A Safe and Tight Estimation of the Worst-Case Execution Time of Dynamically Scheduled Parallel Application

Petros Voudouris, Per Stenström, Risat Pathan

Chalmers University of Technology, Sweden
{petrosv, per.stenstrom, risat}@chalmers.se

Abstract. Estimating a safe and tight upper bound on the Worst-Case Execution Time (WCET) of a parallel program is a major challenge for the design of real-time systems. This paper, proposes for the first time, a framework to estimate the WCET of *dynamically* scheduled parallel applications.

Assuming that the WCET can be safely estimated for a sequential task on a multicore system, we model a parallel application using a directed acyclic graph (DAG). The execution time of the entire application is computed using a breadth-first scheduler that simulates non-preemptive execution of the nodes of the DAG (called, the BFS scheduler). Experiments using Fibonacci application from the Barcelona OpenMP Task Suite (BOTS) show that timing anomalies are a major obstacle to estimate safely the WCET of parallel applications. To avoid such anomalies, the estimated execution time of an application computed under the simulation of BFS scheduler is multiplied by a constant factor to derive a safe bound on WCET. Finally, an anomaly-free priority-based new scheduling policy (called, the Lazy-BFS scheduler) is proposed to estimate safely the WCET. Experimental results show that the bound on WCET computed using Lazy-BFS is not only safe but also 30% tighter than that computed for BFS.

Keywords: parallel program; timing anomaly; time predictability

1 Introduction

There is an increasing demand for more advanced functions in today's prevailing embedded real-time systems like automotive and avionics. In addition to the embedded domains, timeliness is also important in high-performance server applications, for example, to guarantee a bounded response time in the light of increasing number of clients and their processing requests. The need to satisfy such increasing computing demands both in the embedded and in the high-performance domains requires more powerful processing platform. Contemporary multicore processors provide such computing power. The main challenge to ensure timing predictability and maximizing throughput is to come up with techniques to exploit parallel multicore architectures.

Although sequential programming has been the primary paradigm to implement tasks of hard real-time applications [2], such a paradigm limits the extent to which a parallel multicore architecture can be exploited (according to Amdahl's law [3]). On

the other hand, the HPC community has developed several parallel programming models for task parallelism (e.g., Cilk [4]) and for data parallelism (e.g., OpenMp loop [5]). Under parallel programming models, a classical sequential program is implemented as a collection of parallel tasks that can execute in parallel on different cores. The quest for more performance has recently attracted parallel programming models for the design of real-time applications [6, 7]. While the HPC domain is mainly concerned about *average* throughput, the design of real-time system is primarily concerned with the *worst-case* behavior. The blending of high-performance and real-time computing poses a new challenge: *how the worst-case timing behavior of a parallel application can be analyzed?*

Scheduling algorithms play one of the most important roles in determining whether timing constraints of an application are met or not. The timing analysis of real-time scheduling algorithms often assumes that the worst-case execution time (WCET) of each application's task is known [1]. If WCET of a task is not estimated *safely*, then the outcome of schedulability analysis may be erroneous, and could result in catastrophic consequences for hard real-time applications. The estimation of the WCET needs to be also *tight* in order to avoid over-provisioning of computing resources.

The tasks of a parallel application are scheduled either statically or dynamically. Static scheduling binds (offline) each task on a particular core while dynamic scheduling allows a task to execute on any core. Recently, there have been several works on WCET estimation for statically scheduled parallel applications under simplistic assumptions, for example: the number of tasks is smaller than the number of available cores or the maximum number of tasks is two [8, 9, 10]. Such limitations may constrain the programmer to exploit higher inter-task level parallelism and, hence, limits the performance. In addition, a task assigned to a core, which is already highly loaded, may need to await execution while other cores of the platform may be idle; hence, contributing to load imbalance.

While the approaches proposed in [8, 9, 10] are inspiring, the limitations of *static scheduling* motivate us to investigate the problem of estimating the WCET of *dynamically* scheduled parallel application on multicores. To the best of our knowledge, this paper proposes, for the first time, a framework for estimating WCET of a dynamically scheduled parallel application on multicore. The proposed framework is applied to Fibonacci application from the BOTS [11].

