PROCEEDINGS OF

OSPERT 2019

the 15th Annual Workshop on
Operating Systems Platforms for
Embedded Real-Time Applications

July 9th, 2019 in Stuttgart, Germany

in conjunction with

the 31st Euromicro Conference on Real-Time Systems
July 10–12, 2019, Stuttgart, Germany

Editors:
Adam LACKORZYNSKI
Daniel LOHMANN

Contents

Message from the Chairs

Welcome to OSPERT'19, the 15th annual workshop on Operating Systems Platforms for Embedded Real-Time Applications. We invite you to join us in participating in a workshop of lively discussions, exchanging ideas about systems issues related to real-time and embedded systems.

The workshop will open with a keynote by David Haworth, discussing deterministic behavior of modern SoCs and telling from his decade-long experience working on real-time and safety operating system development in the automotive industry.

OSPERT19 received 9 submissions from which 6 were selected by the program commitee to be presented at the workshop. Each paper received four individual reviews. Our special thanks go to the program committee, a team of ten experts for volunteering their time and effort to provide useful feedback to the authors, and of course to all the authors for their contributions and hard work.

OSPERT19 would not have been possible without the support of many people. The first thanks are due to Martina Maggio, Steve Goddard, Gerhard Fohler and the whole ECRTS organizing team for entrusting us with organizing OSPERT19, and for their continued support of the workshop. We would also like to thank the chairs of prior editions of the workshop who shaped OSPERT and let it grow into the successful event that it is today.

Last, but not least, we thank you, the audience, for your participation. Through your stimulating questions and lively interest you help to define and improve OSPERT. We hope you will enjoy this day.

The Workshop Chairs,

Adam Lackorzynski
TU Dresden / Kernkonzept
Germany

Daniel Lohmann
Leibniz Universität Hannover
Germany

Program Committee

Marcus Völp, *Université du Luxembourg*
Olaf Spinczyk, *Osnabrück University*
Richard West, *Boston University*
Rudolfo Pellizzoni, *University of Waterloo*
Michal Sojka, *Czech Technical University in Prague*
Wolfgang Mauerer, *OTH Regensburg*
Björn Brandenburg, *MPI-SWS*
Peter Ulbrich, *Friedrich-Alexander-Universität Erlangen-Nürnberg*
Heechul Yun, *University of Kansas*
Jim Anderson, *University of North Carolina at Chapel Hill*

Keynote Talk

Time-domain determinism using modern SoCs

David Haworth
Senior Expert, Embedded OS, Elektrobit Automotive GmbH

Deterministic behavior is important in any control system. It is essential when the control system has implications for safety. Modern high-performance system-on-chip (SoC) devices have multi-level caches and memory management units that can cause software functions to exhibit variable timing characteristics depending not on the functions themselves, but on the state of the hardware due to other software functions that were executed beforehand.

Using data from a recent project along with a historical perspective, the presentation identifies the causes of non-deterministic behavior and shows that poor design of the time-domain behavior of the system amplifies the effects of the non-deterministic nature of the hardware. The result is to suggest that strict control of the time schedules of the system to avoid non-determinism in the software should reduce the effects of the hardware characteristics as well.

ARA: Automatic Instance-Level Analysis in Real-Time Systems

Gerion Entrup, Benedikt Steinmeier, Christian Dietrich
Leibniz University Hannover
{entrup, dietrich}@sra.uni-hannover.de, benedikt.steinmeier@gmail.com

***Abstract*—Real-time control applications are usually implemented by mapping their real-time model (i.e., tasks, shared resources, and external events) onto software instances of RTOS abstractions, such as threads, locks and ISRs. These instantiated objects and their interactions define what actually happens on the imperative machine; they implement the desired behavior. However, during the lifetime of many projects, the initial real-time model gets lost, becomes outdated, or never existed at all, as all (further) development has been code centric: The source code is all that we have. So developers face a situation where further modifications of the real-time system, but also any attempt of static RTOS tailoring, requires the extraction and the understanding of the employed RTOS instances and their concrete interactions from the source code.**

We present ARA, a tool capable of automatically retrieving instance-level knowledge (e.g., the instanciated threads, locks, or ISRs) from a given (real-time) application. ARA is an RTOS-aware static analyzer that derives, given the application source, a graph of the employed RTOS abstractions, their concrete instances, and how these instances interact with each other at run time. We describe the design principles behind ARA and validate its implementation with four example applications for OSEK/AUTOSAR and FreeRTOS.

I. Introduction

In the domain of real-time systems, application development often begins by mapping tasks, shared resources, and external events onto the underlying real-time operating systems (RTOSs) abstractions, like threads, locks, and interrupt service routines (ISRs). This implementation later executes on the actual machine, leaving the application code as *the* ground truth that defines the system behavior. It consists of concrete *instances* that *interact*, mediated by the RTOS, with each other. For example, an externally-activated ISR activates the data-processing thread after data has been received.

Since developers often use only a small part of the RTOS, leaving a lot of functionality unused, application-specific system specialization bears significant improvements [4], [7], [21]. At OSPERT'18, we presented a taxonomy of specialization levels for (real-time) system software [10]. There, we defined three levels on which the RTOS can be specialized: (1) On the level of abstractions, whole RTOS abstractions can be dropped if applications do not need them. (2) On the level of instances, we can use specialized RTOS data structures and algorithms best suited for the known set of instances. (3) On the level of interactions, we specialize the RTOS with knowledge about individual interactions, like having a single queue writer.

We demonstrated with the example of the GPSLogger[1] application, an real-world real-time application, how instance- and interaction-level knowledge can be used to further specialize the underlying FreeRTOS in order to reduce the memory footprint and startup time. With instance-level knowledge, we were able to initialize all stacks, thread-control blocks, and scheduler data structures statically, which reduced the startup time by 10 percent compared to the unmodified system. With interaction-level knowledge, we could embed a short running thread directly into the activating ISR as it did not interact with any other thread. For this, however, we had to manually analyze the source code of the application and how it exactly utilizes the RTOS; a tedious task. We are convinced that manual specialization is infeasible and that an automatic process to retrieve the used instances and their interactions is needed.

As a follow-up of our previous work, we present ARA (**A**utomatic **R**eal-time system **A**nalyzer), a tool[2] that automatically extracts instance-level knowledge. From the application's source code, ARA extracts all application–operating-system interactions and identifies all instances of RTOS abstractions. Furthermore, ARA retrieves the interactions between the instances and provides further system information, like the number of thread activations. As a result, ARA produces the*instance graph*, a data structure that captures instances as nodes and their interaction as edges. For example, for a queue with only one writer, the instance graph contains only a single write edge with the queue node as target.

Thereby, ARA is not restricted to one specific RTOS. Currently, it is able to analyze applications written against the OSEK/AUTOSAR standard [2], an RTOS API with static instances, and FreeRTOS [3], an RTOS with a POSIX-like interface, where all instances are created dynamically. As both RTOSs name their thread implementation "task", we will use thread and task as interchangeable terms.

The knowledge about instances and interactions cannot only be used for specialization but also for other phases of the development process: When new developers join a project, the instance graph becomes a documentation artifact and provides a fast overview of the code base, easing the introduction phase. Since ARA calculates the instance graph in an automated way, the time-consuming manual extraction and updating of an – often outdated – design document is avoided.

This work was partly supported by the German Research Foundation (DFG) under grant no. LO 1719/4-1

[1] https://github.com/grafalex82/GPSLogger

[2] Available at: https://github.com/luhsra/ara

```
ISR i1 {                    .oil
  CATEGORY = 2;
  PRIORITY = 101;
  SOURCE = "PORTA";
}

TASK t1 {
  PRIORITY = 2;
  SCHEDULE = FULL;
}

TASK t2 {
  PRIORITY = 1;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;
}
```

```
BoundedBuffer bb;           .cpp

ISR(i1) { // priority: 101
  bb.put(readSerial());
  ActivateTask(t1);
}

TASK(t1) { // priority: 2
  while(data = bb.get())
    handleSerial(data);
}

TASK(t2) { // priority: 1
  while (true)
    handleADC(readADC());
}
```

List. 1: OSEK example code

Furthermore, the instance graph, which captures the actual usage of RTOS abstractions, can act as a base for further static analyses, like searching for misused RTOS APIs or protocol violations. For example, interaction knowledge makes it possible to check whether calls that take and release a lock occur pairwise. In ARA, we already provide such checks based on the instance graph.

Summarized, the instance graph has three benefits: It serves as a knowledge base for further RTOS specialization. It gives an overview of the application's code base and becomes a living documentation of the program. It provides knowledge to check the application for incorrect or unusual usage of operating system abstractions.

With this paper, we claim the following contributions:

1) We define the *instance graph* as a knowledge base that captures RTOS instances and their interactions.
2) We present automated methods to statically retrieve an instance graph from a given OSEK or FreeRTOS application.
3) We apply our methodology to four real-world applications to validate our approach.

II. System Model

The input of ARA is a statically configured (real-time) system, so the entire application code is known at compile time. In particular, we have chosen two (real-time) operating-system APIs that meet this requirement: OSEK and FreeRTOS.

A. Overview of OSEK

The OSEK standard defines an interface for fixed-priority RTOSs and has been the dominant industry standard for automotive applications for the last two decades.

It offers two main control-flow abstractions: *ISRs* and *tasks*. Additionally, primitives for inter-task synchronization are provided. All instances must be declared statically in the domain-specific OSEK Implementation Language (OIL) [18], [19].

Listing 1 provides an example OSEK system. We see two tasks and one ISR: task `t1` waits for a notification from ISR `i1` and consumes its input, while task `t2` runs constantly and handles the analog-digital converter. All instances are statically declared in an OIL file (printed on the left side). The scheduler starts task `t2` automatically at boot, while task `t1` gets activated by ISR `i1`. It is noteworthy that the used bounded buffer `bb` is not an RTOS abstraction, but used as global data structure.

```
BoundedBuffer bb;
TaskHandle_t t1, t2;

int main() {
  t1 = xTaskCreate(task_1, 2);
  t2 = xTaskCreate(task_2, 1);
  vTaskStartScheduler();
}

isr_1 { // priority: ∞
  data = readSerial();
  bb.put(data);
  vTaskNotifyGiveFromISR(t1);
}
```

```
task_1 { // priority: 2
  while(1) {
     ulTaskNotifyTake();
     while(data = bb.get())
       handleSerial(data);
  }
}

task_2 { // priority: 1
  while (true)
    handleADC(readADC());
}
```

List. 2: FreeRTOS example code

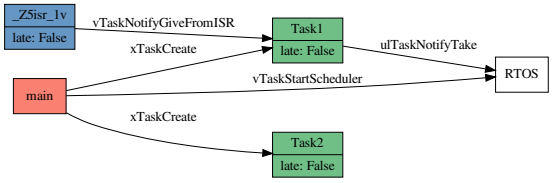

Fig. 1: Instance graph for the given example as generated by ARA. Edge labels always belongs to the edge below them.

B. Overview of FreeRTOS

FreeRTOS is an RTOS stewarded by Amazon to use it together with their cloud instances [11]. One of its core features is the high number of ports to different microcontrollers.

FreeRTOS offers *tasks* as a control-flow abstraction and several synchronization primitives like *semaphores* or *queues*. Unlike OSEK, FreeRTOS does not directly offer an ISR abstraction. Instead, it defines a special class of system calls that can be called from an ISR and they can be recognized by their "FromISR" suffix.

In contrast to OSEK, the FreeRTOS API is dynamic: The application creates all OS instances at run time; either in an initialization phase or during the continued operation. Listing 2 shows the running example using the FreeRTOS API. To foster readability, we have left out some system-call arguments, like the stack size or the name of the created thread (`xTaskCreate()`). Compared to Listing 1, this example contains a main function that sets up the system and starts the scheduler. FreeRTOS is not aware of `isr_1` being an ISR, but we can recognize it by the `vTaskNotifyGiveFromISR` system call.

III. Instance Graph

In this section, we will define the *instance graph* and will present a method to automatically create it. An instance graph describes all instances that will exist in the *whole* application lifetime together with their (flow insensitive) interactions. In Figure 1, we show a simple instance graph that ARA automatically extracted from Listing 2.

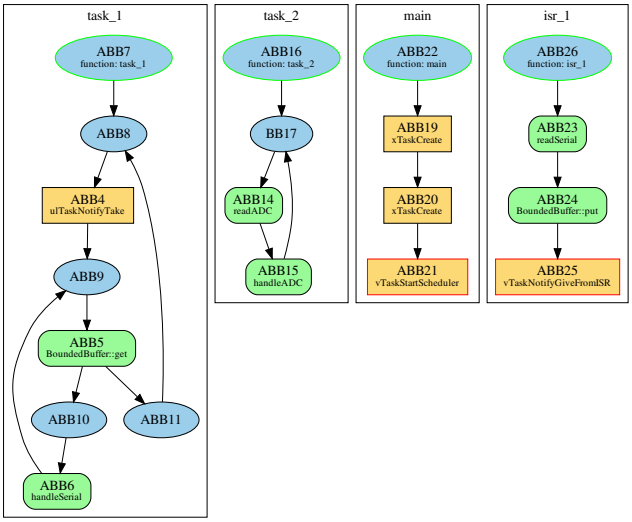

Fig. 2: ICFG for the given example as generated by ARA. System-call blocks are colored in orange (square shape), function-call blocks are colored in green (rounded square shape), computation-blocks are colored in blue (round shape).

The instance graph contains all mentioned instances and their interactions as well as an additional RTOS instance. This pseudo instance collects all interactions that do not take place between two regular instances. Additionally, interactions that originate from the main function reflect the system startup. Since all "late" attributes are set to false, we know that all instances are created before the scheduling begins.

For the construction of the instance graph, two steps are necessary: (1) Building a system-call–aware inter-procedural control flow graph (ICFG). (2) Creating the instance graph based on the ICFG.

A. System-Call Aware Inter-Procedural Control Flow Graph

All interactions between instances originate in system calls; in FreeRTOS also the dynamic instance creation is done via system calls. Therefore, ARA traverses the ICFG of the executed code and then interprets the influence of every system call. For this, it first builds an system-call–centric ICFG that abstracts from the irrelevant code parts.

