Proceedings of the

8^{th} Junior Researcher Workshop on Real-Time Computing

JRWRTC 2014

http://www.cister.isep.ipp.pt/jrwrtc2014/

Versailles, France

October 8-10, 2014

CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Message from the Workshop Chairs

We are delighted to welcome you to the 8^{th} Junior Researcher Workshop on Real-Time Computing taking place in Versailles, October 2014. As part of the 22^{nd} International Conference on Real-Time and Network Systems (RTNS), the workshop provides junior researchers the opportunity to present their work, share and discuss their ideas and meet with the real-time community in a relaxed forum.

We would like to take this opportunity to express our gratitude to the members of the Program Committee listed below for thoroughly reviewing all the submitted papers. We would also like to thank all the authors who submitted their work to the workshop and hence contributed to its success. This year, JRWRTC accepted 13 of 15 peer-reviewed papers, which cover various topics of the real-time field such as scheduling, WCET analysis, time predictability, mixed criticality and sensor networks.

Yet, JRWRTC would not have been possible without the support of many people. We especially thank the General Chairs Mathieu Jan (CEA LIST, Gif-sur-Yvette, France) and Belgacem Ben Hedia (CEA LIST, Gif-sur-Yvette, France) and the local organizing committee, as well as the Program Chairs Joël Goossens (Université Libre de Bruxelles (ULB), Belgium) and Claire Maiza (Grenoble INP / Verimag, France) of RTNS 2014, for their help and support and for allowing this workshop to be, once more, part of the main event of the conference.

On behalf of the Program Committee, we wish you a pleasant workshop. We hope you will enjoy the presentations and invite you to discuss the presented ideas with the authors during the poster session.

Dorin Maxim, Polytechnic Institute of Porto
Geoffrey Nelissen, Polytechnic Institute of Porto
JRWRTC 2014 Workshop Chairs

Program Committee

Bader Alahmad	The University of British Columbia, Canada
Andrea Baldovin	Università degli Studi di Padova, Italy
Hugo Cruz-Sanchez	MyFOX, France
Pontus Ekberg	Uppsala University, Sweden
Glenn Elliott	University of North Carolina, USA
Vikram Gupta	Polytechnic Institute of Porto, Portugal
Junsung Kim	Carnegie Mellon University, USA
Leonidas Kosmidis	Barcelona Supercomputing Center, Spain
Vincent Legout	Virginia Tech, USA
Mitra Nasri	TU Kaiserslautern, Germany
Moritz Neukirchner	TU Braunschweig, Germany
Victor Pollex	Ulm University, Germany
Manar Qamhieh	Université Paris-Est, France
Abhilash Thekkilakattil	Mälardalen University, Sweden
Martijn Van Den Heuvel	TU Eindhoven, The Netherlands

Table of Contents

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).

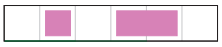


Figure 1: An example task

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.

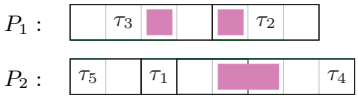


Figure 2: A system schedule

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.

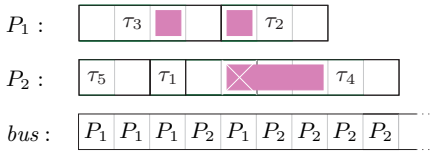


Figure 3: The effect of a bus schedule

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.

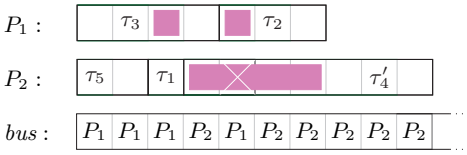


Figure 5: An interrupted and restarted access

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.

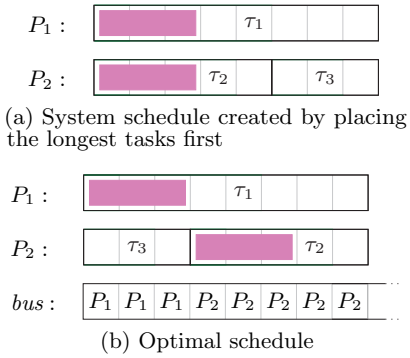
The algorithm takes as input parameters the set of tasks, the number of processor cores, the task selection heuristic *th* and the bus schedule heuristic *bh*. The task selection heuristic *th* selects one of the remaining tasks to be added to the already existing system schedule. It may base its decision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

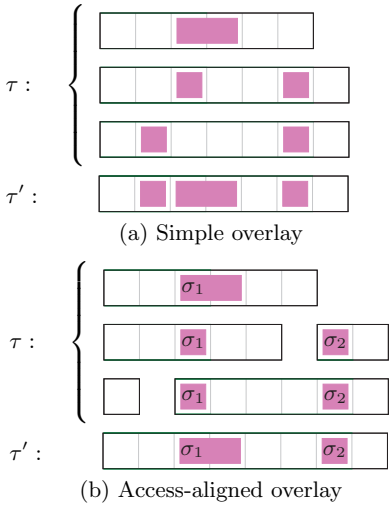
The quality of the results obtained by our framework is mainly determined by the quality of the heuristics used for the construction of the system schedule and the bus schedule.



(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

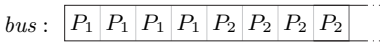
Thus, it will be our main goal to develop and compare different heuristics.

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.

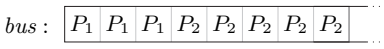
In contrast, Figure 6(b) shows that it is in fact possible to come up with a system schedule that fully utilizes all processor cores without necessarily delaying one of them. Intuitively, this is possible because the system schedule arranges the tasks in a way that no bus accesses overlap. This motivates 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.

Statically Resolving Computed Calls via DWARF Debug Information

Florian Haupenthal
AbsInt Angewandte Informatik GmbH
Science Park 1
66123 Saarbrücken, Germany
haupenthal@absint.com

ABSTRACT

Allowing virtual functions for safety-critical (embedded) systems allows for easier programming, but makes static program analysis harder. Classical analyses are too imprecise or too expensive to resolve computed calls introduced by virtual functions in C++. We present an approach that uses the DWARF debug information as an additional information source to resolve computed calls. We evaluate our approach on a set of example programs on which a value analysis cannot resolve all computed calls. Our approach resolves all the computed calls without increasing the cost of the value analysis, but still needs to be tested on real-world programs.

Categories and Subject Descriptors

D.1.5 [**Software Engineering**]: Object-oriented Programming; D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.2 [**Software Engineering**]: C++; D.4.7 [**Software Engineering**]: Real-time systems and embedded systems

General Terms

Verification

Keywords

Computed calls, static program analysis, DWARF debug information

1. INTRODUCTION

Safety-critical (embedded) software ideally runs through a verification process. This process aims to provide guarantees for every possible execution. The behaviour of a single execution depends on the program, the state of the underlying hardware, and the environment – input or sensor values for example. These configurations can be unknown. A verification by running the program for every possible configuration separately is expensive or infeasible. An abstract interpretation in the form of a static program analysis considers all possible configurations in one analysis run.

One example is a stack analysis. An embedded system provides a limited amount of memory and a program must stay within this limit. Therefore, an analysis has to provide an upper bound on the maximum amount of memory used by a program. A part of this amount is the needed space for local variables. The compiler introduces additional memory use. For example, if one function calls another one, the compiler introduces code that stores the address where the execution of the program has to proceed when the called function returns. A stack frame for each function instance contains this information. For each function call, the program creates an additional stack frame and puts it on top of the stack. If a function returns, its frame gets removed from the stack. Therefore, an analysis has to find the maximum height of the stack.

For some function calls, the call target or whether a concrete call actually takes place depends on run-time information. A stack analysis needs to account for every call target that is possible in at least one execution.

```
1  int main(int argc, char** argv)
2  {
3      foo();
4
5      if(argc != 23) {
6          bar();
7      }
8
9      return 0;
10 }
```

Figure 1: Excerpt of an example program with two function calls and a maximum of two stack frames on the stack at once. Whether the second call takes place depends on run-time information.

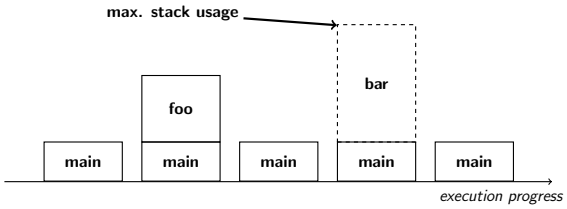


Figure 2: Different stack heights during an execution of the program from Figure 1.

Figure 1 shows an excerpt of an example program with two function calls. The first call happens in every execution. Whether the second call takes place, depends on the input values. For our static analysis, we have to consider the highest possible stack. This is the one for an execution where the second call actually takes place.

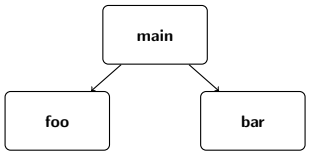


Figure 3: Static call graph of the example program from Figure 1.

Figure 2 shows a graphical representations of the stacks. The size of each box represents the size of the stack frame. The dashed lines indicate that the corresponding frame will only be there in some executions. For a guarantee over all possible executions, we have to consider this call.

Figure 3 shows a static call graph for this program. It is static as it shows a call edge for the second function call. The a^3 tool from AbsInt uses binaries as inputs and transforms them into a graph representation. Besides the stack usage analysis, the value analysis and the worst-case execution time analysis use this representation as well. Note that we use this graph representation without the source code.

Virtual functions as part of C++ can lead to calls with more than one call target. The compiler automatically resolves calls with only one possible target. If there are more possible targets, the compiler introduces a computed call. This is a call, where the program can look up which function to call at run time.

C++ programs generate objects from class descriptions. Every class description is a type. Objects start with a pointer to their corresponding class description. This class description starts with a list of pointers to the functions that this class implements itself or that this class has available because it inherits it from another class. We call this list a vtable.

The example in Figure 4 shows three code snippets. All of them show virtual function calls. The compiler can resolve the first one. Only one call target is possible and therefore, there is a concrete address for the function call. For the remaining ones, it will produce computed calls.

For the second example, a precise value analysis can still track the possible targets after the condition. Using the results of this analysis, we can create a call graph, containing two edges for the two different possible call targets. The value analysis has to keep a separate value for each possible dynamic type. This increases the cost of the whole value analysis.

For the third example, the precision in the value analysis is useless. We analyse only a part of a program. The only information for the first call is the static type. However, the binary contains no types. A value analysis can track the pointer after the assignment in line 4, but after the store in line 6 it loses every information, since this is a store to an unknown memory location.

An analysis using static type information can resolve all computed calls from the example above to generate the call graph. Since compilers can optimise their output, it is hard to interconnect points in the resulting binary and the source code. Therefore, it is hard to transfer the static type information from the source code to the call in the binary.

DWARF is a standardised format allowing a debugger to read additional information for the current state of the execution of an executable. This debug information contains the

```
1  C c;
2  c.cookie();
```

```
1  A* a;
2  B b;
3  C c;
4
5  if(???)
6      a = &b;
7  else
8      a = &c;
9
10 a->cookie();
```

```
1  example(A* a, int* location, int number) {
2      a->cookie();
3      B b;
4      a = &b;
5
6      location* = number;
7
8      a->cookie();
9  }
```

Figure 4: Code snippets with virtual functions of different analysability. The classes B and C in these snippets extend A and have own implementations for cookie(). Note that the store in line 6 is a store to an unknown memory location.

static types of variables and a relation between registers on the binary level and variables in the source code. Combining all this information, we can find a safe (over-)approximation for the set of possible call targets.

An advantage of reading the DWARF debug information is that it is similar for different compilers and platforms. This reduces the platform dependent parts of our code to a minimum.

1.1 Contributions

We introduce an extension to the current analysis framework of the a^3 tool by AbsInt. This extension makes the DWARF debug information available to analyses. We use this additional information to resolve former unresolved computed calls originating from virtual function calls. With this extension, we can analyse programs that were only analysable when manually annotated.

These are our contributions in short:

- We implemented a static reader for DWARF debug information in the a^3 tool by AbsInt. A debugger interprets this information while running an instance of the program. Our implementation can extract information without having a running instance.
- We used the additional information to analyse computed calls originating from virtual function calls for PowerPC.
- Computed calls originating from virtual functions were not automatically analysable. Therefore, programmers omitted using them and example binaries are hard to

find. We evaluated our approach with artificial examples. This shows at least that an automatic analysis of these calls is possible.

2. OUR APPROACH

Our approach consists of two parts. We implemented a static reader for DWARF debug information and scanned the binary for patterns of computed calls.

The DWARF format uses compression. When a debugger uses the DWARF debug information, it has a running instance of a program. It is only interested in information about the current state of execution of the program. The DWARF format takes advantage of these restrictions. A debugger can resolve relative addresses during debugging.

Even the incomplete, relative information can be useful. A debugger can jump from one type description to its base type description, starting from the current **this** pointer. Without a running instance of the program, our implementation can only store all the descriptions and their relations. This suffices to reconstruct the complete type hierarchy, but is separated from a concrete object. A debugger needs to know how to interpret the value of a local variable for a certain state of execution. Therefore, the DWARF debug information can contain a frame base relative location of this variable. The debugger knows the current state of execution and can resolve this location. Our interest is only in the type of the variable, but not the value. Therefore, it suffices to relate the offsets of the load instructions in the binary to these relative locations.

We store all read DWARF debug information into a sqlite database.

```
1 lwz     r9, +0(r9)
2 addi    r9, r9, +4
3 lwz     r0, +0(r9)
4 mtspr   ctr, r0
5 lwz     r3, +8(r31)
6 bctrl
```

Figure 5: Typical assembler sequence for a computed call on a PowerPC.

Figure 5 shows how a typical instruction sequence for a computed call on a PowerPC looks like. The first instruction loads the address of the vtable. The second instruction adds an offset to this address to load the second entry of the vtable. The address of the second entry is the address of the first one plus the size of one pointer, 4 bytes. The third instruction loads the address of the call target from the vtable. Instead of adding an offset to the address of the vtable, optimised code loads with an offset immediately. This platform uses a special purpose register for this function call. The fourth instruction loads the target address into this special purpose register. The fifth instruction copies the **this** pointer. In optimised code, this only happens if the called function uses the **this** pointer. The last instruction is the actual call to the target.

We implemented a pattern, matching this sequence and its variations. Our approach wants to use the static type of the variable via that the virtual function is called. The assembler only shows registers. If there is a load for copying the **this** pointer (line 5), our approach can use the source register from this instruction. If an optimisation removed this load, it can use the load of the vtable address (line 1). We use this order since the pattern matching starts at the unresolved call in the last line and goes backwards. The DWARF debug information contains information on which variable is assigned to which register at which program counter. In addition, it contains information on which variable has which static type. From the addition or the load instruction in line 3 our approach can extract the index of the virtual function in the vtable. With this information, querying the DWARF debug information yields a set of possible call targets, restricted by the static type. At the end, we can add the call targets to the branch instruction.

Querying for the possible call targets consists of two parts. In the first part, a query has to collect all implementations in classes that extends the given static type. Since the dynamic type can at most be equal to the static type, we need a second part. Here, the query needs to add the implementation of the function in static type. If there is none, it has to look for its parents in the inheritance hierarchy. This is similar to the first part, but with the difference that this query recurs only until it finds the first implementation. The first part of the query analyses the children in the hierarchy, but recurs through the whole hierarchy. The reason for this difference is that the static type is a bound on the dynamic type, but only upwards the hierarchy.

Traditionally, a value analysis tries to find out about these call targets, but this has limitations that our approach surpasses. The value analysis has to keep separate values for every possible dynamic type. For our approach, the value analysis can be more abstract and – as presented in the Evaluation – even such a simple value analysis can still refine some results from our approach. A value analysis cannot find any call targets if the creation of the corresponding objects lies outside the analysed part of the program. Our approach only uses the local information at the call site and the DWARF debug information. Therefore, it needs no knowledge about the creation of the object.

3. EVALUATION

We evaluated an implementation of our approach on a set of example programs. In this paper, we only present two of these examples.

Figure 6 shows the inheritance hierarchy of one of our examples and where we have implementations of our function of interest. An additional mark shows where in this hierarchy we can find the static type of the variable, via which we call the function. Our approach adds the implementations from B, F and G to the call graph and omits the implementations in C and E. This is a safe and precise approximation. If the dynamic type for this call is different from D, a following value analysis might declare a call to the implementation in B infeasible. An approach that cannot use the whole inheritance hierarchy has to add all five implementations instead of three.

In the example program in Figure 7, the DWARF debug information knows only the static type of **this**. Therefore, our approach adds two possible call targets to the call graph, the implementations from both classes. If we want to analyse the whole program, then it is already guaranteed that this call will call the implementation in class “Derived”. A simple value analysis on a graph containing both targets from above can declare the superfluous call edge infeasible.

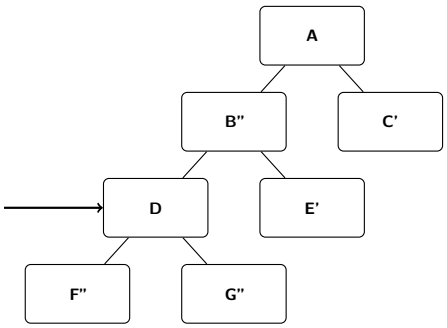


Figure 6: Inheritance hierarchy of an example program. A is the most general class. A virtual function has an implementation in the highlighted classes. The implementations in the double highlighted classes are part of a precise solution. We call the function via a variable of static type "D".

```
1  class Base {
2      public:
3          virtual void functionA()
4          {
5          }
6          void functionB()
7          {
8              this->functionA();
9          }
10 };
11
12 class Derived : public Base {
13     public:
14         void functionA()
15         {
16             Base::functionA();
17         }
18 };
19
20 int main(int argc, char** argv) {
21     Derived derived;
22     derived.functionB();
23
24     return 0;
25 }
```

Figure 7: Program where the static and the dynamic type of "this" differ. The call in "functionB" calls the implementation of "functionA" in class "Derived"

4. RELATED WORK

Both the works of Jones [2] and Shivers [4] introduce control flow analysis (CFA) considering functional and object oriented programming languages. Their work can resolve computed calls, but they have additional information available since they work on source code.

In [1] Dewey and Giffin present an analysis that finds vtable escape vulnerabilities. They search for a pattern of an invocation of a constructor to find vtables. Since we have to be able to analyse parts of programs, we cannot find all vtables with their method.

In [3], Köster finds that it is hard to interconnect program points on source code level and their corresponding instructions on binary level. Current techniques only allow for a mapping between entry and exit points of functions.

In [5], the DWARF debug information is used to find the addresses of vtables. However, they assume code for which the compiler will never produce the computed calls that we want to analyse.

In [6] Tröger and Cifuentes present a static analysis that finds the call sites of computed calls, but for the call targets, they use a dynamic analysis at run time.

5. CONCLUSIONS AND FUTURE WORK

From our examples, we can conclude that our approach is effective. For profound conclusions about its efficiency, we need more real world example programs. Since our approach only looks at small portions of the program under analysis, the efficiency only depends on the size of the inheritance hierarchy. The size of the overall program has no direct influence. For the precision, we have to compare our results against the results of a more expensive value analysis. However, our examples cover all the scenarios that can occur in real programs, but real-world programs, in addition, have different importance and frequency for those different scenarios.

In the future, we plan to examine more instruction sequences for computed calls on other platforms. Furthermore, we plan to perform more experiments.

6. REFERENCES

[1] D. Dewey and J. T. Giffin. Static detection of C++ vtable escape vulnerabilities in binary code. In *NDSS*. The Internet Society, 2012.

[2] N. D. Jones. *Flow analysis of lambda expressions*. Springer, 1981.

[3] M. Köster. Interconnecting Value Analysis Results at the Source and Binary Level. Bachelor's thesis, Saarland University, 2010.

[4] O. Shivers. Control flow analysis in scheme. In *ACM SIGPLAN Notices*, volume 23, pages 164–174. ACM, 1988.

[5] Y. Terashima and K. Gondow. Static Call Graph Generator for C++ using Debugging Information. In *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific*, pages 127–134. IEEE, 2007.

[6] J. Tröger and C. Cifuentes. Analysis of Virtual Method Invocation for Binary Translation. In A. van Deursen and E. Burd, editors, *WCRE*, pages 65–74. IEEE Computer Society, 2002.

Schedulability-Oriented WCET-Optimization of Hard Real-Time Multitasking Systems*

Arno Luppold and Heiko Falk
Institute of Embedded Systems / Real-Time Systems
Ulm University
{arno.luppold|heiko.falk}@uni-ulm.de

ABSTRACT

In multitasking hard real-time systems, each task's response time must provably be lower than or equal to its deadline. If the system does not hold all timing constraints, WCET-oriented optimizing compilers can be used to improve each task's worst-case runtime behaviour. However, current optimizations do not account for a system's schedulability contraints. We provide an approach based on Integer-Linear Programming (ILP) for schedulability-oriented WCET optimization of hard real-time systems.

1. INTRODUCTION

One of the main issues for the compiler is to determine which tasks should be optimized to which amount in order to achieve a schedulable system while complying with the limited hardware resources on embedded platforms. Current approaches on sensitivity analysis are able to provide WCETs for each task which will result in a schedulable system. However, these approaches are performed on a system-level basis, without respecting the compiler's low-level optimization capabilities for each task or the system's specific hardware capabilities and restrictions. As an example which will be used throughout this paper, many embedded platforms provide a very fast but small memory, called scratchpad memory (SPM). Putting a task into the SPM reduces its execution time, but the SPM is usually too small to keep all tasks - sometimes even too small for one complete single task. We therefore aim for integrating schedulability analysis into an optimizing WCET-aware compiler framework.
The key contributions of this paper are:

- We provide an ILP-based method to determine which tasks should be optimized to which amount in order to achieve a schedulable system.
- We provide approaches on both dynamic and fixed priority based systems.
- We provide initial evaluation results which show the potential of our approach.

This paper is organized as follows: Section 2 gives a brief overview of related projects. Section 3 shows our underlying approach. In Section 4 we integrate the approach into a WCET-oriented optimizing compiler framework and provide initial evaluation results. This paper closes with a conclusion and future challenges.

*This work was partially supported by Deutsche Forschungsgesellschaft (DFG) under grant FA 1017/1-2

2. RELATED WORK

In [7], Liu and Layland proposed an iterative approach to calculate the response times of a hard real-time multitasking system.

$$r_i = c_i + \sum_{j=0}^{i-1} \left\lceil \frac{r_i}{T_j} \right\rceil \cdot c_j \tag{1}$$

In their approach, each task τ_i is defined by a priority i, a net worst-case execution time (WCET) c_i, and the minimal period of two consecutive stimuli T_i. The system is defined to be schedulable, if each task's WCRT r_i is lower than or equal to its deadline d_i. Since then, response time analysis has been intensively studied, and more flexible task models have been proposed. Huge progress has been made to provide tight bounds on event-driven task sets which are not strictly periodical [1, 5].

However, those approaches don't give hints on how an unschedulable system could be turned into a schedulable one. To the best of our knowledge, the first work covering the issue of modification of task and system parameters was [6]. In this approach, a constant factor was introduced which scaled all tasks equally up to the system's first deadline miss. Analysing how system parameters may be tuned to keep or to establish a schedulable system is called sensitivity analysis. Since that first approach, sensitivity analysis has been improved massively. A method to sustain schedulability when adding a new task to a feasible system which is scheduled using EDF is described by [11]. Arbitrary activation patterns are analyzed in [8], and [9] proposes a method to perform a sensitivity analysis on distributed systems. Additionally, theoretical approaches have been introduced for parameter tuning of rate-monotonic scheduling in [2].

Although providing a system-level view on a system's sensitivity and robustness, those approaches either focus on task stimuli relaxation or do not offer clear optimization hints for optimizing compilers. They do offer sets of valid system parameters but the system engineer is still required to manually choose a set and try and optimize the tasks accordingly. Depending on the system's size, this may result in a tedious challenge. We therefore aim at providing a new approach which can be tightly coupled to both existing and future WCET-oriented compiler optimizations, allowing for a largely automated design and compilation of hard embedded real-time systems.

3. APPROACH

3.1 Task Model

At the current state of our work we focus on strictly periodical preemptive systems. A task τ_i is defined by its WCET c_i, its deadline d_i, period p_i and execution period T_i. We currently assume that each task's deadline d is lower than or equal to its respective period T. In case of fixed priorities, the index i denotes the task's priority, with 0 being the highest priority. We assume all timing values to be integers. This may be achieved without loss of generality e.g., by using CPU clock cycles as time unit.

3.2 ILP Model

We use the formulation provided by [10] which models the WCET of a task using integer-linear constraints. A task is split into its basic blocks, which are defined as an instruction sequence that must be traversed from top to bottom. Therefore, e.g., any branch instruction must be the very last instruction of a basic block. Variables are introduced for each basic block's execution time. The accumulated execution time w_i of a basic block i is defined to be greater than or equal to i's execution v_i, plus the accumulated execution time of its successor j:

$$w_i \geq v_i + w_j \tag{2}$$

Multiple successors are described by using multiple constraints. The ILP can then be solved using an objective function which tries to minimize the accumulated execution time of a task's entry block. This block provides a safe overapproximation of the task's WCET. The model can be accompanied by further constraints to perform ILP-based WCET optimizations. E.g., [3] used the model to perform a WCET-oriented scratch-pad memory (SPM) allocation. They perform static timing analysis on a given program, once with the whole program in slow Flash memory resulting in a worst-case execution time $v_{i,F}$ for each basic block, and once with the whole program being assigned to the SPM, resulting in $v_{i,S}$. Eq. (2) may then be extended to

$$w_i \geq v_{i,F} - b_i \cdot (v_{i,F} - v_{i,S}) + w_j \tag{3}$$

b_i denotes a binary decision variable which is set to 1 if the block is located in SPM, and 0 else. Additional constraints are added to respect the SPM's overall size and additional jump instructions. Without size constraints, the ILP solver would simply assign the whole program to the SPM. When minimizing the accumulated execution time of the program's entry block w_0, the ILP solver will minimize the overall WCET of the whole program. Additionally, w_0 will be a safe overapproximation of the program's WCET. In a multitasking environment w_0 corresponds to the task's WCET c. Our approach integrates the model by Suhendra et.al. and illustrates its usage using a slightly improved version of the SPM allocation as an example for multitasking optimizations. The improvements are mostly technical and will therefore not be discussed any further.

3.3 Dynamic Priority Systems

We will consider systems which are scheduled using Earliest Deadline First (EDF). For a system which is scheduled using EDF, [7] shows that the system is schedulable iff

$$\sum_i \frac{c_i}{T_i} \leq 1 \tag{4}$$

With c_i being the WCET of task i and T_i being the minimal period between two consecutive task executions.

For dynamic priority systems where each task's deadline equals its period, the system may easily be optimized in a schedulability oriented approach by choosing Eq. (4) as the ILP's objective function. If the optimized system's workload is lower than or equal to 1, the system is schedulable using EDF. Timing overheads inflicted by the task scheduler may be modeled by additional tasks.

3.4 Fixed Priority Systems

Currently our approach for fixed priority systems covers periodical systems, but should be adaptable to more complex systems like event triggered multitasking systems. We define a set of n tasks $\tau_i, 0 \leq i \leq n-1$, each with a period T_i, a WCET c_i, and deadline d_i. τ_0 is defined as the highest priority task and τ_{n-1} the lowest priority task, respectively. r_i is defined as the WCRT of τ_i, i.e., the maximum time span between a task becoming ready for execution and its end of execution, including all blocking times by other tasks. In a first step, response time analysis is performed by iterating over eq. (1) until one of the following conditions occurs:

- The resulting r_i does not change for any τ_i, and each τ_i, $r_i \leq d_i$. In this case, the system is schedulable and no further steps have to be performed.
- The resulting r_i does not change for any τ_i, but for at least one task τ_i, $r_i > d_i$. In this case, the system is stable but is not schedulable.
- At least one r_i might get larger than the task's period T_i. In this case, the fix point iteration is aborted and the system is said to be unstable.

We define the maximum number of preemptions of a task τ_i by another task τ_j as $p_{i,j}$. If the task is not stable, we define $p_{i,j}$ to be the maximum allowed number of preemptions:

$$p_{i,j} = \begin{cases} \left\lceil \frac{r_i}{T_j} \right\rceil & \text{if } r_i \leq T_i, j < i \\ \left\lceil \frac{d_i}{T_j} \right\rceil & \text{if } r_i > T_i, j < i \\ 0 & \text{if } j \geq i \end{cases} \tag{5}$$

Based on eq. (1), the schedulability of a system can be expressed as a set of inequations:

$$\begin{aligned} c_0 &\leq d_0 \\ c_1 + \left\lceil \frac{r_1}{T_0} \right\rceil c_0 &\leq d_1 \\ c_2 + \left\lceil \frac{r_2}{T_0} \right\rceil c_0 + \left\lceil \frac{r_2}{T_1} \right\rceil c_1 &\leq d_2 \\ &... \\ c_{n-1} + \left\lceil \frac{r_{n-1}}{T_0} \right\rceil c_0 + ... + \left\lceil \frac{r_{n-1}}{T_{n-2}} \right\rceil c_{n-2} &\leq d_{n-1} \end{aligned} \tag{6}$$

The system is schedulable if all equations hold. We establish linearity by substituting $\left\lceil \frac{r_i}{T_j} \right\rceil c_j$ with the precalculated $p_{i,j}$. The WCETs c_i are left as ILP variables and may be adapted to achieve a schedulable system. Our approach will now optimize the system performing the following steps:

1. Calculate the maximum number of preemptions $p_{i,j}$ for the original system, as described in eq. (5).

2. Create an ILP as shown in equation (6).

3. Solve the ILP, with eq. (7) as the ILP's objective function. The ILP will provide a c_i for each task i.

4. Re-calculate all $p_{i,j}$ with the newly calculated c_i

5. Modify the ILP as shown in eq. (8) and calculate the relaxed WCETs $c_{i,relaxed}$.

6. Generate the new objective function for any subsequent WCET optimizations, and perform the optimization itself.

To avoid a quadratic problem, it is necessary to assume a constant number of preemptions during the first iteration of the ILP. This will inevitably lead to an underestimation of the maximum allowed WCET for each task. Therefore, optimizations like the aforementioned SPM allocation cannot be integrated in this step of the algorithm. To ease compiler optimizations and to reduce side effects, we use the minimization of the relative change in each task's WCET as objective. The target function will be defined as:

$$\min \sum_i \frac{c_{i,\text{orig}} - c_i}{c_{i,\text{orig}}} \tag{7}$$

$c_{i,\text{orig}}$ is the original WCET of task τ_i without any optimizations. c_i is the optimized WCET of the task which will lead to a schedulable system. Using this method, the ILP solver will propose WCETs for each task which will lead to a schedulable system, while trying to modify the WCETs as little as possible.

