A Framework for the Optimization of the WCET of Programs on Multi-Core Processors

Maximilian John
Saarland University
Saarbrücken, Germany
s9mnjohn@stud.uni-sb.de

Michael Jacobs
Saarland University
Saarbrücken, Germany
jacobs@cs.uni-sb.de

ABSTRACT

For a timing-critical system, it is mandatory to guarantee upper bounds on the execution times of its programs. Such bounds can be derived using *worst-case execution time* (WCET) analysis. WCET analysis for multi-core processors is challenging as the behavior of one processor core in general depends on the behavior of the other cores. A common option to reduce this dependency is the use of *time division multiple access* (TDMA) bus arbitration.

We consider a multi-core processor with a shared TDMA bus. A system schedule for this processor assigns hard real-time tasks to processor cores and determines their execution order. A bus schedule determines which processor core is allowed to access the bus at which points in time. The WCET of a program executed on the processor depends on the choice of the system schedule and the bus schedule. We propose a framework that aims at reducing the overall WCET of the system by simultaneously constructing both schedules. Furthermore, we introduce a system model that allows to describe the considered programs in a simple way. We subsequently discuss how to overcome some restrictions of our system model. Finally, we sketch possible evaluations of our framework.

1. INTRODUCTION

For timing-critical applications, it is mandatory that their response times do not exceed the deadlines defined by the physical environment. A timing-critical application may be implemented by a composition of several programs. If there is a safe estimation of the WCET of each such program, we can give an upper bound on the total response time of the application. In many cases, it is important that these estimates are relatively tight in order to verify the timeliness of the application. WCET analysis is commonly used to derive an upper bound on the execution times of a program and thereby estimates the WCET of the program. The WCET analysis of programs executed on single-core processors is already studied well [1].

Multi-core processors typically share resources—like caches or buses—between several processor cores. Some of the advantages of multi-core processors are reduced weight, reduced production costs and a good ratio between performance and energy consumption. Therefore, it is a current trend to also use them for the design of timing-critical embedded systems. However, the resource sharing leads to the cores behaving in a different manner than with dedicated resources of the same capacities. We refer to these effects as *shared resource interference* [2]. As a consequence of this interference, the behavior of one processor core may depend on the behavior of all other processor cores. In this case, a precise WCET analysis is challenging. It is no longer sufficient to focus on one core's behavior in order to derive a tight upper bound on the execution times of a program executed on it. To bound the complexity of such an analysis, existing approaches mainly concentrate on bounding the direct timing penalties due to shared resource interference, e.g. the time that a processor core is blocked at the shared bus before its access request is granted [3, 4].

In this paper, we focus on the interference caused by shared buses. It is common to assume that the shared bus must not be accessed by more than one processor core at the same time. Therefore, there is typically an instance defining which core is allowed to access the shared bus at a particular point in time—the *bus arbiter*. We say that a processor core is *blocked* as long as one of its access requests is not granted by the arbiter. Obviously, the blocking time contributes to the overall execution time of a processor while executing a particular program. Thus, it is important to also consider the bus blocking in WCET analysis. In general, the precise consideration of the bus blocking experienced by one core requires the examination of the concurrent cores. This makes WCET analysis complex. However, in combination with TDMA bus arbitration, the bus blocking that one processor core suffers from does not depend on concurrent cores. This allows the WCET analysis to precisely model bus blocking without modeling concurrent processor cores. In this paper, we assume a system with a shared bus arbitrated according to a TDMA policy.

TDMA bus arbitration bases its decisions on a static bus schedule that assigns every time slot to the processor core which is allowed to access the bus at that instant. A program's execution time heavily depends on the static choice of this bus schedule. Thus, the eased analyzability comes at the cost of having to choose a bus schedule. This choice of the bus schedule should ideally lead to low WCET bounds for time-critical programs.

We present a heuristical framework that optimizes the system schedule and the bus schedule for a given task set and a given number of processor cores. The optimization goal is to minimize the WCET. The framework is modular in the sense of defining an interface for heuristics that select the task to be executed next on a particular processor core.

Throughout our paper, we make the following contributions:

1. A simple system model for hard real-time tasks with access to a shared bus

2. A modular framework for the optimization of the WCET of hard real-time systems
3. Approaches to apply the framework to real-world systems

2. SYSTEM MODEL

Our model is denoted by the following characteristic parameters. It consists of several equal processor cores, a shared bus and a set of hard real-time tasks. Each task may request access to the shared bus at several points in time during its execution. Assume for the moment that every access request to the bus is granted immediately. Figure 1 depicts an exemplary task. It has two bus accesses (marked purple) at time units 1 and 3, respectively. The second access is twice as long as the first one. The execution time of this task—assuming that both access requests are granted immediately—is 6 time units. Note that it is a fundamental assumption of our system model that every task is characterized by a single execution behavior and thus also by a single execution time (we will sketch in Section 4.2.1 how to support tasks with several execution behaviors).