This paper makes the following contributions. First, a methodology to model a parallel application is proposed. The model captures information regarding what code units can execute in parallel and what must execute sequentially so as to establish the WCET of parallel applications. Second, we identify *timing anomalies* triggered by dynamically scheduling tasks. Third, we contribute with new scheduling policies under which timing anomalies can be avoided. Finally, experimental results are presented to show the tightness by which WCET can be estimated using the proposed scheduling algorithms using Fibonacci from BOTS.

The rest of this paper is organized as follows: Section 2 presents our assumed system model. From this, we identify in Section 3 timing anomalies challenging WCET estimation. A systematic methodology to model parallel applications and the design of a runtime simulator are presented in Section 4. Section 5 then presents our pro-

posed scheduling algorithms (BFS, Lazy-BFS) for the run-time system. Our experimental results are presented in Section 6. Related work is presented in Section 7 before concluding in Section 8.

2 Timing Anomalies

The estimated WCET of an individual task is an upper bound on the WCET meaning it is safe. A task during runtime may take less than its estimated WCET. The overall execution time of a dynamically scheduled parallel application may increase when some tasks take less than their WCETs, which is known as *execution-time-based timing anomaly* [20]. An example of such an anomaly is demonstrated in Figure 1. The C_u value beside each node is the WCET of the corresponding task in Figure 1. The DAG is executed based on non-preemptive BFS on M=2 cores.

For example, consider the DAG and the schedule on the left-hand side of Figure 1. The execution time of the application is 9. Consider the case when node B does not execute for 3 time units but finishes after 1 unit of execution and all other nodes take their WCET. The DAG and the schedule in such case are shown on the right-hand side of Figure 1. The execution time of the application is 10. In other words, the overall execution time of the application is increased when node B takes less than its WCET. This example demonstrates an execution-time-based timing anomaly.

Spawn-Based Timing Anomaly. Execution-time-based timing anomaly made us curious to find scenarios that can result in other types of timing anomalies. In this process, we find a new type of timing anomaly that we call *spawn-based timing anomaly*. In parallel programming, a parallel task may be generated based on conditional statements, for example, depending on specific value of some variable.

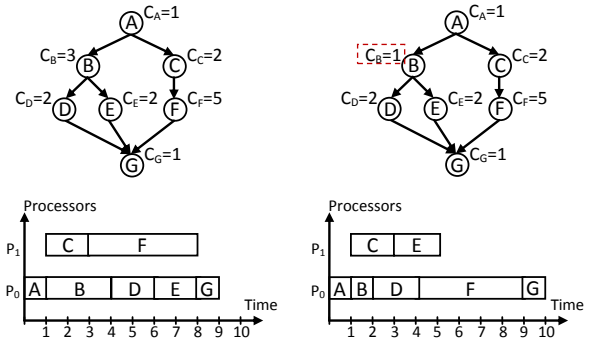

Figure 1. The DAG on the left when executed on two cores has execution time is 9 (schedule on the left). If node B takes 1 time units (the DAG on the right-hand side), the execution time is 10 on two cores. BFS is used in both cases.

A node that is generated based on some conditional statement is called a *conditional node* which may not be always present in the DAG if, for example, values of input changes. A spawn-based timing anomaly occurs if a relatively fewer number of nodes

is generated. Consider the following DAGs in Figure 2 where node C is a conditional node.

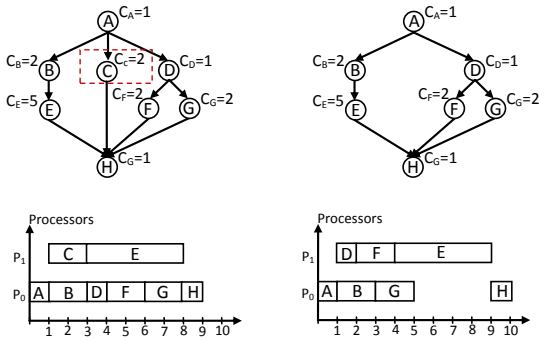

Figure 2**.** When node C is generated, the schedule length is 9. When node C is *not* generated, the schedule length is 10. BFS is used in both cases.