Next, we relax the calculated maximum WCETs: We use eq. (1) to calculate updated values for each $p_{i,j}$. We can now allow each c_i to increase if the increase in execution time will not change the number of maximum preemptions. To achieve this, we modify the set of linear equations from eq. (6). For each inequation, we substitute the task's deadline on the right-hand side of the inequation by

$$\min\left(d_i, \left\lceil \frac{r_i}{T_0} \right\rceil T_0, ..., \left\lceil \frac{r_i}{T_{i-1}} \right\rceil T_{i-1}\right) \tag{8}$$

Each task's WCRT may still never exceed its deadline. As an additional constraint, the task's response time may not be larger than the minimum stimulus interval of each higher-priority task. This prevents the ILP solver from increasing a task's WCET to an amount that would allow for additional preemptions by a higher-priority task. Figure 1 illustrates this for the tasks τ_2 and τ_0 with $\left\lceil \frac{r_2}{T_0} \right\rceil = 2$. Given the new constraints, the ILP is solved a second time, leading to the relaxed WCET $c_{i,relaxed}$ for each task i.

ILP constraints denoting compiler constraints like platform dependent lower bounds on each task's WCET can be directly added to the second ILP. In theory, constraints for WCET-oriented optimizations like [3] could be added as well. However, those optimizations usually do not model the architectural behaviour exactly. For complexity reasons, jump timing penalties are overapproximated, and other behaviour, like changes in the CPU's pipelines, are not modeled at all. This leads to an overapproximation of a task's WCET when moving individual basic blocks to the SPM. We use the results from our proposed sensitivity analysis to formulate a new objective function for WCET based optimizations. Given this objective function, the tasks will be optimized with regard to the system's schedulability, without need for exact WCET estimates within the optimization algorithm. Normalizing the tasks to account for different execution times and considering the necessary relative decrease in the tasks' WCET leads to:

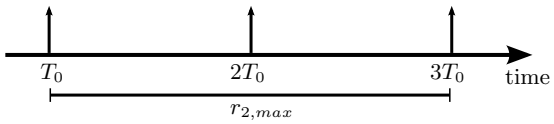


Figure 1: Maximum response time of τ_2 without additional preemptions by τ_0

$$\min \quad \left(\frac{\max\left(c_{0,orig}, ..., c_{n,orig}\right)}{c_0, orig} \cdot \frac{c_{0,orig}}{c_{0,relaxed}} \cdot c_0 + ... + \frac{\max\left(c_{0,orig}, ..., c_{n,orig}\right)}{c_n, orig} \cdot \frac{c_{n,orig}}{c_{n,relaxed}} \cdot c_n \right) \tag{9}$$

This objective can now be applied to WCET-oriented compiler optimizations as proposed in [3].

4. EVALUATION

We illustrate the approach's performance on a set of tasks from the MRTC benchmark suite [4]. We use the WCET aware C compiler framework WCC which was also used by [3]. The target platform was chosen to be the Infineon TriCore 1796 micro processor running at 150MHz. The SPM size was limited to 10% of the total program size. We did not apply any standard compiler optimizations (e.g., loop unrolling) to improve comparability for the reader. All timings printed in this section were obtained using AbsInt aiT (version 14.04, build b217166). Due to the limited space in this paper, we only chose a subset of MRTC benchmarks to show the possibilities of our approach. We tried to model a realistic system with both small, but frequently executed tasks (e.g., to poll sensor values) and much larger, but less frequently called tasks (e.g., I/O operations). We chose st, lms and matmult, as they had the largest WCET out of the MRTC benchmark suite on our platform. We then randomly chose two smaller benchmarks, in this case fibcall and sqrt. To show the effects of our different optimization approaches, we chose the periods to differ by several magnitudes. For simplification reasons, we only evaluated a strictly periodic system where each task's deadline d equals its period T.

i	Name	T	c_{orig}	$\min \sum c_i$	$\min \sum \frac{c_i}{T_i}$
0	fibcall	50	4.84	5.62	2.20
1	sqrt	100	88.79	100.03	50.11
2	st	50,000	3401.87	2570.15	2481.53
3	lms	75,000	9125.07	7002.59	9125.07
4	matmult	100,000	2699.29	1831.07	2699.29

Table 1: Dynamic priorities, times are in μs

Table 1 shows the WCETs for each task, without optimization, with SPM optimization which tries to minimize the sum over all WCETs, and the EDF-oriented SPM optimization. Table 3 shows the corresponding system load. It can be seen that the system is not schedulable without the SPM optimization. When applying the SPM optimization without regarding tasks' periods, the system load even goes up, because c_0 and c_1 increase. This somewhat strange behaviour

Prio.	Name	T	c_{orig}	$c_{relaxed}$	c_{final}
0	fibcall	50	4.84	4.84	4.84
1	sqrt	100	88.79	54.99	51.35
2	st	50000	3401.87	3401.87	3401.87
3	lms	75000	9125.07	9124.61	7002.59
4	matmult	100000	2699.29	2699.29	2699.29

Table 2: Fixed priorities, times are in μs.

	Load
Unopt.	1.20
$\min \sum c_i$	1.27
$\min \sum {}^{c_i}/_{T_i}$	0.74
Opt. Fixed Prio.	0.79

Table 3: System Load

stems from the fact that the TriCore processor performs instruction fetches on 64bit memory block boundaries. The SPM optimization moves parts of the larger tasks st, lms and matmult to the SPM, therefore instruction addresses of the unoptimized tasks fibcall and sqrt change and cause a change in the system's instruction fetch behaviour, leading to an increase of the tasks' WCET. The ILP formulation of the optimization does not account for this, and is therefore not aware of the increase in the tasks' WCET. This is a purely random phenomenon which did not occur on the fixed priority optimization (compare Table 2). When applying the EDF-oriented optimization which tries to minimize the system load, it can be observed that the load drops to 0.74, thus resulting in a schedulable system.

The results for the fixed priority optimization are shown in Table 2. The 5th column, labeled $c_{relaxed}$, shows the target WCET for each task, as it is calculated by subsequently calling the ILP as discussed in section 3.4. It can be seen that only the tasks τ_1 and τ_3 are to be optimized. For the SPM optimization we used, the SPM model only allows either to keep or to decrease a task's WCET. We take this into consideration and adapt eq. (9) by only adding those tasks, of which the WCET must be reduced. This results in the following objective function:

$$\min\left(\frac{9125.07}{88.79}\frac{88.79}{54.99}c_1 + 1 \cdot \frac{9125.07}{9124.61}c_3\right) \approx \min\left(166c_1 + c_3\right) \tag{10}$$

Table 4 shows the resulting response times. We compared the response times using the approach for fixed priority systems with the response times which would arise if the tasks which were optimized for EDF were scheduled using fixed priorities. Due to the different optimization targets, the response times are higher when using the fixed priority approach. However, that approach may be adapted for systems which are not strictly periodical. In this example, both approaches lead to a schedulable system.

5. FUTURE CHALLENGES

We provided an approach to adapt single-task WCET-oriented compiler optimizations to achieve schedulability of multitasking hard real-time systems. Future work will cover an improved optimization strategy for fixed priority tasks, as well as profound analysis of the suboptimality which is inherent for fixed priority systems. We will analyse, if, and under which circumstances, the fixed priority approach will outperform the system load oriented approach. Additionally, we are currently working at integrating the fixed priority approach into event-based systems to perform an improved optimization of systems which are not strictly periodical. In the long run, we are planning on extending the approach to cover scheduler overheads and inter-task dependencies like cache-related preemption delays.

Name	r opt dyn.	r fixed	Deadline
fibcall	2.2	4.84	50
sqrt	54.51	61.03	100
st	5479.77	8772.81	50,000
lms	25562.02	26761.40	75,000
matmult	31477.60	33671.99	100,000

Table 4: Response time using RMS, times are in μs

6. REFERENCES

[1] K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. In *Proceedings of ECRTS*, pages 187–195, 2004.

[2] E. Bini, M. Di Natale, and G. Buttazzo. Sensitivity analysis for fixed-priority real-time systems. *Real-Time Systems*, 39:5–30, 2008.

[3] H. Falk and J. C. Kleinsorge. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In *Proceedings of DAC*, pages 732–737, 2009.

[4] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET Benchmarks – Past, Present and Future. In B. Lisper, editor, *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 137–147, 2010.

[5] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *Proceedings of Computers and Digital Techniques*, 152(2):148–166, 2005.

[6] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings of RTSS*, pages 166–171, 1989.

[7] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[8] M. Neukirchner, S. Quinton, T. Michaels, P. Axer, and R. Ernst. Sensitivity Analysis for Arbitrary Activation Patterns in Real-time Systems. In *Proceedings of DATE*, pages 135–140, 2013.

[9] R. Racu, A. Hamann, and R. Ernst. Sensitivity Analysis of Complex Embedded Real-Time Systems. *Real-Time Systems*, 39:31–72, 2008.

[10] V. Suhendra, T. Mitra, A. Roychoudhury, et al. WCET Centric Data Allocation to Scratchpad Memory. In *Proceedings of RTSS*, pages 223–232, 2005.

[11] F. Zhang, A. Burns, and S. Baruah. Sensitivity analysis of arbitrary deadline real-time systems with EDF scheduling. *Real-Time Systems*, 47:224–252, 2011.

Accounting for Cache Related Pre-emption Delays in Hierarchical Scheduling with Local EDF Scheduler

Will Lunniss
Department of Computer Science
University of York
York, UK
wl510@york.ac.uk

Sebastian Altmeyer
Computer Systems Architecture Group
University of Amsterdam
Netherlands
altmeyer@uva.nl

Robert I. Davis
Department of Computer Science
University of York
York, UK
rob.davis@york.ac.uk

ABSTRACT

Hierarchical scheduling provides a means of composing multiple real-time applications onto a single processor such that the temporal requirements of each application are met. This has become a popular technique in industry as it allows applications from multiple vendors as well as legacy applications to co-exist in isolation on the same platform. However, performance enhancing features such as caches mean that one application can interfere with another by evicting blocks from cache that were in use by another application, violating the requirement of temporal isolation. In this paper, we present some initial analysis that bounds the additional delay due to blocks being evicted from cache by other applications in a system using hierarchical scheduling when using a local EDF scheduler.

1. INTRODUCTION

Hierarchical scheduling provides a means of composing multiple applications onto a single processor such that the temporal requirements of each application are met. This is driven by the need to re-use legacy applications that once ran on slower, but dedicated processors. Each application, referred to as a component, has a dedicated server. A global scheduler then allocates processor time to each server, during which the associated component can use its own local scheduler to schedule its tasks.

In hard real-time systems, the schedulability of each task must be known offline in order to verify that the timing requirements will be met at runtime. However, in pre-emptive multi-tasking systems, caches introduce additional *cache related pre-emption delays* (CRPD) caused by the need to re-fetch cache blocks belonging to the pre-empted task which were evicted from the cache by the pre-empting task. These CRPD effectively increase the worst-case execution time of the tasks. It is therefore important to be able to calculate, and account for, CRPD when determining if a system is schedulable or not. This is further complicated when using hierarchical scheduling as servers will often be suspended while their components' tasks are still active, that is they have started but have not yet completed execution. While a server is suspended, the cache can be polluted by the tasks belonging to other components. When the global scheduler then switches back to the first server, tasks belonging to the associated component may have to reload blocks into cache that were in use before the global context switch.

Hierarchical scheduling has been studied extensively in the past 15 years. Deng and Liu [7] were the first to propose such a two-level scheduling approach. Later Feng and Mok [8] proposed the resource partition model and schedulability analysis based on the supply bound function. Shin and Lee [16] introduced the concept of a temporal interface and the periodic resource model, and refined the analysis of Feng and Mok. When using a local EDF scheduler, Lipari *et al.* [11] [12] have investigated allocating server capacity to components, proposing an exact solution. Recently Fisher and Dewan [9] have developed a polynomial-time approximation with minimal over provisioning of resources.

Hierarchical systems have been used mainly in the avionics industry. For example, the ARINC 653 standard [2] defines temporal partitioning for avionics applications. The global scheduler is a simple Time Division Multiplexing (TDM), in which time is divided into frames of fixed length, each frame is divided into slots and each slot is assigned to one application.

Analysis of CRPD uses the concept of *useful cache blocks* (UCBs) and *evicting cache blocks* (ECBs) based on the work by Lee *et al.* [10]. Any memory block that is accessed by a task while executing is classified as an ECB, as accessing that block may evict a cache block of a pre-empted task. Out of the set of ECBs, some of them may also be UCBs. A memory block m is classified as a UCB at program point ρ, if (i) m may be cached at ρ and (ii) m may be reused at program point q that may be reached from ρ without eviction of m on this path. In the case of a pre-emption at program point ρ, only the memory blocks that are (i) in cache and (ii) will be reused, may cause additional reloads. For a more thorough explanation of UCBs and ECBs, see section 2.1 "Pre-emption costs" of [1].

A number of approaches have been developed for calculating the CRPD when using fixed priority pre-emptive scheduling under a flat, single-level system. A summary of these approaches, along with the state-of-the-art approach is available in [1]. In 2013, Lunniss *et al.* [14] presented a number of approaches for calculating CRPD when using pre-emptive EDF scheduling.

In 2014, Lunniss *et al.* [13] extended previous works to include CRPD analysis under hierarchical scheduling when using a local FP scheduler.

The remainder of the paper is organised as follows. Section 2 introduces the system model, terminology and notation used. Section 3 recaps existing CRPD and schedulability analysis. Section 4 introduces the new analysis for calculating component level CRPD incurred in hierarchical systems when using a local EDF scheduler. In section 5 the analysis is evaluated, and section 6 concludes with a summary and outline of future work.

2. SYSTEM MODEL

We assume a single processor system comprising m components, each with a dedicated server ($S^1..S^m$) that allocates processor capacity to it. We use Ψ to represent the set of all components in the system. G is used to indicate the index of the component that is being analysed. Each server S^G has a budget Q^G and a period P^G, such that the associated component will receive Q^G units of execution time from its server every P^G units of time. Servers are assumed to be scheduled globally using a non-pre-emptive scheduler, as found in systems that use time partitioning to divide up access to the processor. While a server has remaining capacity and is allocated the processor, we assume that the tasks of the associated component are scheduled using pre-emptive EDF.

The system comprises a taskset Γ made up of a fixed number of tasks ($\tau_1..\tau_n$) divided between the components. Each component contains a strict subset of the tasks, represented by Γ^G. For simplicity, we assume that the tasks are independent and do not share resources requiring mutually exclusive access, other than the processor.

Each task, τ_i may produce a potentially infinite stream of jobs that are separated by a minimum inter-arrival time or period T_i. Each task has a relative deadline D_i, a worst case execution time C_i (determined for non-pre-emptive execution). We assume that deadlines are either *implicit* (i.e. $D_i=T_i$) or *constrained* (i.e. $D_i \leq T_i$).

Each task τ_i has a set of UCBs, UCB_i and a set of ECBs, ECB_i represented by a set of integers. If for example, task τ_1 contains 4 ECBs, where the second and fourth ECBs are also UCBs, these can be represented using $ECB_1 = \{1,2,3,4\}$ and $UCB_1 = \{2,4\}$. Each component G also has a set of UCBs, UCB^G and a set of ECBs, ECB^G, that contain respectively all of the UCBs, and all of the ECBs, of their tasks, i.e. $UCB^G = \bigcup_{\forall \tau i \in \Gamma^G} UCB_i$ and $ECB^G = \bigcup_{\forall \tau i \in \Gamma^G} ECB_i$.

Each time a cache block is reloaded, a cost is introduced that is equal to the *block reload time* (BRT). We assume a direct mapped cache, but the work extends to set-associative caches with the LRU replacement policy as described in section 2 of [1]. We focus on instruction only caches.

3. EXISTING SCHEDULABILITY AND CRPD ANALYSIS

Schedulability analysis for EDF uses the *processor demand bound* function [3], [4], in order to determine the demand on the processor within a fixed interval. It calculates the maximum execution time requirement of all tasks' jobs which have both their arrival times and their deadlines in a contiguous interval of length *t*. Baruah *et al.* showed that a taskset is schedulable under EDF iff $\forall t > 0, h(t) \leq t$. We use a modified equation for $h(t)$ from [14] which includes $\gamma_{t,j}$ to represent the CRPD caused by task τ_j that may affect any job of a task with both its release times and absolute deadlines within an interval of length *t*.

$$h(t) = \sum_{j=1}^{n} \left(\max\left\{0, 1 + \left\lfloor \frac{t - D_j}{T_j} \right\rfloor\right\} C_j + \gamma_{t,j} \right) \quad (1)$$

In order to determine the schedulability of a taskset in a hierarchical system, we must account for the limited access to the processor. The *supply bound function* [16], or specifically the inverse of it, can be used to determine the maximum amount of time needed by a specific server to supply some capacity *c*. We define the *inverse supply bound function*, *isbf*, for component *G* as $isbf^G$ [15]:

$$isbf^G(c) = c + (P^G - Q^G)\left(\left\lceil \frac{c}{Q^G} \right\rceil + 1\right) \quad (2)$$

4. NEW CRPD ANALYSIS

In [13] Lunniss *et al.* presented a number of approaches for calculating CRPD in hierarchical systems when using a local FP scheduler. We now describe how CRPD analysis can be adapted for use with a local EDF scheduler. This analysis assumes a non-pre-emptive global scheduler (i.e. the capacity of a server is supplied without pre-emption, but may be supplied starting at any time during the server's period).

The analysis must account for the cost of reloading any UCBs into cache that may be evicted by tasks running in the other components, which we call component level CRPD. To account for the component level CRPD, we define a new term γ_t^G that represents the CRPD incurred by tasks in component *G* due to tasks in the other components running while the server (S^G) for component *G* is suspended. Combining (1), with $isbf^G$, (2), and γ_t^G, we get the following expression for determining the processor demand within an interval of length *t*.

$$h(t) = isbf^G\left(\sum_{j=1}^{n} \left(\max\left\{0, 1 + \left\lfloor \frac{t - D_j}{T_j} \right\rfloor\right\} C_j + \gamma_{t,j} \right) + \gamma_t^G \right) \quad (3)$$

In the computation of γ_t^G, we use a number of terms, described below. We use $E_j(t)$ to denote the maximum number of jobs of task τ_j that can have both their release times and their deadlines in an interval of length *t*, which we calculate as follows:

$$E_j(t) = \max\left(0, 1 + \left\lfloor \frac{t - D_j}{T_j} \right\rfloor\right) \quad (4)$$

We use $E^G(t)$ to denote the maximum number of times server S^G can be both suspended and resumed during *t*. Note that (5) can be used with $t=D_j$ to calculate the maximum number of times server S^G can be suspended and resumed during a single job of task τ_j.

$$E^G(t) = 1 + \left\lfloor \frac{t}{P^G} \right\rfloor \quad (5)$$

We use the term *disruptive execution* to describe an execution of server S^Z while server S^G is suspended that results in tasks from component Z evicting cache blocks that tasks in component G might have loaded and may need to reload in an interval of length *t*. Note that if server S^Z runs more than once while server S^G is suspended, its tasks cannot evict the same blocks twice and as such, the number of disruptive executions is bounded by the number of times that server S^G can be both suspended and resumed. Specifically, we are interested in how many disruptive executions a server can have during an interval of length *t*. We use X^Z to denote the maximum number of such disruptive executions.

$$X^Z(S^G, t) = \min\left(E^G(t), 1 + \left\lceil \frac{t}{P^Z} \right\rceil\right) \quad (6)$$

4.1 Component level CRPD

We first calculate an upper bound on the UCBs that if evicted by tasks in the other components may need to be reloaded. We do this by forming a multiset that contains the UCBs of task τ_k repeated $E^G(D_k)E_k(t)$ times for each task in $\tau_k \in \Gamma^G$. This multiset reflects the fact that server S^G can be suspended and resumed at most $E^G(D_k)$ times during a single job of task τ_k and there can be at most $E_k(t)$ jobs of task τ_k that have their release times and absolute deadlines within the interval of length *t*.

$$M_{G,t}^{ucb} = \bigcup_{\forall k \in \Gamma^G} \left(\bigcup_{E^G(D_k)E_k(t)} UCB_k \right) \quad (7)$$

The second step is to determine which ECBs of the tasks in the other components could evict the UCBs in (7), for which we present three different approaches.

4.1.1 UCB-ECB-Multiset-All

The first option is to assume that every time server S^G is suspended, all of the other servers run and their tasks evict all the cache blocks that they use. We therefore take the union of all ECBs belonging to the other components to get the set of blocks that could be evicted. We form a second multiset that contains $E^G(t)$ copies of the ECBs of all of the other components in the system. This multiset reflects the fact that the other servers' tasks can evict blocks (that need to be reloaded) at most $E^G(t)$ times within an interval of length *t*.

$$M_{G,t}^{ecb-A} = \bigcup_{E^G(t)} \left(\bigcup_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \text{ECB}^Z \right) \quad (8)$$

The total CRPD incurred by tasks in component G due to the other components in the system is then given by the size of the multiset intersection of $M_{G,t}^{ucb}$ (7) and $M_{G,t}^{ecb-A}$ (8).

$$\gamma_t^G = \text{BRT} \bullet \left| M_{G,t}^{ucb} \cap M_{G,t}^{ecb-A} \right| \quad (9)$$

4.1.2 UCB-ECB-Multiset-Counted

The above approach works well when the global scheduler uses a TDM schedule such that each server has the same period and/or components share a large number of ECBs. If some servers run less frequently than server S^G, then the number of times that their ECBs can evict blocks may be over counted. One solution to this problem is to consider each component separately by calculating the number of disruptive executions, $X^Z(S^G,t)$, that server S^Z can have on tasks in component G during t. We form a second multiset that contains $X^Z(S^G,t)$ copies of ECB^Z for each of the other components Z in the system. This multiset reflects the fact that the tasks of each component Z can evict blocks at most $X^Z(S^G,t)$ times within an interval of length t.

$$M_{G,t}^{ecb-C} = \bigcup_{\substack{\forall Z \in \Psi \\ \wedge Z \neq G}} \left(\bigcup_{X^Z(S^G,t)} \text{ECB}^Z \right) \quad (10)$$

The total CRPD incurred by task τ_i, in component G due to the other components in the system is then given by the size of the multiset intersection of $M_{G,t}^{ucb}$ (7) and $M_{G,t}^{ecb-C}$ (10).

$$\gamma_t^G = \text{BRT} \bullet \left| M_{G,t}^{ucb} \cap M_{G,t}^{ecb-C} \right| \quad (11)$$

4.1.3 UCB-ECB-Multiset-Open

In *open* hierarchical systems, the other components may not be known a priori as they can be introduced into a system dynamically. Additionally, even in *closed* systems, full information about the other components in the system may not be available until the final stages of system integration. However, as the cache utilisation of the other components can often be greater than the size of the cache, the precise set of ECBs does not matter. We form a second multiset that contains $E^G(t)$ copies of all cache blocks. This multiset reflects the fact that server S^G can be both suspended and then resumed, after the entire contents of the cache have been evicted at most $E^G(t)$ times within an interval of length t.

$$M_{G,t}^{ecb-O} = \bigcup_{E^G(t)} (\{1,2,..N\}) \quad (12)$$

Where N is the number of cache sets.

The total CRPD incurred by tasks in component G due to the other unknown components in the system is then given by the size of the multiset intersection of $M_{G,t}^{ucb}$ (7) and $M_{G,t}^{ecb-O}$ (12).

$$\gamma_t^G = \text{BRT} \bullet \left| M_{G,t}^{ucb} \cap M_{G,t}^{ecb-O} \right| \quad (13)$$

For all approaches, we calculated the limit (largest value of t that needs to be checked in (1)) using an inflated utilisation in a similar way to that described in section *V. D* of [14].

5. EVALUATION

In this section we compare the different approaches for calculating CRPD in hierarchical scheduling using synthetically generated tasksets. The evaluation was setup to model an ARM processor clocked at 100MHz with a 2KB direct-mapped instruction cache. The cache was setup with a line size of 8 Bytes, giving 256 cache sets, 4 Byte instructions, and a BRT of 8µs. To generate the components and tasksets, we generated n (default of 24) tasks using the UUnifast algorithm [6] to calculate the utilisation, U_i of each task so that the utilisations added up to the desired utilisation level. Periods T_i, were generated at random between 10ms and 1000ms according to a log-uniform distribution. C_i was then calculated via $C_i = U_i T_i$. We assigned implicit deadlines, i.e. $D_i = T_i$. We used the UUnifast algorithm to obtain the number of ECBs for each task so that the ECBs added up to the desired cache utilisation (default of 10). Here, cache utilisation describes the ratio of the total size of the tasks to the size of the cache. A cache utilisation of 1 means that the tasks fit exactly in the cache, whereas a cache utilisation of 10 means the total size of the tasks is 10 times the size of the cache. The number of UCBs was chosen at random between 0 and 30% of the number of ECBs on a per task basis, and the UCBs were placed in a single group at a random location in each task. We then split the tasks at random into 3 components with equal numbers of tasks in each and set the period of each component's server to 5ms. We generated 1000 systems using this technique.

For each system, the total task utilization across all tasks not including pre-emption cost was varied from 0.025 to 1 in steps of 0.025. For each utilization value, we initialised each servers' capacity to the minimum possible value, (i.e. the utilisation of all of its tasks). We then performed a binary search between this minimum and the maximum, (i.e. 1 minus the minimum utilisation of all of the other components) until we found the server capacity required to make the component schedulable. As the servers all had equal periods, provided all components were schedulable and the total capacity required by all servers was $\leq 100\%$, then the system was deemed schedulable at that specific utilisation level. For every approach, the intra-component CRPD (between tasks in the same component) was calculated using the Combined Multiset approach given by Lunniss *et al.* [14].

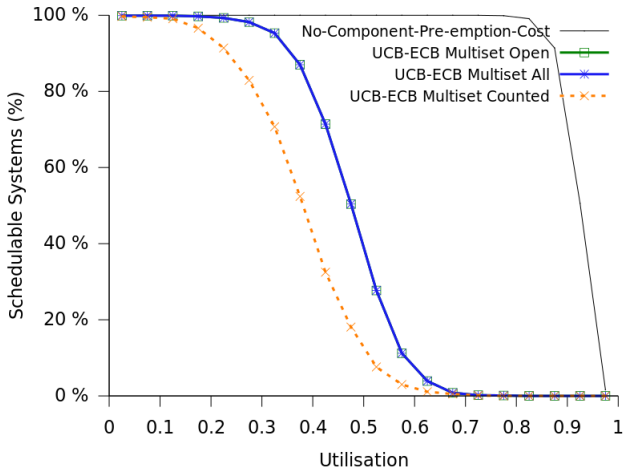


Figure 1. Percentage of systems deemed schedulable

Figure 1 shows that the UCB-ECB-Multiset-All and UCB-ECB-Multiset-Open approaches deem the same number of tasksets schedulable. This is due to the cache utilisation of the other components being greater than the size of the cache, which causes the set of ECBs to be equal, i.e. contain all cache blocks. The UCB-ECB-Multiset-Counted approach deems a lower number of tasksets schedulable because it considers the effects of the other components individually. As the components have equal server periods, each time a component is suspended, it is assumed that each other component will evict it's set of ECBs, when in fact

they may only be evicted once per suspension. We note that the results show that the analysis is somewhat pessimistic, as there is a large difference between the No-Component-Pre-emption-Cost case, and the approaches that consider component pre-emption costs. Examining equation (7), we note that $E^G(D_k)E_k(t)$ is based on the deadline of a task and as such, the analysis effectively assumes the UCBs of all tasks in component G could be in use each time the server for component G is suspended.

The server period is a critical parameter when composing a hierarchical system. The results for varying the server period from 1ms to 20ms, with a fixed range of task periods from 10 to 1000ms are shown in Figure 2 using the weighted schedulability measure [5]. When the component pre-emption costs are ignored, having a small server period ensures that short deadline tasks meet their time constraints. However, switching between components clearly has a cost associated with it making it desirable to switch as infrequently as possible. As the server period increases, schedulability increases due to a smaller number of server context switches, and hence component CRPD, up until around 7-8ms for the best performance. At this point, although the component CRPD continues to decrease, short deadline tasks start to miss their deadlines due to the delay in server capacity being supplied unless server capacities are greatly inflated, and hence the overall schedulability of the system decreases.