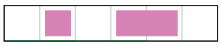


Figure 1: An example task

In our model all tasks are released simultaneously at time unit 0 and have to be executed exactly once. We assume that a task can be started independently of the progress of other tasks. In addition, there is a static assignment from tasks to the processor cores. The tasks assigned to a particular processor core are scheduled non-preemptively following a static task order. The use of non-preemptive scheduling offers several advantages for hard real-time systems [5]. We refer to the combination of the task assignment and the task orders as *system schedule*. Figure 2 shows an example of a system schedule.

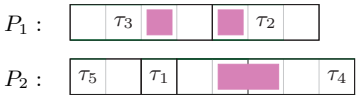


Figure 2: A system schedule

The example system schedule assigns five tasks to two processor cores. Tasks τ_2 and τ_3 are executed on the first processor core whereas the remaining tasks are executed on the second one. Furthermore, the presented system schedule describes the task order per core, e.g. task τ_2 is executed after task τ_3. According to this system schedule, both processor cores request access to the bus at time unit 4. However, the shared bus can only serve one processor core per time unit. In the following, we introduce a bus arbitration to guarantee this.

Our system model uses TDMA bus arbitration. That means, the arbiter has static knowledge about which processor core is allowed to access the bus at which point in time. This static knowledge is present in the form of a *bus schedule* which maps time units to processor cores. Note that we do not rely on periodic bus schedules. An access request of a core that is not allowed to access the bus is blocked. Figure 3 shows the system schedule of Figure 2 supplemented with a bus schedule.

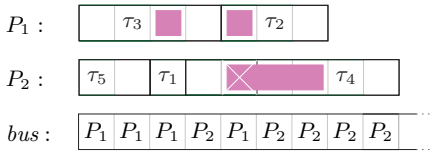


Figure 3: The effect of a bus schedule

Note that processor core P_2 is blocked at time unit 4 because P_1 is allowed to access the bus. Thus, the considered pair of system schedule and bus schedule leads to a response time of 9 time units for task τ_4. As a consequence, the overall execution time (maximum over the response times of all tasks) also amounts to 9 time units.

Intuitively, we assume that the number of time units that a task is blocked just adds up to its execution time. This assumption is commonly known as *timing compositionality*. For a detailed discussion of timing compositionality we refer to an article by Hahn et al. [6].

For the next example, consider the same system schedule and bus schedule as in Figure 3. But this time, we replace τ_4 by τ_4'.

Figure 4: New task τ_4'

This leads to the access of task τ_4' being interrupted. According to our system model, interrupted bus accesses have to be restarted from scratch. Therefore, the system schedule and the bus schedule lead to an overall execution time of 10 time units for the task set (as shown in Figure 5). Thus, the number of time units used for interrupted accesses also adds up to the execution time in a compositional way.

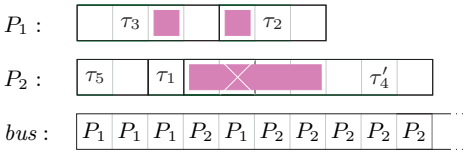


Figure 5: An interrupted and restarted access

Problem statement: Obviously, the system schedule as well as the bus schedule influence the overall execution time of the system. We assume that the task set and the number of processor cores is already given. Based on this input, we try to find a pair of system schedule and bus schedule leading to a short overall execution time for the task set.

As our system model assumes a single execution time per task (ignoring possible bus blocking effects), the overall execution time of the task set and the overall WCET of the task set coincide for our system model. For the sake of generality and comparability, we will only use the term overall WCET in the rest of this paper.

Finding an optimal static multiprocessor schedule is known to be a hard optimization problem already in the absence of accesses to a shared bus [7]. Therefore, we focus on developing a heuristic approach that finds a pair of system schedule and bus schedule leading to an overall WCET close to the possible minimum.