The schedules in Figure 2 show that the execution time of the application is larger when fewer nodes are generated (i.e., node C is not generated). We are not aware of any work where such anomaly has already been identified. Timing anomalies occur only if the execution time of a DAG is computed based on a total ordering of nodes' execution that is different from the ordering during actual execution. This paper proposes a framework to compute a safe estimation of the WCET of parallel applications mitigating the effect of timing anomalies.

3 Proposed Scheduler

Any scheduling algorithm can be plugged into the ExeSIM module. Well-known scheduling strategies are breath-first scheduler [12], work-first scheduler with work stealing [13], etc. We consider two different scheduling algorithms for ExeSIM; BFS and Lazy-BFS.

3.1 Breadth-First Scheduler (BFS)

We have implemented the non-preemptive BFS in ExeSIM. This scheduler dispatches tasks from the ready queue in breadth-first order to the idle cores for execution. Each task executes until completion without any preemption. The output of ExeSIM using BFS scheduler is an estimation of the WCET of the DAG of an application where each node of the DAG takes its WCET. We denote this estimation by EXE_{BFS}. As discussed in Section 3, such an estimation may not be an upper bound on the WCET of the application due to timing anomalies, i.e., the actual execution time may be larger than EXE_{BFS} during runtime when some task take less than their

WCETs. A safe estimation of the WCET of an application executed on an M-core platform under BFS is given (according to Theorem 3 in [20]) as follows:

$$EXE_{BFS}^{safe} = \left(2 - \frac{1}{M}\right) \times EXE_{BFS} \quad (1)$$

The value of EXE_{BFS}^{safe} is a safe bound on the WCET of an application scheduled under BFS. The multiplicative factor (2-1/M) in Eq. (1) may result in too much pessimism in case ExeSIM is close (tight) in estimation of the WCET. In order to derive a tighter estimation of WCET, we propose a priority-based scheduler, called Lazy-BFS.

3.2 Lazy-BFS: Priority-Based Non-Preemptive Scheduler

In Lazy-BFS, each node has a priority and nodes are stored in the ready queue in non-increasing priority order. We now present priority assignment policy for Lazy-BFS and then the details of its scheduling policy.

Priority assignment policy. The priority of a node is denoted as a pair (L, p) where L is the *level* of the node in the DAG and p is *level-priority* value at level L. The first node is assigned level 1 and level-priority 1, i.e. (L, p)=(1,1). The next nodes are given priorities based on the priority of the node that generates them. The new nodes are given different priorities than that of the parent node. Let the total number of new nodes generated from a parent node with priority (L, p) be D (Degree). These D nodes are ordered in BFS order (i.e., the order in which they are created). Each of the new nodes generated from a parent node with priority (L, p) is assigned level (L+1). These D ordered nodes with level (L+1) are respectively assigned level-priorities $p_i = (D * p_{par}) - D + i, i \in [1, D]$ where p_i is the priority of the i^{th} child, p_{par} is the priority of the parent, D the degree and i the position of the child. In Figure 3 an example of priority assignment is presented. It can be seen that all the nodes in different levels are assigned with different levels. The level priority is assigned based on the previous equation. Nodes that are eligible to execute in parallel will not have a tie in both of their level and level-priority pair.

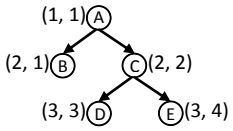

Figure 3 Example of priority assignment

.

We assume that smaller value implies higher priority. The priorities of two nodes A and B are compared as follows. First, the levels of A and B are compared. If node A has smaller level than that of node B, then A has higher priority. If the levels of A and B are equal, then the node with smaller level-priority value has higher priority.