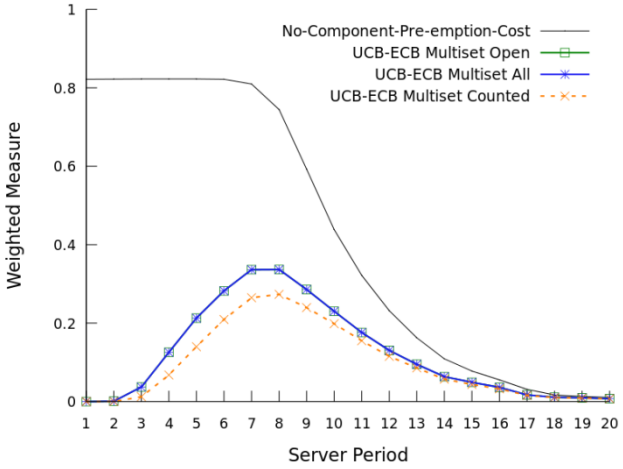


Figure 2. Weighted measure of the schedulability when varying the server period from 1 to 20ms

6. CONCLUSION

In this paper, we have presented some initial analysis for bounding CRPD under hierarchical scheduling when using a local EDF scheduler. This analysis builds on existing work for determining CRPD under single-level EDF scheduling [14], and hierarchical scheduling with a local FP scheduler [13]. We also showed that when taking inter-component CRPD into account, minimising server periods does not maximise schedulability. Instead, the server period must be carefully selected to minimise inter-component CRPD while still ensuring short deadline tasks meet their time constraints. We note that the analysis is somewhat pessimistic due to the use of a tasks' deadline for determining how many times its component could be suspended and resumed during its execution. In future work we would like to investigate ways to resolve this. Furthermore, we believe that the analysis could be optimised when using harmonic server periods, which could lead to an improvement in the UCB-ECB-Multiset-Counted approach. Finally, we would like to extend the analysis for use with a pre-emptive global scheduler.

ACKNOWLEDGEMENTS

This work was partially funded by the UK EPSRC through the Engineering Doctorate Centre in Large-Scale Complex IT Systems (EP/F501374/1), the UK EPSRC funded MCC (EP/K011626/1), the European Community's ARTEMIS Programme and UK Technology Strategy Board, under ARTEMIS grant agreement 295371-2 CRAFTERS, and COST Action IC1202: Timing Analysis On Code-Level (TACLe).

REFERNECES

[1] Altmeyer, S., Davis, R.I., and Maiza, C. Improved Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive Systems. *Real-Time Systems*, 48, 5 (2012), 499-512.

[2] ARINC. *ARINC 653: Avionics Application Software Standard Interface (Draft 15)*. Airlines Electronic Engineering Committee (AEEC), 1996.

[3] Baruah, S. K., Mok, A. K., and Rosier, L. E. Preemptive Scheduling Hard-Real-Time Sporadic Tasks on One Processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS)* (1990), 182-190.

[4] Baruah, S. K., Rosier, L. E., and Howell, R. R. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time Tasks on One Processor. *Real-Time Systems*, 2, 4 (1990), 301-324.

[5] Bastoni, A., Brandenburg, B., and Anderson, J. Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability. In *Proceedings of Operating Systems Platforms for Embedded Real-Time applications (OSPERT)* (2010), 33-44.

[6] Bini, E. and Buttazzo, G. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30, 1 (2005), 129-154.

[7] Deng, Z. and Liu, J. W. S. Scheduling Real-Time Applications in Open Environment. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)* (1997).

[8] Feng, X. and Mok, A. K. A Model of Hierarchical Real-Time Virtual Resources. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)* (2002), 26-35.

[9] Fisher, N. and Dewan, F. A Bandwidth Allocation Scheme for Compositional Real-time Systems with Periodic Resources. *Real-Time Systems*, 48, 3 (2012), 223-263.

[10] Lee, C., Hahn, J., Seo, Y. et al. Analysis of Cache-related Preemption Delay in Fixed-priority Preemptive Scheduling. *IEEE Transactions on Computers*, 47, 6 (June 1998), 700-713.

[11] Lipari, G. and Baruah, S. K. Efficient Scheduling of Real-time Multi-task Applications in Dynamic Systems. In *Proceddings Real-Time Technology and Applications Symposium (RTAS)* (2000), 166-175.

[12] Lipari, G., Carpenter, J., and Baruah, S. A Framework for Achieving Inter-application Isolation in Multiprogrammed, Hard Real-time Environments. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS)* (2000), 217-226.

[13] Lunniss, W., Altmeyer, S., Lipari, G., and Davis, R. I. Accounting for Cache Related pre-emption Delays in Hierarchical Scheduling. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems (RTNS)* (2014).

[14] Lunniss, W., Altmeyer, S., Maiza, C., and Davis, R. I. Intergrating Cache Related Pre-emption Delay Analysis into EDF Scheduling. In *Proceedings 19th IEEE Converence on Real-Time and Embedded Technology and Applications (RTAS)* (2013), 75-84.

[15] Richter, K. *Compositional Scheduling Analysis Using Standard Event Models*. PhD Dissertation, Technical University Carolo-Wilhelmina of Braunschweig, 2005.

[16] Shin, I. and Lee, I. Periodic Resource Model for Compositional Real-Time Guarantees. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)* (2003), 2-13.

Alignment of Memory Transfers of a Time-Predictable Stack Cache

Sahar Abbaspour
Dep. of Applied Math. and Computer Science
Technical University of Denmark
sabb@dtu.dk

Florian Brandner
Computer Science and System Eng. Dep.
ENSTA ParisTech
florian.brandner@ensta-paristech.fr

ABSTRACT

Modern computer architectures use features which often complicate the WCET analysis of real-time software. Alternative time-predictable designs, and in particular caches, thus are gaining more and more interest. A recently proposed stack cache, for instance, avoids the need for the analysis of complex cache states. Instead, only the occupancy level of the cache has to be determined.

The memory transfers generated by the standard stack cache are not generally aligned. These unaligned accesses risk to introduce complexity to the otherwise simple WCET analysis. In this work, we investigate three different approaches to handle the alignment problem in the stack cache: (1) unaligned transfers, (2) alignment through compiler-generated padding, (3) a novel hardware extension ensuring the alignment of all transfers. Simulation results show that our hardware extension offers a good compromise between average-case performance and analysis complexity.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms

Algorithms, Measurement, Performance

Keywords

Block-Aligned Stack Cache, Alignment, Real-Time Systems

1. INTRODUCTION

In order to meet the timing constraints in systems with hard deadlines, the worst-case execution time (WCET) of real-time software needs to be bounded. Many features of modern processor architectures, such as caches and branch predictors, improve the average performance, but have an adverse effect on WCET analysis. Time-predictable computer architectures thus propose alternative designs that are easier to analyze, focusing in particular on analyzable cache and memory designs [13, 8]. One such alternative cache design is the *stack cache* [5, 9], i.e., a cache dedicated to stack data. The stack cache is a complement to a regular data cache and thus reduces the number of accesses through the data cache. This promises improved analysis precision, since unknown access addresses can no longer interfere with stack accesses (and vice versa). Secondly, the stack cache design is simple and thus easy to analyze [5]. The WCET analysis of traditional caches requires precise knowledge about the addresses of accesses [13] and has to take the *complex* replacement policy into account. The analysis of the stack cache on the other hand is much easier and amounts to a simple analysis of the cache's fill level (*occupancy*) [5].

The original stack cache proposes three instructions to reserve space on the stack (`sres`), free space (`sfree`), and to ensure the availability of stack data in the cache (`sens`). The reserve and ensure instructions may cause memory transfers between the stack cache and main memory and thus are relevant for WCET analysis. Since these stack cache control instructions operate on words, the start addresses of memory transfers are not guaranteed to be aligned to the memory controller's burst size. Therefore, unaligned transfers incur a performance penalty that needs to be analyzed. This is in contrast to traditional caches where the cache line size is typically aligned with the burst size. This risks anew to introduce complexity to the otherwise simple WCET analysis of the stack cache. We thus compare three approaches to handle this alignment problem for the stack cache: (1) a stack cache initiating unaligned transfer, (2) compiler-generated padding to align all stack cache allocations (and thus all transfers), and (3) a novel hardware extension that guarantees that all stack cache transfers are block-aligned.

For our hardware extension a burst-sized block of the stack cache is used as an *alignment* buffer, which can be used to perform block-aligned memory transfers. The downside of this approach is that the effective stack cache size is reduced by one block. On the other hand, the block-aligned transfers simplify WCET analysis. In addition, this allows us to perform allocations at word granularity, which improves the cache's utilization. The hardware overhead of our approach is minimal: the implementation of `sres` and `sens` is simplified, while `sfree` requires some minor extensions. The free instructions may need to initiate memory transfers to preserve the alignment of the stack cache content.

Section 2 introduces the stack cache, followed by a discussion of related work. In Section 4, we explain the block-aligned stack cache and its impact on static analysis. We finally present the results from our experiments in Section 5.

2. BACKGROUND

The original stack cache [1] is an on-chip memory implemented as a special ring buffer utilizing two pointers: stack top (`ST`) and memory top (`MT`). The `ST` points to the stack data either stored in the cache or main memory. The `MT` points to the stack data in main memory. The difference

$\texttt{MT} - \texttt{ST}$ represents the occupied space in the stack cache. Since this value cannot exceed the total size of the stack cache ($|SC|$), $0 \leq \texttt{MT} - \texttt{ST} \leq |SC|$ always holds.

Three control instructions manipulate the cache (more details are available elsewhere [1]):

- **sres** x**:** Subtract x from ST. If this violates the equation from above, i.e., the stack cache size is exceeded, a *spill* is initiated, which lowers MT until the invariant is satisfied again.
- **sens** x**:** Ensure that the occupancy is larger than x. If this is not the case, a *fill* is initiated, which increments MT accordingly so that $\texttt{MT} - \texttt{ST} \geq x$.
- **sfree** x**:** Add x to ST. If this results in a violation of the invariant, MT is incremented accordingly. Memory is not accessed.

The analysis of the stack cache [5] is based on the observation that the concrete values of ST and MT are not relevant for the worst-case behavior. Instead the focus is on determining the *occupancy*, i.e., the value $\texttt{MT} - \texttt{ST}$. The impact of function calls is taken to account by the function's *displacement*, i.e. the number of cache blocks spilled to main memory during the function call. The analysis then consists of the following phases: (1) an analysis of the minimum/maximum displacement for call sites on the call graph, (2) a context-independent, function-local data-flow analysis bounding the filling at ensure instructions, (3) a context-independent, function-local data-flow analysis bounding the worst-case occupancy for call sites, and (4) a fully context-sensitive analysis bounding the worst-case spilling of reserves.

The spilling and filling bounds at the stack control instructions can then be taken into account during WCET analysis to compute a final timing bound. Note that the alignment is not considered here and thus handled conservatively.

3. RELATED WORK

Static analysis [12, 3] of caches typically proceeds in two phases: (1) potential addresses of memory accesses are determined, (2) the potential cache content for every program point is computed. The alignment usually is not an issue, as the size can be aligned with the main memory's burst size. Through its simpler analysis model, the stack cache does not require the precise knowledge of addresses, thus eliminating a source of complexity and imprecision. It has been previously shown that the stack cache serves up to 75% of the dynamic memory accesses [1]. An extension to avoid spilling data that is coherent between the stack cache and main memory [9] was presented.

Our approach to compute the worst-case behavior of the stack cache has some similarity to techniques used to statically analyze the maximum stack depth [2]. Also related to the concept of the stack cache, is the register-window mechanism of the SPARC architecture, for which limited WCET analysis support exists in Tidorum Ltd.'s Bound-T tool [11].

Alternative caching mechanisms for program data exist with the Stack Value File [6] and several solutions based on Scratchpad Memory (SPM) (e.g. [7]), which manage the stack in either hardware or software.

4. BLOCK-ALIGNED STACK CACHE

As explained above, the original stack cache operates on words and thus does not automatically align transfers according to the main memory's requirements. With regard to average performance, this is less of an issue and may only lead to a less optimal utilization of the main memory's bandwidth. For the WCET analysis the issue is more problematic. The alignment of the stack cache content needs to be known or otherwise all access have to be assumed unaligned. This information, however, is highly dependent on execution history and thus inevitably increases analysis complexity.

4.1 Hardware Modifications

This work proposes a hardware extension that guarantees that the stack cache initiates aligned memory transfers only, i.e., the start address as well as the length of the memory transfer are multiples of the memory's alignment requirement, i.e. the burst size. This avoids the need to track the alignment of the stack cache content during the WCET analysis, while allowing us to perform all stack cache operations at word granularity, which improves the cache's utilization.

The stack cache is organized in blocks matching the burst size. Moreover, we logically reserve a block in the stack cache as an alignment buffer. Note that this reserved block is not fixed, instead the last block pointed to by MT *dynamically* serves as this alignment buffer. This block is not accessible, for instance, to the compiler, and thus reduces the effective size of the stack cache by one block. The buffer allows us to align all the memory transfers to the desired block size. With regard to the original stack cache (see Section 2), this corresponds to an additional invariant that needs to be respected by the stack cache hardware given a block size BS:

$$\texttt{MT} \mod \texttt{BS} = 0. \tag{1}$$

In order to respect this new invariant the stack control instructions have to be adapted as follows:

- **sres** x**:** Subtract x from ST. If the occupancy exceeds the stack cache size, a *spill* is initiated, which lowers MT by multiples of BS until the occupancy is smaller than $|SC|$.
- **sens** x**:** If the occupancy is not larger than x, a *fill* is initiated, which increments MT by multiples of BS so that $\texttt{MT} - \texttt{ST} \geq x$.
- **sfree** x**:** Add x to ST. If $\texttt{MT} < \texttt{ST}$, set MT to the smallest multiple of BS larger than ST and fill a single block from main memory.

It is easy to see that the modifications to **sres** and **sens** are minimal. Clearly, when Eq. 1 holds, spilling and filling in multiples of BS ensures that the equation also holds after these instructions. The reserved block serving as an alignment buffer, in addition guarantees that sufficient space is available during filling to receive data and sufficient data is available during spilling to transmit data.

The situation is more complex for **sfree**. Whenever a number of **sfree** instructions are executed in a sequence such that the occupancy becomes zero, the MT pointer needs to be updated. In order to satisfy Eq. 1, two options exist: (a) set MT to the largest multiple of BS *smaller* than ST or (b) set MT to the smallest multiple of BS *larger* that ST. The former option would mean that the cache's occupancy becomes negative, which would entail non-trivial modifications to the other stack control instructions. The second option, which represents a non-negative occupancy, thus is preferable. However, in order to guarantee that the content of the stack cache reflects the occupancy ($\texttt{MT} - \texttt{ST}$) a single block has to be filled from main memory.

4.2 Static Analysis

The static analysis proposed for the standard stack cache [5] is in large parts applicable to the new block-aligned stack cache proposed as well. The main difference is that the timing of `sfree` instructions also has to be analyzed.

An `sfree` is required to perform a fill, iff, the minimal occupancy before the instruction is smaller than the instruction's argument x. The analysis problem for free instructions is thus identical to the analysis of ensure instructions [5].

In addition, the displacement of function calls has to be refined to account for data of the caller that is reloaded to the cache by an `sfree` before returning from the call. This is particularly important for the minimum displacement, needed for the analysis of `sens`-instructions, as the original displacement analysis is not safe anymore, unless lazy spilling [9] is used. This information can easily be derived and propagated on the program's call graph.

5. EXPERIMENTS

For our experiments we extended the hardware implementation of the stack cache available with the Patmos processor [10] as well as the cycle-accurate simulation infrastructure and the accompanying LLVM compiler (version 3.4). The average case performance was measured for all benchmarks of the MiBench benchmark suite [4]. The benchmaks were compiled, with optimizations (`-O2`) and stack cache support enabled, and then executed on the Patmos simulator to collect runtime statistics. The simulator was configured to simulate a 2 KB data cache (32 B blocks, 4 way set-associative, LRU replacement policy, and write-through strategy), a 2 KB method cache (32 B blocks, associativity 8), and a 128 B stack cache (with varying configurations). All caches are connected to a shared main memory, which transfers data in 32 B bursts. A moderate access latency of 14 cycles for reads and 12 cycles for writes is assumed.

The benchmarks were tested under three different scenarios: (1) the stack cache performs unaligned memory transfers (`unaligned`), (2) the compiler generates suitable *padding* to align all stack allocations and consequently all memory transfers (`padding`), and (3) the stack cache employs the block-aligned strategy from Section 4 (`block-aligned`). Stack data is usually aligned at word boundaries for Patmos, which applies to the `unaligned` and `block-aligned` configurations. The `padding` configuration, however, aligns all data with the burst size (32 B).

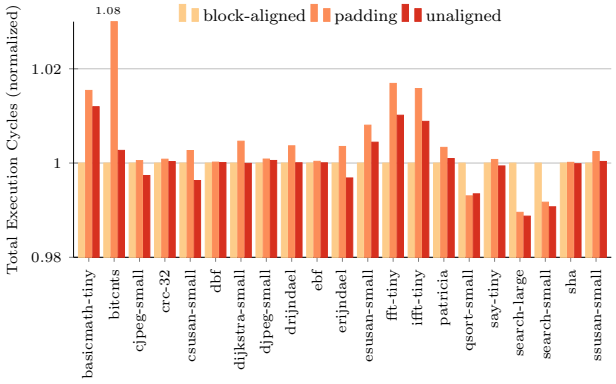


Figure 1: Total execution cycles normalized to the `block-aligned` configuration (lower is better).

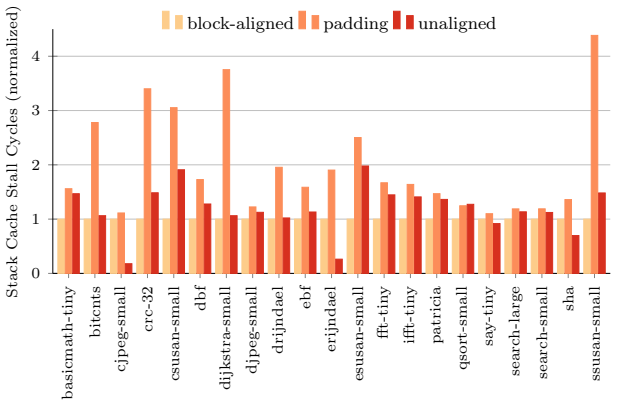


Figure 2: Total number of stall cycles induced by the stack cache normalized to the `block-aligned` configuration (lower is better).

The runtime impact of the various strategies to handle the alignment of memory transfers between the stack cache and the main memory is summarized in Figure 1. Overall, the `unaligned` and `block-aligned` configurations are very close with respect to runtime, while the `padding` configuration performs the least. In particular, the `bitcnts`, `basicmath-tiny`, `fft-tiny`, and `ifft-tiny` benchmarks here show runtime increases of 1.5% and more.

Note that the runtime contribution of the stack cache is relatively small, which in general precludes very large variations in the total runtime due to the stack cache. The simulator thus was extended to collect detailed statistics on the number of stall cycles induced by the stack cache as well as the spilling and filling performed. Figure 2 shows the total number of stall cycles induced by the stack cache, normalized to the `block-aligned` configuration. The `padding` configuration increases the number of stall cycles in all cases in relation to our `block-aligned` strategy (up to a factor of more than 4). The padding introduced by the compiler generally increases the stack cache's occupancy and consequently leads to additional memory transfers. Also for the `unaligned` configuration the number of stall cycles is larger than our new strategy, since the small unaligned memory transfers performed by this configuration induce some overhead. For two benchmarks, `cjpeg-small` and `erijndael`, the number of stall cycles is considerably smaller in this configuration. Our `block-aligned` stack cache here suffers additional filling and spilling due to its reduced effective size, as shown in the following Table.

The impact of the various configurations on the amount of data spilled and filled from/to the stack cache is shown in Table 1. As noted above the `padding` configuration performs additional memory transfers (spills and fills) due to the padding introduced by the compiler to ensure alignment. The `unaligned` configuration on the other hand requires the least filling and spilling as it transfers the precise amount of data needed. In addition, the reduced stack cache size available for the `block-aligned` strategy (recall that one block is reserved as an alignment buffer) plays in favor of the `unaligned` configuration.

To summarize, compiler generated padding is a simple solution to the alignment problem for the stack cache, which is easy to analyze and generally performs reasonably well,

	Block-Aligned		Padding				Unaligned			
Benchmark	Spill	Fill	Spill	rel.	Fill	rel.	Spill	rel.	Fill	rel.
basicmath-tiny	791 288	968 544	1 696 832	2.14	1 981 192	2.05	1 026 895	1.30	1 165 108	1.20
bitcnts	1 201 992	1 202 720	3 603 152	3.00	3 604 328	3.00	826 933	0.69	827 437	0.69
cjpeg-small	96 840	123 048	113 512	1.17	155 672	1.27	12 821	0.13	24 500	0.20
crc-32	2 984	3 312	11 256	3.77	38 688	11.68	2 961	0.99	3 114	0.94
csusan-small	9 184	10 000	32 136	3.50	35 936	3.59	13 403	1.46	13 619	1.36
dbf	11 984	78 136	22 456	1.87	124 672	1.60	10 395	0.87	66 056	0.85
dijkstra-small	98 056	99 520	472 944	4.82	478 152	4.80	49 047	0.50	50 518	0.51
djpeg-small	28 032	29 112	33 688	1.20	36 336	1.25	14 688	0.52	15 369	0.53
drijndael	168 584	212 800	329 600	1.96	373 952	1.76	29 693	0.18	49 823	0.23
ebf	37 536	103 624	65 040	1.73	243 872	2.35	30 618	0.82	114 959	1.11
erijndael	196 240	240 520	366 280	1.87	410 696	1.71	34 258	0.17	54 502	0.23
esusan-small	18 952	19 912	57 896	3.05	61 912	3.11	30 528	1.61	30 852	1.55
fft-tiny	215 936	217 712	448 480	2.08	470 464	2.16	277 788	1.29	279 011	1.28
ifft-tiny	206 464	207 952	425 440	2.06	446 864	2.15	264 063	1.28	264 994	1.27
patricia	3 883 936	4 590 920	6 638 160	1.71	8 089 800	1.76	3 740 659	0.96	4 514 761	0.98
qsort-small	643 296	1 283 680	1 126 664	1.75	2 411 584	1.88	772 978	1.20	1 493 122	1.16
say-tiny	261 016	358 976	303 296	1.16	460 560	1.28	98 239	0.38	118 011	0.33
search-large	216 632	322 344	323 680	1.49	534 824	1.66	207 978	0.96	366 106	1.14
search-small	8 528	14 816	12 688	1.49	24 984	1.69	7 963	0.93	16 214	1.09
sha	5 448	30 248	8 656	1.59	33 608	1.11	1 935	0.36	2 007	0.07
ssusan-small	18 280	19 096	66 736	3.65	71 336	3.74	16 247	0.89	16 463	0.86

Table 1: Words spilled and filled by the stack cache configurations `block-aligned`, `padding`, and `unaligned` (lower is better, rel. indicates the normalized value in comparison to the `block-aligned` configuration).

but may suffer from bad outliers. It generally leads to increased spilling and filling as well as a reduced utilization of the stack cache. Generating unaligned memory transfers naturally performs well for the average case. but, complicates WCET analysis since the alignment of the stack data is highly context dependent. The new solution proposed in this work, the block-aligned stack cache, offers a reasonable trade-off, which combines moderate hardware overhead with good average-case performance and simple WCET analysis.

Acknowledgment

This work was partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST).

6. REFERENCES

[1] S. Abbaspour, F. Brandner, and M. Schoeberl. A time-predictable stack cache. In *Proc. of the Workshop on Software Technologies for Embedded and Ubiquitous Systems*. 2013.

[2] C. Ferdinand, R. Heckmann, and B. Franzen. Static memory and timing analysis of embedded systems code. In *Proc. of Symposium on Verification and Validation of Software Systems*, pages 153–163. Eindhoven Univ. of Techn., 2007.

[3] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[4] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of the Workshop on Workload Characterization*, WWC '01, 2001.

[5] A. Jordan, F. Brandner, and M. Schoeberl. Static analysis of worst-case stack cache behavior. In *Proc. of the Conf. on Real-Time Networks and Systems*, pages 55–64. ACM, 2013.

[6] H.-H. S. Lee, M. Smelyanskiy, G. S. Tyson, and C. J. Newburn. Stack value file: Custom microarchitecture for the stack. In *Proc. of the International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 5–14. IEEE, 2001.

[7] S. Park, H. woo Park, and S. Ha. A novel technique to use scratch-pad memory for stack management. In *In Proc. of the Design, Automation Test in Europe Conference*, DATE '07, pages 1–6. ACM, 2007.

[8] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Proc. of the Conference on Hardware/Software Codesign and System Synthesis*, pages 99–108. ACM, 2011.

[9] S.Abbaspour, A. Jordan, and F. Brandner. Lazy spilling for a time-predictable stack cache: Implementation and analysis. In *Proc. of the International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OASICS*, pages 83–92. Schloss Dagstuhl, 2014.

[10] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. Probst, S. Karlsson, and T. Thorn. *Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach*, volume 18, pages 11–21. OASICS, 2011.

[11] BoundT time and stack analyzer - application note SPARC/ERC32 V7, V8, V8E. Technical Report TR-AN-SPARC-001, Version 7, Tidorum Ltd., 2010.

[12] R. T. White, C. A. Healy, D. B. Whalley, F. Mueller, and M. G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the Real-Time Technology and Applications Symposium*, RTAS '97, pages 192–203, 1997.

[13] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(7):966–978, 2009.

The WCET Analysis using Counters - A Preliminary Assessment

Remy Boutonnet
Verimag, UJF
Grenoble, France
remy.boutonnet@imag.fr

Mihail Asavoae
Verimag, UJF
Grenoble, France
mihail.asavoae@imag.fr

ABSTRACT

The Worst-Case Execution Time (WCET) analysis aims to statically and safely bound the longest execution path of a program. It is also desirable for the computed bound to be as close as possible to the actual WCET, thus making the result of a WCET analysis tight. In this paper we propose a methodology for the WCET analysis using an abstract program (with special program variables called counters), in order to better exploit the underlying program semantics (via abstract interpretation) and to produce potentially tighter WCET bounds.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

WCET Analysis, abstract interpretation, semantic analysis

1. INTRODUCTION

The interaction of hard real-time systems with their external environment is governed by a set of timing constraints. In order to solve these constraints, it is necessary to estimate the worst-case execution time (WCET) of the system components. A WCET analysis computes for a given task or program, a WCET bound which should be safe and tight (close to the actual WCET).

The WCET analysis is performed at the binary level, with knowledge about the underlying architecture. First, the typical WCET analysis workflow as standardized in [18], extracts the control-flow graph (CFG) from the binary code. Subsequently, this CFG is the working structure for both flow- and architecture-related analyses. Their result, which is an annotated CFG, is used to compute the WCET bound in a final phase, called path analysis. In order to achieve tight bounds, a WCET analysis relies on a number of specific analyses, from flow analysis (e.g. detection of loop bounds and infeasible paths) to architecture analysis (e.g. of caches).

In this work, we address the flow analysis from the following angle: how to extract more accurate semantic properties which can help the WCET analysis by removing some infeasible paths. We denote an analysis which extracts semantic properties, invariants on program executions, as a *semantic analysis*. More specifically, in our context a semantic analysis targets invariants which are directly translated into integer linear programming (ILP) constraints, as emphasised by the implicit path enumeration technique (IPET) [17]. Our work uses a program analyzer called Pagai [11], at the LLVM Intermediate Representation level, over the LLVM compiler infrastructure [15]. The key element of our approach is the extraction of invariants from an abstract representation of the input program, as an instrumented code with *counters*. A counter is a special program variable which is attached to a program part (e.g. a basic block) and incremented every time the control flows through that part. We propose the following workflow: after an automated instrumentation of LLVM-IR code, the semantic analysis is performed using Pagai (with a linear arithmetic abstract domain [6]). The invariants, as relations between counters, are transferred to binary code (actually to the path analysis formulation of Otawa [4]), using a block-level traceability tool. We measure the improvements on the WCET bounds on a set of syntactic and standard benchmarks.

```
x = 0; i = 0; s = 0;          // α = 0; β = 0; γ = 0;
while (i < N) {               // α++;
    if(x < 10){
        s += 3;               // β++;
    }
    if(s < N){
        s += 2; x ++;         // γ++;
    }
    i ++;
}
```

Figure 1: Our example program instrumented with counters α, β, γ

We use the program described in Figure 1 to advocate on how the WCET analysis can benefit from a counter-based approach. The counters are α, β, γ, and N is a loop bound.

The following relations can be derived on those counters, which are satisfied at the end of the program:

$$\beta + \gamma \leq \alpha + 10 \quad (1)$$

It shows that the blocks 4 and 6 of the control flow graph of this program, in Figure 2, are both executed in the same iteration of the while loop at most 10 times. Therefore, this information is interesting for us since it leads to refinements in the WCET of the whole program.

Outline. In Section 2 we overview the existing methods on extraction of semantic properties for the WCET analysis.