3. APPROACH

In this section, we present a framework that allows us to heuristically optimize the overall WCET of a task set on a

```
   Data: tasks: set of tasks,
         n: number of processor cores,
         th: task selection heuristic,
         bh: bus schedule heuristic
1  (sys, bus) ← empty schedules for n processor cores;
2  while tasks ≠ ∅ do
3  |  task ← th(tasks, sys, bus);
4  |  p_idle ← find first idle core in (sys, bus);
5  |  sys ← add task in sys to p_idle;
6  |  bus ← bh(sys, bus, "partial");
7  |  tasks ← tasks \ {task};
8  end
9  bus ← bh(sys, bus, "complete");
10 return (sys, bus);
```

Algorithm 1: Optimization procedure

multi-core processor system by choosing a system schedule and a bus schedule. It is centered around Algorithm 1, which is similar to the approach by Rosén et al. [8]. In contrast to the work of Rosén, however, our algorithm is parametric in the task selection heuristic. It integrates the construction of the system schedule and the construction of the bus schedule by alternately adding a task to the system schedule and building a part of the bus schedule.

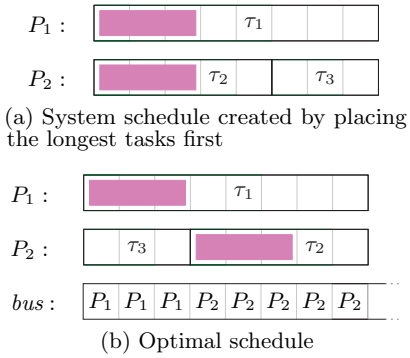
The algorithm takes as input parameters the set of tasks, the number of processor cores, the task selection heuristic *th* and the bus schedule heuristic *bh*. The task selection heuristic *th* selects one of the remaining tasks to be added to the already existing system schedule. It may base its decision on the already constructed parts of the system schedule and the bus schedule. The bus schedule heuristic *bh* continues the construction of the given bus schedule.

Algorithm 1 starts by assuming that none of the processor cores is assigned any of the tasks. We call this an empty system schedule. Analogously, an empty bus schedule is yet undefined for all time slots. In line 3 we select one of the remaining tasks to be added to the system schedule. As a next step, we consider the first point in time for which the bus schedule is yet undefined. Now, let *p_idle* be one of the processor cores which has finished the execution of its assigned tasks up to this point. Line 5 extends the existing system schedule by assigning the selected task to *p_idle*. The task order of the system schedule is extended such that the added task is executed after the tasks previously assigned to *p_idle*. After this extension, there may be points in time for which the bus schedule is not yet defined although no processor core is idle. Therefore, line 6 continues the construction of the bus schedule until one of the processor cores is idle again ("*partial*"). Afterwards, we remove the selected task from the set of remaining tasks (line 7) and repeat the previous lines until no task remains (line 2). As a final step, line 9 continues the construction of the bus schedule up to the point in time at which all processor cores are idle ("*complete*").

4. FUTURE WORK

4.1 Access-Aware Task Selection Heuristics

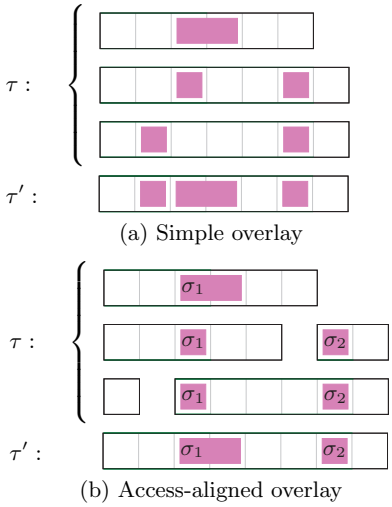
The quality of the results obtained by our framework is mainly determined by the quality of the heuristics used for the construction of the system schedule and the bus schedule.



(a) System schedule created by placing the longest tasks first

(b) Optimal schedule

Figure 6: Influence of the system schedule on the possible execution time



(a) Simple overlay

(b) Access-aligned overlay

Figure 7: Computing a single execution behavior

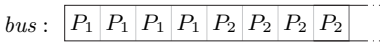
Thus, it will be our main goal to develop and compare different heuristics.

A recent approach by Rosén et al. [8] presents a rather simple task selection heuristic which can be used in our framework. In combination with our system model, this heuristic boils down to selecting the longest remaining task to be scheduled next. An exemplary system schedule created according to this heuristic is depicted in Figure 6(a). Note that any bus schedule added to this system schedule leads to a blocking of at least 3 time units for at least one of the processor cores.

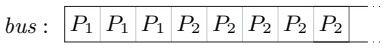
In contrast, Figure 6(b) shows that it is in fact possible to come up with a system schedule that fully utilizes all processor cores without necessarily delaying one of them. Intuitively, this is possible because the system schedule arranges the tasks in a way that no bus accesses overlap. This motivates us to develop task selection heuristics which try to reduce the access overlaps. In order to do so, it is mandatory to take into account the access behavior of the different candidate tasks.