First, ARA extracts the control-flow graph, which covers the application code with its basic-block nodes. Then, it partitions the control-flow graph into atomic basic blocks (ABBs), a concept introduced by Scheler and Schröder-Preikschat [20], to abstract from the application's micro structure. As an adaptation of the original ABB concept, ARA constructs and connects the ABBs differently, for the whole application at once:

1) Split every basic block (BB) that contains a function or system call. The split is done directly before and after the function or system call. Therefore, all function and system calls reside in their own BB.
2) Each BB gets a type assigned: system-call block, function-call block, or computation block.
3) Merge several computation BBs into a single computation node, if they form a single-entry-single-exit (SE-SE) region, which can only be entered via one distinguished entry BB and left via exactly one exit BB.

Each block constructed with this technique forms an ABB. Afterwards, we have a local ABB-graph for each function within the application code. By assigning a type to every ABBs, we focus on the application logic that is visible to the operating system and all irrelevant computation is subsumed into computation ABBs. Interaction with the kernel is only possible in system-call blocks. Figure 2 shows all application ABBs derived from Listing 2.

Every system call is the intention of an interaction and gets represented by an edge in the instance graph. Usually, the system-call arguments identify the source and the target node together with the exact system-call semantics. Sometimes, the source and target are also defined by the calling instance itself (e.g. the `vTaskDelay` call in FreeRTOS). In order to deduce the system-call arguments, we perform a value analysis: Starting from the call site, we search backwards in the function-local def–use chains and follow the callee–caller relationship if we hit a function beginning. With this interprocedural search, ARA recognizes arguments that have an unambiguous value. In the current implementation, we do not support operations, like an addition with a constant, that change propagated values deterministically.

B. Instance Graph Creation

With the information about the system calls and their arguments, an interpretation of their semantics can be performed. In the first step ARA creates all instances. Here, OSEK and FreeRTOS are handled differently. Since OSEK requires that all instances must be declared in the OIL file, it directly provides information about all instances. As all instances in FreeRTOS are created via system calls, ARA needs to find all instance-creation system calls: It traverses the ICFG, beginning from the system's entry point (usually the main function). Whenever ARA detects an instance-creation system call, it emits a corresponding node in the instance graph. Since it is possible that system calls are invoked in a loop or under a condition, it can happen that the concrete number of actually existing instances cannot be determined ahead of time. This also applies if the system call is contained in another function that is called in a loop or condition. ARA detects such situations, creates exactly one instance, and labels it as being a template for multiple, or an optional, instances.

Additionally, FreeRTOS needs a special handling for ISRs. For all system calls that are recognized as ISR system calls, ARA assumes that they are called within an ISR. To find the actual function that defines the ISR, we traverse the call hierarchy up to a root node.

In FreeRTOS, instance creation can happen anywhere in the application. Therefore, it is important to differentiate whether a creation takes place before or after the start of the scheduler. Since the system entry point is executed exactly once, the code block executed between the entry and the scheduler start gets executed exactly once. For code within a task context, we do

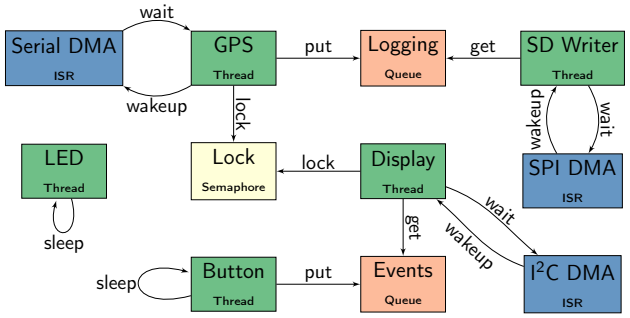

Fig. 3: Manually derived interaction graph for GPSLogger

not have this guarantee, since the task cannot run not at all or it can execute multiple times. Therefore, we analyze all system calls, with respect to their call graphs, and deduce if they are called before or after the start of the scheduler. Also, the call to the scheduler start is a dynamic one. If this call is made in a condition or a loop, no statement can be made. For all instances that are created after the scheduler start, we set the "late" attribute of the instance (see Figure 1).

After the instance creation, we analyze the interactions between them; a step that is equal for FreeRTOS and OSEK. For this, we traverse the ICFG of all tasks and, dependent on the system call, create an edge between the corresponding instances. Since we aim to find all *possible* interactions, the context (loop or condition) of the system call is irrelevant for the edge creation. Interactions whose source or target cannot be determined are assigned to the RTOS instance.

The combination of the system-call–aware ICFG extraction and the subsequent instance and interaction extraction is an automated process and results in the instance graph.

IV. Program Architecture

ARA uses LLVM [14] and reads multiple files in the LLVM intermediate representation as input format (e.g., clang can create these files easily). The initial control flow graph (CFG) extraction into BBs is performed entirely in LLVM. Additionally, analyses already implemented in LLVM, like dominator analysis or loop information, are used. The LLVM-specific and performance-critical parts of ARA are written in C++, while we default to Python for fast prototyping.

V. Experimental Validation

To validate the correctness of ARA, we create instance graphs of four real-world applications: The I4Copter with (1) and without (2) events (based on OSEK), the SmartPlug (3, based on FreeRTOS), and the GPSLogger (4, based on FreeRTOS). The generated instance graphs are rather big and can therefore be found in the appendix.

A. I4Copter

The I4Copter [26] is a safety-critical embedded control system (quadrotor helicopter) developed by the University of Erlangen-Nuremberg in cooperation with Siemens Corporate Technology. We used it as a validation base for OSEK systems. The I4Copter exists in two variants: an implementation for OSEK extended conformance class 1 (ECC1, with events) and another one that runs on the simpler basic conformance class 1 (BCC1, without events). We analyzed both systems with ARA and the instance graphs can be found in Figure 5 and Figure 7. For the event variant, 14 tasks, 4 alarms, 11 events, and 1 resource were identified. From the variant without events, 11 tasks, 4 ISRs, and 1 resource were extracted. We showed the results to an author of the I4Copter who confirmed the results.

B. SmartPlug

The SmartPlug[3] is a hardware power switch controllable via Wi-Fi. It runs on an ESP8266 and uses FreeRTOS to orchestrate its tasks. The project does not provide any building documentation and depends on several unprovided libraries. We therefore replaced all library calls that do not perform any kind of RTOS interaction with stubs. When analyzing the source code, ARA found 11 tasks, 2 queues, 1 semaphore and 1 ISR, presented in Figure 6. ARA detects 4 tasks that are always created and 7 tasks that are created only if some condition is met (indicated by the question mark at the creation system call). We performed a manual validation which confirmed that these optional tasks are created depending on a configuration option, which is retrieved at run time by reading a file.

C. GPSLogger

The GPSLogger is a freely available application to collect GPS information.

It runs on a "STM32 Nucleo-F103RB" evaluation board that is equipped with a STM32F103 MCU. It is connected to a graphical display (I^2C), a GPS receiver (UART), an SD card (SPI), and two buttons (GPIO). Due to a broken SD card library, we had to replace the SD card operations with a `printf()`. In a previous work [10], we created the instance graph manually as shown in Figure 3. The application consists of 5 tasks, 3 ISRs, 2 blocking queues, and one binary semaphore.

The instance graph as created by ARA is shown in Figure 4. Both graphs are almost isomorph. The automatically-derived graph contains an additional main instance to show all creation system calls and an RTOS instance that captures unassignable interactions. As a main difference, ARA detects the ISR interactions but assigns them to the RTOS instance. ARA does this as a fallback, since the correct instance that the `vTaskNotifyGiveFromISR` call gets as argument is not a global variable but derived dynamically. Also, ARA does not detect one interaction of the "Display Task" with the RTOS (`ulTaskNotifyTake`), since it occurs in a function that ARA's reachability analysis cannot find due to an unresolved function-pointer call.

While we saw some specialties in the analyzed systems, we were able to construct instance graphs from all applications. All instance graphs are providing a compact system overview and can be used as knowledge base for further analysis.

[3] https://github.com/KKoovalsky/Smartplug

VI. Discussion

In the previous section, we have seen how ARA can extract interaction graphs from different unknown applications, and we validated the results by comparing them with manually extracted graphs and by manual code inspection. While the ARA approach has a great potential to foster application knowledge, its static nature has some limitations; both aspects will be discussed in the following.

A. Limitations

The main limitation of ARA lays in limitations of its value analysis. On the one hand, this is seen during the extraction of argument values. Values retrieved as result of a function-pointer call or an unambiguous assignment (e.g., in a branch) cannot be retrieved.

On the other hand, ARA does not decide whether to take a branch or how often to execute a loop, so the amount of therein created instances cannot be retrieved. In the current implementation, ARA detects these cases and marks the result appropriately.

In the future, we want improve the recognition by using already implemented compiler techniques such as dead-code elimination and constant folding to remove branches or loop unrolling to determine loop iterations. Another known technique is symbolic execution, which, however, comes with high costs [5]. Nevertheless, we believe that most embedded systems, while programmed against a dynamic API, are rather static in their OS interactions. Mostly, tasks are defined in some kind of main function before the actual scheduler starts and system calls only interpret constant values or global arguments. The analyzed real-world systems are developed this way, except for the SmartPlug, where one task acts as a configuration instance that creates several other tasks. Nevertheless, ARA recognizes this creations but cannot make a statement about the exact amount of instances.

ARA performs a reachability analysis, beginning at the system and task entry points to decide whether an interaction is executed or not. In this analysis, ARA does not resolve any function pointers. In the current implementation, ARA stops the traversal at this point, resulting in possibly unanalyzed system-calls, if they are only reachable via a function pointer. This can lead to unrecognized instances and missing interactions. In the specialization use case unrecognized instances lead to a more generic implementation and thus only to a weaker specialization. However, missing interactions can lead to the selection of the wrong specialization for the corresponding instances and, thus, provoke incorrect system behavior. One way to solve this problem is to retrieve a restricted set of possible call targets by comparing function signatures. A better value analysis will further limit the call-target set of a function pointer. We want to address this limitation in our future work.

The described limitations are inherent for static analysis. Tracing an actual run of the system would circumvent these problems. However, tracing does not detect dynamically *uncreated* instances. This can be seen in the SmartPlug where on an actual run only a subset of all tasks, which are found by static analysis, is created due to dynamic configuration. We plan to extend ARA to additionally support traces.

When we analyzed the real-world applications, we saw different code qualities. Especially the GPSLogger seems like a hobby project that was developed incrementally without a real-time system model. For example, the source base contains two copies of FreeRTOS; both of them are used. Additionally, the analyzed applications are rather small in its code size. We see a threat to validity of ARA's analysis results, that we want to address in our future work with the evaluation of more and larger applications.

ARA considers only interactions that involve the RTOS. For example, ARA does not detect an communication via shared memory like the bounded buffer in Listing 1. Since our main goal is a knowledge gain for RTOS specialization, this is not a limitation. While ARA currently only supports FreeRTOS and OSEK, we plan to extend it to more RTOS APIs.

B. Advantages

Application overview is given by a good visualization of the system composition. This is on the one hand useful to get an overview of the developer's own application as seen by a machine. Often, applications are developed with a program-design model in mind. The instance graph can be used as visualization of this model and prove that it was actually implemented. If the model gets outdated in further development, ARA can serve as a tool to retrieve it in an automated manner. On the other hand, the instance graph is useful as program-design documentation for external developers. Especially for big code bases, it provides the unexperienced developer a compact overview about system composition so she is able to quickly find parts in the source code that are responsible for an observed behavior. ARA is a tool to generate this design document in an automated manner and can, therefore, be integrated into the continuous integration (CI).

Application verification is provided by automatic checks. With knowledge about used operating system (OS) abstractions, ARA is able to check for their correct usage. To demonstrate this, we have implemented two verifications in ARA. The first one checks if an ISR in FreeRTOS only uses ISR-enabled system calls. The second one verifies if system calls to enter and exit a critical region always occur pairwise. Automatic lock verification is a topic of ongoing research [13], [16], [15], [9]. Our approach does not try to verify correct lock usage, but it is able to detect lock misuse. Given the already retrieved instance graph, these checks are easy to implement. Again, this functionality of ARA is useful for a CI process.

Knowledge gain for specialization is achieved in an automated manner. Instance knowledge at compile time can be used to create a specialized variant of the underlying RTOS. For example, instances can be statically initialized and preallocated at compile time. More efficient data structures (like arrays instead of lists) can be used when the number of instances is known. Algorithms can be improved when the communicating instances are known beforehand. For example, queue synchronization can be reduced if only one producer and one consumer is detected.

VII. Related Work

There are several other solutions to statically extract the application knowledge that is required to specialize the RTOS. Bertran et al. [4] track – based on the binary application image – those control system that cross the system-call boundary and eliminate dead system- and library calls from Linux and L4. However, they do not extract instance knowledge or try to interpret the system calls. In [7], we built the global control flow graph (GCFG), which we used for excessive system-call specialization. However, the GCFG captures the system only on a flow-sensitive interaction level (instead of the feature and instance level), proved to be computationally more expensive than ARA, and was only implemented for OSEK. Schirmeier et al. [21] transform the CFG in Kripke structures to be able to apply model checking in computational temporal logic (CTL) for patterns that lead to an automatic OS configuration. They apply the method to eCos and its powerful configuration framework. While CTL may be usable to extract instances, the authors aim to use eCos' existing configuration framework and do not try to extract instances or interactions.

A classical approach to document a program structure are UML diagrams. With StarUML [22], BOUML [6], and ArgoUML [1] several tools exist to automate the diagram generation by performing a static analysis on the application source code. Class diagrams are another program-structure visualization, as generated by Structurizr [24], Structure101 [23], NDepend [17], or Doxygen [8] in an automated fashion. However, all these tools extract no instance knowledge, are control-flow agnostic, and do not consider the RTOS.

Especially for the RTOS domain, several tools like Grasp [12] and Tracealyzer [25] exist that retrieve information from the real-time system to show timing behavior of RTOS instances. Therefore, they build an implicit form of an instance graph but with focus on actual execution times. Nevertheless, they use tracing information to retrieve instances and timing behavior and do not perform any form of static analysis. As a result, they only retrieve all actual executed instances. Instances that are defined in the application but not executed in the trace are not retrieved.

VIII. Conclusion

In this paper, we have presented the instance graph, which is capable of describing all instances of RTOS abstractions together with their interactions. With ARA, we presented a tool to automatically generate an instance graph for applications written against the FreeRTOS or the OSEK API.

We validated the correctness of ARA with four real-world applications and compared the automatically extracted instances graphs to manually extracted knowledge. While having limitations, mainly stemming from the value analysis, ARA was able to recognize all instances and most of its interactions. We have discussed the utility of the instance graph to assist programmers during the application development, to provide an knowledge base for further static analyses, and to foster further RTOS specialization.