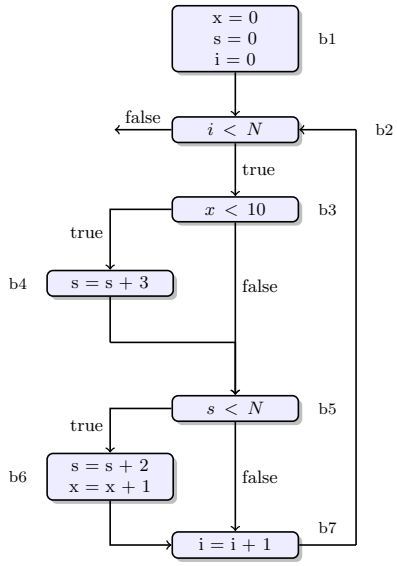


Figure 2: The control flow graph corresponding to this example.

In Section 3 we elaborate on how our system is designed, implemented and experimented with. We draw conclusions and discuss about directions of future work in Section 4.

2. RELATED WORK

The WCET analysis requires knowledge about loop bounds and infeasible paths in order to compute tight WCET bounds. The existing support for automatic extraction of semantic properties like the aforementioned ones is usually coming in two flavours: as a result of abstract interpretation or of symbolic execution. More recent approaches also use state of the art satisfiability modulo theory (SMT) solving to address parts of the WCET analysis.

Abstract interpretation, introduced in [5] is one of the major program reasoning techniques. It relies on abstract domains, like linear arithmetic [6] and fixpoint computation to generate invariants at the program points of interest. In the context of WCET analysis, abstract interpretation plays a key role [19] in both control-flow (e.g. in value analysis) or processor behaviour analyses (e.g. in cache analyses).

Symbolic execution [13] is a technique which uses arbitrary values as program inputs and allows program reasoning at the level of execution paths. Symbolic execution has drawn interest from the WCET analysis community [9, 14], but while it is potentially very precise, it suffers from scalability issues when it is applied on large programs. As a consequence, symbolic execution in WCET analysis is coupled with search space reduction [9] techniques or it is applied on code fragments [14].

Recent works [14, 10] rely on SMT-solving techniques to compute WCET bounds. For example, WCET squeezing [14] employs a form of symbolic execution on paths returned by an external WCET analyzer. This technique embeds the path analysis into a CEGAR loop, allowing an incremental strengthening of the WCET bound. The approach in [10] computes WCET bounds as solutions of optimisation modulo theory problems (i.e. extensions of the SMT to maximisation problem). The program semantics are encoded as an SMT formula. To maintain the scalability of the analysis, the original SMT formula is augmented with additional constraints called "cuts", which express summaries of portions of code.

Using counters to extract semantic properties is not new, existing counter-based approaches have been proposed, for example in [8] using a single counter and in [12] with multiple counters. The former considers one counter which represents time and accumulates the program semantics into it. The later overcome the issue with the single-counter annotations to work with complex invariant generation tools, CFG graph transformation and generation of progress invariants, it computes loop bounds and infeasiblilty relations.

The existing WCET analyzers span from industrial strength platforms, like aiT [1] to academic tools like Otawa [4], SWEET [2] and Chronos [16]. The oRange tool [7] complements the Otawa timing analyzer by computing loop bounds using static analysis with abstract interpretation [5, 3] on C programs. The SWEET tool supports an implementation of the abstract execution over the domain of intervals. The Chronos timing analyzer integrates a pattern-based semantic analysis which keeps track, for a particular branch, of which assignments or other branches may influence it. The industrial timing analyzer aiT uses a similar pattern-driven analysis to identify code portions (e.g. loop or branch shapes) and apply the appropriate analysis. While we conducted limited experiments on SWEET, Chronos and aiT, these tools seem capable to detect certain types of bounds and infeasibility relations: for SWEET - up to the strengths of the abstract domain and for Chronos and aiT - up to the code structure which exhibits certain syntactic patterns. However, using a specialised static analyzer to compute semantic properties and to transfer these properties seems to offer power (through various abstract domains) as well as flexibility (i.e. driven by the strengths of the static analyzer).

3. SYSTEM DESIGN AND IMPLEMENTATION

3.1 General System

In the most general context, a counter-based methodology for WCET analysis is driven by two elements: a compilation toolchain (which also fixes the input language) and a WCET analyzer (which includes the necessary processor behaviour analyses). Next in Figure 3 we describe an implementation of this methodology over the LLVM infrastructure and the Otawa timing analyzer.

The standard WCET analysis workflow, in Figure 3 (left) computes the WCET bound of the binary code (in our case it is ARM code) generated from LLVM-IR code. The Otawa timing analyzer relies on an ILP formulation of the path analysis and embeds an ILP solver to compute the result (i.e. the WCET bound). Our counter-based analysis workflow, in Figure 3 (right) could be seen as a plugin for the WCET analysis. For the purpose of semantics extraction, the Pagai static analyzer uses the initial LLVM-IR code, instrumented with counters. Pagai computes invariants on the counters, at the LLVM-IR level, using either abstract interpretation or its combination with SMT solving. The invariants are directly translated into ILP constraints. Fi-

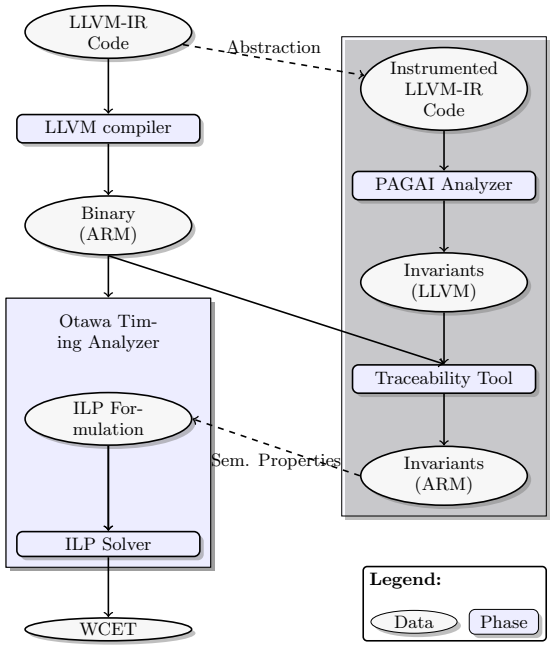


Figure 3: General workflow for Counter-based WCET Analysis

nally, a traceability tool maps the LLVM-IR blocks to ARM basic blocks and facilitates the transfer of Pagai invariants on counters at the binary level. These properties are integrated into the path analysis of Otawa and solved in order to obtain the WCET bound. The WCET analysis is performed over the initial program, the instrumented code is used only to extract invariants w.r.t. program semantics. Moreover, our workflow does not consider code optimisations between the LLVM-IR and ARM levels, however the code could and should be optimized from C (i.e. not represented in the workflow) and LLVM-IR levels. Our counter-based methodology works when replacing Otawa with the aiT, SWEET or Chronos timing analyzers (all using IPET), up to some supported architectures. The counters approach can be used in difference scenarios: to find or refine loop bounds or infeasible paths, particularly those created by mutually exclusive conditions.

By processing the example of Figure 1 through the PAGAI static analyzer, we automatically obtain the following constraint between the counters α, β and γ:

$$10 + \alpha - \beta - \gamma \quad \geq \quad 0 \tag{2}$$

as well as $-\alpha + 3\beta + 2\gamma \geq 0$, $\alpha - \beta \geq 0$ and $\alpha - \gamma \geq 0$.

This relation shows what we have derived by hand for our example: the then-parts of the two conditions in the while loop are both executed in the same iteration at most ten times. Therefore, the counters approach with a static analyzer is able to automatically find a non-trivial case of infeasible path: the path containing the blocks b4, b5, b6, b7 is executed at most ten times.

We can also use the counters approach to find or refine a loop bound where other tools like oRange cannot find one. In our example in Figure 1, if the condition of the while loop $i < N$ is replaced by $x < 10$ and the second condition $s < N$ by $s < 150$, the PAGAI static analyzer outputs the relation $-10 + \alpha = 10$ which enables us to show that the while loop is executed at most 10 times in that case.

3.2 Experiments

Our set of benchmark programs covers a wide range of applications (though of small size - column **LOC** in Figure 4). We include automatically generated code from high-level designs - (e.g. a snapshot called `selector` and avionics-specific controllers in `roll-control` and `cruise-control`) as well as several syntactic programs with complicated infeasible paths (e.g. `sou`, `even`, `break`, and `rate_limiter`). In order to extract semantic properties using the Pagai static analyzer, we automatically instrument the LLVM IR code (the working level for Pagai) with a number of counter variables (in column **#Cntrs**) and a set of invariants (in column **#Inv**) which are fed into the ILP representation of path analysis. In this paper, we use the processor behavior analysis as provided by Otawa, our main concern being the program-level semantic analysis.

The set of invariants covers relations between basic blocks (represented by their respective counter variable) of several forms. First, there are the loop bounds types of relations, like for the benchmarks `break` and `selector`. Second, the path infeasibility relations are expressed either as invariants on two counter variables (in the case of pairwise exclusive branches) - for the benchmark programs `sou` and `even` - or as a counter value which is equal to zero (i.e. the paths going through the particular basic block are infeasible) - for the benchmark `rate_limiter`. Moreover, for some benchmarks like `selector`, the infeasibility relations are more expressive as an invariant on three counter variables than the pairwise relations. Third, the set of invariants also includes relations which do not contribute to a reduction of the WCET bound. Overall, the experiments show promising results because, in general it is difficult to obtain improvements of more than several percents (indeed, the code size is rather small).

4. CONCLUSIONS

In this paper we addressed the problem on how to tune the WCET analysis so as to produce tighter WCET bounds. As such, we proposed a methodology to extract semantic information via special program variables called counters. We used a program analyzer, called Pagai, to compute flow relations (as relations between these counters). Finally, we transferred these relations into an IPET formulation of the path analysis and observed encouraging results with improvements over 20% on certain benchmarks. The methodology is still under development as we would like to investigate how our counter-based WCET analysis compares with and could complement existing WCET analyzers using automated extraction of semantic properties.

5. ACKNOWLEDGMENTS

The authors thank Fabienne Carrier and Claire Maiza for their valuable comments on the paper content and Julien Henry for his help with the Pagai static analyzer. This work was partially funded by grant W-SEPT (ANR-12-INSE-0001) from the French *Agence national de la recherche*.

6. REFERENCES

[1] AbsInt Angewandte Informatik: aiT Worst-Case Execution Time Analyzers.

Name	Program Description	LOC	#Cntrs	#Inv	WCET init	WCET final	Red %
`selector`	Fragment of SCADE design	134	14	14	1112	528	52.6%
`roll − control`	From the SCADE Suite	234	25	19	501	501	0%
`cruise − control`	From the SCADE Suite	234	35	31	881	852	3.3%
`sou`	Syntactic benchmark 1	69	3	3	99	67	47.8%
`even`	Syntactic benchmark 2	82	9	8	2807	2210	21.3%
`break`	Syntactic benchmark 3	114	4	5	820	820	0%
`rate_limiter`	Program from [10]	35	2	2	43	29	32.6%

Figure 4: Set of benchmarks for the counter-based WCET analysis

[2] http://www.mrtc.mdh.se/projects/wcet/sweet/index.html.

[3] Z. Ammarguellat and W. L. H. III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *PLDI*, pages 283–295, 1990.

[4] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *SEUS*, 2010.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[6] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.

[7] M. de Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static loop bound analysis of c programs based on flow analysis and abstract interpretation. In *RTCSA*, 2008.

[8] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.

[9] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *RTSS*, 2006.

[10] J. Henry, M. Asavoae, D. Monniaux, and C. Maiza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In *LCTES*, pages 43–52, 2014.

[11] J. Henry, D. Monniaux, and M. Moy. Pagai: A path sensitive static analyser. *Electr. Notes Theor. Comput. Sci.*, 289:15–25, 2012.

[12] N. Holsti. Computing time as a program variable: a way around infeasible paths. In *WCET*, 2008.

[13] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[14] J. Knoop, L. Kovács, and J. Zwirchmayr. Wcet squeezing: on-demand feasibility refinement for proven precise wcet-bounds. In *RTNS*, pages 161–170, 2013.

[15] C. Lattner and V. S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[16] X. Li, L. Yun, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Sci. Comput. Program.*, 69(1-3):56–67, 2007.

[17] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–307, 1995.

[18] R. Wilhelm and all. The worst-case execution-time problem—overview of methods and survey of tools. *ACM TECS*, 7(3):1–53, 2008.

[19] R. Wilhelm and B. Wachter. Abstract interpretation with applications to timing validation. In *CAV*, pages 22–36, 2008.

Adaptation of RUN to Mixed-Criticality Systems

Romain GRATIA
Technological Research Institute SystemX
romain.gratia@irt-systemx.fr

Thomas ROBERT
Institut Mines-Telecom
thomas.robert@telecom-paristech.fr

Laurent PAUTET
Institut Mines-Telecom
laurent.pautet@telecom-paristech.fr

ABSTRACT

Mixed-criticality scheduling provides a method to better allocate CPU resources between tasks, whose criticalities, e.g. importance, are not the same for the well functioning of the overall system. Considering that tasks have different CPU consumption modes, the core idea is to accept that when tasks enter higher CPU consumption modes then lesser criticality tasks may no longer be scheduled. Hence, it allows deploying more tasks on fewer processors accepting that in the worst mode only most critical tasks will respect their deadlines.

This paper presents a method to adapt RUN, an optimal scheduling algorithm for multicore processor, to cope with mixed-criticality task sets. The RUN approach deeply relies on a data structure called reduction tree computed off-line to elect tasks [6]. We adapt this tree to define a scheduler compatible with mixed-criticality requirements. We briefly present how these modifications impact the on-line part of the original RUN algorithm. Even if optimality is lost, preliminary experiments are encouraging, even with task sets identified as difficult for our proposal.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics

General Terms

Theory

Keywords

Mixed-Criticality, Multiprocessor, Global Scheduling

1. INTRODUCTION

Automotive industry is currently undergoing a major evolution, as the number of embedded software applications in cars is skyrocketing. This is due to the multiplication of Advanced Driver Assistance Systems (ADASs) and telecommunication capabilities. With current architecture, integrating these new functionalities will become more and more complex as each one of them would require the integration of its own execution platform. Therefore, a new architecture has to be found, so that this integration becomes practicable and tractable. The current trend leads to the use of multicore platforms, as uni-processors are less and less produced by hardware manufacturers. These platforms are associated with virtualization layers for the re-usability of previously developed applications. Such a context offers the opportunity to consider new scheduling problems. Indeed, this architecture will result in the execution of hard real-time and multimedia applications on the same platform while abiding by stringent safety standards.

Scheduling theory has become mature on problems related to multicore scheduling. Partitioned scheduling has often been preferred to global scheduling as its low utilization bound is counterbalanced by the high overheads, due to migrations or preemptions, induced by most global scheduling algorithms. However, recent papers have proposed very interesting global algorithms which, apart from being optimal, also limit the number of preemptions and of migrations. One of those algorithms is the algorithm RUN [6].

The completion of critical tasks is absolutely required for the safety of a system. To ensure this, safety requirements tend to impose highly conservative hypotheses when deriving their WCETs. The use of these hypotheses leads to very pessimistic WCETs, which results in under-utilization of hardware platforms in average case. Mixed-Criticality scheduling aims at reducing this under-utilization as much as possible. This field of research is pretty active [2], yet multicore scheduling remains an issue. We propose a method to adapt RUN algorithm to Mixed-Criticality systems (further denoted MC).

The rest of the paper is organized as follows: section 2 gives a brief overview of how the scheduling algorithm RUN works and recalls MC objectives. Section 3 briefly presents our method. Finally, section 4 discusses preliminary results.

2. PRINCIPLES OF RUN AND MIXED CRITICALITY

This section presents a brief overview of RUN algorithm, and some background on MC knowledge.

2.1 RUN scheduling

RUN is a global scheduling algorithm that is optimal. It means that whenever a task set is schedulable by any method on a given set of processors, then RUN proposes a schedule relying on less or the same number of processors. Besides that, RUN is known to entail few context switches and migrations which are usually the curse of global scheduling algorithms [1].

We consider a set of tasks denoted $\mathcal{T}$ complying with the Periodic Task model with implicit deadlines, similar to [5]. $\mathcal{T}$ contains tasks $\tau_1, ..., \tau_n$ with the following parameters defined for each task τ_i:

- A period P_i which is the time between each activation of the task. "Implicit deadlines" means that a task must have completed its execution before its next activation.
- An execution time budget C_i, corresponding to the Worst Case Execution Time of the task.

Tasks are first activated synchronously when the system starts. $U(\tau_i)$ is the utilization of a task τ_i:

$$U(\tau_i) \triangleq \frac{C_i}{P_i}$$

This definition can be extended to task sets $\mathcal{T}$ considering $U(\mathcal{T}) = \sum_{\tau \in \mathcal{T}} U(\tau)$. RUN ensures that the number of processors required for scheduling a task set $\mathcal{T}$ is the smallest integer greater than $U(\mathcal{T})$, denoted $\lceil U(\mathcal{T}) \rceil$.

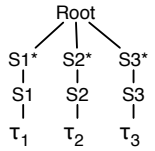
RUN scheduler relies on a hierarchy of scheduling servers to decide which tasks should actually be running. This hierarchy is organized as a tree, called "Reduction" tree, and is computed off-line. The root of this tree is a server and its leaves are servers comprising only one task of $\mathcal{T}$. Each server S is characterized by activation times and a list of servers to schedule, referred as "children", and a constant rate $r(S)$. Activation times of a server is the union of activation times of its children (and thus may not be simply periodic). On each time interval I between two activations of a server S such that the duration of I is D_i, server S time budget is defined as $r(S) * D_i$. Yet, there are two kinds of servers, each with a specific behavior.

The first kind of servers ensures that if the server is executed then one of its children is selected for execution until the server depletes his or its executed children budget, or is preempted. This server is called primal server as it corresponds to usual definition of scheduling servers. Primal server rate is defined as the sum of its children rates (or utilization if its children are tasks). For this reason, we use for both servers and tasks U() to either denotes server rates or task utilizations. The root of the tree is a primal server. In the remainder, EDF is used to schedule primal server children considering activation time as deadlines.

The other kind of server has a single child and is seen as the dual of its child as the child is executed only when its parent is not (detailed mathematical justifications are detailed in the seminal paper). A dual server utilization equals 1 minus its child utilization (always positive if its child utilization is lower than 1). The dual server of a server S is noted S^*.

During execution, the scheduler is called each time a task or a server is activated, a task completes its execution, or a server exhausts its time budget. The root server is executed as long as its budget is not depleted. All scheduling decisions made by servers end up by the correct scheduling of tasks thanks to the way the Reduction tree is built.

RUN proposes a method for building the Reduction tree, and typing servers so the scheduler is optimal, i.e. schedules $\mathcal{T}$ with $\lceil U(\mathcal{T}) \rceil$ processors. More formally, two operations are defined: *pack* and *dual*. They are applied on sets of servers and produce sets of servers Γ. The result of the first function, pack(Γ), is a set of primal servers that enforces three properties. All servers in Γ are the child of exactly one element of $pack(\Gamma)$. Any server in $pack(\Gamma)$ has an utilization lower than 1. For any two servers in $pack(\Gamma)$ their total utilization is greater than 1. The second function creates for each server S in Γ a dual server S^*, such that S^* schedule S with same activation times, and $U(S^*) = 1 - U(S)$.



Name	Period(s)	U(-)
Root	2,6,12	1
$S1^*$	2	0.23
$S2^*$	6	0.29
$S3^*$	12	0.48
$S1 \equiv \tau_1$	2	0.77
$S2 \equiv \tau_2$	6	0.71
$S3 \equiv \tau_3$	12	0.52

Figure 1: Example of a Reduction tree

A Reduction tree is computed applying the composition of dual and pack, $dual \circ pack$, on a set of servers until the result is a singleton. The initial set of servers is denoted as $pack(\mathcal{T})$. The server in the resulting singleton will be the root server. An example of such tree is detailed in figure 1. This task set will be reused and extended to illustrate mixed criticality concepts and our approach. Due to lack of space, we only present this simple Reduction tree with each task having its own server. It also requires only one application of the composition of dual and pack operations.

Next subsection introduces some definitions on Mixed-Criticality systems.

2.2 Background in Mixed-Criticality Concepts

We presented and motivated the use of MC systems in the introduction, and we now give an overview of its theory. MC scheduling algorithms often use several modes of operation and by changing mode when necessary. One mode in which critical tasks consume few CPU resources and another one in which they are considered to consume their pessimistic and safe WCETs. A mode change with respect to a scheduler consists in a modification of task attributes while they are scheduled. It consists in leaving a mode in which critical tasks consume few CPU resources to another one in which they are considered consuming their pessimistic and safe WCETs. The gain of such an approach lies in the fact that the first mode would be more likely, and allows scheduling additional non critical tasks, e.g. tasks that can be stopped without causing harm, when the second mode is detected.

First formalization of this principle has been proposed as an extension of a Periodic Task model [8]. Each task τ_i is extended as follows:

- Each task has a criticality level χ_i in the ordered set $\{Low, Hi\}$ such that $Low < Hi$.
- Each timing attribute (P_i, C_i, and $U(\tau_i)$) becomes a function depending on the criticality level: any attribute A turns in $A(L)$, L being a criticality level.

Usually only WCETs, and consequently utilizations, vary according to the criticality that is why we limit ourselves to this case, leading to constant period attributes. Hence, C_i is a function that always verifies $C_i(Low) \leq C_i(Hi)$ and $C_i(Hi) = C_i(\chi_i)$. Task set presented in figure 1 can be extended by two tasks of Low criticality, considering τ_1, τ_2, τ_3 from figure 2 of criticality Hi.

At run time, the scheduler that provides a correct schedule for $\mathcal{T}$ has to cope with the following constraints:

	Period	χ_i	$U(\tau_i)(Low)$	$U(\tau_i)(Hi)$
τ_1	2	Hi	0.53	0.77
τ_2	6	Hi	0.05	0.71
τ_3	12	Hi	0.32	0.52
τ_4	4	Low	0.54	0.54
τ_5	12	Low	0.56	0.56

Figure 2: Example of a MC task set

- As long as each task τ_i execution time is smaller than $C_i(Low)$ then all tasks should meet their deadlines whatever their criticality level is.
- As soon as one task execution time exceeds its Low criticality time budget, then only deadlines of Hi criticality tasks should be ensured from this instant.

The time at which one task exceeds its low time budget is called a timing failure event and triggers a mode change. Hence, even an activated task may have its attributes changed altering the scheduler objectives. It is a particular case of dynamic scheduling where scheduling objectives and task attributes are piece-wise constant along time. Given this setting, MC scheduler behavior is often described through two operational modes: before and after the timing failure, said Low and Hi modes. However, mode changes must be carefully done.

Indeed, if a timing failure event occurs when the system is in Low mode, a mode change is triggered. Hi tasks can now use their Hi WCETs. And they should be able to use it before their deadlines. Thus the mode change should be performed while Hi tasks have still enough time to execute up their Hi WCETs diminished by its potential execution time done while in Low mode. Therefore, the scheduling performed in the Low and Hi modes are not independent. Next section details how we developed this idea, creating RUN schedules for Hi and Low modes compatibles with this constraint.

3. ADAPTING RUN

This section details how RUN is adapted to support modes, and how to use them to implement MC. Finally preliminary results are discussed.

Designing a MC scheduler is easier to handle if each time the timing failure occurs all Hi tasks have either completed their execution or have not even started to be executed. A solution to ensure this property is to transform tasks as it has been done in [7]. Each Hi task $\tau_i \in \mathcal{T}$ is decomposed in two tasks τ_i^s and τ_i^f, called *start* and *finish* tasks respectively. Attributes of these tasks are defined as $P_i^s = P_i^f = P_i$, $C_i^s(Hi) = C_i^s(Low) = C_i(Low)$, and $C_i^f(Hi) = C_i(Hi) - C_i(Low)$ and $C_i^f(Low) = 0$. Low criticality tasks remain unchanged. This task set will be referred as $split(\mathcal{T})$. As long as start and finish tasks of same index are not scheduled simultaneously and τ_i^s is executed before τ_i^f, scheduling this task set is equivalent to schedule $\mathcal{T}$.

We propose to compute a Reduction tree, denoted $HiLowTree$, to schedule start and finish tasks with Hi attributes respecting the exclusion condition. To do so, we first compute a Reduction tree of Hi tasks of $\mathcal{T}$ denoted $HiOnlyTree$. Then we replace each task τ_i by its corresponding pair of start and finish tasks in $HiOnlyTree$. These tasks pairs will be scheduled by the same primal server and thus cannot be simultaneously executed. Besides, when replacing the original Hi task, the start task is executed first. Thus, we respect the two conditions previously exposed.

Then, we enhance the RUN approach by introducing a new kind of server: modal server. Such a server is used as a replacement for finish tasks in the Reduction tree $HiLowTree$. Its activation time and rate equals the finish task Hi attributes. Besides, a modal server has two sets of tasks to schedule depending of the mode in which the system is. In Hi mode, it schedules the finish task it has replaced. But, in Low mode, a modal server schedules a set of Low tasks whose utilisations and periods respect some conditions.

With our method, a MC system is designed by following these steps off-line :

1. Replace each finish task in the Reduction tree $HiLowTree$ by its corresponding modal server with an empty set of Low tasks, noted MS_i.
2. Determine subsets LT_i of Low tasks of $split(\mathcal{T})$ that MS_i would be able to schedule LT_i with its budget.
3. Build another Reduction tree, denoted $LowOnlyTree$, for Low tasks not in $\bigcup_{i|\chi_i=Hi} LT_i$.

Then the scheduling is executed on-line as follows:

1. At run-time in Low mode, schedule $LowOnlyTree$ according to original RUN rules, and $HiLowTree$ with modal servers in Low mode. Both trees are configured on a distinct set of processors.
2. At run-time in Hi mode, stop executing $LowOnlyTree$ and execute $HiLowTree$ with its modal servers in High mode.

Due to point 2 and 3 all Low tasks will meet their deadline in Low mode. Deadlines of Hi tasks are ensured in Low mode due to use of $HiLowTree$ that generates a correct schedule of start tasks. Hi mode scheduling is correct as the scheduler behaves just as if it would have used $HiOnlyTree$. In the next section, we present the first evaluations of our method.

4. EVALUATION

In this section we describe how we evaluate the performances of our method and describe the first results of our experiments.

4.1 Evaluation

We first need criteria to compare our results with. These criteria could be the number of processors that are necessary to schedule a task set $\mathcal{T}$ by other algorithms. In our case, we compare with unmodified RUN. For example, if we consider the task set described in figure 2, a RUN scheduler requires up to 4 processors to schedule all tasks according to Hi values for WCETs. Scheduling only Hi tasks takes only 2 processors. From these observations, we introduce three constants computed from MC task set attributes to assess the performances of a MC scheduler:

- U_{Hi+Low}: task set utilization for all tasks of $\mathcal{T}$ considering most pessimistic WCETs (notice that for Low task $C_i(Hi) = C_i(Low)$).

- U_{Hi}: task set utilization restricted to tasks of Hi criticality in $\mathcal{T}$.
- U_{Low}: task set utilization restricted to Low criticality tasks in $\mathcal{T}$.

As RUN is optimal, we know that all tasks could be scheduled in their worst case with $\lceil U_{Hi+Low} \rceil$ processors. Moreover, any optimal MC scheduler should handle a timing failure occurring at time 0, and need at least $\lceil U_{Hi} \rceil$ processors.

Now, let us denote $\lceil U_{bundled} \rceil$ the amount of processors required by our method. First, a sanity check consists in comparing the behavior of the approach in the worst case: no Low tasks are integrated to modal servers. It leads to $\lceil U_{bundled} \rceil = \lceil U_{Hi} \rceil + \lceil U_{low} \rceil \leq \lceil U_{Hi+Low} \rceil + 1$. This upper bound is not as bad as it seems as we observed that being able to schedule Hi and Low tasks on $\lceil U_{Hi+Low} \rceil + 1$ processors offers better performances than other MC schedulers. The interest of this approach lies in the fact that we do not only compare to other MC schedulers but also to classical ones (and especially RUN) using most pessimistic scenarios. Thus, it enables us to judge if the use of a MC scheduling algorithm is opportune or not. In the next section we present the results of our method.

4.2 First Experiments

Due to space limitation, we only present the results of the method on the task set introduced in figure 2. The application of our method resulted in the allocation of task τ_5 in the modal server of task τ_2. However, the task τ_4 can not be executed in any modal server. Finally, we have the resulting modal servers described in figure 3.

Hi Primal Servers	Hi Tasks	Modal Server
S_1	τ_1	$\emptyset$
S_2	τ_2	τ_5
S_3	τ_3	$\emptyset$

Figure 3: Created Modal Servers

Thanks to our method, we can save up to a processor to execute this system, compared with the four processors needed in Hi+Low mode as can be seen in figure 4.

Task Set	Utilization	Number of processors required
Hi+Low	3.1	4
Bundled	3	3
Low Task $\notin$ modal server in Bundled task set	0.54	1

Figure 4: Results of our method

More experiments are under processed, but until now our method has never required more processors than the Hi+Low mode. For these experiments, values used for utilization were drawn from applying the uunifast algorithm adapted to multi-core platforms [3]. Besides, generation of periods has been bone by using an algorithm described in [4].

5. CONCLUSION

In this paper, we find a way to adapt the algorithm RUN to the scheduling of MC systems. This has been achieved by enabling the use of modes by RUN. First experiments provided some promising results, even in the case of unfavorable task sets.

Further work will be to enhance our method by deriving a more sophisticated assignment policy of the Low tasks into modal servers.

6. ACKNOWLEDGMENTS

This research work has been carried out in the framework of the Technological Research Institute SystemX, and therefore granted with public funds within the scope of the French Program "Investissements d'Avenir".