4.2 Generalizing the Approach

4.2.1 *Tasks with Multiple Execution behaviors*



(a) Legal bus schedule



(b) Illegal bus schedule

Figure 8: Bus schedule with bus-processor ratio 4

Our system model assumes tasks with a single execution behavior (ignoring possible bus blocking effects). This assumption guarantees the efficiency of our approach as there is no need to enumerate many different execution behaviors per task.

However, real-world tasks executed on modern hardware platforms typically exhibit various execution behaviors. We aim at supporting such tasks without giving up the efficiency and simplicity of our system model. In order to support a task with a set of execution behaviors, we propose to replace this set by a single execution behavior. For every possible bus schedule, this single execution behavior should lead to an execution time at least as high as the maximum over the execution times of all members of the original set.

One possible way to obtain the single execution behavior is to overlay the execution behaviors of the original set. The principle of overlaying is depicted in Figure 7(a). Essentially, every relative position of the resulting behavior is marked as access if at least one of the original behaviors has an access at this position. Note that the resulting behavior may contain strictly more time units of bus accesses than every original behavior.

Another approach to the construction of the single execution behavior aligns the accesses of the original behaviors before performing the overlay. Figure 7(b) illustrates this approach for the same set of behaviors as already used in the example of Figure 7(a). The intuition is that we number the accesses in the increasing order of their appearance per execution behavior. Subsequently, we add the minimal amount of margin to the execution behaviors such that all accesses with the same number start at the same instant. We see that the resulting execution behavior contains one time unit of bus access less than the result in Figure 7(a). However, this comes at the cost of a longer execution time (8 time units compared to 7 in Figure 7(a)).

4.2.2 Task Dependencies

So far, we consider a scenario without task dependencies. However, supporting such dependencies in our approach is straight-forward. The task selection heuristic simply has to return a task for which all predecessors in the dependency graph already finished their execution.

This treatment of the task dependencies may—in certain situations—lead to the task selection heuristic not being able to select any of the remaining tasks. We can simply solve this problem by allowing the heuristic to return a dummy task of length 1 without bus access in such cases.

4.2.3 Less Fine-Grained Bus Schedules

In our system model, we assume that we can define the bus schedule at the same granularity of time units as the execution behavior of the tasks. If we assume that our tasks are defined at the granularity of a processor cycle, then for many realistic hardware platforms the bus schedule will not be definable at the same granularity. It is common to have an integer factor K defining the bus-processor ratio for a given hardware platform. Then the value of the bus schedule may only change at integer multiples of K.

$$\forall n \in \mathbb{N}.\ n \not\equiv 0 \mod K \Rightarrow bus(n) = bus(n-1)$$

Consider for example a bus-processor ratio of 4. Figure 8(a) shows a bus schedule that conforms to this ratio. The bus schedule in Figure 8(b) does not conform to this as it changes its value at time unit 3.

Our approach naturally supports such restrictions by using bus schedule heuristics that only create allowed bus schedules.

4.3 Evaluation

We plan to extract execution behaviors (as defined by our system model) from real-world programs. Subsequently, we intend to construct task sets based on these behaviors. We will use these task sets to compare the effectiveness and efficiency of different task selection and bus schedule heuristics. Additionally, we will compare the different heuristics to provably optimal results for relatively small examples.

Furthermore, we are interested in how the different ways to generalize our approach (cf. Section 4.2) influence the overall WCET obtained by our approach. For example, we want to find out which is the best way to replace a set of execution behaviors by a single behavior.

5. REFERENCES

[1] R. Wilhelm *et al.*, "The worst-case execution-time problem — overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 36:1–36:53, 2008.

[2] A. Abel *et al.*, "Impact of resource sharing on performance and performance prediction: A survey," in *CONCUR*, 2013, pp. 25–43.

[3] R. Pellizzoni and M. Caccamo, "Impact of peripheral-processor interference on wcet analysis of real-time embedded systems," *IEEE Transactions on Computers*, vol. 59, pp. 400–415, 2010.

[4] R. Pellizzoni *et al.*, "Worst case delay analysis for memory interference in multicore systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2010, pp. 741–746.

[5] M. Marouf and Y. Sorel, "Scheduling non-preemptive hard real-time tasks with strict periods," in *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*, 2011, pp. 1–8.

[6] S. Hahn *et al.*, "Towards compositionality in execution time analysis – definition and challenges," in *Proceedings of the International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2013.

[7] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1990.

[8] J. Rosén *et al.*, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, 2007, pp. 49–60.