References

[1] ArgoUML - Homepage. http://argouml.tigris.org/.

[2] AUTOSAR. Specification of operating system (version 5.1.0). Technical report, Automotive Open System Architecture GbR, February 2013.

[3] Richard Barry. *Using the FreeRTOS Real Time Kernel*. Real Time Engineers Ltd, 2010.

[4] Ramon Bertran, Marisa Gil, Javier Cabezas, Victor Jimenez, Lluis Vilanova, Enric Morancho, and Nacho Navarro. Building a global system view for optimization purposes. In *WIOSCA'06*, Washington, DC, USA, June 2006. IEEE Computer Society Press.

[5] Armin Biere, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. The auspicious couple: Symbolic execution and WCET analysis. In *WCET'13*, 2013.

[6] BOUML - a free UML tool box. https://bouml.fr/.

[7] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. Global optimization of fixed-priority real-time systems by rtos-aware control-flow analysis. *ACM Transactions on Embedded Computing Systems*, 16(2), 2017.

[8] Doxygen - Homepage. http://www.doxygen.nl/index.html.

[9] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP'03*, New York, NY, USA, 2003. ACM Press.

[10] Björn Fiedler, Gerion Entrup, Christian Dietrich, and Daniel Lohmann. Levels of specialization in real-time operating systems. In *OSPERT'18*.

[11] FreeRTOS FAQ - What is the difference between FreeRTOS and Amazon FreeRTOS? https://freertos.org/FAQ_Amazon.html.

[12] Mike Holenderski, Reinder J. Bril, and Johan J. Lukkien. Grasp: Tracing, visualizing and measuring the behavior of real-time systems. In *in Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2010.

[13] D. Hutchins, A. Ballman, and D. Sutherland. C/c++ thread safety analysis. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014.

[14] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO'04*, Washington, DC, USA, March 2004. IEEE Computer Society Press.

[15] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. LockDoc: Trace-based analysis of locking in the Linux kernel. In *EuroSys'19*, New York, NY, USA, March 2019. ACM Press.

[16] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *ICSE'09*, ICSE '09, Washington, DC, USA, 2009. IEEE Computer Society.

[17] NDepend - Homepage. https://www.ndepend.com/.

[18] OSEK/VDX Group. OSEK implementation language specification 2.5. Technical report, OSEK/VDX Group, 2004. http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf, visited 2014-09-29.

[19] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2014-09-29.

[20] Fabian Scheler and Wolfgang Schröder-Preikschat. The RTSC: Leveraging the migration from event-triggered to time-triggered systems. In *ISORC'10*, Washington, DC, USA, May 2010. IEEE Computer Society Press.

[21] Horst Schirmeier, Matthias Bahne, Jochen Streicher, and Olaf Spinczyk. Towards eCos autoconfiguration by static application analysis. In *ACoTA'10*, Antwerp, Belgium, September 2010. CEUR-WS.org.

[22] StarUML Homepage. http://staruml.io/.

[23] Structure101 - Homepage. https://structure101.com/.

[24] Structurizr - Homepage. https://structurizr.com/help/about.

[25] Percepio Tracealyzr - Homepage. https://percepio.com/tracealyzer/.

[26] Peter Ulbrich, Rüdiger Kapitza, Christian Harkort, Reiner Schmid, and Wolfgang Schröder-Preikschat. I4Copter: An adaptable and modular quadrotor platform. In *SAC'11*, New York, NY, USA, 2011. ACM Press.

IX. Appendix

In the following, the instance graphs of all tested real-world examples are shown. We decided to present them here exactly as generated by ARA. Due to their size, they are probably difficult to read on printed paper but, of course, zoomable in digital form.

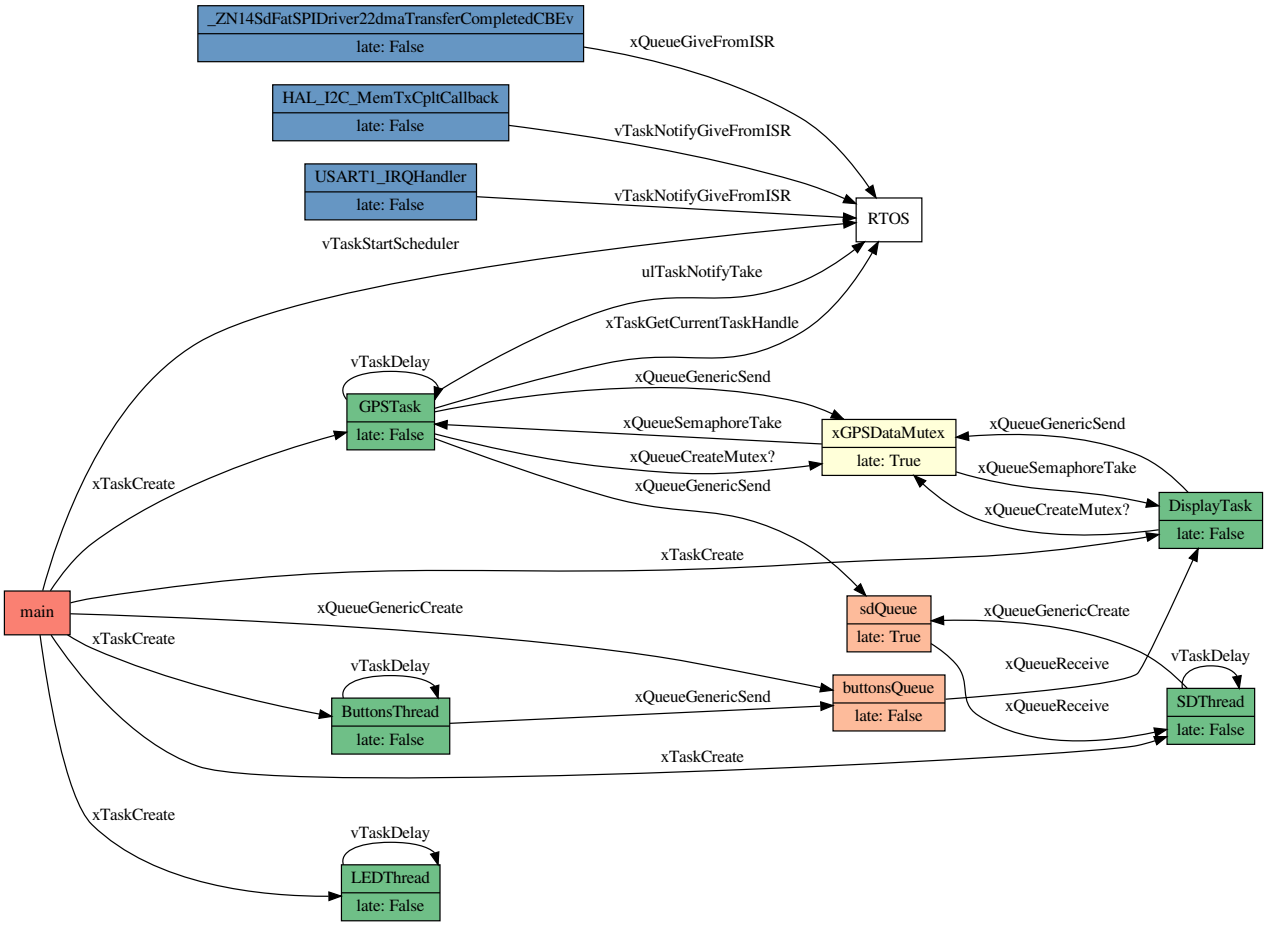

Fig. 4: GPSLogger

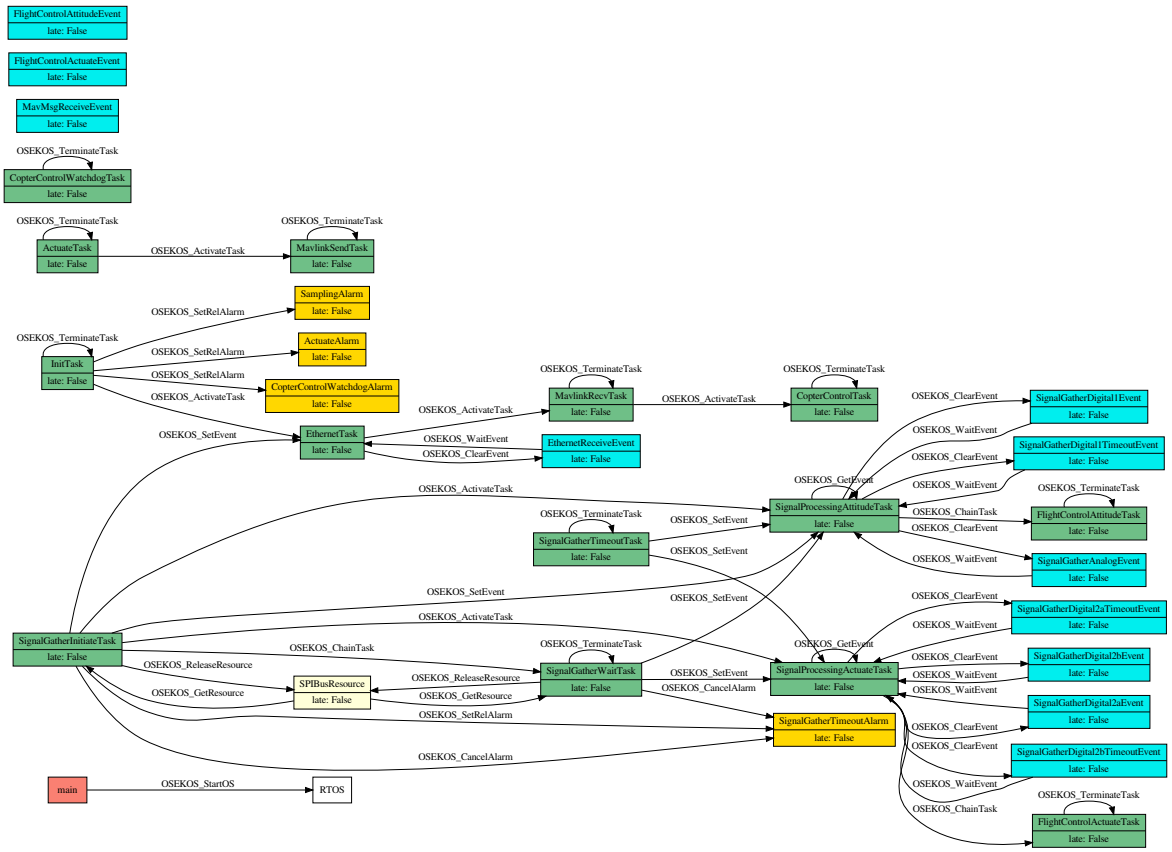

Fig. 5: I4Copter

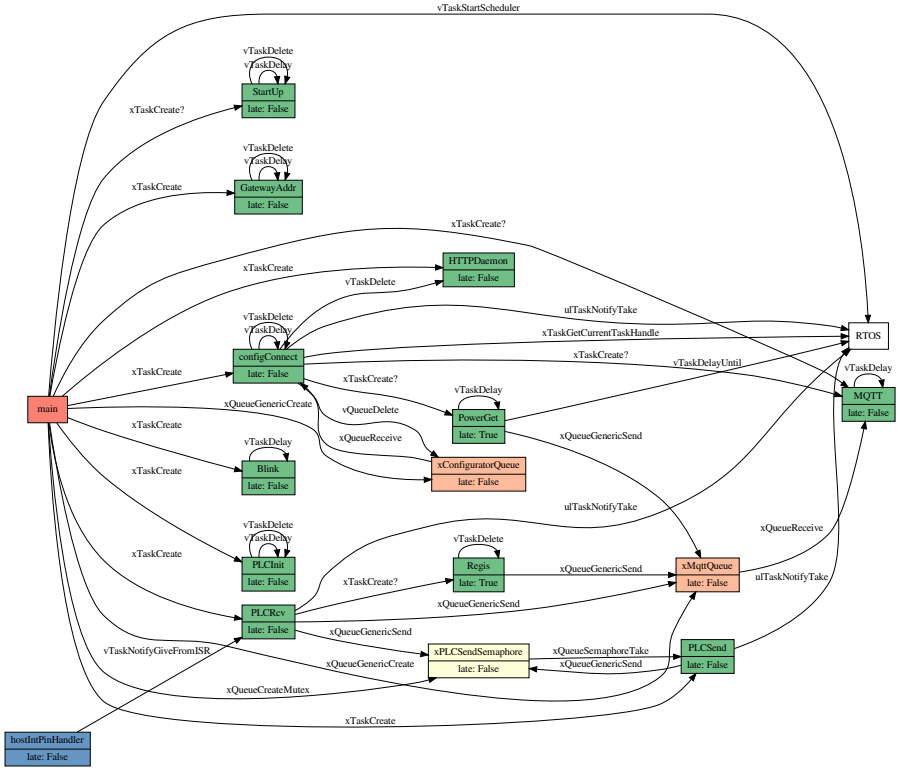

Fig. 6: Smartplug

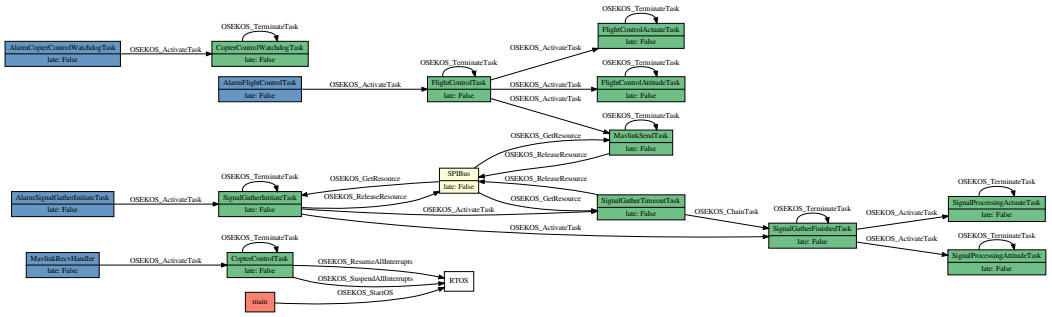

Fig. 7: I4Copter without Events

Boosting Job-Level Migration by Static Analysis

Tobias Klaus, Peter Ulbrich, Phillip Raffeck, Benjamin Frank,
Lisa Wernet, Maxim Ritter von Onciul, Wolfgang Schröder-Preikschat
Department of Computer Science, Distributed Systems and Operating Systems
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

***Abstract*—From an operating system's perspective, task migration is a potent instrument to exploit multi-core processors. Like full preemption, full migration is particularly advantageous as it allows the scheduler to relocate tasks at arbitrary times between cores. However, in real-time systems, migration is accompanied by a tremendous drawback: poor predictability and thus inevitable overapproximations in the worst-case execution-time analysis. This is due to the non-constant size of the tasks' resident set and the costs associated with its transfer between cores. As a result, migration is banned in many real-time systems, regressing the developer to a static allocation of tasks to cores with disadvantageous effects on the overall utilization and schedulability.**

In this paper, we tackle the shortcomings of full migration in real-time systems by reducing the associated costs and increasing its predictability. Our approach is to analyze a task's source code to identify beneficial migration points considering the size of scheduling units and the associated migration costs. Consequently, we can do both: generate schedules that benefit from static migration as well as provide information about advantageous migration points to dynamic scheduling, making full migration more predictable. Our experiments show improved schedulability and a reduction in transfer size of up to 76 percent.