7. REFERENCES

[1] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[2] A. Burns and R. Davis. Mixed criticality systems - a review. www-users.cs.york.ac.uk/ burns/review.pdf, 2013.

[3] R. Davis and A. Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 398–409, Dec 2009.

[4] P. Emberson, R. Stafford, and R. I. Davis. Techniques For The Synthesis Of Multiprocessor Tasksets. 1(1):6–11, 2010.

[5] C. L. J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard- Real-Time Environment Scheduling Algorithms for Multiprogramming. (1):46–61, 1973.

[6] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. RUN: Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor. *2011 IEEE 32nd Real-Time Systems Symposium*, pages 104–115, Nov. 2011.

[7] J. Theis, G. Fohler, and S. Baruah. Schedule table generation for time-triggered mixed criticality systems. In *1st Workshop on Mixed Criticality Systems, IEEE Real-Time Systems Symposium*, December 2013.

[8] S. Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 239–243, Dec. 2007.

Study of Temporal Constraints for Data Management in Wireless Sensor Networks

Abderrahmen BELFKIH, Bruno SADEG, Claude DUVALLET, Laurent AMANTON

University of Le Havre
BP 1123, 76063 Le Havre cedex

{Abderrahmen.Belfkih, Bruno.Sadeg, Claude.Duvallet, Laurent.Amanton}@univ-lehavre.fr

ABSTRACT

Over the last years, extensive research has been carried out, to study data processing in wireless sensor networks (WSN). Real-time aspect is an important factor for sensor applications in order for the data to reflect the current state of the environment. Several research efforts have focused on effective query processing in WSN and others are focused on the use of abstract database technology (Cougar, TinyDB,...) to manage data efficiently in sensor networks. However, there is some work dealing with the temporal constraints when managing sensor data.

In this paper, we study some temporal constraint parameters like response time, period and computing time through two data processing techniques for WSN. The first is based on a warehousing approach, in which data are collected and stored on remote database. The second is based on query processing using an abstract database like TinyDB. This study has allowed us to identify the factors which enhance the respect of temporal constraints.

Keywords

wireless sensor network, data collection, query processing, sensors database, time constraints.

1. INTRODUCTION

Wireless sensor networks (WSN) can be considered as a type of ad hoc wireless networks. They are composed of small wireless nodes which are manually or randomly deployed in a region of interest to sense different physical characteristics of the environment like temperature, humidity, pressure, light, etc. The nodes transmit the sensed data to a sink node referred to as a Base Station. This process is called data collection, it must be able to meet certain deadlines and the timely data delivery, to reflect the current state of the environment.

Actually, the sensor networks are deployed without considering the transmission delays of data or their deadlines. Generally, researchers are interested in data processing techniques to reduce the number of data transmissions and to offer an efficient solution to increase the lifetime of the network. They have proposed news methods for data processing like data aggregation techniques, efficient transmission control protocols and query processing systems such as Cougar [4] and TinyDB [6], to simplify the extraction and management of data. Sensor network loses its intended function (consistency) if data is delivered late. Many WSN applications like industrial process control and monitoring, can be considered time critical. They require a strict deadline for data delivery to the sink.

The idea in this paper is to study temporal constraints and data arrival times from sensors to users/actuators and the factors that may influence this time using two approaches: abstract database and periodic data collection.

Data collection and query processing are two different technologies which are two popular techniques used for data processing in WSN. The first present a traditional technique used in many applications where all sensors send data periodically to the sink at predetermined times. The second is a recent technique for data processing in WSN based on SQL-like query language, where queries are issued to the sensors to get the required information which has been redefined in the query. Many research are interested in this technology, because it has shown good results so far.

The remainder of this paper is structured as follows. Section 2 reviews the related work. In Section 3, we describe the two approaches cited previously. In Section 4, we give simulation results and we discuss them. Finally, in Section 5, we conclude the paper by showing how to effectively exploit the two techniques in WSN and we give some perspectives to this work.

2. RELATED WORK

In literature, many research works have been interested to improve the time of data management in WSN. Kshama *et al.* [7] suppose that the freshness of data is maintained as the process of data collection is as much faster as possible. In their paper, they discuss different methods of data collection in WSN like the methods for fair data collection and a various TDMA scheduling schemes. Choi *et al.* [3] propose a new data aggregation algorithm in WSN with time constraint. Their goal is to provide data while respecting their constraint time and to limit the number of transmissions which reduce the energy consumption in WSN. In [8], authors have used the Time Division Multiple Access (TDMA) schedule to reduce the latency of data collection and to conserve energy at sensor nodes. They have divided the time into slots and the duration of a time slot allows a sensor node to transmit one packet.

Data query processing using an abstract database has been cited for the first time by Bonnet *et al.* in their paper [2]. They present a declarative approach to facilitate

the description and processing of queries in the WSN. The data required by the base station form a virtual table, in which columns represent the data requested by the user. TinyDB and Cougar represent the first generation of query processing systems in wireless sensor networks. Corona [5] is a distributed query processor. It makes the WSN like a table in a relational database and it uses SQL-style syntax to formulate a queries.

3. NETWORK MODEL

The basic model is composed of three parts: (i) sensor nodes, randomly deployed over an area. They can retrieve data like humidity, temperature and light and send values to a base station, (ii) the base station periodically collects data from sensor nodes and inserts them into a database, (iii) users can connect to the database to get information about WSN. We have update the model in the second scenario and we have added an abstract database for WSN named TinyDB, which allows the user to send queries to the base station and to get responses. Our goal is to compare the timing properties using a static database versus an abstract in-network database abstraction.

3.1 First scenario: data collection with remote database

In this scenario, sensors send data periodically to the base station. The data collected by WSN are streamed to a remote database, where can be stored. Next, user can query stored data at any time.

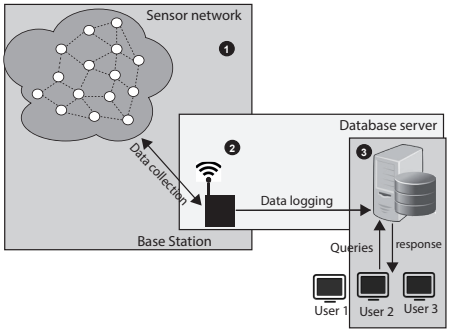


Figure 1: Data collection with remote database

3.2 Second scenario: query processing with WSN abstract database

This scenario is based on a query processing technique, in which we use an abstract database system named TinyDB connected to the base station of WSN. Users specify the data they want to collect through SQL-like declarative queries, to send them to the base station via the the abstract database interface. The base station will broadcast these queries over the network. Each sensor that receives a query, responds with the values requested for a duration fixed in the query by the user. Finally, the base station send the responses back to the user via the abstract database system, which can be stored also in a remote database.

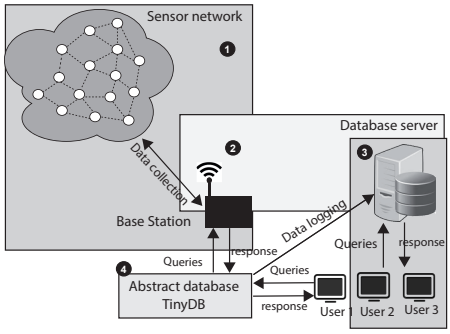


Figure 2: Query processing with WSN abstract database

In the next section, we implement the two scenarios described above and we discuss some time parameters such as network convergence time and data collection time. We analyze also the impacts of network topologies and the impact of choosing the database on average response time.

4. SIMULATION AND RESULTS

For the first scenario, we have used COOJA network simulator for Contiki [9] to create the WSN simulation. We have created one sink node and a set of Sky Motes sensors. The nodes are randomly distributed in a area within 80m*80m and the sink is placed at the center of the deployment area. The number of sensors vary from 10 to 100 with step of 10. Each node sends temperature and humidity to the base station (sink node) every 60 seconds. We have used RPL routing protocol [11] to provide the communication between sensor nodes and the base station. RPL is more efficient for data delivery time than other existing known protocols such as LOAD (LOAD is derived from AODV), DSR and DSDV [10, 1]. RPL provides efficient routing paths guaranteeing data delivery before deadline in WSN. It is based on a Destination-Oriented Directed Acyclic Graph (DODAG) anchored at one or more nodes (DAG root(s)). Each node computes its rank in the RPL tree and maintains a routing table to reach all nodes in this sub-DODAG[1].

The second scenario is based on Mica2 sensors randomly scattered in the field. We have varied the number of sensors from 10 to 100 by step of 10. We use the UDP[2] to make communication between sensors and the base station. We have used TinyDB system using Tossim the simulator of TinyOS. TinyDB assumes a fixed tree-based topology, where all nodes try to form a tree to the root node based upon the link quality information between the node and its neighbors.

4.1 Impact of number of nodes on network convergence time

Network convergence is defined as the time needed to start the different devices, to make connection between sensors and base station and to build routing tables of nodes. This

[1]The sub-DODAG of a node is the set of other nodes whose paths to the DODAG root pass through that node.
[2]User Datagram Protocol

step comes just before the process of data collection. It can affect the arrival time of data. The shorter the convergence time, the quicker the availability of the data.

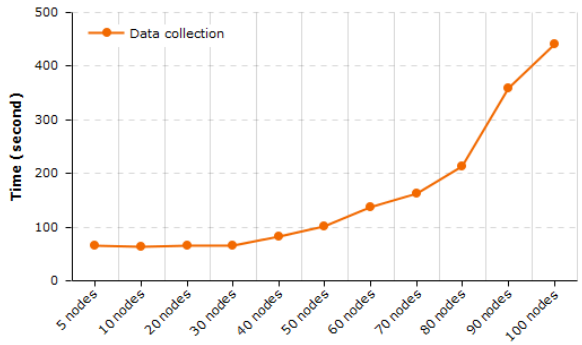


Figure 3: Network convergence time for data collection

In Figure 3, the network convergence time increases when the number of nodes increases. Usually, network convergence time depends on the DODAG building process, which begins at the root (base station). The root starts the dissemination of information about the structure using DIO (DODAG Information Object) message. The nodes not connected to the tree (without communication with the root) receive the message DIO and process it. Then they decide to join or not the structure. Once the node has joined the structure, it has a route to the root of the DODAG structure. Building the final routing table depends on the number of nodes and the path cost between nodes.

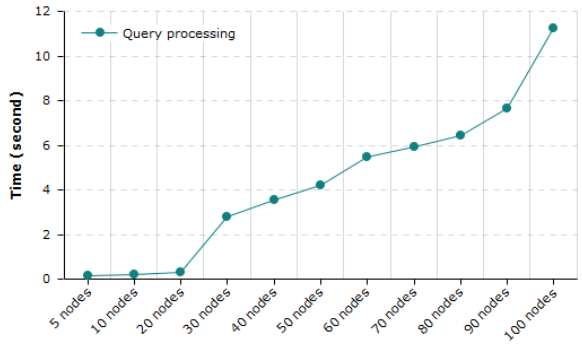


Figure 4: Network convergence time for TinyDB

We note also that the convergence time for TinyDB is less than that of data collection (cf. Figure 4). TinyDB constitutes a routing tree or Directed Acyclic Graph (DAG) with all sensors in which the root is the sink (gateway). TinyDB uses a selecting multi-hop tree based on efficiency of routing which depends on energy consumption. Sensors make intelligent routing decisions, where every node selects a parent based on link quality to the base station. The nodes keep both a short list of neighbors which have a recently heard transmit, and some routing information about the connectivity of those neighbors to the rest of the network.

4.2 Impact of number of nodes on data collection time

Data collection time present the time taken to get the responses from all sensors connected to the base station. It can affects the data validity time and not reflect the current state of the environment. The time required to send the data from sensor to the base station depends on the sensors capacity, its position and the path quality which can be the number of hops to the base station. Generally, the sender uses multi-hop paths to send its data when the base station is far located which leads to transmission delay and they can cause a failure to respect the data validity time. Figure

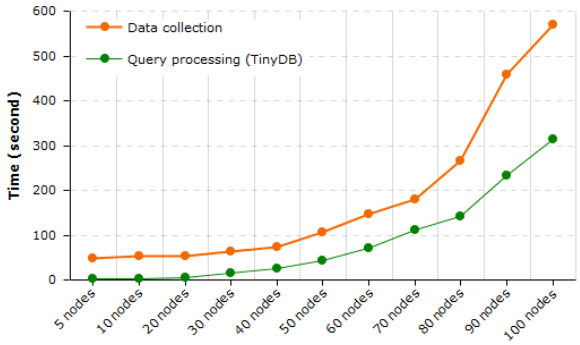


Figure 5: Completed Cycle Time

5 shows the time spent by the nodes to complete the sending of their data, using data collection technique and query processing technique. We observe that the curve increases when the number of sensors increases for the two techniques. The time required to send the data depends of the number of hops and the availability of parent nodes to send data received from children. Their availability is not guaranteed all time because the choice of parent node depends on the node decision to join or not the structure.

We also notice in Figure 5 that the time required to obtain results from all sensors by using the technique of processing request is less than that of data collection. With TinyDB, the data is regularly reported and aggregated by a tree or a directed acyclic graph from the nodes to an access point network. It includes aggregation and filtering operations inside the sensor network to maintain all routing information. The parent nodes near the root put agreements with their children on a time interval for the data listened to them. The whole process is repeated for each period and query.

4.3 Impact of choosing the database on average response time

We use in this test three DBMS[3]: PostgreSQL, MySQL, SQLite, to evaluate queries response time.

Table 1: Average response time (ms)

100 insertion queries from base station to database	9.397	48.626	72.788
100 random select queries from users to database	0.992	0.690	0.225

[3]DataBase Management System

Table 1 shows that SQLite database gives the best average response time for the SELECT queries, next we find MySQL and finally PostgreSQL. For the INSERT queries, PostgreSQL database has the less average response time compared with MySQL and SQLite. SQLite takes a lot of time to insert data, because it does not have a central server to coordinate accesses. It must close and reopen the database file, and invalidate its cache, for each transaction. It does not allow multiple concurrent writes in the same time. PostgreSQL allows multiple transactions to proceed with inserts, concurrently. MySQL is better than PostgreSQL for reading tables because it has a "Query Cache" to make queries faster. SQLite is faster than MySQL because it uses a disk access to read and write directly from the database files on disk.

4.4 Impact of network topologies on data collection time

We have tested 15 sender nodes and one sink node with four network topologies. The table below shows the time spent to collect data from all sensors during a complet cycle of data collection for the two technologies.

Table 2: Complet cycle time (ms)

Topologies	Data collection	Query processing
Star	62312	1832
Mesh	56002	1362
Grid	54121	2163
Tree	53515	2143

We note that the tree network topology for data collection technology is more faster versus the other topologies. In fact, RPL forms a tree-like topology rooted at the sink, reflecting the above results. Star, mesh and grid topologies have not shown good results. RPL takes a time to to define DODAG networks and to build path to the root. It can not fully exploit, which affects the quality of paths. For the query processing technology, the mesh topology gives the best complet cycle time versus the other topologies. In mesh topology, all nodes cooperate in the distribution of data in the network. The same technology is used by TinyDB. The nodes make an intelligent routing decisions, where every node selects a parent based on link quality to the base station.

5. CONCLUSION

In this work, we study and we compare the timing properties of the data collection from a sensor network using a static database versus an abstract in-network database abstraction, named TinyDB. We evaluate temporal constraints such as data collection time, database response time, and network convergence time in both approaches. We have found according to the tests performed that many factors can affect the temporal constraints in a WSN. We have determined that the network topology and the routing protocol, together may play an important role on data collection time. The convergence time also has an impact on the process of data collection. We have shown clearly the timing-response advantage of using a TinyDB approach compared to accessing the data stored in an external database. So we can conclude that the great choice of the network topologie and the routing protocol with the right approach can improves the temporal constraints in WSN. We plan to work in this direction in a future works and we plan to take data temporal consistency.

6. REFERENCES

[1] L. Ben Saad, C. Chauvenet, and B. Tourancheau. Simulation of the RPL Routing Protocol for IPv6 Sensor Networks: two cases studies. In *International Conference on Sensor Technologies and Applications SENSORCOMM 2011*, Nice, France, 2011. IARIA.

[2] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proceedings of the International Conference on Mobile Data Management*, pages 3–14, London, UK, UK, 2001. Springer-Verlag.

[3] J. Y. Choi, J. Lee, K. Lee, S. Choi, W. H. Kwon, and H. S. Park. Aggregation time control algorithm for time constrained data delivery in wireless sensor networks. In *Proceedings of the 63rd IEEE Vehicular Technology Conference, VTC Spring 2006, 7-10 May 2006, Melbourne, Australia*, pages 563–567, 2006.

[4] W. F. Fung, D. Sun, and J. Gehrke. Cougar the network is the database. In *International conference on Management of data*, pages 621–621, New York, NY, USA, 2002. ACM.

[5] R. Khoury, T. Dawborn, B. Gafurov, G. Pink, E. Tse, Q. Tse, K. Almi'Ani, M. M. Gaber, U. Röhm, and B. Scholz. Corona: Energy-efficient multi-query processing in wireless sensor networks. In *DASFAA (2)*, pages 416–419, 2010.

[6] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 491–502, New York, NY, USA, 2003. ACM.

[7] J. D. Pandya and T. Vasavda. Data collection in tree-based wireless sensor network using tdma scheduling. In *International Journal of Advanced Research in Computer Science and Software Engineering*, pages 101–105, 2014.

[8] V. R. Yogeswari. A survey on efficient data collection in wireless sensor networks. In *International Journal of Innovative Research in Computer and Communication Engineering*, pages 2181–2184. IJIRCCE, 2013.

[9] N. Tsiftes, J. Eriksson, and A. Dunkels. Low-power wireless ipv6 routing with contikirpl. In *International Conference on Information Processing in Sensor Networks*, pages 406–407, New York, 2010. ACM.

[10] M. Vucinic, B. Tourancheau, and A. Duda. Performance comparison of the rpl and loadng routing protocols in a home automation scenario. In *WCNC*, pages 1974–1979, 2013.

[11] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander. RPL: IPV6 Routing Protocol for Low-Power and Lossy Networks, mar 2012.

An Approach for Verifying Concurrent C Programs

Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia
CEA, LIST,
91191 Gif-sur-Yvette, France
first.last@cea.fr

Kamel Barkaoui
CEDRIC Laboratory, CNAM,
Paris, France
kamel.barkaoui@cnam.fr

Serge Haddad
LSV, ENS Cachan, & CNRS &
INRIA, France
haddad@lsv.ens-cachan.fr

ABSTRACT

As software system and its complexity are fast growing, software correctness becomes more and more a crucial issue. We address the problem of verifying functional properties of real-time operating system (microkernel) implemented with C. We present a work-in-progress approach for formally specifying and verifying concurrent C programs directly based on the semantics of C. The basis of this approach is to automatically translate a C code into a TLA+ specification which can be checked by the TLC model checker. We define a set of translation rules and implement it in a tool (C2TLA+) that automatically translates C code into a TLA+ specification. Generated specifications can be integrated with manually written specifications that provide primitives that cannot be expressed in C, or that provide abstract versions of the generated specifications to address the state-explosion problem.

1. INTRODUCTION

Formal software verification has become a more and more important issue for ensuring the correctness of a software system implementation. The verification of such system is challenging: system software is typically written in a low-level programming language with pointers and pointer arithmetic, and is also concurrent using synchronization mechanisms to control access to shared memory locations. We address these issues in the context of formal verification of operating systems microkernels written in C programming language. We note that we are interested to check functional properties of the microkernel and not timed properties.

In this paper we present a work-in-progress approach for formally specifying and verifying C concurrent programs using TLA+ [11] as formal framework. The proposed approach is based on the translation from C code to a TLA+ specification. We propose a translator called C2TLA+ that can automatically translate from a given C code an operational specification on which back-end model checking techniques are applied to perform verification. The generated specification can be used to find runtime errors in the C code. In addition, they can be completed with manually written specifications to check the C code against safety or liveness properties and to provide concurrency primitives or model hardware that cannot be expressed in C. The manually written specifications can also provide abstract versions of translated C code to address the state space explosion problem.

Why TLA+?.

The choice of TLA+ is motivated by several reasons. TLA+ is sufficiently expressive to specify the semantics of a programming language as well as safety and liveness properties of concurrent systems [10]. Its associated model checker, TLC, is used to validate the specifications developed and is also supported by the TLAPS prover. TLA+ provides a mechanism for structuring large specifications using different levels of abstraction and it also allows an incremental process of specification refinement. So we can focus on relevant details of the system by abstracting away the irrelevant ones.

Outline.

The rest of the paper is organized as follows. We discuss related work in Section 2. We give an overview of the formal language that we used (TLA+) in Section 3. Section 4 presents the global approach and our current work. Section 5 concludes and presents future research directions.

2. RELATED WORK

There exist a wealth of work on automated techniques for formal software verification. The seL4 [9] is the first OS kernel that is fully formally verified. The verification of seL4 has required around 25 person-years of research devoted to developing their proofs and more than 150,000 lines of proof scripts. Deductive techniques are rigorous but require labor-intensive as well as considerable skill in formal logic. We focus here on more closely related work based on the model checking technique.

SLAM [2] and BLAST [6] are both software verification tools that implement the *counter-example-guided predicate abstraction refinement* (CEGAR) approach [5]. They use an automatic technique to incrementally construct abstractions i.e. abstract models cannot be chosen by user. But, SLAM cannot deal with concurrency and BLAST cannot handle recursion.

Another approach consists to transform the C code into the input language of a model checker. Modex [8] can automatically extract a Promela model from a C code implementation. The generated Promela model can then be checked with SPIN [7] model checker. As Promela does not handle pointer and has no procedure calls, Modex handles these missing features by including embedded code inside Promela specifications. On the other hand, the embedded code fragments cannot be checked by SPIN and might contain a division by zero error or null pointer dereference, Modex instruments additional checks by using assertions. But, not all errors can be anticipated and the model checker can crash [8]. CBMC [3] is a bounded model checker for ANSI C programs. It translates a C code into a formula (in Static Single Assignment form) which is then fed to a SAT or SMT solver to check its satisfiability. It can only check safety properties. CBMC explores program behavior exhaustively but only up to a given depth, i.e. it is restricted to programs without

deep loops [4].

In this work, we propose a methodology to specify and verify C software systems using TLA+ as formal framework. With TLA+, we can express safety and liveness properties unlike SLAM, BLAST and CBMC which have limited support for concurrent properties as they only check safety properties. Our approach uses abstraction and refinement in order to structure specifications and mitigate the state explosion problem for modular reasoning which is not the case of Spin and CBMC.

3. AN OVERVIEW OF TLA+

TLA+ is a formal specification language based on the Temporal Logic of Actions (TLA) [10] for the description of reactive and distributed systems. TLA combines two logics: a logic of actions and a temporal logic. To specify a system in TLA, one describes its allowed behaviors. A *behavior* is an infinite sequence of states that represents a conceivable execution of the system. A *state* is an assignment of values to variables. A *state predicate* or a *predicate* for short is a boolean expression built from variables and constant symbols. An *action* is an expression formed from unprimed variables, primed variables and constant symbols. It represents a relation between old states and new states, where the unprimed variables refer to the old state and the primed variables refer to the new state. For example, $x = y' + 2$ is an action asserting that the value of x in the old state is two greater that the value of y in the new state.

Formulas in TLA are build from actions using boolean connectives, TLA quantification and temporal operators $\Box$ ($always$). The expression $[A]_{vars}$ where A is an action and $vars$ the tuple of all system variables, is defined as $A \lor vars' = vars$. It states that either A holds between the current and the next state or the values of $vars$ remain unchanged when passing to the next state.

A TLA+ specification consists on a single mathematical formula *Spec* defined by:

$$Spec \triangleq Init \land \Box[Next]_{vars} \land Fairness \quad (1)$$

where

- *Init* is the predicate describing all legal initial states,
- *Next* is the next-state action defining all possible steps of the system,
- *Fairness* is a temporal formula that specifies fairness assumptions about the execution of actions.

The formula *Spec* is true of a behavior iff *Init* is true of the first state and every state that satisfies *Next* or a "stuttering step" that leaves all variables *vars* unchanged.

To show that a property holds for a program, we must check that the program *implements* the property ϕ, which is formalized as $Spec \Rightarrow \phi$. This formula is said to be valid iff every behavior that satisfies *Spec* also satisfies ϕ.

Moreover, TLA+ has a model checker (TLC) that allows to check if a given model satisfies a given TLA formula. TLC can handle a subclass of TLA+ specifications that we believe includes most specification that describe our systems.

4. SPECIFICATION AND VERIFICATION APPROACH

The specification and verification approach is illustrated in Figure 1. The first step of the approach is to automatically translate a C code implementation to a TLA+ specification using the translator that we developed C2TLA+.

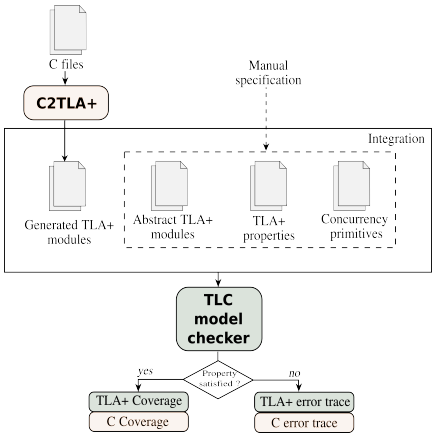


Figure 1: Specification and verification approach

C2TLA+ uses CIL [13] to transform intricate constructs of C into simpler ones. After obtaining the Abstract Syntax Tree (AST) of the normalized C code, C2TLA+ generates a TLA+ specification according to a set of translation rules that we define in Subsection 4.2. The generated specifications can be checked by TLC without any user interaction for potential C errors.

TLA+ specifications are organized into modules that can be reused independently. The methodology provides for the user the possibility to connect generated modules to other manually specified modules. These latter can model synchronization primitives like "compare-and-swap" and "test-and-set" instructions, model hardware like interruptions, or provide an abstract model of a specification. All modules are integrated together to form the whole system to verify. The user defines a set of safety and liveness properties expressed in TLA and TLC explores all reachable states in the model, looking for one in which (a) an invariant is violated, (b) deadlock occurs (there is no possible state), (c) the type declaration is violated. When a property is violated, TLC produces the minimal length trace that leads from the initial state to the bad state. To improve usability, we reimplement this trace in the C code. In addition, TLC also collects coverage information by reporting the number of times each action of a specification was "executed" to construct a new state. This information is used to generate the C code coverage of an implementation, which may be helpful for finding untested parts of the C code.

4.1 Considered features

We handle a subset of C according to simplifications done by CIL. The C aspects that we consider include basic datatypes (`int`, `struct`, `enum`), arrays, pointers, pointer arithmetic, all kinds of control flow statements, function calls, recursion and concurrency. We do not yet consider floating point operations, non-portable conversions between objects of different types, dynamic allocation, function calls through pointers, and assignment of structs. We note that these features are not needed by the system that we aim to check.

4.2 Translation from C to TLA+

4.2.1 Concurrency and memory layout

A concurrent program consists in many interleaved sequences of operations called *processes*, corresponding to

threads in C. C2TLA+ assigns to each process a unique identifier *id*.

The memory of a concurrent C program is divided by C2TLA+ into four regions:

- A region that indicates for each process where is in its program sequence. This region is modeled by a TLA+ variable *stack_regs*, associating to each process a stack of records. Each record contains two fields:
  - *pc*, the program counter, points to the current statement of the function being executed, represented by a tuple ⟨*function name*, *label*⟩;
  - *fp*, the frame pointer, contains the base offset of the current stack frame.

  The top of each stack register (*Head(stack_regs[id])*) indicates the registers of the function being currently executed.
- A region that contains global (and static) variables and called *mem*. It is modeled by an array and it is shared by all processes.
- A region called *stack_data* and contains stack frames. Each process has its own stack which contains the parameters of a function and its local variables. Stack frames are pushed when calling a function and popped when returning.
- A region that contains values to be returned by processes and it is modeled by an array indexed by the process identifier, called *ret*.

Figure 2 gives an example of a C code in which one process (with *id* equals *"p1"*) executes `p1()` function and the second one (with *id* equals *"p2"*) executes `p2()` function. C2TLA+ assigns to each C variable a unique constant that we called "address". This latter specifies the memory region where data is stored (local or global) and the offset of the data in the memory region. For example, the TLA+ expression $[loc \mapsto "mem", offs \mapsto 0]$ denotes the record *Addr_count* such that *Addr_count.loc* equals "*mem*" and *Addr_count.offs* equals 0.