Scheduling policy. Lazy-BFS executes tasks based on their priority in a non-preemptive fashion. In Lazy-BFS, a task is allowed to start its execution if each of its higher priority tasks has already been dispatched for execution. Note that if a relative-

ly higher priority task is not generated yet, a relatively lower priority task, which may be in the ready queue, cannot start its execution even if some core is idle. This ensures that tasks are executed strictly in their decreasing priority order. The policy is non-greedy (lazy) in the sense that ready task may not be executed even if a core is idle. We may have a situation where some higher priority task would not be created (e.g., due to conditional spawn) while a relatively lower priority task waits in the ready queue. This may create a deadlock situation. We avoid deadlock as follows: if all cores become idle, then the highest priority task from the ready queue is dispatched for execution even if some of its (non-existent) higher-priority task is not yet dispatched for execution.

Whenever a new task starts execution, the priority of that task is stored in variable (L_{lowest}, p_{lowest}) in the runtime system. If multiple tasks are ready to execute in the ready queue, Lazy-BFS starts executing the highest priority-task with priority (L, p) non-preemptively on an idle core if one of the two following conditions are satisfied:

(C1) If at least one core is busy when (L_{lowest}, p_{lowest}) = (L, p-1), then each of the tasks having priorities higher than (L, p) have either finished execution or are currently in execution. In such case, the highest-priority ready task with priority (L, p) is allocated to the idle core for execution. We also set (L_{lowest}, p_{lowest}) = (L,p) to specify that the lowest-priority task that is already given a core has priority (L,p).

(C2) If all the cores become idle, then the highest-priority ready task with priority (L, p) is allocated to the idle core for execution. We also set (L_{lowest}, p_{lowest})=(L,p).

4 Analysis Framework

We consider a multicore platform with M identical cores such that each core has a (normalized) speed 1. We consider a time-predicable multicore architecture [15, 23] such that an upper bound on time to access any shared resource, for example, memory controllers [16], cache [17, 18], inter-connection network [19] is known.

We focus on parallel applications assuming a task-based dataflow parallel programming model (e.g. Cilk [4], OpenMP [5]). A parallel application is modeled as a directed acyclic graph (DAG) denoted by G = (V, E) where V (the set of nodes) is the set of tasks and E (the set of edges) is the set of dependencies between tasks. If there is an edge from node $u_i \in V$ to node $u_k \in V$, then the edge specifies that execution of task u_k can start only after execution of task u_i completes.

We assume that the WCET of each node of the DAG is known (please see [10] where such an approach is proposed). The WCET of a node $u \in V$ is denoted as C_u. The WCET of each node includes any synchronization delay due to critical sections (please see [21] that proposes time-predictable synchronization primitives). The overheads related to scheduling decisions and managements of tasks in the ready queue have been incorporated in the WCET of the corresponding task.

In addition to the occurrences of timing anomalies, another major challenge in analyzing a dynamically scheduled parallel program is the many possible execution interleavings of different parallel nodes. We present a methodology to model the structure

of a parallel application to capture information about such inter-leavings as a directed acyclic graph (DAG).

The structure (i.e., nodes and edges) of the DAG of a parallel application depends on the input parameters. The main challenge is to determine the DAG that will have the *longest* execution time, called, the *worst-case DAG*, for some given key input. Such key input parameters are also used in computing the WCET of sequential program (e.g., loop bounds, number of array elements, etc.) [14]. Modeling an application as a DAG from a time-predictability perspective is the first building block, called, the *GenDAG* module, of our proposed framework.

We use three types of nodes to model the application's different parts.

Spawn nodes: It models the "#omp pragma task" and generates new nodes. It has a set of nodes connected in series. A node models the execution time that is required to generate a task. When the "#omp pragma task" is included in a loop or before a recursive call then multiple nodes are created. So, for example, a loop from 0 to 3, will generate 4 nodes connected in series.

Basic Node: It models the execution time of a sequentially executed piece of code.