I. Introduction

To date, real-time scheduling on multi-core systems while fully utilizing the cores is a challenging task. Initially, static allocation of tasks to cores suffers from the well-known Dhall's effect [1]: adverse utilization characteristics of tasks may lead to poor overall utilization up to the point where the system becomes unschedulable. Consider for example, a task set comprising three tasks τ_a, τ_b, and τ_c, where τ_a and τ_b have a processor utilization of 70 percent and τ_c of 40 percent. Allocating this task set on a system with two cores is infeasible. This problem of static allocation can be overcome by dynamic allocation and the possibility to migrate work between different cores, as it enables to spread work among cores. Such migration is conceptually possible at multiple levels of granularity: task, job, and instruction level [2]. While the former two are relatively easy to implement, they are incapable of solving the general issue. They still fail to find a feasible schedule for the previous example, as the infeasibility stems from the adversly large utilization on task and job level. Migration on instruction level, on the other hand, succeeds to utilize the system fully and thus to find a feasible schedule, as it allows split up a job and distribute its utilization across cores. An abundance of multi-core scheduling algorithms [3] rely on such fine-grained migration to exploit this potential.

However, migration comes at considerable costs in practice, as the operating system not only has to preempt a task but also transfer its resident set (i.e., active working set) between different core-local memories. In contrast to core-local preemption, these costs are non-constant and highly dependent on the point of migration [4]. Thus, migration at the instruction level carries the risk to migrate at adverse points that are associated with high overheads. Choosing an inapt migration point may even jeopardize deadline tardiness and feasibility [5]. Even worse, worst-case execution time (WCET) analysis is forced to assume a pessimistic bound on the migration cost. To conserve predictability, migration is thus banned in many real-time operating systems, restricting the developer to a static allocation of tasks to cores with disadvantageous effects on the overall utilization and schedulability.

Without resorting to migration, however, the granularity of scheduling units is a crucial factor for the overall schedulability of a given system as it is typically easier to find a valid schedule for smaller scheduling units. For runtime migration, only the direct cost of the migration mechanism itself can be influenced by the operating systems. However, the fundamental issue is in the variability of the indirect cost, that is resident-set size. Consequently, we identified the following two major challenges to overcome the predictability issues and to boost migration in real-time systems.

A. Challenge #1: Adverse Size of Scheduling Units

Considering our previous example, finding a schedule is infeasible only because of the adverse granularity of the three tasks although, in theory, the overall utilization is not exceeded. Splitting tasks into smaller scheduling units solves the problem, for example, by splitting task τ_c into two parts with a utilizaton of less than 30 percent each. However, cutting code to match a certain scheduling granularity is a tedious process as the execution time is a non-functional property and thus hard to correlate with the source code. The fundamental challenge is to identify *split points* that are valid for all possible execution paths across branches and preserve the task's functional properties.

Our Approach: We perform static analysis on the system at compile time to identify potential split points with the desired granularity based on execution cost estimations. Additionally, we ensure that program semantics are preserved across all control-flow branches. We leverage heuristic estimation of execution cost to keep the analysis overhead manageable. Subsequently, a target-specific WCET analysis can verify the granularity of the scheduling units.

B. Challenge #2: Minimize Migration Overhead

Choosing scheduling units only by size still suffers from the same issues as instruction-level migration, namely potentially high migration cost. A split point that results in optimally sized scheduling units may coincide with an unfavorably large resident set. In the worst case, the additional migration cost may again jeopardize the schedulability gained by changing the granularity in the first place. The challenge is in the identification of split points that benefit both aspects.

Our Approach: We optimize both the scheduling-unit size and the associated migration cost simultaneously by extending the search for split point candidates to the vicinity of the optimal scheduling-unit granularity. This way, we are able to choose split points with beneficial resident-set sizes.

C. Contribution and Paper Structure

In this paper, we present an approach to facilitate the use of migration in real-time systems by reducing the associated costs and increasing its predictability. Our toolchain allows for automated static analysis of existing source code to identify beneficial migration points by simultaneously considering both the size of scheduling units and the associated migration costs. We transform the control-flow graph into a split-point graph that models scheduling units and holds information on potential split points and their costs. Consequently, our analysis can be used to both generate static schedules with migration as well as provide hints on beneficial migration points to dynamic scheduling thus supporting online migration.

The remainder of this paper is structured as follows: In Section II, we present our approach to solve the aforementioned challenges. Section III gives an overview of the implementation of our prototype, which is evaluated in Section IV. Section V outlines related work in the context of job migration and Section VI concludes.

II. Approach

In this section, we detail our approach to the identification of beneficial migration points by static code analysis. We, therefore, first introduce the system model and fundamental assumptions that we demand and give an overview of the general concept of our analysis. Finally, we detail the handling of loops and branches, which are in particular challenging and require specific cutting schemes.

A. System Model and Assumptions

We consider a real-time system with m cores that allows full preemption and full migration. We define the latter to permit migration at each instruction of the application code but to prohibit migration during the execution of system calls and other operating-system code. A set τ of n sporadic tasks with occurrence rate T_i is scheduled on m processors. Each task τ_i contains a set of l *scheduling units* J (a.k.a. jobs) and has a processor utilisation U_i. The number of tasks n, the period and their respective *worst-case execution time* (WCET) C_i determine the theoretical schedulability. A system is theoretically schedulable on m cores if the total utilization of all tasks is less or equal than the number of cores m, i.e., if the following equation holds:

$$\sum_{i=1}^{n} U_i = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq m \text{ with } C_i = \sum_{k=1}^{l} C_i^{J_k} \tag{1}$$

Here, $C_i^{J_k}$ denotes the WCET of the scheduling unit k of τ_i. We extend the inequality by the overhead α to express the (data transfer) costs associated with migration for each task:

$$\sum_{i=1}^{n} \frac{C_i + \alpha_i}{T_i} \leq m \tag{2}$$

As a non-functional requirement, we assume that upper bounds on the number of iterations for all loops are given. We further assume a RISC processor without out-of-order execution as the target and that cache-related overhead is already covered by the overapproximations required to incorporate preemption effects and delays.

With the notion of a *resident set*, we refer to the currently active (i.e., alive) part of a process's working set; the latter is often used interchangeably in the literature. That is, for example, local and global variables or the state of the stack. We restrict this definition to comprise only core-local data and assume that all other data is globally accessible.

B. General Concept

We leverage static code analysis to identify interactions with the operating system's scheduler, that is system calls, from the tasks' source code. By incorporating knowledge about the semantics of the targeted operating system and its scheduling, we can thereof derive all truly existing scheduling units and their respective control-flow graphs; irrespective of the development model (e.g., process or run-to-completion) and style the developer pursued.

We further infer all additional information that is required to identify potential split points. First and foremost, this is the active resident set at all times. We, therefore, perform a liveness analysis of all local variables and compute the resident-set size for every instruction. Furthermore, we perform a heuristic timing analysis of all nodes to estimate their size in terms of execution time. Finally, the control-flow graphs are transformed into *split-point graphs* that hold all additional information. In this graph, edges correspond to the possible split points, and nodes represent all instructions between them.

The identification of split points consecutively transforms the split-point graph such that all nodes are at or below a predefined target size at minimal migration costs. To achieve this, we assess each possible split point according to two criteria: distance (δ) to the intrinsic split point and the associated resident-set size (ω), that is migration costs. Figure 1 illustrates the interplay of these two parameters. Recall that we assume global variables to be globally accessible from all cores.

We define the intrinsic split-point as the point, where the estimated worst-case execution time since the beginning of the scheduling unit, or, equivalently, the last split point, approximately equals the target scheduling-unit size. In our

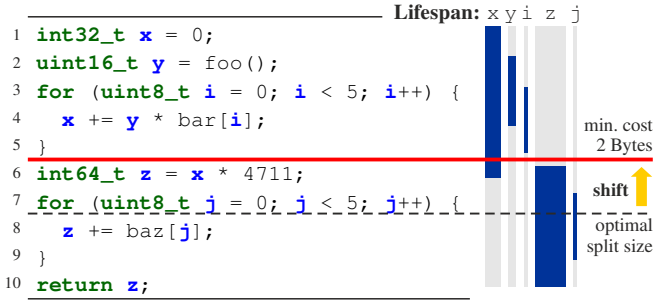

Figure 1: Working set and lifespan of automatic variables, showcasing the need to simultaneously consider both scheduling-unit size and resident-set size to obtain optimal split points with minimal overhead.

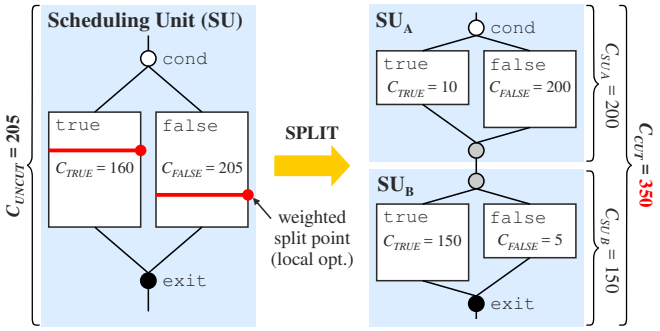

Figure 2: Bloated WCET estimate due to an unbalanced cut, caused by a branch-local optimization of split points.

example, splitting between lines seven and eight would lead to optimal size. However, in this case, we suffer from significant migration costs, as `j` and `z` have to be transferred.

We, therefore, employ the size of the resident set at a given program point to further consider the migration cost. We acquire ω by a liveness analysis to identify all local variables that are referencable from a given program point. In the example in Figure 1, the state is minimal between lines five and six, where just the intermediate result stored in `x` has to be transferred.

By simultaneously optimizing both criteria, we are, in general, able to obtain best-suited points for cutting the scheduling unit locally. However, we have to assess possible split points also from a global point of view. Only by that, we can guarantee that the resulting cut is both correct and suitable in cases where different possibilities of program flow exist, for example in branches and loops. Therefore, we search for a minimal cut on the split-point graph to find split points in all branches that result in global split points associated with minimal migration costs.

In the following sections, we give further insights on how our approach handles specific program constructs.

C. Splitting Branches

Assessing a sequential control flow according to our criteria only requires a straightforward assessment of all split-point candidates and selection of the optimal one. In contrast, branches are harder to split as we need to maintain global relations between split points in all branches belonging to the same conditional construct to preserve program semantics. A further challenge is to avoid an increase in the overall WCET by an unbalanced subdivision of branches. Figure 2 illustrates the underlying problem: For the original, uncut branch (left), we have a WCET of $C_{uncut} = 205$. In this example, the liveness analysis reveals minimal migration costs at the beginning of the `true` and the end of the `false` branch. Consequently, the cut scheduling units SU_A and SU_B contain the greater part of the `true` and `false` branches respectively. Ultimately, the WCET analysis suffers from a pessimistic overapproximation in both branches yielding an overall WCET of $C_{cut} = 350$.

This unfavorable behavior is rooted in the branch-local optimization of the migration cost. We solve this issue by considering both criteria (i.e., size and migration cost) simultaneously across all branches to find globally suitable split points that lead to a balanced cut even for nested branches. Throughout this paper, we call this a *horizontal cut*.

D. Splitting Loops

Further measures are required for the splitting of loops as splitting inside the loop body is entirely ineffective as the related costs outweigh the benefits. Additionally, an individual loop iteration typically contributes only a small fraction to the loop's overall WCET. Therefore, we subdivide loops at a granularity of whole iterations using index set splitting [6] to separate the index range of a loop into smaller subranges until the individual execution time fits the targeted size. As the resident-set size is usually the same for all loop iterations, we consider all possible split points as equally suited[1].

In general, we require an upper bound on the number of loop iterations $iter_{max}$ to estimate a loop's overall execution time, which is commonly available knowledge in (safety-critical) real-time systems. Considering the WCET C_{loop} of a single loop iteration, we can derive the number of iterations $iter_{fit}$ required to fit the target size (C_{target}) of the scheduling units:

$$iter_{fit} = \lceil C_{target} / C_{loop} \rceil \tag{3}$$

The number of required cuts n_{cut} results from the total number of iterations $iter_{max}$ and the fitting size $iter_{fit}$:

$$n_{cut} = \lfloor iter_{max} / iter_{fit} \rfloor \tag{4}$$

By splitting loops by their index range, we obtain suitable scheduling-unit size while avoiding the potentially disadvantageous effects of splitting the loop body.

In summary, by employing our concept of split-point graphs, we can successfully subdivide tasks into scheduling units of smaller size. We can efficiently cut both composite branches and loops. Liveness analysis allows us to identify split points with minimal migration costs. Thereby, we improve schedulability in multi-core settings and reduce the otherwise inevitable overapproximation of indirect migration overheads.

[1]Theoretically, loop constructs exist that violate this assumption. In that case, we overapproximate the resident-set size by the overall maximum.

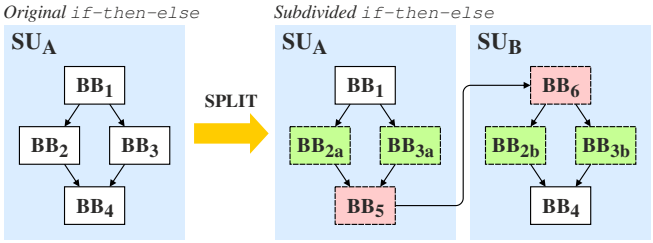

Figure 3: Schematic depiction of the split procedure for `if-then-else` branches.