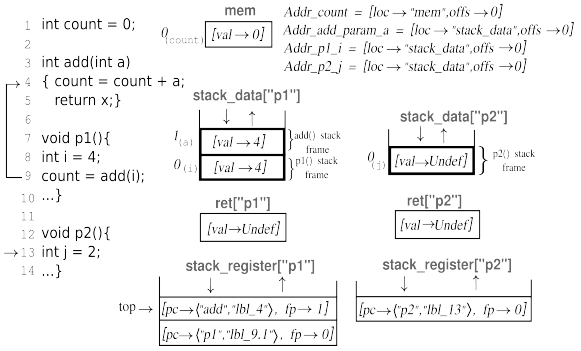


Figure 2: Example of a C code and its memory representation in TLA+

4.2.2 Loading and assignment

Variable names, fields data structure and arrays in C are lvalues. C2TLA+ translates an lvalue into an address. Loading an lvalue is performed by the TLA+ operator *load*(). An assignment of an lvalue is translated by C2TLA+ using the *store*() operator which saves the value of the right-hand operand into the memory location of the lvalue. The definition of *load*() and *store*() are given in Figure 3. For example, accessing to the value of `count` is expressed by the TLA+ expression *load(id,Addr_count)*.

$$
\begin{array}{l}
load(id,\ ptr)\ \triangleq \\
\quad \text{IF } ptr.loc = \text{“mem”} \text{ THEN } mem[ptr.offs] \\
\quad \text{ELSE } stack\_data[id][Head(stack\_regs[id]).fp + ptr.offs] \\
\\
store(id,\ ptr,\ value)\ \triangleq \\
\lor\ \land\ ptr.loc = \text{“mem”} \\
\quad \land\ mem' = [mem \text{ EXCEPT } ![ptr.offs] = value] \\
\quad \land \text{ UNCHANGED } stack\_data \\
\lor\ \land\ ptr.loc = \text{“stack\_data”} \\
\quad \land\ stack\_data' = [stack\_data \text{ EXCEPT} \\
\quad\quad ![id][Head(stack\_regs[id]).fp + ptr.offs] = value] \\
\quad \land \text{ UNCHANGED } mem
\end{array}
$$

Figure 3: Definition of *load()* and *store()* operators

4.2.3 Function definition

A C function definition is translated into an operator with the process identifier *id* as argument. Translating the function body consists of the disjunction of translating each statement that contains. C2LTA+ uses label values given by CIL to identify statements and each C statement is translated into an atomic action defined as a conjunction of sub-actions. At a given state one and only one action is evaluated to true. Each action updates the *stack_regs* variable by modifying its head by the label value of the action done once the call has finished.

4.2.4 Function call

Each function call results in a creation of a stack frame onto the local memory *stack_regs[id]* of the process *id*. The stack frame contains local variables and formal parameters which are initialized with the values with which the function was called. Then, the program counter is updated by changing its head to a record whose *pc* field points to the action done once the call has finished (the instruction following the function call). At the top of the stack register is pushed a record whose *pc* field points to the first statement of the called function, and *fp* field points to the base address of the new stack frame.

4.2.5 Return statement

Once the function returns, the returned value is stored on value returning memory *ret*. Its stack frame and the top of the stack register are popped and the program control is returned to the statement immediately following the call.

4.3 Checking the specification

C2TLA+ generates the main specification *Spec* that describes the execution of the C program.

- The *Init* predicate which specifies the initial values of all variables.
- The tuple of all variables $vars \triangleq \langle mem, stack\_data, stack\_regs, ret \rangle$.
- The predicate $process(id)$ defines the next-state action of the process *id*. It asserts that one of the functions is being executed until its stack register becomes empty. For the example of Figure 2, the *process*() predicate is defined as:

$$process(id) \triangleq \begin{array}{l} \land\ stack\_regs[id] \neq \langle\rangle \\ \land(\ add(id) \lor p1(id) \lor p2(id)\ ) \end{array}$$

- The *Next* action states that one process is non-deterministically selected among those which can take an execution step or that leaves all variables unchanged when all processes terminate.

$Next \triangleq$
$\lor \exists\ id \in ProcSet : process(id)$
$\lor(\forall\ id \in ProcSet : (stack\_regs[id] = \langle\rangle) \land$ (UNCHANGED $vars$))

- The main specification is defined by $Spec \triangleq Init \land \Box[Next]_{vars} \land WF_{vars}(Next)$. To check liveness properties in the system, we must consider fairness assumptions.

The generated specification can be directly checked by TLC. In that case, errors reported by TLC correspond to runtime errors in the C code, e.g. dereferencing null-pointer, uninitialized data access and division by zero.

4.4 Integrating abstract models

When checking whether a concurrent program satisfies a given properties, the size of the state space of the program limits the application of the model checking. A way to tackle the state space explosion problem is *abstraction*. A program usually have internal actions which need not to be considered in the verification process. Ignoring such actions reduces the state space of the program and makes the model checking feasible. With TLA+, it is possible to define different levels of abstraction and model check the existence of refinement relationship between two specifications. A specification R is a refinement of an abstract specification S iff $R \Rightarrow S$. This is true iff there exists a refinement mapping between the two specifications R and S. The refinement mapping [1] maps states of the concrete specification with states of abstract specification. From a generated TLA+ specification by C2TLA+, TLC can check if this specification refines an abstract model w.r.t. a refinement relation which preserves the properties of the abstract system.

4.5 Results and current work

Currently, we developed the translator C2TLA+ which automatically generates a TLA+ specification from C code. The translator is based on the semantics of C. We assume that generated specification behaves exactly as the C program. We tried many academic examples of C code that we checked using C2TLA+ and TLC. Actually, we are applying the methodology on a critical part of the microkernel of the PharOS [12] real-time operating system (RTOS). This part consists of a distributed version of the scheduling algorithm of the RTOS tasks. Examples of properties that we aim to check include safety properties e.g. that all spinlocks protect the critical sections, at any instant of time, the system schedules the (ready) task having earliest deadline, and also liveness properties e.g. if a thread entered its critical section, it will eventually leave it.

5. CONCLUSION AND FUTURE WORK

We have proposed an approach for specifying and verifying concurrent C programs based on an automated translation from C to TLA+. The goal of the approach is to make concrete and abstract specifications interact. Abstract models can define aspects not expressed in C code like concurrency primitives, or define an abstract specification of a concrete one. Using model checking technique, we can check the refinement relations between two specifications and the correctness properties of the whole system.

We aim to extend this work along several directions. We plan to further study the use of TLA+ modules with different levels of refinement. We also plan to check equivalence between a C code and another simplified C code. The simplified code contains less steps which would reduce the state space of the system to verify. Another avenue of future work include updating the translator to support missing features. It would be interesting to profit from data analysis in C in order to generate TLA+ code with less interleaving between the processes. Finally, we plan to use the TLA+ proof system to prove properties on an abstract TLA+ specification and prove that a generated specification by C2TLA+ is a refinement of this abstract specification.

We must remind that we are reporting the current state of a work in progress and we shall further improvement in the approach process and making it applicable on a considerable case study.

6. REFERENCES

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] T. Ball and S. K. Rajamani. The SLAM project: Debugging System Software via Static Analysis. *SIGPLAN Not*, 2002.

[3] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[4] V. D'Silva, D. Kroening, and G. Weissenbacher. A Survey of Automated Techniques for Formal Software Verification . *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[5] E. A. Emerson and A. P. Sistla, editors. *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2000.

[6] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. pages 235–239. Springer, 2003.

[7] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[8] G. J. Holzmann. Trends in Software Verification. In *Proceedings of the Formal Methods Europe Conference*, Lecture Notes in Computer Science, pages 40–50. Springer, 2003.

[9] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *SOSP*, pages 207–220, New York, USA, 2009.

[10] L. Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[11] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[12] M. Lemerre, E. Ohayon, D. Chabrol, M. Jan, and M.-B. Jacques. Method and Tools for Mixed-Criticality Real-Time Applications within PharOS. In *Proceedings of AMICS 2011: 1st International Workshop on Architectures and Applications for Mixed-Criticality Systems*, 2011.

[13] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.

Resource Sharing Under a Server-based Semi-Partitioned Scheduling Approach

Alexandre Esper [†¶]
[†] Critical Software S.A.
Portugal
aresper@criticalsoftware.com

Eduardo Tovar [¶]
[¶]CISTER/INESC-TEC
Portugal
emt@isep.ipp.pt

ABSTRACT

The rapid evolution of commercial multicore platforms has raised the industry interest in developing and running applications independently on the same platform. However, in realistic industrial settings, tasks belonging to different applications share resources that are not limited to the CPUs. Applications share hardware components (e.g. co-processors and actuators), and need to access portions of code that may be protected by semaphores or mutexes.

In this paper we address the important challenge of resource sharing on multicore platforms through the use of servers, i.e, through a hierarchical scheduling approach, which is an effective technique to ensure the integration of independently developed applications on the same computing platform as well as the isolation of tasks. To solve that problem we adapt and extend the MrsP [6] resource sharing protocol and further combine it with the NPS-F scheduling algorithm [5], which employs a server-base approach. A schedulability analysis is then provided for the resulting framework.

1. INTRODUCTION

The interest of industry in real-time embedded applications has recently gained strength. This has been mainly driven by the important need of taking as much benefit as possible of the processing power offered by multicore platforms, which can now be easily found in products ranging from portable cell phones and smartphones to large computer servers. Additionally, current software development processes often involve more than one independent development team (e.g. subcontractors) that produce software components, which are further integrated to form a final product.

In order to reach that goal, server-based techniques emerged as an intuitive solution to effectively ensure the temporal isolation and protection, while respecting all real-time constraints of the applications. *Servers* allow for the reservation of a portion of the embedded system capacity for a specific application. Therefore, it allows applications to run independently through time partitioning.

The research on real-time systems for uniprocessor (single core) is well established and consolidated, and there are multiple works extending the uniprocessor scheduling algorithms to multicore [7]. However, in practice, several challenges emerge when considering all the resources that must be accessed by tasks running in a multicore environment.

In a realistic industrial application, the resources shared by different tasks are not limited to the CPUs. Applications share hardware components and need to access data or execute portions of code that may be protected by semaphores or mutexes. This adds an additional layer of complexity to the scheduling problem and requires the introduction of *Resource Sharing Protocols*.

The objective of this work is to set the basis for the design of a framework that is able to effectively handle the hierarchical scheduling of tasks on multicore platforms whilst taking the shared logical resources into consideration. Hence, we adapt and combine the MrsP [6] resource sharing protocol with the NPS-F [5] scheduling algorithm.

2. SYSTEM MODEL

We consider the general *sporadic task model*, where each task τ_i in a system τ is characterized by its minimum inter-arrival time T_i, relative deadline D_i, and worst-case computation time, C_i. That is, each task τ_i can generate a potentially infinite number of *jobs* at least T_i time units apart, and each job must execute for at most C_i time units before its deadline occurring D_i time units after its release. *Arbitrary deadlines* are assumed, but jobs from the same task can never execute in parallel. The utilization u_i of a task τ_i is defined as $u_i = C_i/T_i$ and the system utilization, $U(\tau)$, is defined as $U(\tau) = \sum_{i=1}^{n} u_i$.

The execution platform is composed of m identical physical processors, uniquely numbered $P_1 \dots P_m$. We also consider a set of k *servers*, uniquely numbered $S_1 \dots S_k$. The tasks are first mapped to a server, which are then allocated to the processors. The server utilization $U(S_q)$ is defined as:

$$U(S_q) = \sum_{\tau_i \in \tau(S_q)} u_i \quad (1)$$

where $\tau(S_q)$ is the set of tasks assigned to S_q. We assume that the utilization of a server never exceeds 1 and that it never executes on more than one processor at a time.

Shared Resources (denoted as r^j) are defined as the data structures that are shared between tasks. They are divided in two types; those that are shared by tasks mapped to the same server, called *local resources*, and those that are shared between tasks mapped to different servers, which are called *global resources*. The code associated with a resource is called a *critical section* and must be accessed under *mutual exclusion*. The blocking time experienced by a task τ_i when accessing a locked local resource is defined as the *local blocking time*. Similarly, the blocking time experienced by τ_i when it tries to access a locked global resource is defined as the *global blocking time*.

The relation between tasks and resources is given by two functions: $F(\tau_i)$ and $G(r^j)$. $F(\tau_i)$ returns the set of re-

sources used by task τ_i and $G(r^j)$ returns the set of tasks that use resource r^j. The parameter c^j is used to denote the worst case execution time of the resource r^j when accessed by any task.

3. RELATED WORK

This section provides an overview of the MrsP protocol [6] and the NPS-F scheduling algorithm [5], which constitute the basis for the present work.

3.1 Review of MrsP

One of the most recent resource sharing protocols for multicore platforms is MrsP [6]. MrsP is restricted to *fully partitioned systems* where tasks are scheduled using *fixed priorities*. The general *sporadic task model* is employed and each processor P_k implements a local extension of the Stack Resource Policy (SRP) applied to the Priority Ceiling Protocol (PCP) [11] (denoted as PCP/SRP), where all resources r^j are assigned a set of ceiling priorities, one for each processor P_k. The ceilings are defined as the maximum priority of all tasks allocated to P_k that use r^j. Whenever a task τ_i attempts to access r^j, its priority is raised to the local ceiling of r^j. For local resources, MrsP behaves as an implementation of the uniprocessor PCP/SRP. For global resources, the access to a resource is granted through a FIFO queue. While waiting to gain access to resource r^j that is already locked by another task τ_k, τ_i remains active, i.e., it busy-waits for the lock to become available (*spin-based* locking).

The characteristics of MrsP reviewed so far are similar to MSRP [9], of which it is a variant. Its main difference is that tasks busy-waiting may use their "spin" time to undertake the execution of other waiting tasks. This means that although MrsP is defined for partitioned systems, the tasks still have the ability to migrate from one processor to another at run-time. If a task τ_i is preempted whilst accessing a resource r^i, then τ_i can migrate to any processor on which a task is waiting to gain access to r^i. The authors claim that this property effectively leads to a schedulability analysis that presents an identical form to the response-time analysis for uniprocessor, thus providing several desirable properties of the single processor PCP/SRP [11][2]. Therefore, under MrsP, tasks can execute requests from other tasks, thus preventing a degradation of the system performance that would result from a preemption of the task holding the resource.

It was proved in [6] that the MrsP resource sharing protocol can be incorporated in the Response-Time Analysis (RTA)[1] in the following way:

$$R_i = C_i + \max\{\hat{e}, \hat{b}\} + \sum_{\tau_j \in \mathbf{hpl}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j \qquad (2)$$

where $\mathbf{hpl}(i)$ is the set of *local* tasks with priority greater than τ_i. The parameter $\hat{e}$ is the maximum execution time of a resource used by a local task with priority less than that of τ_i and a local task with an equal or higher priority than τ_i. The parameter $\hat{b}$ is the maximum non-preemptive execution time induced by the *Real-Time Operating System* (RTOS).

The C_i parameter for each task is given by:

$$C_i = WCET_i + \sum_{r^j \in F(\tau_i)} n_i \times e^j \qquad (3)$$

where $WCET_i$ is the worst-case execution time of the task, ignoring the time it takes to access resources (but including all time spend in the RTOS). The second term of Equation 3 accounts for the increased cost of the potential parallel access to resource r^j due to tasks running on different processors. n_i is the number of times τ_i uses r_j and parameter e^j is the maximum amount of time τ_i might need to execute r^j, including its spinning time, given by: $e^j = |map(G(r^j))| \times c^j$, where function $|map(G(r^j))|$ returns the number of processors onto which tasks that use resource r^j can execute.

3.2 Review of NPS-F

NPS-F is a semi-partitioned scheduling algorithm [5]. The semi-partitioned approach allows the development of algorithms with a higher utilization bound than the partitioned approach (better work balance between processors) and also reduces the number of migrating tasks by avoiding the use of global shared queues for scheduling tasks to processors, when compared to the global scheduling approaches.

This algorithm employs a server-based approach, considering a set of k *servers*. In the so-called flat-mapping, the tasks are assigned to the servers and each server is then assigned to one or two processors at most. A server has a utilization upper-bounded by 1 and can never execute on more than one processor at a time. Hence, each sever S_q is equivalent to a unicore processor with a computing capacity $U(S_q)$. Each server serves one or more tasks using EDF as the internal scheduling policy.

The NPS-F algorithm is composed of 4 steps. The first step is the assignment of tasks to the servers, based on the task's utilization (u_i). The second step is the computation of the capacity of each server S_q. NPS-F ensures the schedulability of all tasks allocated to S_q, even under the most unfavorable arrival phasings, i.e. when there are ready tasks, but their associated servers are not executing. This is achieved by *inflating* the utilization of the server, given by Equation (1) (see [5][13] for more details). The third step of the off-line procedure is the allocation of the servers to the processors, following a semi-partitioned approach. Servers that are assigned to only one processor are called *non-split servers*, whereas servers that are assigned to two processors each are called *split servers*. Note that the servers that serve the spli-tasks must be carefully positioned within the time slots in order to avoid their overlapping in time. The last step of the algorithm is performed at run-time, when the dispatching inside each server is performed under EDF policy.

Under NPS-F, it is the execution time of the servers which is split - not directly that of the underlying tasks served. In principle, this allows an improved efficiency in the utilization of a multiprocessor system. NPS-F has a utilization bound of 75% configurable up to 100% at the cost of more preemptions and migrations.

Recently, a new schedulability test for mapping tasks to servers for NPS-F has been proposed in [13]. But since we aim at defining a hierarchical scheduling framework that allows resource sharing between tasks, we first present some useful concepts defined in [12]. The *Resource Demand* of a task set $\tau(S_q)$ represents the collective workload resource requirements that the tasks in $\tau(S_q)$ request within a certain interval of time t. The *Demand Bound Function*(DBF) [4] of $\tau(S_q)$ calculates the maximum possible resource demands required to satisfy its timing requirements within a time interval of length t. The *Resources Supply* represents the amount of time the system can provide for $\tau(S_q)$'s execu-

tion, which is in fact the execution time provided by a server S_q, onto which $\tau(S_q)$ has been assigned. The *Supply Bound Function* (SBF) calculates the minimum possible resources supplies provided by a server S_q during a time interval of length t. A server S_q is said to satisfy $\tau(S_q)$'s execution demand if:

$$\mathrm{DBF}(S_q, t) \leq \mathrm{SBF}(S_q, t), \forall t > 0 \quad (4)$$

Inequality (4) is then used as the new NPS-F schedulability test, meaning that the execution demand by all jobs assigned to a server (computed using the DBF) cannot exceed the amount of time supplied by the server for their execution, for every time interval of length t. Since the test is based on the concept of the DBF (exact test for unicore platforms) rather than on the utilization, it allows to overcome many sources of pessimism that existed in the previous analysis [5].

Assuming sporadic task sets with arbitrary deadlines and ignoring all overheads (which can however be easily accounted for as shown in [13]), the $\mathrm{DBF}(S_q, t)$ is given by:

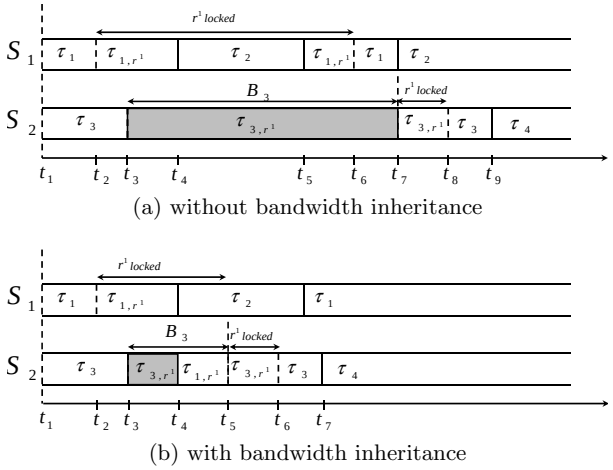
$$\mathrm{DBF}(S_q, t) = \sum_{\tau_i \in S_q} \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) \times C_i \quad (5)$$

To perform the assignment of the tasks to the servers, NPS-F iterates over the set of all system tasks and attempts to fit each one of them (according to the bin-packing heuristic used, e.g., Next-Fit (NF) or First-Fit (FF)) in the servers. Each task τ_i is provisionally added to the chosen server S_q and the length of the testing time interval t is calculated. The schedulability test defined by Equation (5) is then applied and if successful for some server S_i, the task τ_i is permanently mapped to it. Otherwise, a new server is opened and the task τ_i is added to it. If the schedulability test fails for a server with only one task, then the task set is considered unschedulable.

4. ACCOUNTING FOR SHARED RESOURCES IN NPS-F

A current limitation of NPS-F is that it does not consider the interaction between tasks (i.e. the access to shared resources). The solution we propose in this paper complements NPS-F in that sense. It is based on an extension and further adaptation of the MrsP [6] resource sharing protocol that takes the particularities of NPS-F into account. In Section 4.1 we explain how the schedulability test of NPS can be adapted to introduce MrsP resource sharing protocol. Then, the problem of mapping tasks to servers is briefly addressed in Section 4.2.

We adapt MrsP to work with servers by instantiating the concept of *bandwidth inheritance* [8]. With that solution, one server may undertake the processing of a resource critical section on behalf of another task assigned to another server. In order to better visualize the impact of such bandwidth inheritance protocol, consider a simple system composed of two servers and four tasks. It is assumed that both servers are assigned to different physical processors and that the global resource r^1 is shared between the servers. Tasks τ_1 and τ_2 are allocated to server S_1; task τ_1 uses the local resource r^1 and task τ_2 does not use any shared resource (apart from processor). Tasks τ_3 and τ_4 are allocated to server S_2; task τ_3 also uses the global resource r^1 and task τ_4 does not use any resource (apart from the processor).



(a) without bandwidth inheritance

(b) with bandwidth inheritance

Figure 1: Example schedule with shared resource.

Two simple example schedules (with and without bandwidth inheritance) for the system are provided in Fig.1 to illustrate the solution. In Fig.1(a), tasks τ_1 and τ_3 are executing at instant t_1. At instant t_2 task τ_1 locks resource r^1 and execute its critical section, represented by the notation τ_{1,r^1}. At instant t_3, task τ_3 tries to access r^1 but gets blocked (represented in gray) because the resource is already locked by τ_1. Even worse, task τ_1 is preempted by task τ_2 at t_4. Task τ_3 will only be able to continue its execution at t_7, after τ_2 has finished its execution (t_5) and after τ_1 releases r_1 (t_6). The blocking interval of task τ_3 is represented by B_3. In Fig.1(b), the notion of bandwidth inheritance is depicted. The main difference in relation to the previous schedule is that server S_2 is able to undertake the processing of task τ_1 when it is preempted by τ_2 at instant t_4. Task τ_1 then executes the critical section and releases r^1 at $t = 5$. Task τ_3 is then able to lock r_1 much earlier than in the schedule presented in Fig.1(a) and hence B_3 is reduced.

4.1 Adaptation of the NPS-F Schedulability Analysis

The main adaptation required is related to the restriction adopted in MrsP [6] where tasks are scheduled using *fixed priority*. Due to the fact that under NPS-F each server serves one or more tasks employing an EDF scheduling policy [10], the scheduling analysis defined by Equation (2) can no longer be applied. Considering this, a schedulability analysis that accounts for resources sharing under the EDF scheduling policy needs to be defined.

Thus we need to investigate how to account for the effects of (global and local) shared resources into Equation (4). As shown in [3], EDF-scheduled systems in which accessess to shared resources are arbitrated by SRP can be integrated into the analysis using a *Blocking Function* $B(t)$. This function provides the longest blocking time a higher priority task can experience when blocked by a lower priority task. Based on [3], the *Blocking Function* $B(t)$ can be approximated by a function $B^L(S_q)$ and incorporated into Equation (4), resulting in the following schedulability test:

$$\forall t : \mathrm{B}^L(S_q) + \mathrm{DBF}(S_q, t) \leq \mathrm{SBF}(S_q, t) \quad (6)$$

where $\mathrm{B}^L(S_q) = \max\{\hat{e}, \hat{b}\}$ is the blocking term due to local resources used by tasks served by S_q. In order to account for the global resources shared under the MrsP protocol, the DBF can be expanded as follows:

$$\mathrm{B}^L(t) + \sum_{k=1}^{n} \max\left(0, \left\lfloor \frac{t - D_k}{T_k} \right\rfloor + 1\right) \times C_k \leq \mathrm{SBF}(S_q, t) \quad (7)$$

where C_k is given by:

$$C_k = WCET(\tau_k) + \sum_{r^j \in F(\tau_k)} e^j \quad (8)$$

where e^j is now given by $e^j = |mapserv(G(r^j))| \times c^j$. Function *mapserv* returns the set of servers onto which the tasks accessing r^j are assigned. Therefore, similarly to Equation (4), the worst-case execution time of a task τ_i is augmented by the maximum number of parallel access to each resource used by τ_i. Since global accesses are performed in a FIFO manner and because a priority ceiling protocol is used locally to each server, there may be at most one parallel access per server in which the resource is used. The C_i of each task is therefore influenced by the number of servers accessing the resource rather than the number of processors as it was the case in Equation (4).

4.2 Mapping of tasks to servers

Equation (8) shows that the execution time e^j of resource r^j depends on the number of servers that have parallel access to r^j. This leads to one of the key challenges foreseen when applying Inequality (7), i.e. how to perform the mapping of tasks into servers under NPS-F. This mapping uses the schedulability test provided by Inequality (7). However, to perform that schedulability test on a task τ_i, it is necessary to know where the tasks accessing the same resources than τ_i are assigned (through function *mapserv*). Therefore, this leads to a circular dependency between the calculation of the parallel access time to the resources and the assignment of tasks to the servers.

One of the possible solutions to overcome this circularity issue for global resources is to assume a worst case scenario in terms of parallelism, when computing the schedulability test of each task τ_i. This can be achieved by assuming that all the tasks sharing resources with τ_i are mapped to different servers. Under this scenario, two kinds of upper bounds on the amount of parallelism for the access to the resource can be considered: (i) the number of tasks that share resources with τ_i; (ii) the maximum number of servers onto which the tasks can be allocated.

The smallest of these two values can then be used as an upper bound on $|mapserv(G(r^j))|$. In this way the circularity issue is broken. The allocation of the subsequent tasks can be improved by taking into consideration the mapping decisions already taken, and not the worst case any more. However, we still have to consider the worst case scenario for the tasks that have not been allocated to a server yet, but that share resources with τ_i. Note that, through the definition of the parameter $\hat{e}$, local resources exhibit a similar circular dependency between the mapping decisions and the schedulability test. Again, this dependency could be broken by considering the worst-case value for $\hat{e}$, which can be computed taking the maximum blocking time that τ_i can suffer from tasks already assigned to the same server S_q and those that are not yet assigned to any server.

5. CONCLUSIONS

In this paper we present a framework for scheduling real-time tasks in multicore platforms with resources sharing. The solution was based on an adaptation of MrsP resource sharing protocol to work with the server-based semi-partitioned scheduling algorithm NPS-F . The schedulability analysis of tasks assigned to a server is provided, taking into account the blocking time due to shared resources. The next foreseen step is the mapping of the tasks to the servers, which has circular dependencies with the schedulability test provided. The method designed for NPS-F could then be extended to any server based scheduling algorithm for multicore architectures and hence be used to design assignment techniques ensuring the isolation of different applications sharing the same computing platform.

Acknowledgements

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project FCOMP-01-0124-FEDER-037281 (CISTER), and by National Funds through FCT and the EU ARTEMIS JU funding, within projects ref. ARTEMIS/0003/2012, JU grant nr. 333053 (CONCERTO) and and ref. ARTEMIS/0001/2013 (JU grant nr. 621429 - EMC2).

6. REFERENCES

[1] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[2] T. P. Baker. Stack-based scheduling of realtime processes. *Real-Time Syst.*, 3(1):67–99, 1991.

[3] S. K. Baruah. Resource sharing in edf-scheduled systems: A closer look. In *RTSS 2006*, pages 379–387. IEEE, 2006.

[4] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *RTSS 1990*, pages 182–190. IEEE, 1990.

[5] K. Bletsas and B. Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. *Real-Time Syst.*, 47(4):319–355, 2011.

[6] A. Burns and A. Wellings. A schedulability compatible multiprocessor resource sharing protocol–mrsp. In *ECRTS 2013*, pages 282–291. IEEE, 2013.

[7] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35, 2011.

[8] D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *ECRTS, 2010 22nd Euromicro Conference on*, pages 90–99. IEEE, 2010.

[9] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *RTSS 2001*, pages 73–83. IEEE, 2001.

[10] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[11] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

[12] I. Shin and I. Lee. Compositional real-time scheduling framework. In *RTSS 2004*, pages 57–67. IEEE, 2004.

[13] P. Sousa, K. Bletsas, E. Tovar, P. Souto, and B. Åkesson. Unified overhead-aware schedulability analysis for slot-based task-splitting. *Real-Time Syst.*, pages 1–56, 2014.

Externalisation of Time-Triggered communication system in BIP high level models

Hela Guesmi, Belgacem Ben Hedia
CEA, LIST
firstname.lastname@cea.fr

Simon Bliudze
EPFL
Simon.bliudze@epfl.ch

Saddek Bensalem
Verimag, UJF
Saddek.Bensalem@imag.fr

ABSTRACT

To target a wider spectrum of Time-Triggered(TT) implementations of hard real-time systems, we consider approaches for building component-based systems that provide a physical model from a high-level model of the system and TT specifications. The obtained physical model is thus suitable for direct transformation into languages of specific TT platforms. In addition, if these approaches provide correctness-by-construction, they can help to avoid the monolithic a posteriori validation.

In this paper, we focus on the TT interface concept of the TT paradigm. And we present a method that transforms the interaction in classic BIP (Behavior, Interaction, Priority) Model into a TT interface by source-to-source transformations. The method is based on the successive application of two types of source-to-source transformations; Transfer functions internalisation and $n+1$-ary connector to TT interface transformation. The first simplifies the connector transfer functions by modifying components automata. The second transforms connector with simple transfer function into TT interfaces.

Keywords

TT paradigm; correctness-by-construction; Source-to-source transformation; BIP; interaction expressions; connectors;

1. INTRODUCTION

In hard real time computer systems, correctness of a result depends on both the time and the value domains.
With the increasing complexity of these systems, ensuring their correctness using a posteriori verification becomes, at best, a major factor in the development cost and, at worst, simply impossible. An error in the specifications is not detectable. We must, therefore, define a structured and simplified design process which allows the development of correct-by-construction system. Thereby, monolithic a posteriori verification can be avoided as much as possible.