Synchronization Node: It models the time that is required to identify that all the related nodes are synchronized. For example Figure 4 presents the generation of the graph for Fibonacci with input 3. The code of Fibonacci is presented below.

```
int fib(int n){
       int x,y;
       if(n < 2) return n;          } Basic
       #pragma omp task             }
         x = fib(n-1);              } Spawn
       #pragma omp task             }
         x = fib(n-2);              }
       #pragma taskwait             }
         return x+y                 } Sync
}
```

❶ Initially only the spawn node for Fib(3) is ready for execution and the first node of the spawn node is executed. ❷ Next Fib(2), which is also a spawn node, is generated. Since, Fib(3) is a spawn node, also the corresponding synchronization node (S3) is generated. ❸ At the next step, the second node in the Fib(3) and the first node of Fib(2) are executed in parallel. Consequently, the Fib(1) and the corresponding synchronization node (S2) are generated from Fib(2). S2 now points to the synchronization node that its parents was pointing (S3). At ❹ the second node from Fib(2), the Fib(1) from Fib(2) and the Fib(1) from Fib(3) are executed in parallel and similarly Fib(0) is generated. Since the two Fib(1) nodes were executed their dependencies are released. At ❺ Fib(0) is executed and S2 becomes ready since all its dependencies have been released. ❻ When S2 finishes S3 can start its execution since all its dependencies have been released.

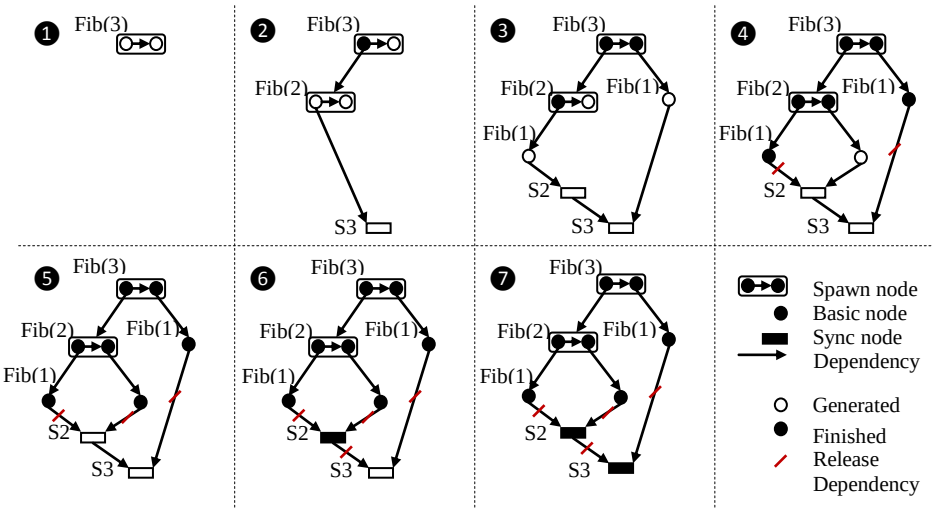

Figure 4 Example of graph generation for Fibonacci 3

The second building block of our proposed framework is the DAG execution simulator called, the ExeSIM module. The purpose of this module is to simulate the execution of the worst-case DAG to find the WCET of parallel applications under some scheduling policy. Throughput-oriented run-time systems have various sources of time unpredictability, for example, random work stealing. We implement the ExeSIM module from scratch to avoid such sources of timing unpredictability. ExeSIM is an event-based simulator that mimics the execution of the tasks of a parallel application. The input to ExeSIM is the root node of the worst-case DAG of an application and the output is the execution time of the entire application. In Figure 5 is presented an abstract view of the simulator. The GenDag module inserts new ready nodes to the ready queue. Based on the scheduling policy and the available processors the appropriate nodes are selected for execution. The results are feedback to GenDag to progress the execution of application.

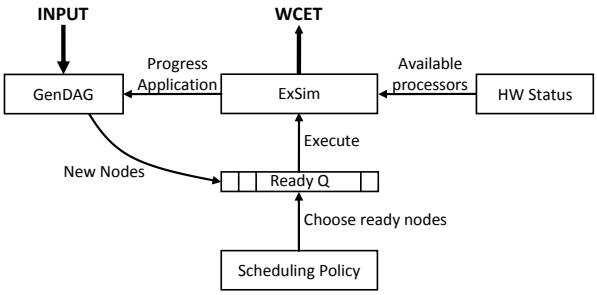

Figure 5 Abstract view of ExeSim simulator.