III. Implementation

We based the implementation of our approach on the *Real-Time Systems Compiler* (RTSC) [7]. The RTSC is an LLVM-based [8] toolchain that's characteristic feature is the automated manipulation of non-functional properties of real-time systems. For this purpose, the RTSC employs static code analysis to transform a given source system into an OS-agnostic intermediate representation, which in turn is based on the LLVM intermediate representation (LLVM-IR). As a result of this, the application's program-flow is represented as single-entry single-exit regions[2] between system calls, that is scheduling units that do not affect the internal state of the OS. Based on this intermediate representation, the RTSC facilitates the systematic manipulation of non-functional properties. For example, a conversion from event to time-triggered execution. Currently, the RTSC supports various real-time operating systems with its front and backends respectively [10], [11].

A. Split-Point Graph Generation

Currently, only the application code is part of the intermediate system representation. Thus, with our prototype, we focused on the identification of suitable split point within the application, which is the main subject of migration. Conceptually, however, our approach can be extended to the operating-system code, which we, however, consider as future work. To identify optimal split points, we need to derive the split-point graph from the intermediate representation and enrich it with execution time estimates and resident-set sizes.

In a first step, we perform a liveness analysis of all variables on the intermediate representation of LLVM. The results of this analysis provide information about the size of the resident-set size ω, which we then utilize as one optimization criterion.

In a second step, we estimate the execution time per instruction in the LLVM representation[3] to determine the distance δ from the intrinsic split points for each instruction. For this estimation, we rely on a simple model of the execution platform and assign execution costs to each instruction, giving more weight to classes of instruction, which are likely to be more expensive, for example, load and store instructions. This heuristic oversimplifies the process of modeling the WCET but is a reasonable approximation at the abstraction level of the intermediate representation. Deriving better estimates for the execution time is beyond the scope of this paper.

[2]A concept introduced as *Atomic Basic Blocks* (ABB) in [9].

[3]We inline all function calls to ease execution time estimation.

B. Split Point Optimization

With the foundations laid, we can assess the local aptness of potential split points by a cost function that combines our two criteria with appropriate weights as shown in Equation 5. δ thereby denotes the distance to the intrinsic scheduling granularity, ω the migration costs in bits, and w_δ and w_ω the respective weights. Using complex weight functions, the interaction of weights and their effect on the assessment of split points can be influenced, for example, to exponentially punish an increasing distance δ. For simplicity, we resort to a constant factor of one for both weights in our prototype:

$$w_\delta \cdot \delta + w_\omega \cdot \omega \tag{5}$$

In the case of sequential code, we can directly choose the point that is most suitable according to our cost function. However, as outlined in Section II, in the presence of loops and branches, we have to perform the global assessment of all branches to ensure a horizontal cut. For the search for the minimal horizontal cut among the split-point candidates in different branches, we apply the Ford-Fulkersson algorithm [12] on the split-point graph. After having identified a globally suitable split point across all branches, we then perform the actual splitting in `if-then-else` branches as depicted in Figure 3. At the split point in each branch, we cut the existing basic block (e.g., BB_2) in two parts by inserting instructions around the split point. In the first part (e.g. BB_{2a}), we set a flag indicating we executed that branch and jump to a newly created basic block terminating the first scheduling unit (BB_5). In the second part (e.g., BB_{2b}), we introduce a label which we can use as a jump target from the newly created entry basic block (BB_6) of the second scheduling unit.

For loops, we use the procedure shown at the example in Listings 1 and 2. Using the method outlined in Section II-D, we identify the number of loop iterations $iter_{fit}$ that should constitute one scheduling unit (5 in the example). We then split the loop by duplicating the body and adjusting the loop condition to preserve the program semantics. For this, we introduce a counter for the split loops (`sCo`) that has to be less than $iter_{fit}$ for the loop condition to be evaluated to `true`.

IV. Evaluation

To assess our approach's ability to cope with the initial two challenges, its real-world usability ,and runtime, we conducted two experiments and one theoretical consideration.

```
1 LOOP_Bound(x:10);
2 for(int i = 0;
3  i < x; ++i)
4 {
5  ....
6 }
```

Listing 1: Loop in the original state.

```
1  int i = 0, C = 5;
2  for(; i < x && C; ++i) {
3   --C;
4   ....
5  }
6   ....
7   C = 5;
8  for(; i < x && C; ++i) {
9   --C;
10  ....
11 }
```

Listing 2: Loop after the splitting procedure.

A. Runtime Overheads of Splitting

Since splitting of scheduling units is realised by the insertion of specific instructions, e.g., jump statements, at the designated cut point, it comes with an additional overhead. This overhead differs depending on the type of control flow. In the case of sequential control flow, we insert only one instruction per cut. The number of additional instructions for the splitting of branches i^+_{if} depends on the number of cuts n_{cut} and the number of branches n_{branch}. The first term of the sum shown in Equation 6 refers to instructions for setting a flag marking the active branch. The second term represents the jump instruction that terminates the first scheduling unit and the third term contains instructions for checking the stored flag and proceeding with the correct branch.

$$i^+_{if} = n_{cut} * n_{branch} * 2 + n_{cut} * 1 + n_{cut} * 3 \quad (6)$$

Splitting loops adds i^+_{loop} instructions to the instructions of the original loops. The first term of Equation 7 refers to instructions necessary to maintain an additional counter for the iterations planned in each scheduling unit of the loop. The second term corresponds to the end of each scheduling unit at a split point and comprises instructions for exiting the scheduling unit and resetting the additional iteration counter. The third term contains instructions to decide whether to execute the following part of the loop after each split point.

$$i^+_{loop} = (n_{cut} + 1) * 5 + n_{cut} * 2 + n_{cut} * 3 \quad (7)$$

The runtime overhead introduced by splitting is therefore minor compared to the execution time of most scheduling units. Exceptions are the extreme cases of conditional constructs with a large number of branches and loops with small bodies. However, we can detect and avoid these cases in the search for suitable split points, which we consider future work.

B. Schedulability of Synthetic Benchmarks

We assessed the validity of our approach concerning schedulability by generating 12000 synthetic benchmark systems with a utilization between 3.5 and 4.0 comprising few OSEK-compliant tasks. We tried to find a feasible allocation and schedule for each task set on a system with four processor cores by employing specialized versions [13] of the branch-and-bound allocation algorithm and the minimax scheduling algorithm, both by Peng et al. [14]. Figure 4 shows the achieved relative schedulability of the generated task sets both with (left bars) and without (right bars) automated splitting of scheduling units. The results show that the relative schedulability increases through splitting, leading to 70 percent more schedulable task sets for the highest utilization. This confirms the general capability of our approach to employ migration automatically to achieve better utilization.

C. Minimize Migration Costs

To evaluate our approach regarding Challenge #2, we analyzed possible splitting points in real-world benchmarks taken from the TACLeBench suite [15]. For this, we converted the benchmarks to OSEK tasks and created OSEK systems

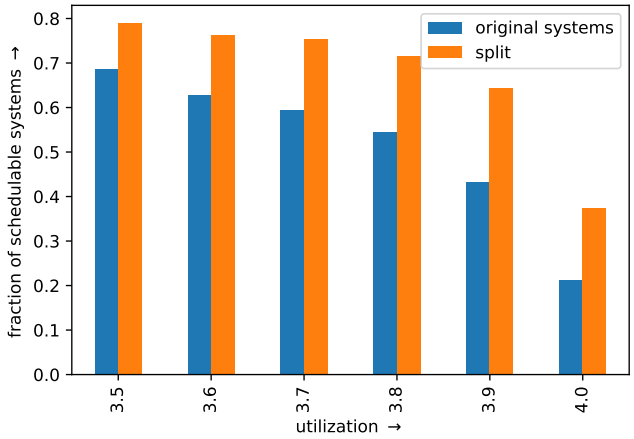

Figure 4: Relative schedulability of generated task sets achieved with (left bars) and without (right bars) splitting of scheduling units, showing an increased schedulability reached through automated splitting.

comprising one benchmark task and two load tasks. We adjusted the amount of load to achieve a system which is unschedulable on two cores without migration to force our RTSC extension to split the benchmark task. We recorded the worst-case dynamic migration costs for every instruction as well as the statically composed migration costs of split points that comprise all branches of the horizontal cut. Table I shows an overview of both the worst-case migration cost observed in all possible split-point candidates as well as the migration cost of the split point chosen by our approach for several TACLeBench benchmarks we analyzed. The costs represent the size of the resident set in bits based on LLVM IR types, which allow for a variable type width from 1 to $2^{23} - 1$ bits. The results indicate that our approach is capable of providing worst-case migration costs for a whole horizontal split that are beneath the dynamic worst-case of single branches, with a reduction of up to 76 percent. These improvements in worst-case migration overhead ultimately allow reducing the pessimism in the response-time analysis.

V. Related Work

Studies on migration in real-time systems agree that migration costs vary significantly with the resident-set size [4], [16],

Benchmark	Worst-case Resident-set Size [bits]	Split-point Resident-set Size [bits]
binarysearch	225	224
bitonic	65	64
complex_update	480	288
countnegative	2176	1568
filterbank	60 736	60 704
iir	432	400
insertsort	544	128
minver	17 568	16 800
petrinet	5057	5056

Table I: Comparison of the worst-case dynamic resident-set size and the resident-set size at the split point chosen by our approach in bits on the basis of LLVM IR types.

[17]. There exists a large body of related work on scheduling algorithms [2], [18] that assume a constant upper bound on migration overheads. To increase predictability and reduce costs, various approaches focused on restricting preemption and finding thresholds or placements for preemption points [17], [19]–[23]. Anderson et. al [24] extended this concept to restricted migration, but lack a practical implementation. Automatic analysis and generation of multi-core systems [13], [25], [26] for non-preemptive scheduling has been studied. All these approaches either ignore migration or are accompanied by substantial pessimism, yet they provide a good starting point for the practical application of our migration hits at runtime.

Orthogonal to our approach of (horizontal) splitting is (vertical) slicing of real-time applications. Here, the time-sensitive code is separated from time-insensitive code to enhance schedulability [27]. In contrast to our approach, it is difficult to control and optimize the output of code slicing timing-wise. In a safety-critical systems, checkpointing [28], [29] is considered to partition tasks, wherein the state to be saved is minimized. However, checkpointing is specific to the application and not universally applicable. Sarkar et al. [5] proposed hardware-assisted migration, from which our approach would also benefit but on which it does not depend.

VI. Conclusion & Outlook

In this paper, we presented an approach to boost migration in real-time systems by an automated analysis of tasks at the source-code level. Our analysis reveals beneficial split points with minimal migration costs. On the one hand, this knowledge can be exploited by the RTOS at runtime and thus is an enabler for migration thresholds; analogous to preemptive thresholds or limited preemptive scheduling [20]. Consequently, static WCET analysis can infer tighter bounds on migration overheads. On the other hand, our toolchain supports the subdivision of tasks into smaller scheduling units already at compile time, thereby improving overall schedulability of static allocation schemes.

We continue to make migration in real-time systems more accessible and predictable and are currently working on the following topics: (1) Improving the WCET heuristics used for splitting, for example, by adding more precise hardware models that allow for calculating migration costs based on cache lines instead of data bits. (2) Adapting an existing RTOS to support migration thresholds. (3) Extending the analysis to the OS implementation and the system calls respectively.

Source code is available:
`www4.cs.fau.de/Research/RTSC/experiments/abbslicing/`

Acknowledgment

This work is supported by the German Research Foundation (DFG) under grants no. SCHR 603/14-2, SCHR 603/13-1, SCHR 603/9-2, the CRC/TRR 89 Project C1, and the Bavarian Ministry of State for Economics under grant no. 0704/883 25.

References

[1] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[2] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. H. Anderson, and S. K. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms," in *Handbook of Scheduling*, 2004.

[3] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comp. Sur.*, vol. 43, no. 4, p. 35, 2011.

[4] A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," *Proceedings of OSPERT '10*, pp. 33–44, 2010.

[5] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan, "Push-assisted migration of real-time tasks in multi-core processors," *Sigplan Notes*, vol. 44, no. 7, pp. 80–89, 2009.

[6] M. Griebl, P. Feautrier, and C. Lengauer, "Index set splitting," *Int'l Journal of Parallel Prog'ing*, vol. 28, no. 6, pp. 607–631, Dec. 2000.

[7] F. Scheler and W. Schröder-Preikschat, "The real-time systems compiler: Migrating event-triggered systems to time-triggered systems," *Software: Practice and Experience*, vol. 41, no. 12, pp. 1491–1515, 2011.

[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the Int'l Symp. on Code Generation and Optimization*, Washington, DC, USA, 2004.

[9] F. Scheler and W. Schröder-Preikschat, "Synthesising real-time systems from atomic basic blocks," in *Proc. of RTAS '06 WIP*, 2006.

[10] M. Stilkerich, J. Schedel, P. Ulbrich, W. Schroder-Preikschat, and D. Lohmann, "Escaping the bonds of the legacy: Step-wise migration to a type-safe language in safety-critical embedded systems," in *Proc. of ISORC '11*, March 2011, pp. 163–170.

[11] S. Vaas, P. Ulbrich, M. Reichenbach, and D. Fey, "Application-Specific Tailoring of Multi-Core SoCs for Real-Time Systems with Diverse Predictability Demands," *Signal Processing Systems*, Jul. 2018.

[12] L. R. Ford and D. R. Fulkerson, "A simple algorithm for finding maximal network flows and an application to the hitchcock problem," *Canadian Journal of Mathematics*, vol. 9, pp. 210–218, 1957.

[13] T. Klaus, F. Franzmann, M. Becker, and P. Ulbrich, "Data propagation delay constraints in multi-rate systems: Deadlines vs. job-level dependencies," in *Proc. of RTNS '18*, 2018, pp. 93–103.

[14] D.-T. Peng, K. G. Shin, and T. F. Abdelzaher, "Assignment and scheduling communicating periodic tasks in distributed real-time systems," *IEEE Trans. on Software Eng'ing*, vol. 23, no. 12, pp. 745–758, 1997.

[15] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research," in *Proc. of WCET '16*, vol. 55, 2016, pp. 2:1–2:10.

[16] C. Burguière, J. Reineke, and S. Altmeyer, "Cache-related preemption delay computation for set-associative caches–pitfalls and solutions," in *OASIcs-OpenAccess Series in Informatics*, vol. 10, 2009.

[17] E. W. Briao, D. Barcelos, F. Wronski, and F. R. Wagner, "Impact of task migration in noc-based mpsocs for soft real-time applications," in *Int'l Conf. on Very Large Scale Integration*, Oct. 2007, pp. 296–299.

[18] A. Burns and R. Davis, "Mixed criticality systems – a review," Tech. Rep. 9. Edition, 2016.