Two fundamentally different paradigms for the design of real-time systems are identified; Event-Triggered(ET) and TT paradigms. In ET paradigm, all communication and processing activities are initiated whenever a considerable event, i.e., change of state in the observed variable, is noticed. It doesn't cope with demands for predictability and determinism that must be met in hard real-time systems. Activities in TT paradigm are initiated periodically at predetermined points in time. These statically defined activation instants enforce regularity and make TT systems more predictable than ET systems. This approach is well-suited for hard real-time systems.

A system model of this paradigm is essential to speed-up understanding and smooth design task. It requires explicitly manipulating not only the value domain specifications, but also temporal constraints for which high abstraction level primitives are not provided. Kopetz [7] presents a TT-Model of computation, based on essential properties of the TT paradigm: **the global notion of time** that must be established by a periodic clock synchronization in order to enable a TT communication and computation, **the temporal structure of each task**, consisting of predefined start and worst-case termination instants attributed statically to each task and **TT interfaces** which is a memory element shared between two interfacing subsystems. TT-Model separates the design of interactions between components from the design of the components themselves.

To target a wider spectrum of TT implementations, we consider approaches for building component-based systems that provide a physical model from a high-level model of the system and TT specifications. In addition, if these approaches provide correctness-by-construction, they can avoid the monolithic a posteriori validation. We focus in particular on the framework BIP [1]. It is a component framework for constructing systems by the superposition of three layers: Behaviour, Interaction, and Priority. The Behaviour layer consists of a set of atomic components represented by transition systems. The second layer describes possible interactions between atomic component. Interactions are set of ports and are specified by a set of connectors. The third layer includes priorities between interactions using mechanisms for conflict resolution. In this paper, we consider Real-Time BIP version [2] where atomic components are represented by timed automata. We limit ourselves to connectors and leave priorities for future work.

From a high-level BIP system model, a physical model containing all TT concepts (such as TT interfaces, the global notion of time and the temporal structure of each task) is generated using a set of source-to-source transformations. This physical model (called also BIP-TT model) is then translated to the programming language specific to the particular TT platform. The program in this language is then compiled using the associated compilation chain. Thus, BIP-TT model is not dedicated to an exemplary architecture.

There have been a number of approaches exposing the relevant features of the underlying architectures at high level design tool. [8] presents a design framework based on UML diagrams for applications running on Time Triggered Architecture(TTA). This approach doesn't support earlier architectural design phase and needs a backward mechanisms for the generated code verification. Since BIP design flow is unique due to its single semantic framework used to support application modelling and to generate correct-by-construction code, many approaches tend to use it to translate high level models into physical models including architectural features. In [5], a distributed BIP model is generated from a high level

one. In [4], a method is presented for generating a mixed hardware/software system model for many-core platforms from an application software and a mapping. These two approaches take advantages from BIP framework but they do not address the TT paradigm. To the best of our knowledge, our approach is the first to address the problem of generating TT application from BIP high level models.

In this paper we address the issue of source-to-source transformations that explicit TT communications in the physical model, in BIP framework. Other TT concepts (the global synchronized time and task temporal structure) transformations are beyond the scope of this paper.

The remainder of this paper is structured as follows: Section 2 introduces BIP framework and explains the relevant TT concepts. Section 3, presents a method using a set of source-to-source transformations for generating a BIP model expliciting TT communication interfaces, from a high level classic BIP model. In Section 4, we conclude the paper by discussing advantages and downsides of our method.

2. RELATED CONCEPTS

In this section, we present first the basic semantic model of BIP, and main TT concepts that must clearly appear in the final BIP-TT model.

2.1 The BIP component framework

In the BIP framework, for each layer, a concrete model is provided. Atomic components model the behaviour layer. The interaction layer is modelled with connectors and finally Priorities is a mechanism for scheduling interactions.

An atomic component consists of a timed automaton with local data and an interface consisting of ports. Transitions in the component automaton are labelled by ports and can execute C code to transform local data. Let $\mathcal{P}$ be a set of ports. We assume that every port $p \in \mathcal{P}$ has an associated data variable x_p. This variable is used to exchange data with other components, when interactions take place.

Definition 1. ***(atomic component):***
An atomic component B is defined by $B = (L, P, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$, where,

- *(L, P, T) is a labelled transition system, that is:*
  - *L is a set of control states*
  - *P is a set of communication ports,*
  - *$T \subseteq L \times 2^P \times L$ is a set of transitions*
- *X is a set of variables and for each transition $\tau \in T$, g_τ is a guard and f_τ is an update function that is state transformer defined on X.*

Interactions which are sets of ports allowing synchronizations between components, are defined and graphically represented by connectors. The execution of interactions may involve transfer of data between the participating components. For every interaction, data transfer functions of an interaction a are specified by an Up and a $Down$ actions. The action Up is supposed to update the local variables of the connector, using the values of variables associated with the ports. Conversely, the action $Down$ is supposed to update the variables associated with the ports, using the values of the connector variables.

Definition 2. *(Connector) A connector γ defines sets of ports of atomic components B_i which can be involved in an interaction a. It is formalized by $\gamma = (P, a, q, g, Up, Down)$ where:*

- *P is the support set of synchronized ports of γ with $P = a$*
- *q is its exported port.*
- *g is the boolean guard expression.*
- *Up is the upward transfer function of the form $x_q := Up(\{x_p\}_{p \in a})$,.*
- *and $Down$ is the downward transfer functions of the form $x_p := Down_p(x_q)$ for each $p \in a$.*

The interaction presented by this connector is of the form:

$$(q \leftarrow a).[g(\{x_p\}_{p \in a}) : x_q := Up(\{x_p\}_{p \in a}) // x_{p \in a} := Down_p^d(x_q)]$$

2.2 TT Paradigm [6, 7]

TT paradigm encompasses these 3 key concepts;

The global synchronized time: It allows definition of instances when communication and computation of tasks take place in a TT system. It is established by a periodic clock synchronization from which other clocks can be derived.

The temporal control structure of the task sequence: The TT paradigm is based on a set of static schedules. These schedules have to provide an implicit synchronization of the tasks at run time. This introduces a fixed task activation rates during system design. Thus to each task is allocated predefined start instant (T_b) and the worst-case termination instant (T_e). These instants are triggered by the progression of the global time.

Time-Triggered interface(Firewall): It is a data-sharing boundary between two communicating subsystems. Exchanged messages are state messages, informing about the state of the relevant variable at a particular point in time. A new version of a state message overwrites the previous version. State messages are not consumed on reading and they are produced periodically at predetermined points in real-time. Thus TT interfaces contain real-time data which is a valid image of the observed variable.

These three notions should clearly appear in the final BIP-TT model to facilitate its translation into the programming language specific to the particular TT platform.

3. TIME-TRIGGERED ARCHITECTURES IN BIP

The methodology that integrates TT concepts in BIP, is based on the transformation of an arbitrary BIP model with additional TT annotations (task, TT interfaces) into more restricted models called BIP-TT, which are suitable for direct transformation into languages of specific TT platforms.

In order to understand the transformation process of a BIP model into BIP-TT one, we present first the original BIP and final BIP-TT models and then we detail the transformation rules that transform the former into the latter.

3.1 The original BIP Model

We assume that the considered original BIP model consists only of atomic components and flat connectors, example cf.Figure1. Indeed, these assumptions do not impose restrictions on the components since we can use the "component flattening" transformation [5] to replace every composite component by its equivalent set of atomic components.

Figure 1 shows a BIP model, made up of five atomic components executing four different tasks. We assume that a task is a set of elementary actions. Thus two or more components can execute separately elementary actions belonging to the same task. Each component is annotated by the

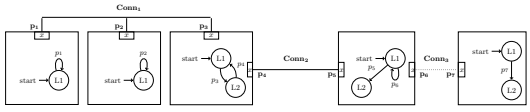
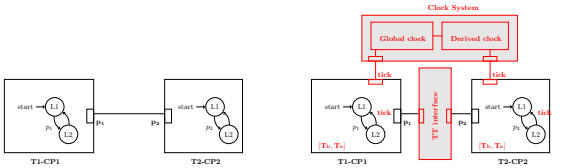


Figure 1: High level BIP model

task it is executing and the component identifier. Take for example the first component, annotated by "T1-CP1", i.e., "CP1" is its identifier and "T1" is the executed task identifier. Two different components may execute the same task, e.g components "CP4" and "CP5". The connector relating such components is shown by dotted lines. To simplify the presentation of figures' automata in this paper, the temporal aspect is not displayed.

3.2 BIP_TT Model

The final BIP-TT Model presents a hand-made translation of the TT paradigm, introduced by Kopetz, into a BIP model. It clearly includes TT three main concepts.
Figure 2b shows roughly how should be the BIP-TT model of the BIP model of Figure 2a. Red components are BIP components and presents TT concepts.



(a) Original BIP model (b) Final BIP-TT model

Figure 2: Modelling TT paradigm in BIP

As here we settle for studying source-to-source transformations to obtain TT interfaces from BIP connectors, we model in Figure 3 the TT interface in BIP. It is an atomic two-port component which behaviour is modelled by a labelled automaton with one state and two transitions, one for reading action (labelled by the port W_{ITT}) and one for writing(labelled by the port R_{ITT}).

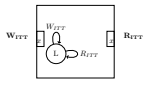


Figure 3: BIP model of the TT interface

3.3 Transformations from BIP classic model to BIP-TT model: from the communication concept point of view

The high level BIP model refinement process is based on the operational semantics of BIP [3] which allows to compute the meaning of a BIP model with simple connectors as a behaviourally equivalent BIP model that contains TT interfaces cf.Figure 3. The transformation process follows these two steps: ***1)*** Transfer functions internalisation and ***2)*** $n+1$-ary connector to TT interface transformation $\mathcal{F}_n$.

These two transformations are described in reverse, from the most specific to the most general $N-$ary connector case. We use the high level BIP model in Figure 1 as a running example throughout the paper to illustrate these transformation rules.

3.3.1 $\mathbf{n+1}$-ary connector to TT interface transformation ($\mathcal{F}_\mathbf{n}$)

This transformation is applied only on $n+1-$ary connector with only one writer, n readers, and with simple assignation transfer functions, i.e., we just copy the value of the associated variable to the writer port in the local variable of the connector (the Up function), and copy the latter in readers' ports' variables($Down$ functions). Note that this behaviour is similar to the TT interface one which is used to make and transfer copy from the producer to consumers. We denote this transformation function by $\mathcal{F}_n$, it transforms an $n+1-$ary connector $C = (P_C, a_C, q_C, g_C, Up_C, Down_C)$, in the source model, into the triplet; binary connector C_1, TT interface I_{TT}, and an $n-$ary connector C_2^n, in the resulting model. These are defined below in function of the initial connector C. Let P_C be the set of ports of the connector C such as $P_C = \{p_{W_C}, \{p_{R_{Ci}}\}_{i\in[1..n]}\}$.

Rule 1. $\mathbf{C_1}$

C_1 is formalized by $C_1 = (P_1, a_1, q_1, g_1, Up_1, Down_1)$. The interaction presented by this connector is then of the form:

$$(q_1 \leftarrow a_1 = \{p_{W_C}, p_{W_{ITT}}\}) \ . \ [g_C(x_{p_{W_C}}) : x_{q_1} := Up_1(x_{p_{W_C}}) = x_{p_{W_C}} // x_{p\in a_1} := Down_1(x_{q_1}) = x_{q_1}]$$

Rule 2. $\mathbf{I_{TT}}$

The atomic component $I_{TT} = (L, P, T, X, \{g_\tau\}_\tau \in T, \{f_\tau\}_\tau \in T)$ where $L = \{l\}$, $P = \{p_{W_{ITT}}, p_{R_{ITT}}\}$, T is a set of the two possible transitions, each labeled by one of the two ports.

Rule 3. $\mathbf{C_2^n}$

C_2^n is formalized by $C_2^n = (P_2^n, a_2, q_2, g_2, Up_2, Down_2)$. The interaction presented by this connector is of the form:

$$(q_2 \leftarrow a_2 = \{p_{R_{ITT}}, \{p_{R_{Ci}}\}_{i\in[1..n]}\}) \ . \ [g_C(\{x_{p_{R_{Ci}}}\}_{i\in[1..n]}) : x_{q_2} := Up_2(x_{p_{R_{ITT}}}) = x_{p_{R_{ITT}}} // x_{p\in a_2} := Down_2(x_{q_2}) = x_{q_2}]$$

Example 1. *If we suppose that there exists only one writer among the first three components in the example of Figure 1 (for example CP1), then this transformation will transform connectors $Conn_1$ (using $\mathcal{F}_2$) and $Conn_2$ (using $\mathcal{F}_1$) as shown in Figure 4.*

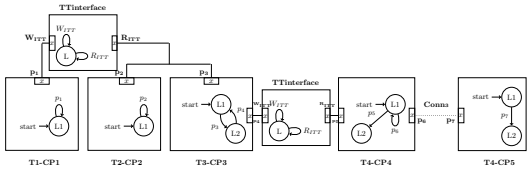


Figure 4: $Conn_1$ and $Conn_2$ connectors to TT interfaces transformation

3.3.2 Transfer functions internalisation

This transformation takes an arbitrary $N-$ary connector with transfer functions different from the simple assignation and produces a connector with simple assignations transfer functions. Then the transformation function $\mathcal{F}_n$ can be applied on the obtained connector. Up and $Down$ functions are internalised by modifying components' automata. In this transformation readers and writers are detected. Suppose that there are m writers, $m \geq 1$ and n readers, $N \leq n+m$, i.e., a component can be both reader and writer. One component writer is randomly chosen to be "the maestro" (in the rest of the paper the maestro is the m^{th} writer). It is then connected to all the rest of writers via $m-1$ binary connectors, so that to aggregate all their data, and to readers via an $n-$ary connector.

Automata of Writers $W_{j,j\in[1,m-1]}$ and readers $R_{i,i\in[1,n]}$, are modified so that to internalize their concerned $Down$ functions. The maestro M component and automaton are modified by adding ports, variables, states and transitions. We denote their refined models respectively by $W^r_{j,j\in[1,m-1]}$,

$R^r_{i,i\in[1,n]}$and M^r. Thus the initial connector $C = (P_C, a_C, q_C, g_C, Up_C, Down_C)$ is split into $m-1$ binary connectors $C^b_{i,i\in[1,m-1]}$ if $m > 1$, and an $n-$ary connector C^n.

We denote the sets of ports and interactions of the initial connector C respectively by $P_C = \{\{p_{W_i}\}_{i\in[1..m]} \bigcup \{p_{R_j}\}_{j\in[1..n]}\}$ and a_C. Ports p_M and p_{R_i} are respectively ports of the maestro and the component R_i involved in the interaction a_C. The derived connectors after transformation and the refined components are defined below.

Rule 4. ***Connector*** $\mathbf{C^b_{i,i\in[1,m-1]}}$

C^b_i is formalized by $C^b_i = (P^b_i, a^b_i, q^b_i, g^b_i, Up^b_i, Down^b_i)$.

The interaction presented by this connector is then of the form: $(q^b_i \leftarrow a^b_i = \{p_{W_i}, p_{M_i}\}) \cdot [g_C(x_{p_{W_i}}) : x_{q^b_i} := Up^b_i(x_{p_{W_i}}) = x_{p_{W_i}} // x_{p_{M_i}} := Down^b_i(x_{q^b_i}) = x_{q^b_i}]$

C^n connector, relating the maestro writer component to the n reader components is defined below;

Rule 5. ***Connector*** $\mathbf{C^n}$

C^n is formalized by $C^n = (P^n, a^n, q^n, g^n, Up^n, Down^n)$, where:

The interaction presented by this connector is then of the form: $(q^n \leftarrow a^n = \{p_M, \{p_{R_j}\}_{j\in[1..n]}\}) \cdot [g_C(\{p_{R_j}\}_{j\in[1..n]}) : x_{q^n} := Up^n(x_{p_M}) = x_{p_M} // x_{p_{R_j}} := Down^n(x_{q^n}) = x_{q^n}]$

We now present how we transform a writer component M in original BIP model, into a maestro component M^r that is capable to aggregate all other writers data, to internalize Up transfer function of the initial connector and then to send the result of this function to readers. The maestro component M^r, has $m-1$ ports p_{M_i} allowing its connection with the rest of writers and a port p_M relating the maestro to readers. Old exported variable x is kept as a local variable, and a new variable z is associated with the port p_M. To be able to internalize the Up function of the initial connector, $m-1$ states and transitions are added before each transition labelled by the port p_M. Each new transition is labelled by a port P_{M_i}. Then Up function is executed in the last new transition. Then, after executing the interaction involving P_M port, we copy z variable to x variable. Figure 5 shows an example of the maestro transformation, in case of a connector with 2 writers $m = 2$, and which transfer functions are Up and $Down$.

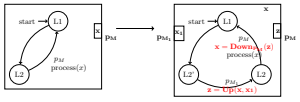


Figure 5: Example of a writer to a maestro transformation with $m = 2$

Example 2. *Based on the example model of figure1, we suppose that the connector $Conn_1$ have the following transfer functions. The transfer functions of the connector C_2 are simple assignations; Up :* $x_{Conn_1} = U(x_{p_1})$ *and* $Down : x_{p_1} = D_1(x_{Conn_1}), x_{p_2} = D_2(x_{Conn_1})$ *and* $x_{p_3} = D_3(x_{Conn_1})$.

By applying the transformation to this connector, we obtain the model of Figure 6. Since the initial model contains just one writer, the connector topology remains intact, only its transfer functions and component behaviours are modified in that example. Functions U and D_1 will be integrated to $CP1$ component. D_2 (resp. D_3) function will be internalized in $CP2$ (resp. $CP3$). In each component, we export a new variable z (instead of x) in ports p_i, $i \in [1,3]$. For down functions (D_1, D_2 and D_3), we add a C function in every transition labelled by port p_i. This function is of the form $x = D_i(z)$, $i \in [1,3]$. Concerning Up function, a state and a transition are added before each transition labelled by the writing port p_1 in the component $T1 - CP1$. The new transition executes a C function of the form $z = U(x)$. The new connector $Conn'_1$ have the following transfer functions: $Up :$ $x_{Conn'_1} = z_{p_1}$ *and* $Down : z_{p_1} = z_{p_2} = z_{p_3} = x_{Conn'_1}$.

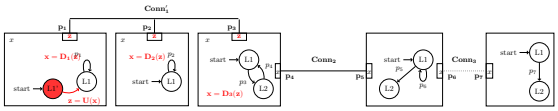


Figure 6: $Conn_1$ transfer functions internalisation

4. DISCUSSION & CONCLUSION

BIP connectors, can be transformed into TT interfaces by successive application of two types of source-to-source transformations; **Transfer functions internalisation** and **n + 1-ary connector to TT interface transformation**. The first simplifies the connector transfer functions by modifying components automata while keeping the same general behaviour of the model. The second transforms connector with simple transfer functions to TT interfaces.

The major asset of these source-to-source transformations, is that we don't add new components requiring adding new tasks, a part from TT interfaces. These transformations focus on transforming atomic components by adding new ports, new variables and extending automata with new states and transitions. The number of added states strongly depends on the number of writers in the model and the number of transitions labeled by the port involved in the interaction.

For that we propose in our future work to study different cases and to decide whether to modify components automata or add a task that orchestrates all interactions without altering components' automata. Then, based on system constraints, a trade-off can be defined.

5. REFERENCES

[1] *BIP2 Documentation*, July 2012.

[2] T. Abdellatif, J. Combaz, and J. Sifakis. Model-based implementation of real-time applications. pages 229–238, May 2010.

[3] A. Basu, P. Bidinger, M. Bozga, and J. Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *Formal Techniques for Networked and Distributed Systems–FORTE 2008*, pages 116–133. Springer, 2008.

[4] P. Bourgos. Rigorous design flow for program-ming manycore platforms.

[5] M. Bozga, M. Jaber, and J. Sifakis. Source-to-source architecture transformation for performance optimization in bip. *Industrial Informatics, IEEE Transactions on*, 6(4):708–718, 2010.

[6] H. Kopetz. The time-triggered approach to real-time system design. *Predictably Dependable Computing Systems. Springer*, 1995.

[7] H. Kopetz. The time-triggered model of computation. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 168–177. IEEE, 1998.

[8] K. D. Nguyen, P. Thiagarajan, and W.-F. Wong. A uml-based design framework for time-triggered applications. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 39–48. IEEE, 2007.

Towards Exploiting Limited Preemptive Scheduling for Partitioned Multicore Systems*

Abhilash Thekkilakattil, Radu Dobrin and Sasikumar Punnekkat

{*abhilash.thekkilakattil, radu.dobrin, sasikumar.punnekkat*}*@mdh.se*

Mälardalen University, Västerås, Sweden

ABSTRACT

Limited-preemptive scheduling has gained popularity due to its ability to guarantee predictable behaviors with respect to preemption overheads on uniprocessors, e.g., minimize cache related preemption delays. However, only few techniques exist that extend limited-preemptive scheduling to multicore systems where the challenge of predictably sharing the cache is further exacerbated.

In this paper, we propose a cache aware partitioning strategy for limited-preemptive real-time tasks. We use the concept of feasibility windows that are specified by a period, relative offset and deadline, and a processor core, to ensure the behavior of the schedule that minimizes cache effects, as well as achieve efficient partitioning. Optimization is used to derive the feasibility windows for non-preemptive regions of the tasks. The feasibility windows can be instantiated to the context of table driven, earliest deadline first or fixed priority based schedulers.

1. INTRODUCTION

The multicore revolution has lead to a revived interest in multiprocessor real-time scheduling. The deployment of multiprocessors in real-time systems is beneficial due to the increased processing capacity that these platforms provide. Many multiprocessor scheduling algorithms have been proposed in the literature that can be broadly classified into global and partitioned multiprocessor scheduling paradigms. Under the global scheduling paradigm, the real-time tasks are dispatched from a global queue based on some rule, e.g., global preemptive Earliest Deadline First. Under the partitioned scheduling paradigm, the real-time tasks are first partitioned onto the available processors, and each processor locally schedules the tasks according to some rule, e.g, partitioned EDF. However, both global and partitioned scheduling paradigms have disadvantages that prevent the total utilization of the processing capacity that multiprocessing platforms provide. For example, global scheduling suffers from the Dhall effect [2] in which deadline misses cannot be avoided even though the taskset utilization is orders of magnitude less than the available processing capacity. Similarly, under partitioned scheduling on an m-processor platform, there exist tasksets that can only use half the available processing capacity (e.g., m tasks each with utilization over 50%) [3]. Such a problem arises because partitioning (bin packing) becomes less efficient when there are high utilization tasks. Several semi-partitioned scheduling algorithms were proposed that build on the best of global and partitioned paradigms by permitting restricted migrations across the processors. However, many of the associated schedulability analyses assume negligible runtime overheads such as preemption related delays, or require over-provisioning of resources.

Multicore processors offer a cheap and flexible multiprocessing platform, that reduces the Size, Weight and Power (SWaP) constraints. The deployment of multicore processors in real-time applications is complicated by various hardware features whose worst-case performance diverges significantly from the average case. For example, the use of shared caches speed-up data access times and in turn the processing speed. However it can significantly increase the overhead in case of a cache miss. Multi-core systems consist of a number of groups of cores that typically share caches at different levels.

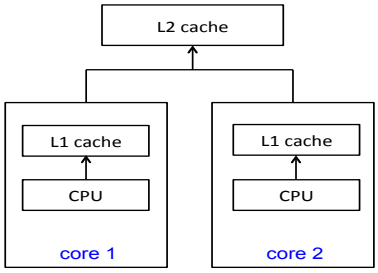


Figure 1: Example of a multicore system.

An example of a multicore system is given in Figure 1, in which cores 1 and 2 share the $L2$ cache. As a consequence of such an architecture, the worst-case execution time (WCET) of a task allocated to a particular core will depend on the allocation of tasks on the other core. For example, in the above figure, if a task A is allocated on core 1, since it is sharing a cache with core 2, task A's WCET may depend on the tasks allocated on core 2. This is because the tasks executing on core 2 can evict the $L2$ cache lines used by task A introducing temporal overheads in the schedule, e.g., Cache Related Preemption Delays (CRPDs), potentially leading to deadline misses. Note that in this case, non-preemptively scheduling task A on any of the core has no particular benefit since a task that is executing in parallel on the other core can invalidate the $L2$ cache for task A. One solution is to 'schedule' the cache preemptively or non-preemptively,

*This work was supported by the Swedish Research Council project CONTESSE (2010-4276).

such as done by Ward et al. [13]. Such an approach implies blocking on some tasks (when using non-preemptive locks on caches) or increased overheads (when preemptively scheduling caches), which can be avoided by slicing the task into subtasks. However, this approach requires modification of the Operating System (OS) to support cache management techniques. Additionally, since tasks are assumed to be already partitioned, the method is less flexible with respect to controlling blocking that occurs when tasks on a particular core have to wait for tasks on another core to unlock the shared cache lines. This is because partitioning tasks makes it difficult to move around the execution windows of the corresponding jobs to minimize blocking. Another approach is to partition the cache among the competing tasks such that each task may use only the cache lines allocated to it [10]. However, such an approach requires compiler support which complicates the system development. Finding alternate methods that do not require extra support, such as modifications to the OS or specialized tools, to make the shared caches more predictable is interesting in cases where the development platforms cannot be modified.

Limited-preemptive scheduling has gained popularity due to the combined benefit of preemptive and non-preemptive scheduling offered by such schedulers [5]. Under fixed preemption point scheduling, the preemptions are restricted to certain locations in the code in order to improve predictability with respect to cache behavior. It may be possible to identify data intensive regions of code that requires the use of $L2$ cache by looking at the code characteristics. Whereas, some regions of code may require only the $L1$ cache. Therefore preemption points can be placed at locations where preemption related overheads are negligible, or to demarcate regions of code accessing the $L2$ cache (e.g., using methods similar to [11]) and consequently enable co-scheduling of non-preemptive regions of all the tasks that access the $L2$ cache. If it is not possible to demarcate such regions, the entire task is tagged as non-preemptive. As noted by Anderson et al. [1], the overheads due to $L1$ cache misses are not significant compared to $L2$ misses. Scheduling non-preemptive regions that access $L2$ cache on the same processor can increase efficiency while decreasing preemption related overheads since they can potentially make use of the $L1$ cache instead. If they cannot make use of the $L1$ cache, temporal partitioning can be enforced so that such non-preemptive regions do not execute in parallel. To summarize, co-scheduling non-preemptive regions has the advantage that it can satisfy the following constraints: a) it can potentially improve allocation and scheduling efficiency since smaller objects enable efficient bin packing b) temporal separation can be enforced for the non-preemptable blocks that access the same $L2$ cache lines without any modifications to the OS c) the non-preemptive regions can potentially share the $L1$ cache instead of the $L2$ cache minimizing $L2$ cache accesses by allocating them to the same processor. Marinho et al. [9] and Davis et al. [6] proposed schedulability analyses for global limited-preemptive FPS. Later, we [12] proposed a schedulability analysis for global limited-preemptive EDF.

In this paper, we propose a methodology to partition non-preemptive regions of a task onto a multicore platform such that the effects of the shared cache ($L2$ in this case) on the execution times are minimized, thereby making the system more predictable. Mathematical optimization is used to minimize the number of processors required to schedule the non-preemptive regions.

2. SYSTEM MODEL

In this paper, we consider a set of n periodic real-time tasks $\Gamma = \{\tau_1, \tau_2, ...\tau_n\}$ to be scheduled on identical multiprocessors. Each τ_i is characterized by an inter-arrival time T_i, and a relative deadline $D_i \leq T_i$, and consists of p_i non-preemptive blocks that have to be executed sequentially. The Worst Case Execution Time (WCET) of the k^{th} non-preemptive block of τ_i, $k \in [1, p_i]$, is denoted by $\beta_{i,k}$, and $\sum_{k=1}^{p_i} \beta_{i,k} = C_i$. This can be calculated using currently available uniprocessor WCET tools [8]. The preemption occurring at the end of a non-preemptive block is assumed to incur negligible preemption overhead or is accounted for in the WCET of the following block. The notation $\beta_{i,k}$ also denotes the k^{th} non-preemptive block of τ_i. Although the blocks need to execute sequentially, different blocks may execute on different cores. The overhead due to these "migrations" can be controlled (minimized) by appropriately placing preemption points (using methods similar to [11]).

3. PROPOSED METHODOLOGY

We build on our previous works [7] to derive feasibility windows for the non-preemptive blocks of every task, that guarantee a specified behavior of the schedule, e.g., timeliness and temporal separation, to achieve a predictable cache behavior. We first present the methodology applied at a task level, and later extend it to job level to finely control partitioning.

DEFINITION 1. *The **feasibility window** of a non-preemptive block $\beta_{i,k}$ of a task τ_i is defined by a time period, relative offset and deadline, and a processor core on which the non-preemptive block $\beta_{i,k}$ must execute in order to guarantee a specified temporal isolation from the non-preemptive blocks that access the same cache lines scheduled on other cores.*

The execution of the non-preemptive blocks within the associated feasibility windows guarantees their non-preemptive execution, efficient partitioning, as well as temporal separation from non-preemptive blocks that access the same cache lines, while requiring no modification to the OS. Since feasibility windows are derived per non-preemptive block, the non-preemptive blocks are converted to artifact tasks. These artifact tasks have task parameters specified by the associated feasibility window, and are scheduled just like normal tasks. The period of an artifact tasks is an integer multiple of the original task.The non-preemptive blocks can be scheduled using the deadlines that specify the associated feasibility window, or fixed priorities can be derived using the method presented in [7].