5 Experiments

The code of Fibonacci from BOTS is analyzed to generate its worst-case DAGs based on GenDAG module. Recall that we assume that the WCET of each individual node is known. The WCET of each spawn, sync, and basic node is assumed to 300, 100 and 400 time units, respectively.

We computed the WCET of each application using ExeSIM considering variation in input size (denoted by n), number of available cores (denoted by M), and scheduling policies (BFS or Lazy-BFS). Since ExeSIM is implemented as a sequential program, currently it is capable of handling small inputs. It is expected that the experimental results for large input and other applications from BOTS will follow similar trends presented here. In execution-time-based timing anomaly, some nodes of the DAG take less than their WCETs. To capture such behavior, we consider two additional parameters: pN and pW that are defined as follows.

Parameter pN ranges in [0, 1] and represents the percentage of nodes of a DAG for which the "actual" execution time is less than their WCETs. A node that takes smaller execution time than its WCET is called an "anomaly-critical" node. Parameter pW captures the actual execution time of an anomaly-critical node as the percentage of its WCET. These two new parameters pN and pW are used as follows.

When a new node is generated by the GenDAG module, a random number using some built-in function rand() in the range [0,1] is generated. If rand() is larger than pN, then the new node's actual execution time is set to its WCET; otherwise, the new node's actual execution time is set to pW times its WCET. For example, assume that the WCET of a new node is 20. Let pN=0.3 and pW=90% for some experiment. If rand() generates 0.75, then the new node executes for 20 time units that is equal to its WCET because 0.75 > pN=0.3. If rand() is 0.15, then the node's execution time is set to (pW × 20)=(90% × 20)=18 time units.

The ExeSIM simulator determines the execution time of the entire application based on the actual execution time of each node. If the computed execution time of the application is larger than the estimated WCET for a specific scheduling policy, then a timing anomaly is detected for that scheduling policy.

Each experiment is characterized by four parameters (n, M, pN, pW). We considered 20 different values of pN $\in$ {0.05, 0.1, …, 1.0} and pW=98%. For some given values of n and M, we compute the execution time 10.000 times for BFS. At each value of pN for some given n and M, the percentage of cases where the computed execution time is larger than the computed WCET, an anomaly is detected. This percentage of 10.000 executions is the "percentage of timing anomaly".

The results are presented in Figure 5 for Fibonacci with input n= 10, 11, 12, and13 considering M = 4, 8 and 16 cores. The x-axis in the graphs Figure 5 represents the percentage of anomaly-critical node (pN) and the y-axis represents the percentage of timing anomalies under BFS. It is evident that for all input parameters and number of cores, BFS suffers from timing anomalies. In summary, timing anomalies are frequent and we need mechanism to mitigate them.

A safe bound using BFS is given in Eq. (1). The estimation of the WCET under Lazy-BFS is safe by construction since timing anomalies cannot occur in Lazy-BFS.

The estimation under Lazy-BFS is denoted as $EXE_{LazyBFS}$. In Figure 6 we compare $EXE_{LazyBFS}$ with EXE_{BFS}^{safe} for Fibonacci. The x-axis is the input clustered by the number of processors. The vertical axis shows the WCET estimation. Form the results it can be seen that for all the cases the WCET estimation with Lazy-BFS is smaller compared to BFS. In addition, by increasing the input size the WCET estimation increases also and similarly by increasing the number of processors the WCET estimation is decreased. The WCET estimation using Lazy-BFS is around 30% tighter than that of using BFS.