[19] B. Peng, N. Fisher, and M. Bertogna, "Explicit preemption placement for real-time conditional code," in *Proc. of ECRTS*, 2014, pp. 177–188.

[20] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. a survey," *IEEE Trans. on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013.

[21] M. Bertogna, G. Buttazzo, and G. Yao, "Improving feasibility of fixed priority tasks using non-preemptive regions," in *Proc. of RTSS '11*. IEEE, 2011, pp. 251–260.

[22] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo, "Optimal selection of preemption points to minimize preemption overhead," in *Proc. of ECRTS '11*, Jul. 2011, pp. 217–227.

[23] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Proc. of RTCSA '99*. IEEE, 1999, pp. 328–335.

[24] J. H. Anderson, V. Bud, and U. C. Devi, "An edf-based restricted-migration scheduling algorithm for multiprocessor soft real-time systems," *Real-time Systems*, vol. 38, no. 2, pp. 85–131, Feb. 2008.

[25] F. P. Franzmann, T. Klaus, P. Ulbrich, P. Deinhardt, B. Steffes, F. Scheler, and W. Schröder-Preikschat, "From Intent to Effect: Tool-based Generation of Time-Triggered Real-Time Systems on Multi-Core Processors," in *Proc. of ISORC '16*, 2016.

[26] F. Nemati, J. Kraft, and T. Nolte, "A framework for real-time systems migration to multi-cores," Tech. Rep., 2009.

[27] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, 2005.

[28] K. G. Shin, T.-H. Lin, and Y.-H. Lee, "Optimal checkpointing of real-time tasks," *IEEE T. on Comp.*, vol. 100, no. 11, pp. 1328–1341, 1987.

[29] S. W. Kwak, B. J. Choi, and B. K. Kim, "An optimal checkpointing-strategy for real-time control systems under transient faults," *IEEE Trans. on Reliability*, vol. 50, no. 3, pp. 293–301, 2001.

Experiments for Predictable Execution of GPU Kernels

Flavio Kreiliger*, Joel Matějka*†, Michal Sojka† and Zdeněk Hanzálek†
Faculty of Electrical Engineering/Czech Institute of Informatics, Robotics and Cybernetics†*
Czech Technical University in Prague
Prague, Czech Republic
{kreilfla,matejjoe}@fel.cvut.cz, {michal.sojka,zdenek.hanzalek}@cvut.cz

***Abstract*—Multi-Processor Systems-on-Chip (MPSoC) platforms will definitely power various future autonomous machines. Due to the high complexity of such platforms, it is difficult to achieve timing predictability, reliability and efficient resource utilization at the same time. We believe that time-triggered scheduling in combination with PRedictable Execution Model (PREM) can provide strong safety guarantees, and our longer-term goal is to schedule execution on the whole MPSoC (CPUs and GPU) in time triggered manner.**

To extend PREM to GPUs, we compare two synchronization mechanisms available on the NVIDIA Tegra X2 platform: one based on pinned memory and another that uses a GPU timer (so-called *globaltimer*). We found that running the NVIDIA profiler (nvprof) reconfigures the resolution of the globaltimer from 1 µs to 160 ns. By using time-triggered scheduling with such a resolution, it was possible to reduce execution time jitter of a tiled 2D convolution kernel from 6.47% to 0.15% while maintaining the same average execution time.

***Index Terms*—predictable execution, gpu, nvidia, tx2, prem**

I. Introduction

Autonomous machines such as self-driving cars will certainly be a part of our future. Nowadays, both industry and researchers work heavily on various aspects of those machines. One aspect that is still not satisfactorily addressed is how to ensure their safe operation. Those machines require vast computational power to process all the sensor data, and reason about them in real-time, however, safety systems are traditionally implemented with slow, simple, but reliable computing elements. In contrast to that, autonomous machines are powered with heterogeneous computing architectures, where a multi-core CPU is accompanied by one or more accelerators such as GPUs or FPGAs, often on the same chip. These are called Multi-Processor Systems-on-Chip (MPSoC). In this work, we use a popular representative of these systems: NVIDIA Tegra X2 (TX2), which features a GPU.

While FPGAs can offer precise timing, GPUs seem to be more popular in these applications, perhaps due to their easier programmability. However, GPUs originate from industrial domains, where average-case performance was traditionally more important than real-time and safety guarantees.

This work was supported by the grant no. SGS19/175/OHK3/3T/13 and by the THERMAC project, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 832011.

To reason about safety properties, the functional safety standard for road vehicles ISO 26262 [1] defines the term "freedom from interference", as the absence of certain faults, one of them being "incorrect allocation of execution time". This means that predictable timing is a prerequisite for achieving safety according to this standard.

We believe (and many safety standards agree) that time triggered scheduling gives stronger safety guarantees than online event triggered scheduling. In the case of MPSoC platforms, time-triggered scheduling makes it easier to control contention on shared hardware resources (caches, buses, memories) and thus to control the inter-task interference. Our longer-term goal is to schedule execution on the whole MPSoC (CPUs and GPU) in time triggered manner. In our past work [2], we reduced interference between tasks on a multi-core CPU by time triggered scheduling. This paper is our starting step to doing the same for GPU tasks.

In this paper, we first evaluate synchronization mechanisms for GPU workload, with the conclusion that time-triggered synchronization has the potential of having much lower overhead than lock-based synchronization via so-called pinned memory. However, the overhead of time-triggered execution can be low only when estimates of worst-case execution time are tight. For this reason, we analyze the interference between tasks running on the GPU and try to reduce it by using two main techniques: 1) prefetching data from global memory to local shared memory [3] and 2) scheduling the GPU execution in time triggered manner. Our experimental evaluation shows that these techniques are able to reduce the interference and execution time jitter without significantly increasing total execution time.

More specifically, we adopt the concept of Predictable Execution Model (PREM) proposed by Pellizzoni et al. [4], where computation is split into memory and compute phases, and these phases are scheduled to not interfere with each other – for example, by not running two memory phases in parallel. For GPU workloads, this would result in severe underutilization of memory bandwidth. Therefore, we aim to allow multiple kernels to access memory simultaneously while preserving predictable execution time.

Our overall approach has two main assumptions: a) Executed workload is time-deterministic, meaning that the amount of computation can be determined ahead of time and does not

depend on processed data. This holds for many algorithms used for autonomous machines such as neural network inference [5] or visual object tracking [6]. b) Time-triggered scheduling is used for the whole MPSoC platform to ensure that interference between all on-chip processors can be controlled. On the other hand, it is known that time-triggered scheduling often lacks the required flexibility. For this reason, we envision the use of both time- and event-triggered approaches together. Time-triggered execution will be used for shorter, non-preemptive intervals (e.g., for processing of one camera frame), and multiple of those intervals will be executed in a more dynamic way based on online scheduling and synchronization mechanisms.

II. Related work

Similarly to our work, recent research by Cavicchioli et al. characterized interference on main memory and communication bus level between the CPU and GPU [7]. Other researchers [8], [9] developed various microbenchmarks to understand GPUs and their memory system. Our work differs from those by using time triggered scheduling.

The scheduling behavior of many GPUs is unknown in most cases due to a lack of publicly available and open documentation. Therefore, GPUs are mostly treated as black boxes, and different approaches for predictable execution of different workloads have been developed to bypass this uncertainty. An often used method is to ensure that only one process can access the GPU resources at a time by use of a locking mechanism [10]. The cost of this approach may be an underutilization of powerful GPUs. Dividing the workload into smaller preemptable chunks could reduce this problem [11], [12]. Others evaluated techniques to manage accesses to memory [13] to reduce contention between GPU and CPU applications.

Further Otternes et al. assessed the NVIDIA TX1 platform regarding real-time behavior concerning co-scheduling of multiple kernels [11] [12], and additionally, Amert et al. derived a set of the GPU scheduling rules used in the Jetson TX1 and TX2 platforms to brighten up the black box nature of those platforms [14]. They ran different experiments to understand how the GPU schedules the work if submitted from the same or different processes. They found that the GPU workload launched from different processes shares the GPU by the use of multiprogramming, where each kernel runs exclusively on the GPU during its assigned time slice and does not overlap other GPU computation. For GPU workload submitted from the same process, the computation can overlap and is scheduled according to the derived rules. Bakita et al. proposed a validation framework to validate those derived rules for future GPU generations [15].

Capodieci et al. [16] changed how the GPU workload is scheduled by using an EDF scheduler combined with a Constant Bandwidth Server. Their scheduler is implemented in a hypervisor and works by replacing the run list inside the GPU-host.

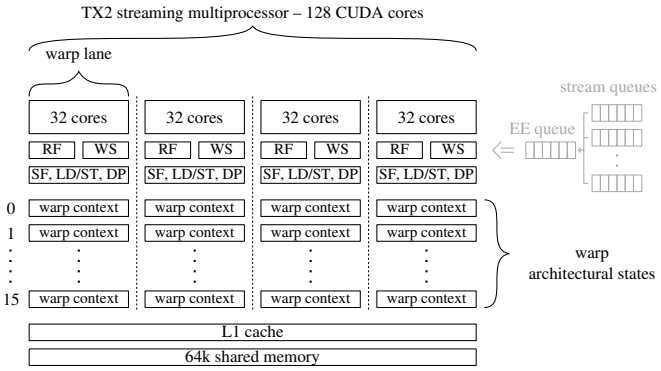

Figure 1. Estimated architecture of one SM of TX2

III. Background

A. GPU/TX2 architecture

NVIDIA Tegra X2 (TX2) is a high-performance embedded MPSoC consisting of two CPU clusters and one Pascal GPU with 256 CUDA cores. The memory bus is shared across the entire chip. However, each CPU cluster and GPU have a separate L2 cache. These caches are not coherent. The GPU is composed of two independent computing blocks called Streaming Multiprocessors (SM), each having its L1 cache, shared memory, and four warp lanes. Each warp lane executes a *warp*, i.e., a group of up to 32 threads performing the same instruction on different data. Since NVIDIA does not publish all details about their GPU architectures, it is difficult to estimate architecture details, and how GPU workload is scheduled on the available warp lanes. Based on publicly available documentation, previous work by Amert [14] and Capodieci [16], and our experiments, we assume the architecture of one streaming multiprocessor to be as depicted in Figure 1. The workload is inserted by CPUs into stream queues, then by rules revealed by Amert [14], put into the execution engine queue and assigned to an SM if enough resources are available. We assume that up to 16 warps can be assigned to a single warp lane. The warps from CUDA blocks (see III-B) are placed in the available *warp context* slots, which store their architectural state, and are run by the hardware warp scheduler (WS) as soon all their dependencies are satisfied. The warp scheduler issues and interleaves instructions from the associated warps, hiding latencies caused by waiting for shared resources. After all warps in a block have finished, the occupied warp context slots are freed and can be reused by warps from the next queued block. Warp scheduling is similar to hyperthreading used in CPUs. Multiple running warps share CUDA cores and other resources such as multiple load/store units, special function units (SF) and double precision units (DP) in one warp lane.

The GPU also features a clock source, called *globaltimer*, which provides synchronous time to all SMs.

In this paper, we do not consider graphics jobs and how the hardware is shared between them and compute jobs.

B. CUDA programming model

To offload computation to the GPU, NVIDIA offers the C/C++ API called CUDA. Programmers write so called *kernels*, i.e., functions that execute in parallel on the GPU. When a kernel is launched, the programmer specifies, with a special syntax, the kernel execution configuration: the number of threads and how those threads are organized into groups (CUDA-blocks or thread-blocks). Each block is executed independently without a built-in possibility to synchronize with other blocks or kernels. Only threads within a single block can be synchronized. Launched CUDA kernels are placed into queues called streams from where they are executed in FIFO order. By default, there is one stream per process. More streams can be created to execute kernels in parallel if enough resources are available. All kernel launches are asynchronous, meaning that if a CPU needs to wait for kernel completion, it has to invoke explicit synchronization operation.

IV. Evaluation goals

In this section, we explain the goals of this paper in more detail.

A. Synchronization mechanism

A precondition for applying PREM to GPU workloads is the availability of fast synchronization between all blocks running at the same time.

In our previous work, we used locks in shared memory to synchronize PREM phases on the CPU [2]. Shared memory offered a fast communication channel since multiple CPU cores share the same cache and the synchronization bypasses the main memory. On the TX2 GPU, an equivalent approach would be to use pinned memory, which is accessed in non-cached manner, to arbitrate accesses to the main memory.

An alternative approach would be to use time based synchronization using the *globaltimer*. We are interested in finding the overhead of the mechanisms and assessing their suitability for whole-GPU synchronization.

B. Benchmark selection

Polybench-ACC [17] is a collection of computational kernels such as matrix multiplication, 2D or 3D convolution, or linear equation solver, used to evaluate the performance of compilers and similar software. Mentioned algorithms are the core of many high-performance applications such as neural networks or image processing. To see the potential benefit of our interference reduction approach, we want to evaluate it on a benchmark highly sensitive to memory interference. Therefore, we evaluated the sensitivity of all polybench kernels to memory interference from CPU and selected 2D convolution as a good candidate.

C. Reduction of intra-GPU interference

As the Polybench 2D convolution kernel is accessing the global memory, it is hard to reduce the interference directly. Therefore, we first apply *tiling* – a technique commonly used to coalescence memory accesses in global memory to speed up the GPU execution [3], [18]. The tiling is done by splitting the input data into multiple tiles which fit into the shared memory segment within a CUDA block. The computation is then performed on the tile previously prefetched from the global memory into the shared memory. At the end, the processed tile is written back. This tiled implementation naturally maps to the three PREM-phases: *prefetch*, *compute* and *writeback*.

Further, we want to assess the interference between the individual phases of tile processing if scheduled synchronously in parallel.

V. Experimental evaluation

We ran all experiments on the Jetson TX2 board in NV Power Mode MAXN and with all frequencies configured to the maximum values by running *jetsonclock.sh* (a script provided by NVIDIA to configure board clocks). All source code we used for the experiments can be found in the git repository: *https://github.com/CTU-IIG/tt-gpu*

A. Pinned memory synchronization evaluation

We evaluated synchronization based on locks in pinned memory with two experiments. First, we measured the ping-pong round-trip time between two GPU kernels and later the experiment was repeated to collect the round-trip times between CPU and GPU since the synchronization mechanism should offer a possibility to be used for CPU to GPU synchronization. Both experiments have been repeated for 1000 times. We had to add the *membar* instruction to ensure that one GPU kernel sees the updates from the other GPU kernels.

