. To obtain predictable multi-core architectures, several configuration and design recommendations are presented. These include the recommendations for cache replacement policies, shared bus protocol, private caches and private memories.

Similarly, a WCRT analysis method is presented [18] for concurrent programs running on shared cache multi-cores. The concurrent programs are visualized in the system model as graphs of Message Sequence Charts. The L1 cache behavior is analyzed for each task in each core independently (intra-core cache analysis). By using a filter, only the memory accesses that miss in the L1 cache are analyzed at the L2 cache level. Similar to L1, the L2 cache behavior is analyzed for each task in each core independently, assuming no conflict from other tasks in other cores. An iterative solution (L2 cache conflict analysis and WCRT analysis) is proposed to overcome the conflicts of L2 cache (by estimating and exploiting the lifetime information for each task in the system).

There also exists some research projects related to designing time predictable multi-core architectures, e.g., MERASA [27], PREDATOR [31], CERTAINTY [32]. In the MERASA project, a timing predictable and WCET analyzable embedded multi-core processor has been designed with the appropriate system software. The MARASA hardware guarantees a time bounded access to shared resources, where the processor executes concurrent threads in isolation. The MERASA system software that provides an abstraction between application software and embedded hardware uses a hardware-based real-time scheduling and a thread control to provide thread isolation. The experiences in evaluating the WCET of parallel application, at the MERASA project, are reported in [25]. The recommendation of this study is to determine the parallelism and synchronization in the parallel code for its WCET analysis. However, the process described in the study is completely guided by the user and is specific to the estimation of WCET of one component of the parallel application (3D multigrid solver). Therefore, it needs to be investigated further before it can be generalized for other parallel applications.

Conclusion

To ease the temporal validation of real-time embedded multi-core systems, this paper addresses the issue of analyzing the timing behavior of parallel applications. For that the

Timing Analysis tool for Parallel Embedded Software (TAPES) is presented. TAPES provides novel ways to capture the timing properties of parallel executing threads such as inter-thread flow information, thread dependencies, and threads execution times. On one side, TAPES allows to compute the WCET estimates using a hybrid measurement-based analysis. While performing the hybrid analysis, a new program flow analysis method is proposed for parallel applications that describes their high-level execution semantics. On the other side, TAPES allows performing dynamic analysis to capture the end-to-end execution times of threads and the entire application. The applicability of TAPES is demonstrated by the timing analysis of three of the benchmarks from an embedded parallel benchmark suite. The results showed less pessimism (i.e., over-approximation is below 16.77% on average) in the computed WCET estimates, when compared to the measured estimates. In the future, we aim to perform the timing analysis of remaining benchmarks in the suite using TAPES.

Acknowledgment

This work is funded by the program of strategic technologies of the national plan for science, technology, and innovation in Saudi Arabia, under Project No. 11-INF1705- 10. The authors would like to thank Science and Technology Unit at Umm Al-Qura University, for providing the necessary support to conduct this research.

References

[1] Bate, I., & Kazakov, D. (2008, June). New directions in worst-case execution time analysis. In Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on (pp. 3545-3552). IEEE.

[2] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., ... & Sen, R. (2011). The gem5 simulator. ACM SIGARCH Computer Architecture News, 39(2), 1-7.

[3] Burger, D., & Austin, T. M. (1997). The SimpleScalar tool set, version 2.0. ACM SIGARCH computer architecture news, 25(3), 13-25.

[4] Engblom, J. (2003, May). Analysis of the execution time unpredictability caused by dynamic branch prediction. In Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE (pp. 152-159). IEEE.

[5] Gustafsson, J., & Ermedahl, A. (2008). Merging techniques for faster derivation of WCET flow information using abstract execution. In OASIcs-OpenAccess Series in Informatics (Vol. 8). Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[6] Gustafsson, J., Betts, A., Ermedahl, A., & Lisper, B. (2010). The Mälardalen WCET benchmarks: Past, present and future. In OASIcs-OpenAccess Series in Informatics (Vol. 15). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[7] Gustavsson, A., Ermedahl, A., Lisper, B., & Pettersson, P. (2010). Towards WCET analysis of multicore architectures using UPPAAL. In OASIcs-OpenAccess Series in Informatics (Vol. 15). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[8] Gustavsson, A. (2011). Worst-case execution time analysis of parallel systems. Real Time in Sweden, 104-107.

[9] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., & Brown, R. B. (2001, December). MiBench: A free, commercially representative embedded benchmark suite. In Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on (pp. 3-14). IEEE.

[10] Healy, C., Sjodin, M., Rustagi, V., & Whalley, D. (1998, June). Bounding loop iterations for timing analysis. In Real-Time Technology and Applications Symposium, 1998. Proceedings. Fourth IEEE (pp. 12-21). IEEE.