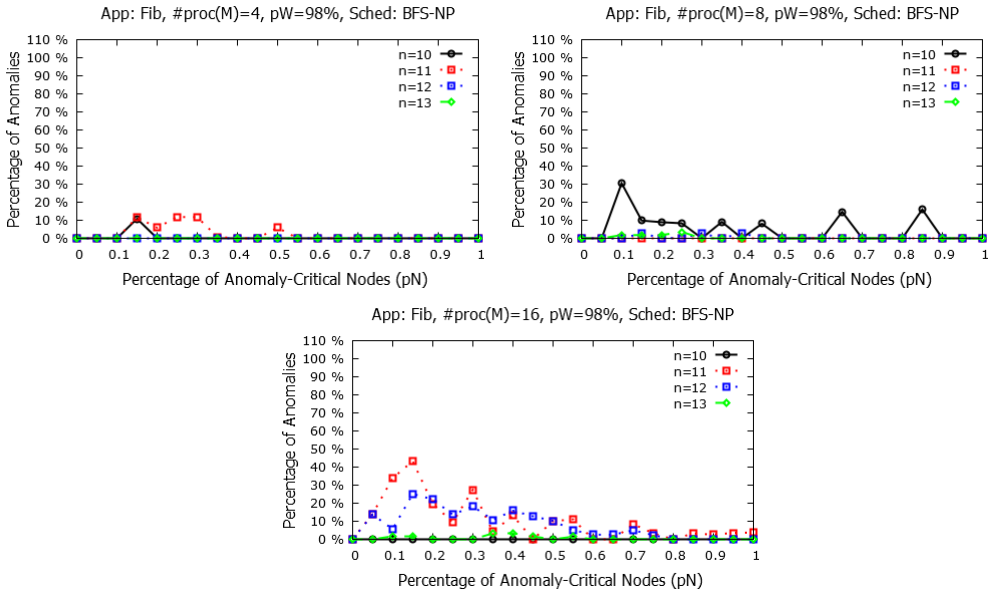

Figure 6 Percentage of Anomalies for Fibonacci using BFS schedule.

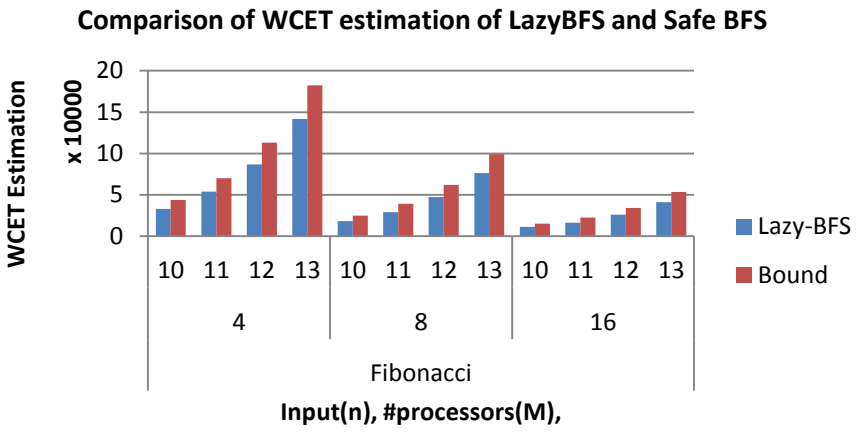

Figure 7 WCET estimation for Fibonacci is presented. The graph shows configurations for inputs 10, 11, 12 and 13 clustered for 4, 8 and 16 cores.

6 Related Work

A parallel program is relatively more complex to analyze than a sequential program due to many possible inter-leavings of the threads. Recently, the design of many time-predictable hardware has been proposed by many researchers [15, 23]. In such an architecture, the upper bound on accessing a shared hardware resource is known (predictable). Time-predictable hardware is increasingly receiving attractions in analyzing timing behavior of parallel programs [8, 9, 10, 21, 22]. Model checking is used in [22] by modeling spinlocks, private and shared caches to derive the WCET of small parallel program. However, the approach in [22] suffers from state space explosion for higher number of parallel tasks.