Between GPU kernels the average round-trip time was 2.065 µs (min: 1.92 µs, max: 2.24 µs) and the CPU to GPU round trip time was in average 1.94 µs (min: 1.47 µs max: 2.56 µs). These times are not sufficient for synchronizing PREM phases on the GPU, because, as discussed later in Section V-E, the length of the phases is in the range of 1 to 4 µs and compared to this, the overhead of this synchronization mechanism would be too high.

B. GPU timer granularity

We evaluated the *globaltimer* as a synchronization mechanism between GPU tasks. According to the documentation [19], the globaltimer should have a resolution in the nanosecond level. The main criteria for the *globaltimer* to be used as a synchronization mechanism are its resolution and that it is running synchronously on both SMs. To evaluate these properties, we ran a kernel from Listing 1 with four blocks of one thread each. Each block retrieves the globaltimer timestamps in a for loop, storing them into its shared memory segment. The shared memory was selected for two reasons: 1) its access time is short enough to not influence timestamp precision much and 2) allocating shared memory segments to occupy half of the available shared memory on an SM ensures that two blocks execute on one SM and two on the other.

Figure 2 shows a zoom into the first few iterations of collected timestamps. Running the experiment in the default settings gives disappointing results. The measured resolution

Listing 1. Simplified kernel to retrieve global timer jitter

```
__shared__ uint64_t times[NOF_STAMPS];
for (int i = 0; i < NOF_STAMPS; i++)
  asm volatile("mov.u64 %0, %%globaltimer;" \
                       : "=l"(times[i]));
```

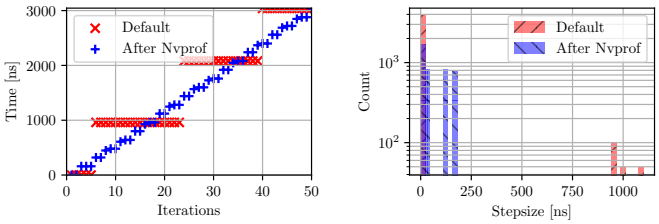

Figure 2. Timestamps and step sizes of the globaltimer after reboot and after one run of nvprof. The retrieved timestamps of the other blocks exhibited the same resolution.

was only 1 µs. The "Default" points on the left side show the timestamps collected by the first block. The right side of the figure shows the histogram of the differences between two subsequent timestamps. By coincidence, we found that running nvprof[1] once on an arbitrary kernel reduces the measured resolution of the globaltimer to 160 ns, as shown with "After Nvprof" points in Fig. 2. The use of nvprof seems to reconfigure the globaltimer on the GPU without reconfiguring it back at the end. Although this behavior is not documented and not really intuitive, it helped us to increase the resolution of the globaltimer.

It is important to highlight, that nvprof needs to run only once on an arbitrary kernel. After this run, the further kernels can run without the instrumentation with nvprof to still profit from the higher resolution.

C. Time triggered execution of tiled 2D Convolution

To see how the execution jitter occurs and if it can be reduced if multiple kernels (4 in our experiments) run in parallel, we compare the original 2D Convolution polybench benchmark (later denoted as *legacy* implementation) and our *tiled* version of it. Each kernel was run 1000 times then the average, minimum and maximum execution times have been calculated. Both implementations apply a 3x3 convolution mask on a dataset consisting of 1026x1022 float elements. The kernels were launched with a configuration of two blocks with 512 threads. The tiled implementation tiles the input data into 512 tiles of 4x512 elements. Each tile is processed in the following phases: first, the tile is prefetched from global memory into the CUDA shared memory segment, then the computation takes place, and in the end, the resulting data is written back to global memory. Since the streaming multiprocessor on the TX2 offers 64 kB of shared memory, we dimensioned our kernel blocks to use 16 kB of shared memory to allow the execution of 4 kernels in parallel. To investigate the possibility of interference reduction, we use the globaltimer to synchronize the running blocks and to control the start times of the tile processing. Figure 3 shows how

[1] *nvprof* is the profiling tool offered by nvidia to analyze traces and timings of called CUDA API and launched kernels

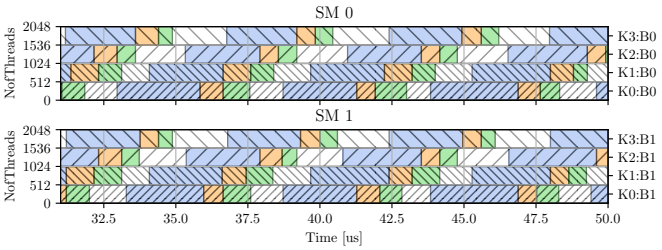

Figure 3. Zoom into the execution of 4 tiled 2D convolution kernels (K0–K3). The total execution time was 2.87 ms. The tiles are scheduled on both streaming multiprocessors with an offset of 1.4 µs against each other. Both blocks of the same kernel (B0–B1) are scheduled at the same time instance and are processing multiple tiles in sequence. Blue, orange and green colors represent *prefetch*, *compute* and *writeback* phases and blocks with the same hatch correspond to the same kernel. During the white phases the blocks are spinning on the *globaltimer* until they are allowed to process the next tile.

the tile processing start times are shifted with an offset of 1.4 µs against each other. The two blocks inside a kernel start processing their current tiles always at the same time, the white spaces between the tile processing phases represent the time a block is spinning on the globaltimer until it is allowed to start with the next prefetch phase.

To have a more elaborate overview of the influence of the tile scheduling offset to the observed execution jitter, we run four kernels of the tiled 2D convolution in parallel with different tile offsets. All kernels recorded their block start/end times using the globaltimer. The difference between the latest block end time and the earliest start time is called scenario execution time.

We can see in Figure 4 the average scenario execution time with the min-max execution jitter (blue) and the corresponding execution jitter compared to the scenario execution time in percentage (red). The dotted black line represents the average scenario execution time of the baseline (4 legacy kernels in parallel). As we can see, the scenario execution time and execution jitter remain relatively stable at 2.5 ms respectively 1.4% until the tile offset exceeds 1.2 µs. After this point, the scenario execution time increases and the execution jitter decreases. Based on these results, we classify the tile offsets of 1.3 µs and 1.4 µs as able to reduce the execution jitter while still having a acceptably low scenario execution time.

Further, the 2D convolution kernels were launched in the next scenarios: i) The original (legacy) implementation with 1 kernel running on the GPU, ii) the legacy implementation with 4 kernels running in parallel, iii) the tiled version with 4 kernels running in parallel but without synchronization and iv) the tiled version with the tile processing shifted by 1.3 µs and v) by 1.4 µs offset. Figure 5 shows the average execution time and execution jitter of the scenarios. The blue bars show the average scenario execution time. The minimum and maximum scenario execution times are represented by the small error bars on top of the blue bars. The red bars represent the min-max jitter in percentage relative to the average scenario execution time. It can be seen, that the legacy implementation suffers from high contention in the four kernel configuration. The worst-case observed execution time (WOET) is still slightly shorter than the WOET of the single

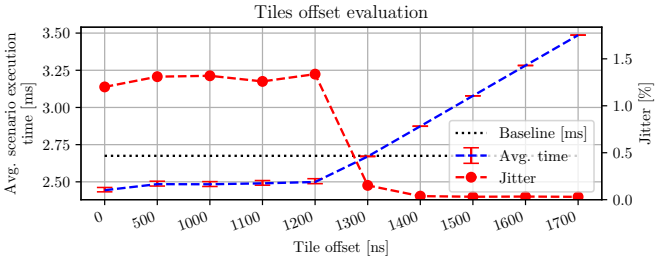

Figure 4. Influence of tile scheduling offset to the scenario execution time and the execution jitter.

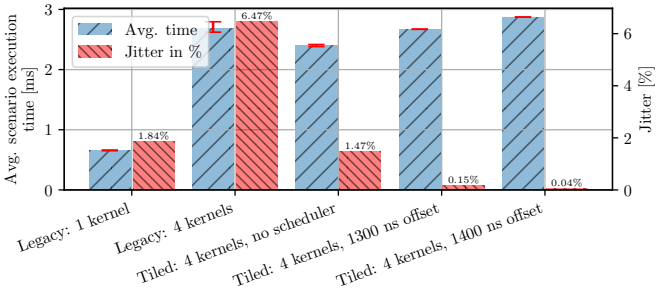

Figure 5. Comparison of scenario execution time. Tiles are scheduled against each other.

kernel version executed 4 times in a row, but the execution jitter is around 6.47% of the average execution time. The tiled implementation with 4 kernels already performed faster than the legacy implementation and its execution jitter is only 1.47%.

The tiling concentrates the accesses to the main memory of the kernels. Therefore, the kernels do not have to access the main memory in all phases, which leads to less contention and lower jitter. The scheduled tiled versions have a bit higher average scenario execution times than the legacy four-kernel version, but with the advantage of execution jitter reduced to 0.15% and 0.04% for the scheduling offset of 1.3 µs and 1.4 µs respectively. Still, one could argue that the WOET of the tiled version (2.42 ms) without scheduling is still shorter than the minimum execution time of the scheduled version (2.87 ms). However, the version without the scheduler offers no future possibilities to synchronize the GPU with the CPU, and the whole execution on the GPU would need to be treated as a single memory phase for CPU PREM scheduling.

D. Phase evaluation

In the tiled implementation, each block processes sequentially multiple tiles, each consisting of three PREM phases. To analyze in more detail how the phases interfere, we added another synchronization point, as shown in Figure 6, between compute and writeback phases to allow independent evaluation of phase interference. By shifting the phase start times, we measured the interference of: i) the prefetch and compute phases (WB is scheduled later not to run concurrently), and of ii) the writeback phases (PF and C are scheduled earlier not to run concurrently).

In Figure 7, we can see how the prefetch and compute phases interfere with each other. The average compute time bars are stacked on top of the average prefetch time bars. The

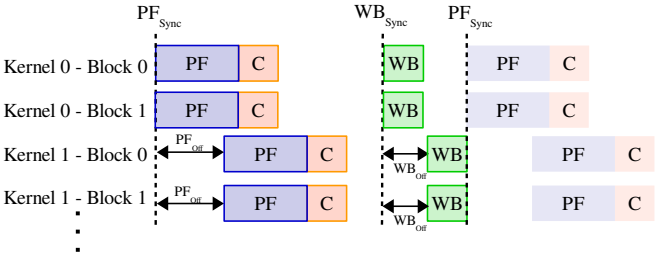

Figure 6. Synchronization points to schedule the PREM phases independently.

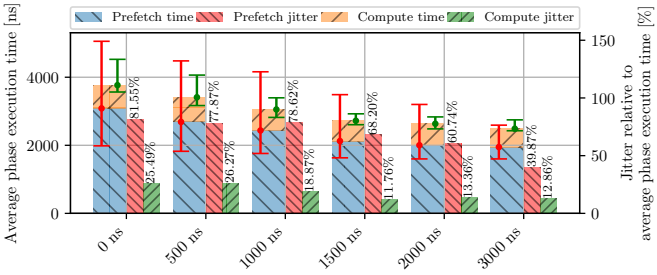

Figure 7. Only *prefetch* and *compute* phases are scheduled against each other (X-axis shows shift offset PF_{Off}). *Writeback* phases are moved away by the schedule and do not influence the previous two phases. In this experiment the two blocks running in a kernel are scheduled at the same time instance.

bars on the right represent the phase execution jitter of the two phases compared to the total average phase execution time (PF + C). One can see that the average phase execution time and the jitter are reduced the less the phases overlap. This effect is dominant in the *prefetch* phases. An interesting fact is that the *compute* phases have the biggest jitter when they overlap with other compute phases (no shift). This indicates some contention on the shared memory or other resources in the streaming multiprocessor. It also prevents the straightforward application of the PREM model, which assumes that compute phases do not interfere.

Figure 8 shows the interference of the *writeback* phases. Similarly to the *prefetch* phases, the less the *writeback* phases overlap, the more the phase execution time is reduced.

Even though the phase execution jitter appears to be high (e.g. 81% in Fig. 7 on the left), the kernel scenario execution jitter percentage is much smaller (1.2% in Fig. 5) since it is relative to longer scenario execution time.

E. Comparison of PREM scheduling on CPU and GPU

When we compare the above-described results with our previous application of PREM on the ARM CPUs of the Jetson

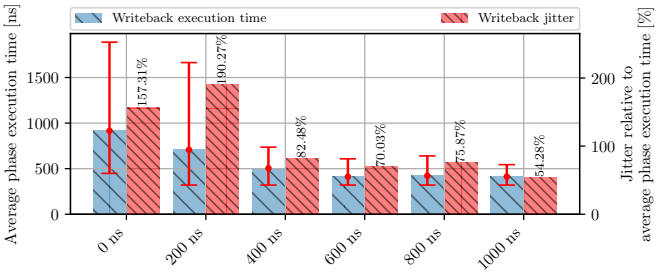

Figure 8. Execution time and jitter of *writeback* phases scheduled against each other (X-axis shows shift offset WB_{Off}). *prefetch* and *compute* phases are scheduled away to isolate the *writeback* phases

TX1 [2], the *prefetch* and *writeback* phases took around 100 and 400 µs respectively and *compute* phases up to 3 ms. This allowed to schedule a sequence of *memory* phases in parallel with one or more longer *compute* phases and the CPUs were efficiently utilized. On the GPU side, the phases execution times are much shorter and differently distributed. Namely, the *writeback* phase has the shortest phase execution time followed by the *compute* and the *prefetch* phases. Therefore, the approach used for CPU PREM scheduling, is not generally applicable to the GPU. When combined with the fact, that the execution time of *compute* phases is influenced by overlapping with other *compute* and *prefetch* phases, it is clear that the PREM scheduling rules need to be changed to be properly applicable to the GPU execution.

The experiment, where the whole tiles were scheduled against each other (Fig. 5), showed that the jitter could already be significantly reduced without introducing big increase of average execution time of all participating kernels. Therefore, a solution to predictable execution times on the GPU requires a different (less strict) set of co-scheduling rules than on the CPU. It remains to be seen whether/how such rules can be used as a proof for *freedom from interference*.