[11] Healy, C. A., Arnold, R. D., Mueller, F., Whalley, D. B., & Harmon, M. G. (1999). Bounding pipeline and instruction cache performance. IEEE Transactions on Computers, 48(1), 53-70.

[12] Kästner, D., Schlickling, M., Pister, M., Cullmann, C., Gebhard, G., Heckmann, R., & Ferdinand, C. (2012, September). Meeting real-time requirements with multi-core

processors. In International Conference on Computer Safety, Reliability, and Security (pp. 117-131). Springer, Berlin, Heidelberg.

[13] Kebbal, D. (2006, August). Automatic flow analysis using symbolic execution and path enumeration. In Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on (pp. 8-pp). IEEE.

[14] Performance analysis of embedded software using implicit path enumeration. In ACM SIGPLAN Notices (Vol. 30, No. 11, pp. 88-98). ACM.

[15] Li, Y. T. S., Malik, S., & Wolfe, A. (1999). Performance estimation of embedded software with instruction cache modeling. ACM Transactions on Design Automation of Electronic Systems (TODAES), 4(3), 257-279.

[16] Li, X., Roychoudhury, A., & Mitra, T. (2004, December). Modeling out-of-order processors for software timing analysis. In Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International (pp. 92-103). IEEE.

[17] Iqbal, S. M. Z., Liang, Y., & Grahn, H. (2010). Parmibench-an open-source benchmark for embedded multiprocessor systems. IEEE Computer Architecture Letters, 9(2), 45-48.

[18] Liang, Y., Ding, H., Mitra, T., Roychoudhury, A., Li, Y., & Suhendra, V. (2012). Timing analysis of concurrent programs running on shared cache multi-cores. Real-Time Systems, 48(6), 638-680. http://dx.doi.org/10.1007/s11241-012-9160-2

[19] Lisper, B., Ermedahl, A., Schreiner, D., Knoop, J., & Gliwa, P. (2013). Practical experiences of applying source-level WCET flow analysis to industrial code. International Journal on Software Tools for Technology Transfer (STTT), 1-11.

[20] Lo, D., & Suh, G. E. (2012, June). Worst-case execution time analysis for parallel run-time monitoring. In Proceedings of the 49th Annual Design Automation Conference (pp. 421-429). ACM.

[21] Marref, A. (2009). Predicated Worst Case Execution Time Analysis (Ph.D. dissertation). York, UK.

[22] Ozaktas, H., Rochange, C., & Sainrat, P. (2013, July). Automatic wcet analysis of real-time parallel applications. In 13th Workshop on Worst-Case Execution Time Analysis (WCET 2013) (pp. pp-11).

[23] Park, C., & Shaw, A. (1990). Experiments with a program timing tool based on source-level timing schema. [1990] Proceedings 11th Real-Time Systems Symposium. doi:10.1109/real.1990.128731

[24] Potop-Butucaru, D., & Puaut, I. (2013). Integrated worst-case response time evaluation of multicore non-preemptive applications (Rep. No. RR-8234). Retrieved from https://hal.inria.fr/hal-00787931/document

[25] Rochange, C., Bonenfant, A., Sainrat, P., Gerdes, M., Wolf, J., Ungerer, T., ... & Mikulu, F. (2010). WCET analysis of a parallel 3D multigrid solver executed on the MERASA multi-core. In OASIcs-OpenAccess Series in Informatics (Vol. 15). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[26] Theiling H., Ferdinand C., Wilhelm R. (2000). Fast and precise wcet prediction by separated cache and path analyses. Real-Time Systems 18 (2-3) 157–179.

[27] Ungerer, T., Cazorla, F., Sainrat, P., Bernat, G., Petrov, Z., & Rochange, C. et al. (2010). Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. IEEE Micro, 30(5), 66-75. http://dx.doi.org/10.1109/mm.2010.78

[28] Wilhelm, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., . . . Heckmann, R. (2008). The worst-case execution-time problem—overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems, 7(3), 1-53. doi:10.1145/1347375.1347389

[29] Wu, L., & Zhang, W. (2012). A Model Checking Based Approach to Bounding Worst-Case Execution Time for Multicore Processors. ACM Transactions on Embedded Computing Systems, 11(S2), 1-19. doi:10.1145/2331147.2331166

[30] On-target software verification solutions. (2017, July 25). Retrieved January 19, 2018, from https://www.rapitasystems.com/products/rapitime.

[31] (alex@absint.com), A. A. (n.d.). Design for

predictability and efficiency. Retrieved January 19, 2018, from http://www.predator-project.eu/

[32] "Certainty." Certainty - Welcome to the CERTAINTY Website !, Retrieved January 19, 2018, www.certainty-project.eu/.