The work in [8] considers computing WCET of a hard real-time parallel 3D multigrid solver running on a time-predictable MERASA multicore processor. Similar to our approach, [8] also considers dividing the code in parts that can execute in parallel. The main challenge addressed in [8] is to estimate an upper bound on delay due to synchronization. Ozaktas et al. [9] also proposed techniques to compute an upper bound on stall time due to synchronization. The work in [10] proposed an approach to compute WCET of parallel application where sequential tasks execute on different cores and they communicate via messages. The main idea in [10] is that the entire application is analyzed using a graph that connects the control-flow graphs of each task using edges used to model communication channel across threads. However, these works [8, 9, 10] assume that (i) the number of threads is no larger than the number of cores, and (ii) each thread is statically assigned to one core. There exists, to the best of our knowledge, no work that considers computing the WCET of a *dynamically* scheduled parallel application.

7 Conclusion

This paper proposes a framework to compute the WCET of a dynamically scheduled parallel application. The framework has two major modules: GenDAG and ExeSIM. The GenDAG module is used to model a parallel application as a DAG. The ExeSIM is an event based simulator designed from scratch to avoid any timing-unpredictability feature often found in throughput-oriented runtime systems.

It has been shown that even if the runtime system and hardware is time predictable, execution-based timing anomalies can occur in BFS. A safe margin is added to the estimation of BFS to mitigate the effect of timing anomalies. The Lazy-BFS scheduler is free from any timing anomalies (i.e., safe) and around 30% tighter in estimating the WCET compared to the safe estimation of BFS.

8 References

[1] R.I. Davis and A. Burns, “A Survey of Hard Real-Time Scheduling for Multiprocessor Systems”, ACM Computing Surveys, 43(4), 2011.

[2] A. Burns and A. Wellings, “Real-Time Systems and Programming Languages”, 4th ed., Addison Wesley Longmain, Reading, MA, 2009.

[3] Gene M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities”, in Proc. of AFIPS, 1967.

[4] Robert D. Blumofe and Charles E. Leiserson; “Scheduling multithreaded computations by work stealing”; Journal of the ACM, 46(5):720–748, September 1999.

[5] Openmp application program interface. Version 4.0, Jul 2013.

[6] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, C. Gill, **“**Parallel real-time scheduling of DAGs**”,** IEEE Trans. on Parallel and Distributed Systems; 25(12); 2014

[7] K. Lakshmanan, S. Kato, and R. R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in Proc. of RTSS, 2010.

[8] Christine Rochange et al., “WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-core” in Proc of the WCET, 2010.

[9] H. Ozaktas et al. “Automatic wcet analysis of real-time parallel applications”, in Proc. of the WCET, 2013.

[10] Dumitru Potop-Butucaru and Isabelle Puaut, “Integrated Worst-Case Execution Time Estimation of Multicore Applications”, in Proc. of WCET Analysis, 2013.

[11] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade, “Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP”, In Proc. of ICPP, 2009.

[12] Girija J. Narlikar, “Scheduling threads for low space requirement and good locality”, Proc. of the symposium on Parallel algorithms and architectures, 1999.

[13] Matteo Frigo , Charles E. Leiserson , Keith H. Randall, The implementation of the Cilk-5 multithreaded language, Proc. of conference on Programming language design and implementation, 1998.

[14] Reinhard Wilhelm et al “The worst-case execution-time problem—overview of methods and survey of tools.” ACM Trans. Embed. Comput. Syst. 7(3), 2008.

[15] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In Proc. of ISCA, 2009.

[16] J. Staschulat, S. Schliecker M. Ivers, R. Ernst, “Analysis of Memory Latencies in Multi-Processor Systems” In proc of WCET, 2007.

[17] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures", in Proc. RTAS, 2013.

[18] Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches, Proc of RTSS, 2009.

[19] Jakob Rosén, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In Proc. of the RTSS, 2007.

[20] R. L. Graham. Bounds on multiprocessing timing anomalies. SIAM Journal of Applied Mathematics, 17(2), 1969.

[21] Wolf, J.; Gerdes, M.; Kluge, F.; Uhrig, S.; Mische, J.; Metzlaff, S.; Rochange, C.; Cassé, H.; Sainrat, P.; Ungerer, T., "RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-core Processor," Proc. of the ISORC, 2010

[22] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. *Proc. of WCET, 2010.*

[23] Martin Schoeberl. Time-predictable computer architecture. *EURASIP J. Embedded Syst.* 2009.