3.1 Motivating Example

Consider three tasks τ_1, τ_2 and τ_3 as shown in table 1. In order to partition these tasks on a multicore system, 3 processors are required since all the three tasks have utilization greater than 50%. It is now possible to use cache management technique proposed earlier e.g., by [13] to temporally separate the executions of those regions of code that create cache conflicts if executed in parallel. The resulting schedule is given in Figure 2, in which the regions of code that generate cache conflicts when executed in parallel are

Task	C_i	$D_i = T_i$	$\beta_{i,k}$
τ_1	3	5	$\beta_{1,1} = 3$
τ_2	6	10	$\beta_{1,1} = 3$, $\beta_{1,2} = 3$
τ_3	6	10	$\beta_{1,1} = 3$, $\beta_{1,2} = 3$

Table 1: Example task set

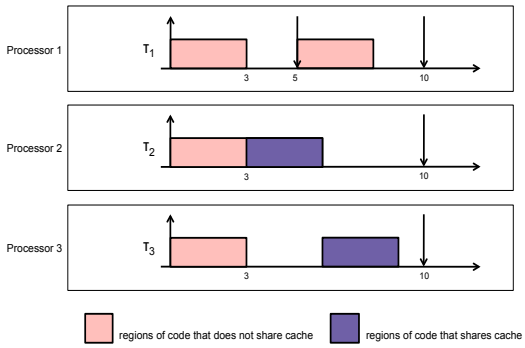


Figure 2: Schedule under Ward et al's [13] method.

highlighted using the darker color. These regions can be temporally seperated as shown in the figure e.g., by using resource sharing protocols [13] or cache partitioning. Such a methodology requires the OS to have control of the cache management.

Instead, our methodology exploits real-time scheduling to guarantee temporal separation by deriving appropriate task parameters such as priorities, release times and deadlines such that the regions of code that accesses the same cache blocks are separated in time.

Consider the partitioning of the example taskset given in table 1. In this case, τ_2 can be split into two artifact tasks each with a period 10, release times 0 and 6 respectively, and deadlines 6 and 10 respectively. These artifacts can be allocated to different processors as shown in Figure 3 The artifact tasks can now be allocated to 2 different processors, instead of 3, to achieve an efficient packing. The resulting tasksets can now be scheduled using standard uniprocessor scheduling algorithms, e.g., EDF, by converting the feasibility windows to job level or task level priorities. This method achieves a predictable cache behavior without the need to modify the OS, e.g., to incorporate support for cache management such as done by [13].

3.2 Methodology Overview

In this section, we present the methodology to derive the feasibility windows for the non-preemptive blocks of every task. We use mathematical optimization to derive the feasibility windows that guarantees the required execution pattern that minimizes cache effects while achieving efficient partitioning of the non-preemptive blocks.

Constraints on the feasibility windows: We first present constraints on the release times and deadlines of $\beta_{i,k}$, denoted by $\gamma_{i,k}$ and $\delta_{i,k}$.

Calculating release times: Every task τ_i has p_i non-preemptive blocks denoted by $\beta_{i,k}$, $k \in [1, p_i]$. The latest start-time (lst) of the last non-preemptive block of τ_i denoted by β_{i,p_i}

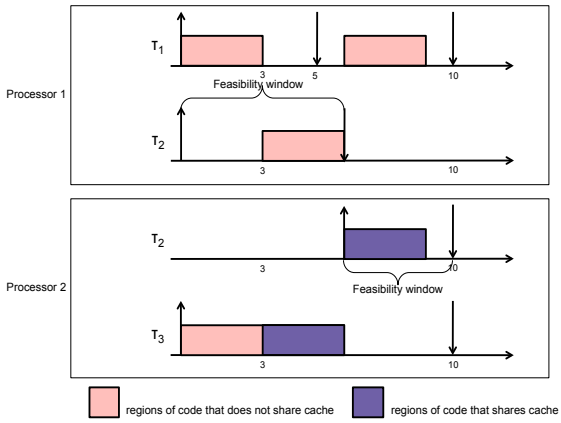


Figure 3: Schedule under our methodology.

is given by:

$$lst(\beta_{i,p_i}) = D_i - \beta_{i,p_i}$$

Only then can the p_i^{th} non-preemptive block complete before the deadline D_i— a deadline miss occurs if it starts later. The p_i^{th} block can immediately start its execution once all the $p_i - 1$ blocks complete their execution, and hence its earliest start-time (est) is given by,

$$est(\beta_{i,p_i}) = \sum_{j=1}^{p_i-1} \beta_{i,j}$$

Similarly, the latest start-time and earliest start-time of any k^{th} such non-preemptive block of τ_i is given by:

$$lst(\beta_{i,k}) = D_i - \sum_{j=k}^{p_i} \beta_{i,j}$$

and

$$est(\beta_{i,k}) = \sum_{j=1}^{k-1} \beta_{i,j}$$

Therefore, the actual release time $\gamma_{i,k}$ of $\beta_{i,k}$ is defined by the following inequality:

$$est(\beta_{i,k}) \leq \gamma_{i,k} \leq lst(\beta_{i,k}) \quad (1)$$

Calculating deadlines: The k^{th} non-preemptive block of τ_i should finish before the release time of the $k + 1^{th}$ block of τ_i. Therefore, the deadline $\delta_{i,k}$ of $\beta_{i,k}$ is given by:

$$\delta_{i,k} \leq \gamma_{i,k+1} \quad (2)$$

In the above equation when $k = p_i$, $\gamma_{i,k+1} = D_i$ to ensure that the deadline of the last non-preemptive block is no greater than the original task τ_i deadline.

Size of feasibility windows: Additionally the feasibility window specified by $\gamma_{i,k}$ and $\delta_{i,k}$ should be large enough to execute the k^{th} block, i.e.,

$$\delta_{i,k} - \gamma_{i,k} \geq \beta_{i,k} \quad (3)$$

Temporal separation guarantees: In order to guarantee temporal separation of blocks $\beta_{i,k}$ and $\beta_{j,l}$, we need to ensure

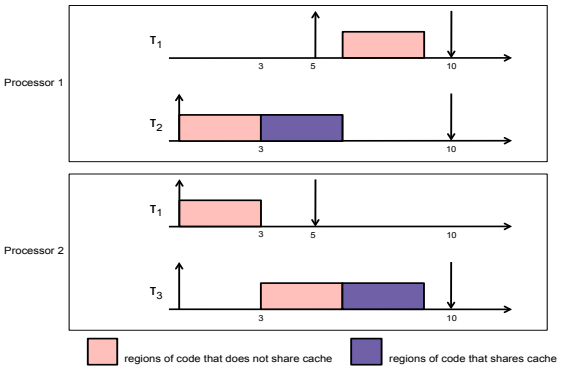


Figure 4: An alternate schedule using job level analysis.

that the release time of either one of the block is later than the deadline of the other, i.e.,

$$\gamma_{i,k} \geq \delta_{j,l} \qquad or \qquad \gamma_{j,l} \geq \delta_{i,k} \tag{4}$$

Optimization: We can now use mathematical optimization along with the constraints on the feasibility windows derived above to partition the non-preemptive regions and derive associated values for all $\gamma_{i,k}$ and $\delta_{i,k}$ of all the tasks. In case sufficient computing power is not available, heuristics can be used to find an appropriate partitioning scheme, and associated feasibility windows specified by the constraints presented above. We do not go into the details of the optimization formulation as it can be formulated using existing techniques (such as done by Baruah and Bini [4]).

3.3 Job level analysis

In this following, we give some insights into adapting the above method to job level, thereby enabling better control on the partitioning. For example, if the system designer wants to avoid splitting of task τ_2 in Table 1, an alternative schedule can be obtained. An example is given in Figure 4 where the jobs of task τ_1 is split into two artifact tasks each with a period 10 and release times 0 and 5 respectively, and deadlines 5 and 10 respectively. By guiding the optimization using additional constraints on the different jobs, many such partitioning schemes can be obtained allowing a system designer to achieve various trade-offs. However, a downside of using a job level analysis is the increase in complexity due to the increased number of constraints. For example, the constraints 1 to 4 should be further broken down into a set of job level constraints, and additional constraints should be added to ensure that all non-preemptive blocks of all jobs of the same task execute on the same processor (e.g., using binary variables that indicate whether they need to be on the same processor).

4. CONCLUDING REMARKS

In this paper, we present a methodology to exploit limited-preemptive scheduling in multicore systems to achieve predictable cache behavior while ensuring efficient partitioning of non-preemptive regions on the cores. Our method can be adapted to a wide range of contexts by defining suitable constraints on the feasibility windows for the non-preemptive regions. Additionally, the task partitioning and scheduling can be finely controlled by breaking down the task level analysis into a job level analysis and specifying additional constraints on the non-preemptive blocks of the different jobs.

The proposed method enables trade-offs between efficiency and complexity.

5. REFERENCES

[1] J. Anderson, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. In *The Real-Time and Embedded Technology and Applications Symposium*, 2006.

[2] B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *The7 th International Conference on Real-Time Computing Systems and Applications*, 2000.

[3] B. Andersson and J. Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50In *The 15th Euromicro Conference on Real-Time Systems*, July 2003.

[4] S. Baruah and E. Bini. Partitioned scheduling of sporadic task systems: an ilp-based approach. In *The International Conference on Design and Architectures for Signal and Image Processing*, 2008.

[5] R. Davis and M. Bertogna. Optimal fixed priority scheduling with deferred pre-emption. In *The Real-Time Systems Symposium*, 2012.

[6] R. Davis, A. Burns, J. Marinho, V. Nelis, S. Petters, and M. Bertogna. Global fixed priority scheduling with deferred pre-emption. In *The International Conference on Embedded and Real-Time Computing Systems and Applications*, 2013.

[7] R. Dobrin, S. Punnekkat, and H. Aysan. Maximizing the fault tolerance capability of fixed priority schedules. In *The International Conference on Embedded and Real-Time Computing Systems and Applications*, 2008.

[8] B. Lisper. Sweet âĂŞ a tool for wcet flow analysis. In *The 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, 2014.

[9] J. Marinho, V. Nelis, S. Petters, M. Bertogna, and R. Davis. Limited pre-emptive global fixed task priority. In *The International Real-time Systems Symposium*, 2013.

[10] F. Mueller. Compiler support for software-based cache partitioning. In *In Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1995.

[11] B. Peng, N. Fisher, and M. Bertogna. Explicit preemption placement for real-timeconditional code. In *The Euromicro Conference on Real-Time Systems*, July 2014.

[12] A. Thekkilakattil, S. Baruah, R. Dobrin, and S. Punnekkat. The global limited preemptive earliest deadline first feasibility of sporadic real-time tasks. In *The Euromicro Conference on Real-Time Systems*, July 2014.

[13] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *The Euromicro Conference on Real-Time Systems*, 2013.

Multi-Criteria Optimization of Hard Real-Time Systems*

Nicolas Roeser, Arno Luppold and Heiko Falk
Institute of Embedded Systems / Real-Time Systems
Ulm University
{nicolas.roeser|arno.luppold|heiko.falk}@uni-ulm.de

ABSTRACT

Modern embedded hard real-time systems often have to comply with several design constraints. On the one hand, the system's execution time has to be provably less than or equal to a given deadline. On the other hand, further constraints may be given with regard to maximum code size and energy consumption due to limited resources. We propose an idea for a compiler-based approach to automatically optimize embedded hard real-time systems with regard to multiple optimization criteria.

1. INTRODUCTION

In hard real-time systems, a task's Worst-Case Execution Time (WCET) must be lower than or equal to its deadline. However, many embedded systems have to comply with additional requirements. Minimizing energy consumption is an issue on mobile systems in order to save battery and allow for longer operation of devices. Additionally, embedded devices commonly have to cope with limited memory and therefore have to adhere to code and data size constraints.

Current optimizing compilers like GCC or LLVM mostly base their optimization techniques on heuristics. They cannot guarantee that the compiled and linked binary will adhere to all constraints. Therefore, the system developer manually has to analyze the resulting binary for compliance with the requirements. If one or more requirements are violated, the developer manually has to modify the compiler input and analyze the system's behaviour iteratively until all requirements are fulfilled. This may be a tedious approach, especially in cases with tight requirements on a given hardware architecture.

In order to ease system development, we provide a concept for an automated compiler-based multi-criteria optimization framework. If the compiler is unable to generate a sufficient solution, it will provide direct feedback, allowing the developer to remove functionality or upgrade hardware capabilities.

Although our approach is flexible enough to be adapted to different constraints, at least those which can be expressed by linear functions, we currently focus on WCET, code size and energy consumption, as we have identified them to be both academically challenging and practically relevant.

The key contributions of this paper are:

- We propose an idea on a compiler-based approach for a multi-criteria optimization framework.
- We illustrate our approach using existing compiler optimization techniques.
- We demonstrate its capabilities by applying it to a synthetic example.

This paper is organized as follows: section 2 gives a brief overview of related projects. Section 3 explains our underlying approach. Section 4 shows an example to illustrate this approach. This paper closes with a conclusion and a view on future challenges.

*This work was partially supported by Deutsche Forschungsgesellschaft (DFG) under grant FA 1017/1-2.

2. RELATED WORK

An optimization method trying to reduce the execution time and size of the code generated by a standard compiler is suggested in [9]: NSGA-II, a genetic algorithm for multi-criteria optimization, chooses which of GCC's optimizations should be enabled. Embedded systems are not targeted, so energy is not a criterion. The proposed iterative method needs several compilations; in contrast, non-heuristic algorithms can provide results in one optimization run.

Li and Malik [4] propose an ILP-based method for the estimation of a program's WCET. Their implicit path enumeration technique (IPET) uses flow constraints to model the program's control flow. The ILP objective function is then set to maximize the execution time while adhering to the flow constraints. While this approach is able to calculate tight and safe approximations for the WCET, the *maximization* objective prevents it from being used as basis for optimizations which try to *minimize* the WCET. Suhendra et al. [8] have solved this issue by implicitly summing up the execution times of basic blocks. Branches are modeled using multiple constraints, and the WCET can be calculated by trying to minimize one ILP variable which holds the execution time of the whole program. This approach will be further described in section 3.

Many embedded platforms feature a scratchpad memory (SPM), which is a relatively small but very fast and energy efficient memory. Steinke et al. [7] show in an example that with a scratchpad of 128 bytes, the total energy consumption of a system (CPU, main memory and scratchpad) can be reduced by two thirds.

Function specialization [6] (also known as function cloning or procedure cloning), as it is used in this paper, is a common basic compiler optimization. If a function is called with one or more known arguments, the compiler can add specialized versions of this function to the program text and replace the corresponding calls to them, which usually helps to reduce the program runtime, because some overhead at call time and inside the specialized functions can be saved.

3. APPROACH

Our approach combines and extends WCET-aware function specialization [5] and SPM allocation [3] to a multi-criteria optimization which optimizes for WCET, energy consumption and code size. SPM allocation affects both energy consumption and WCET with very small influence on the total code size. Function specialization increases code size and may decrease the WCET, but has a negligible effect on energy consumption. These properties are taken into account by the integer linear programming formulas described in the following sections. The ILP program decides which functions should be specialized and which ones are to be put into SPM in order to

- meet the system's timing deadline,
- fit the program into the available memory, and
- minimize the average-case energy consumption of the program.

At the current stage of our work, we only consider single-tasking systems running on memory without any caches. Functions specialized in more than one way are not considered, although adding support for that case to our formulas should be possible by interpreting such specializations as additional functions. We restrict the SPM allocation to function level. Therefore, each function may either be completely located in SPM or in Flash.

Prior to performing our optimization, safe approximations for WCET and average-case energy consumption have to be obtained for each function, both specialized and not, and whether it is located in SPM or in Flash memory. Additionally, the code size must be determined for both the specialized and the original version of each function. This analysis may be performed using tools like AbsInt aiT [1] for the system's WCET or by measurements for the energy consumption. The analysis itself is not part of our approach and will not be discussed any further.

An integer linear programming (ILP) problem is used to model the challenge and calculate an ideal solution. Energy consumption is usually a soft constraint, because hard upper bounds on a system's operational time cannot be easily predicted, so minimization of the average energy consumption is usually chosen as the multi-criteria optimization goal.

Throughout this paper, we use the notational conventions shown in table 1. Capital letters are used for constants in the ILP formulas, while small letters depict ILP variables.

3.1 ILP Description of the System

Suhendra et al. [8] introduce an ILP-based method for optimizing hard real-time systems. In order to model a program's control flow as set of integer linear constraints, the task is split into its basic blocks. A basic block is defined as a set of instructions that must be traversed from top to bottom without any jumps or branches. The control flow is then modeled by calculating the accumulated WCET w_B of a basic block B as the sum of the net execution time c_B of B and the accumulated execution time of its successor. This implicit summation stops at function borders. Therefore, for a block B that calls a function, $w_B = c_B + w_U + w_T$ (U being the succeeding block within B's surrounding function and T being the called function's entry block). Multiple successors can be modeled by using multiple constraints. The ILP's objective function is then defined to minimize the accumulated execution time of the task's entry basic block.

$\mathcal{F}$	set of all functions in the program after function specialization ($\mathcal{F}$ of course always includes the entry point, function `main`),
f	a function,
f_0	a specialized form of function f,
$N_{f,g}$	number of times function f calls function g,
S_f	code size of function f,
s_{Flash}	amount of non-SPM main memory (like flash EEPROM) needed by the program code,
s_{SPM}	amount of SPM needed by the program code,
E_f	energy consumption of function f,
D	deadline of the task,
W_f	accumulated WCET of function f,
r_f	binary decision variable for SPM usage of f,
p_f	binary decision variable for specialization of f.

Table 1: Notational conventions

For an illustration of how the control flow graph is transformed to enable this ILP-based analysis of the WCET, refer to our `bench` example program in section 4: the original control flow graph is shown in figure 1. Figure 2 depicts the final step of the graph transformation. The resulting graph is of course no longer suitable for generation of the program binary, it is solely used for WCET analysis.

3.2 WCET Optimization

ILP-based optimizations like [3] assign individual basic blocks to the SPM. Due to the linearity inherent to ILP and to reduce overall complexity, effects of platform-dependent details like timing costs for jump instructions to and from blocks on the SPM are highly overapproximated. The optimizations usually aim at leading the ILP solver in the right direction to minimize the overall WCET, but due to the overestimated timing penalties, the ILP program's estimation of the system's WCET will be a huge overapproximation. When optimizing for one single criterion, this problem is easily handled by trying to minimize the WCET, and afterwards performing a static WCET analysis to calculate tight WCET estimates. With multi-criteria optimizations, however, we may be using the WCET as a design constraint, and not as the primary optimization goal. As stated above, our approach focuses on minimizing the average-case energy consumption, while still meeting the system's timing constraints. Therefore, instead of minimizing for the WCET, we add an additional constraint to the ILP rules to define an upper bound for the system's maximum execution time:

$$w^*_{\texttt{main}} \leq D \tag{1}$$

$w^*_{\texttt{main}}$ denotes the system's accumulated WCET in the ILP formulation, D is defined as the system's deadline. To avoid the aforementioned overestimation of the system's WCET, we limit the SPM optimization to whole functions. We define a binary decision variable r_f which is set to 1 if function f is located on SPM, and 0 else. We use aiT to calculate each function's WCET $W_{f,\text{SPM}}$ if the function is located in SPM, and $W_{f,\text{Flash}}$ if the function is located in flash memory. These costs contain the execution time of the function f without the costs of any functions which are called by f. This way, we mostly avoid overestimations of the WCET in the ILP formulations and are not bound to the WCET minimization as the ILP's objective function.

Function specialization is handled by introducing another binary decision variable into the ILP problem. We define p_f to be 1 if the function is to be specialized, and 0 else. With this in mind, the accumulated execution time of function f may be written as

$$w_f^* \geq N_{f,f}\left[W_{f,\text{SPM}} \cdot r_f + W_{f,\text{Flash}} \cdot (1 - r_f)\right] + \sum_{g \in \mathcal{C}_f} N_{f,g} \cdot w_g^* \quad (2)$$

for all calls to the unspecialized, original function f, and as

$$\begin{aligned} w_{f_0}^* \geq & N_{f,f}\left[W_{f,\text{SPM}} \cdot r_f\,(1 - p_f) + \right. \\ & W_{f,\text{Flash}} \cdot (1 - r_f)\,(1 - p_f) + \\ & \left. W_{f_0,\text{SPM}} \cdot r_f p_f + W_{f_0,\text{Flash}} \cdot (1 - r_f)\,p_f\right] + \\ & \sum_{g \in \mathcal{C}_f} N_{f,g} \cdot w_g^* \end{aligned} \quad (3)$$

for calls to a potentially specialized version f_0. In both cases, $\mathcal{C}_f$ is defined as the set of functions which are called by f, and is determined by the compiler. If a function g is called in its unspecialized and in a potentially specialized version, g and g_0 are added to $\mathcal{C}_f$. $N_{f,f}$ is 1 for non-recursive functions, otherwise it denotes the maximum recursion depth, as determined by the writer of the task, and e.g. annotated in the source code. If the control flow of a function is split into several branches which call different functions, we use several inequations, one for each possible path. Though equation (3) seems to be of quadratic complexity, the multiplication of binary ILP variables equals a logical AND operation and can easily be linearized, as shown in [2]. $w_{\texttt{main}}^*$ can be used as safe overapproximation of the WCET of the whole task.

3.3 Energy Optimization

To calculate and minimize the average energy consumption of the system, we need detailed knowledge about the execution count of each function. These information may be obtained from simulation or measurements. The maximum energy savings are not necessarily achieved by optimizing the functions which need most energy. Instead, the product of a function's energy consumption E_f and the function's estimated execution count over the operation time of the system needs to be taken into account.

The energy consumption of a function heavily depends on whether the function is located in SPM or not. We assume that energy profiling data is available for both cases. Just as for the WCET formulation, E_f does not include the energy consumed by functions which f calls. The total energy consumption by function f, including all called functions, is given by a formula along the lines of the one presented as equation (3). Again, we directly include the distinction whether a specialized function is called or not.

$$\begin{aligned} e_f^* \geq & N_{f,f} \cdot E_f \\ & + \underbrace{\sum_{g \in \mathcal{F}} N_{f,g} \cdot e_g^*}_{\text{if the call to } g \text{ can not use a specialized form}} \\ & + \underbrace{\sum_{g \in \mathcal{F}} N_{f,g} \cdot \left(e_{g_0}^* \cdot p_g + e_g^* \cdot (1 - p_g)\right)}_{\text{if the call to } g \text{ can use the specialized form } g_0} \end{aligned} \quad (4)$$

With this formula, $e_{\texttt{main}}^*$ is an overapproximation of the system's typical energy consumption.

3.4 Code Size

Choosing to specialize a function increases the overall code size of the program. For each function, we a priori determine the effect on the code size if this function is chosen to be specialized.

In order to make our integer linear program aware of how much SPM and main memory the code takes, we add ILP constraints to model the code size:

$$s_{\text{Flash}} \geq \sum_{f \in \mathcal{F}} S_f \cdot (1 - r_f) + S_{f_0} \cdot p_f \cdot (1 - r_{f_0}) \quad (5)$$

$$s_{\text{SPM}} \geq \sum_{f \in \mathcal{F}} S_f \cdot r_f + S_{f_0} \cdot p_f \cdot r_{f_0} \quad (6)$$

Again, linearization of the ILP formulas has to be performed, cf. equation (3).

Additional constraints can be added to limit the maximum amount of memory that may be used by the program:

$$s_{\text{Flash}} \leq M_{\text{Flash}} \quad (7)$$

$$s_{\text{SPM}} \leq M_{\text{SPM}} \quad (8)$$

With M_{Flash} and M_{SPM} being the physically available SPM and Flash memory.

Finally, the ILP's objective function may be chosen. As stated above, usually the energy consumption is chosen to be minimized:

$$\min\left(e_{\texttt{main}}^*\right) \quad (9)$$

4. EXAMPLE

We will illustrate our approach with a hypothetical benchmark `bench` which is to be run on an embedded microprocessor equipped with 200 bytes of scratchpad memory. Execution of the `bench` task starts with function `main`. This function either calls function `f` twice, or function `g` twice; which function is actually called depends on some external data. If functions are specialized, one call of the two calls to `f`, or `g`, respectively, can use the specialized function. Figure 1 shows the control flow with the basic blocks of the unmodified function `main` in the example program.

Analysis of the `bench` task as described in section 3.2 results in the data given in table 2 as input for our algorithm.

f	`main`	`f`	`g`	`f0`	`g0`
S	100	50	100	50	100
For function in main memory:					
W	50	50	40	30	9
E	50	50	60	50	50
For function in scratchpad memory:					
W	10	10	9	5	5
E	20	20	30	20	20

Table 2: Net WCET and energy values and sizes per function

We assume that collecting profile data over several thousand executions of the task reveals that the two branches in function `main` have very different execution probabilities: the branch calling `f` is executed in 10 out of 1010 cases, the path over the calls to function `g` is executed 1000 out of 1010 times. The probability distribution *non-specialized*:*specialized* for which form of the function is used is 10:10 for `f` and

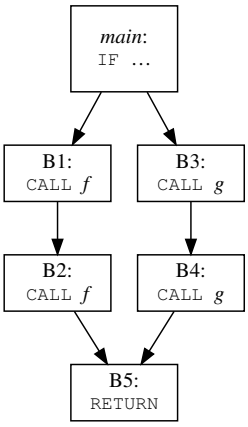


Figure 1: Control flow graph showing the `main` function of the `bench` task and its basic blocks.

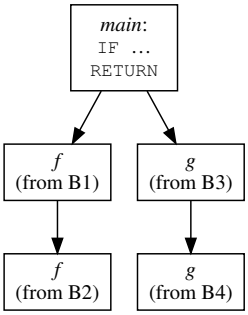


Figure 2: WCET analysis graph for `main`, i.e., the entire task, generated from the original CFG.

200:1800 for `g`. The collected profiling data has no impact on the WCET of the task, which is 150 cycles if no optimizations are applied.

An optimization which solely targets minimization of the WCET will specialize `g` and put `main` and `f`, but not `g` or `g0` (the specialized version of function `g`), into the SPM, which results in a WCET of 59 cycles.

The best solution when optimizing only for energy is to specialize `g` and `f` and put functions `main` and `g0` into SPM. The WCET for the example program optimized in this way is 90 cycles. For comparison, the best WCET we could get if the system had no SPM at all would be 130 cycles.

When applying our multi-criteria optimization algorithm explained in section 3 to the example program, we will now assume the deadline of 70 cycles, until which the task must have completed its work. Table 3 shows the optimization results which are achieved by applying single optimizations or the multi-criteria algorithm with the policy to minimize overall energy consumption. In the case of more than one possible solution which is considered equal by the relevant optimization, the figures in this table have been chosen to show the solution which is also better in terms of the other possible optimization criteria.

Target	WCET	s_{SPM}	s_{Flash}	Energy
unoptimized	150	0	250	85,750
WCET	59	150	200	61,300
Energy	90	200	200	34,600
Multi-criteria	70	150	100	55,450

Table 3: Results of optimizing the `bench` program for different optimization targets (WCET in cycles; energy consumption averaged over 505 executions)

As can be seen, if the WCET is selected as the only optimization target, optimization will produce a program binary that satisfies the deadline, but wastes a lot of energy.

If the system's energy consumption is selected as optimization objective, the execution path which is taken most of the time is optimized, and code size is increased, but the compilation result is still worthless, as it cannot be guaranteed to adhere to the deadline.

If `bench` is not optimized for execution time, code size, and energy consumption at the same time, we get unsatisfying results. Only a multi-criteria optimization like the algorithm presented by us above allows us to create a program which stays within all our optimization constraints.

5. CONCLUSION

We have provided a concept for multi-criteria oriented compiler optimizations which respect both multiple hard constraints and an additional optimization target without any hard upper bound. By means of a synthetical example, we have illustrated the potential benefits of our concept.

In the future, we are going to implement our approach into an existing compiler framework and evaluate its capabilities using real-world benchmarks. We aim at constructing a generic interface for multi-criteria compiler optimizations in the long run which can make use of arbitrary code optimizations. A further challenge is the extension of multi-criteria optimization to multi-tasking systems.

6. REFERENCES

[1] AbsInt Angewandte Informatik GmbH. aiT Worst-Case Execution Time Analyzers. *Available online at* `http://www.absint.com/ait/`, Aug 2014.

[2] J. Bisschop. *AIMMS. Optimization Modeling*. Paragon Decision Technology, Haarlem, The Netherlands, AIMMS 3 edition, June 2009.

[3] H. Falk and J. C. Kleinsorge. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In *Proceedings of the 46th Design Automation Conference*, DAC '09, pages 732–737, New York, NY, USA, 2009. ACM.

[4] Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd Design Automation Conference*, DAC '95, pages 456–461, New York, NY, USA, 1995. ACM.

[5] P. Lokuciejewski, H. Falk, M. Schwarzer, P. Marwedel, and H. Theiling. Influence of Procedure Cloning on WCET Prediction. In *Proceedings of the 5th International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '07, pages 137–142, New York, NY, USA, 2007. ACM.

[6] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 2011.

[7] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, DATE 2002, pages 409–415, 2002.

[8] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proceedings of the 26th Real-Time Systems Symposium*, RTSS 2005, pages 223–232, 2005.

[9] Y.-Q. Zhou and N.-W. Lin. A Study on Optimizing Execution Time and Code Size in Iterative Compilation. In *Proceedings of the 3rd International Conference on Innovations in Bio-Inspired Computing and Applications*, IBICA 2012, pages 104–109, 2012.