VI. Conclusion and future work

In this paper, we evaluated mechanisms for the low-overhead application of predictable execution model (PREM) to GPU kernels. We compared two synchronization mechanisms for synchronization of PREM phases. The memory-based synchronization achieves round-trip time of around 2 µs, which would result in too high overhead for short PREM phases on the GPU. Synchronization based on the globaltimer allows reaching lower overhead, but only after running *nvprof*, which magically increases the globaltimer resolution to 160 ns. Furthermore, we have shown that by using a tiled implementation of the 2D convolution kernel and tightly synchronizing execution all blocks across multiple kernels by using the globaltimer, we can reduce the execution time jitter from 6.47% to 0.15% while maintaining almost the same average execution time. We have also shown that the duration and interference of the PREM phases are different on the GPU compared to CPU. Namely, the phases are 100 to 1000 times shorter on the GPU and the execution time of *compute* phases can be influenced by other overlapping PREM phases. This and the short compute phase times make it impossible to execute a sequence of *memory* phases in parallel with a *compute* phase. On the other hand, simple scheduling the whole tiles with fixed offsets, investigated in this paper, resulted in sufficiently predictable execution with low jitter. Therefore, we believe that applying more advanced scheduling can lead to even more predictable execution, especially when combined with time-triggered CPU scheduling.

Since we performed the first experiments only on the 2D Convolution kernel, we plan to analyze in more detail how various execution phases influence each other in other kernels. Especially we would like to evaluate the behavior of the PREM phases of more compute intensive kernels. Based on such an evaluation, we want to come up with scheduling rules whose application will lead to low execution time jitter and acceptable performance at the same time. Later we plan to evaluate our scheduling concept on a real application commonly used in autonomous driving. Combining predictable GPU execution with PREM-based CPU execution is also planned.

References

[1] ISO, "ISO 26262 Road vehicles – Functional safety," 2011.

[2] J. Matějka, B. Forsberg, M. Sojka, P. Šůcha, L. Benini, A. Marongiu, and Z. Hanzálek, "Combining PREM compilation and static scheduling for high-performance and predictable MPSoC execution," *Parallel Computing*, 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167819118301789

[3] M. Harris, "Using shared memory in CUDA C/C++," NVIDIA, accessed: 2019-04-09.

[4] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011, pp. 269–279.

[5] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[6] J. F. Henriques, R. Caseiro, P. Martins, and J. Batista, "High-speed tracking with kernelized correlation filters," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 3, pp. 583–596, March 2015.

[7] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2017, pp. 1–10.

[8] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, Jan 2017.

[9] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, March 2010, pp. 235–246.

[10] Y. Xu, R. Wang, T. Li, M. Song, L. Gao, Z. Luan, and D. Qian, "Scheduling tasks with mixed timing constraints in GPU-powered real-time systems," 06 2016, pp. 1–13.

[11] C. Basaran and K. Kang, "Supporting preemptive task executions and memory copies in GPGPUs," in *2012 24th Euromicro Conference on Real-Time Systems*, July 2012, pp. 287–296.

[12] J. Zhong and B. He, "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1522–1532, June 2014.

[13] B. Forsberg, A. Marongiu, and L. Benini, "GPUguard: Towards supporting a predictable execution model for heterogeneous SoC," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017, pp. 318–321.

[14] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2017, pp. 104–115.

[15] J. Bakita, N. Otterness, J. H.Anderson, and F. D. Smith, "Scaling up: The validation of empirically derived scheduling rules on NVIDIA GPUs," 2018.

[16] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for GPU with preemption support," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2018, pp. 119–130.

[17] University of Delaware, "Polybench-ACC," https://github.com/cavazos-lab/PolyBench-ACC, accessed: 2019-04-09.

[18] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: Optimizing GPU memory bandwidth via warp specialization," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2011, pp. 1–11.

[19] NVIDIA, "Parallel thread execution ISA version 6.4," https://docs.nvidia.com/cuda/parallel-thread-execution/#special-registers-globaltimer, accessed: 2019-04-09.

Event-Driven Multithreading Execution Platform for Real-Time On-Board Software Systems

Zain A. H. Hammadeh, Tobias Franz, Olaf Maibaum, Andreas Gerndt, Daniel Lüdtke
Simulation and Software Technology
German Aerospace Center (DLR)
Braunschweig, Germany
{zain.hajhammadeh, tobias.franz, olaf.maibaum, andreas.gerndt, daniel.luedtke}@dlr.de

***Abstract*—The high computational demand and the modularity of future space applications make the effort of developing multithreading reusable middlewares worthwhile. In this paper, we present a multihreading execution platform and a software development framework that consists of abstract classes with virtual methods. The presented work is written in C++ following the event-driven programming paradigm and based on the inverse of control programming principle. The platform is portable over different operating systems, e.g., Linux and RTEMS. This platform is supported with a modeling language to automatically generate the code from the given requirements. Our platform has been used in already flying satellites, e.g., Eu:CROPIS.**

We present in this paper an example that illustrates how to use the proposed platform in designing and implementing an on-board software system.

***Index Terms*—RTOS, Multithreading, Event-driven**

I. Introduction

Modern space applications demand high performance computing resources to carry out the increasing computational requirements of on-board data processing and sophisticated control algorithms. On the one hand, *multicore* platforms are promising to fulfill the computational requirements properly [1] as they provide high performance with low power consumption compared with high frequency uniprocessors. However, it is quite often not easy to write applications that execute in parallel. On the other hand, sensors are slow and cannot be on a par with the computing resources. Self-suspending processes are usually used to read from sensors, which makes timing more complicated and presents high pessimism and, thus, high over-provisioning.

In this paper, we present an event-driven multithreading execution platform, which is written in C++ following the inverse of control programming principle to improve reusability. We call our execution platform *Tasking Framework*. Tasking Framework provides abstract classes, which facilitates the implementation of space applications as event-driven task graphs. It also provides a multithreading execution based on POSIX, C++11 threading, and OUTPOST [2], which makes Tasking Framework compatible with Linux, RTEMS and many other real-time operating systems (RTOS).

Tasking Framework is motivated with lessons learned from the Bispectral Infra-Red Detection (BIRD) [3] attitude control system. The BIRD satellite launched in 2001. BIRD used a distributed Kalman filter [4] to estimate the attitude state vector of the satellite. This filter comprises several estimation and prediction modules executed by the controller thread. Each estimation module computes one value in the attitude state vector, for example, the sun vector from the sun sensor input values, the predicted sun vector and expected control effect from the last control cycle, a rate from the new sun vector or magnetic field vector, or the best rate by cross checking magnetic field vector rate, sun vector rate and measured rate from gyroscopes. The computation order is given by the data flow between the estimation modules. The order was given by a call sequence of the estimation modules in the controller thread.

During the development of the BIRD attitude control system some timing issues arose from the limited computing power of the on-board computer, and the timing requirements imposed by the sensors. BIRD used for all threads a predefined time slot in 500 ms cycle. The star tracker reported the attitude quaternion after 375 ms. The output buffers of the five actuators have to arm at 450 ms in the control cycle. By this, only 75 ms remain for all the computations inside the attitude control system. With the means of the event-driven paradigm, an as soon as possible scheduling of the computations is possible, which realizes timing constraints in the end of the control cycle. Only the computations that depend on the star tracker data have to be computed after 375 ms in the control cycle.

Tasking Framework has been used in the following projects: Autonomous Terrain-based Optical Navigation (ATON) [5], Euglena Combined Regenerative Organic food Production In Space (Eu:CROPIS) [6], Matter-Wave Interferometry in Weightlessness (MAIUS) [7], and Scalable On-Board Computing for Space Avionics (ScOSA) [8].

The rest of the paper is organized as follows: We present the basic concepts of Tasking Framework in Section II. In Section III, we elaborate the execution model. Our modeling language is presented in Section IV. Section V presents a case study of using Tasking Framework with the proposed Tasking Modeling Language (TML). After presenting our execution platform, we address the related work in Section VI, and we conclude in Section VII.

II. Tasking Framework

Embedded system applications are often described as a graph, which illustrates the software components and the de-

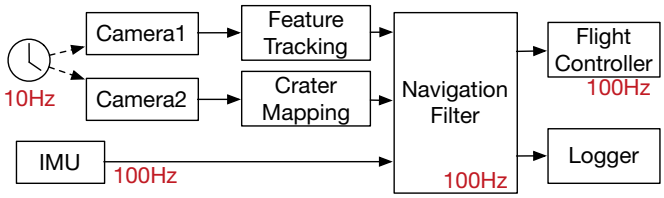

Fig. 1. Optical navigation system

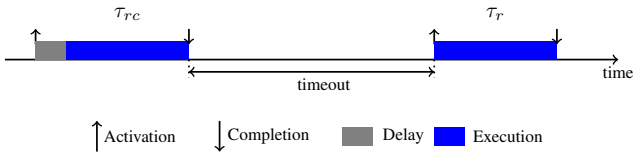

Fig. 2. An example of using the relative time. τ_{rc} sends a request command to the sensor. After the timeout occurrence, the following task τ_r reads the response sent by the sensor.

pendencies among them. Figure 1 shows an optical navigation system for spacecraft, which is a part of the ATON project [5]. Real-time capabilities are necessary to analyze optical sensor data and to react on the system's estimated position. The Tasking Framework is used to periodically trigger the cameras and to execute the image analyzer modules as soon as all required input data is available.

In this system, two cameras are triggered by a periodic timer and the images are then transferred to different analyzer components. The first one is a *feature tracking* component that estimates a relative movement, the second is a *crater navigation* component that tries to match craters on the Moon in the input images with a catalog of craters. The output of these components is then transmitted to the *navigation filter*. The navigation filter uses a Kalman filter to fuse the inputs with data from an additional inertial measurement unit (IMU) to get an estimated output position, which is then logged and sent to the flight controller. Tasking Framework is used in this example to integrate all these components.

Tasking Framework is implemented as a *namespace* `Tasking`, which comprises abstract classes with few virtual methods. It consists of the *execution platform* and the *application programming interface (API)*. Using the Tasking Framework, applications are implemented as a graph of *tasks* that are connected via *channels*, and each task has one or more *inputs*. Periodic tasks are connected to a source of *events* to trigger the task periodically, see Figure 7. In practice:

- Each computation block of a software component is realized by the class `Tasking::Task`. The virtual method `Task::execute()` will be overridden by the code of the software component;
- Each input of a task is realized by the class `Tasking::Input`;
- Each input object is associated with an object of the class `Tasking::Channel`;
- Each task may have multiple inputs and multiple outputs;
- A set of tasks, inputs and channels are framed in a scheduler entity, which is realized by the class `Tasking::Scheduler`;
- Each scheduler entity is provided with a scheduling policy;
- Each scheduler entity has threads to execute the assigned tasks according to the specified scheduling policy. The number of threads is specified by the software developer.
- Tasks can be activated also periodically by the means of the class `Tasking::Event`.

Although space software standards discourages virtual methods, the `execute()` method of tasks should be virtual to let the developer implement different tasks. A few other methods are intentionally virtual to add application code, e.g. synchronization of channel data.

To simplify setting up an object w.r.t. static memory management, we designed templates for the main classes.

A. Activation model

A task τ_i is activated and an instance of it will be queued when all inputs are activated (*and* semantic). *Or* semantic is also supported by providing the *final* flag. When the final flag is set for an input, the task will be activated regardless of other inputs.

The j-th input in_{ij} of task τ_i is activated when a predecessor task or other sources, e.g. the main thread, calls `Channel::push()` on the associated channel with in_{ij}. In the context of `Channel::push()`, the input in_{ij} will be set to active and if the final flag is set then τ_i will be activated, otherwise, the other inputs will be checked and τ_i will be activated only when all inputs associated to it are set to active.

Although we design our platform to be event-driven, time-triggered activation is supported by presenting the class `Tasking::Event`. Two time-triggered activation methods are supported: *periodic* and *relative time*. In the periodic method, the given time duration represents the distance between two successive events. Relative time method is used, for instance, when sensor data is needed. A task τ_{rc} sends a request command to the sensor then it sets the timer to a predefined time duration (relative time) and terminates. After the timeout occurrence, the following task τ_r reads the response sent by the sensor. Note that, this solution is similar to using *self-suspending* tasks [9]. Using relative time (in general using self-suspending tasks) requires to tightly bound the timeout. However, using channels connected to Interrupt Service Routines (ISR) of IO drivers (event-driven programming), in which τ_r is activated only when the sensor data is available, can improve the utilization. Figure 2 illustrates the relative time.

B. More features

1) Task group: The default call semantics among tasks that is supported in Tasking Framework is *asynchronous*, in which a task τ_i activates the successor tasks, then it can be executed again regardless of the status of the successor tasks. However,

in some applications, the graph of tasks or a subset of it has a *synchronous* call semantics such that τ_i activates the successor tasks and it will not be executed again till all tasks in the synchronous subset finish their execution. To support the synchronous call semantics, the class `Tasking::Group` is provided.

2) Task barrier: The number of activations at an input is declared at compile time. In situations, where the number of data elements is only known at run time, the activation cannot be adapted. This can be the case when, for example, a data source have states where no data is sent. The class `Tasking::Barrier` is a mean to control the activation of tasks with an unknown number of data packets.

By default the barrier can be instantiated with a minimum number of expected push operations on the barrier. After the minimum number of pushes happens, the barrier will activate all associated inputs, as long as data sources did not increase the number of expected push operations on the channel. If it is increased, more push operations are expected.

3) Unit test: We provide a special scheduler `SchedulerUnitTest` with step operation to support unit testing. Using Googletest (gtest) [10], we provide twelve classes to test the API.

Note that, the execution model has to be tested separately by the developer using other means, e.g. stress test.

III. Execution Model

Tasking Framework is a multithreading execution platform. The software developer should specify the number of threads, called *executors*. Therefore, there will be $n + 1$ threads: the main thread plus n executors. The implementation of the execution model is platform specific. We have three implementations of the execution model: the POSIX threading model (targeting Linux), C++11 threading and OUTPOST-core [2] (targeting RTEMS and FreeRTOS).

The execution model is represented by 4 classes:

- `Tasking::SchedulerExecutionModel`: Creating, scheduling, managing the executor threads and interfacing to the API.
- `Tasking::ClockExecutionModel`: Managing the time for time events. In embedded Linux, the clock is represented by a thread.
- `Tasking::Mutex`: An encapsulation of the mutex.
- `Tasking::Signaler`: An encapsulation of the conditional variables.

Tasking Framework schedules the ready task instances to the available executors according to the following scheduling policies: First-In-First-Out (FIFO), Last-In-First-Out (LIFO), and Static Priority Non-Preemptive (SPNP). The software developer can assign a priority to each task to be used by the SPNP queue.

An executor thread goes to sleep, i.e. waits on a conditional variable, after being created till it gets a signal from the clock thread (or a timer) in case of time-triggered tasks, or from other sources, e.g. the main thread. Figure 3 shows the life cycle of an executor thread.

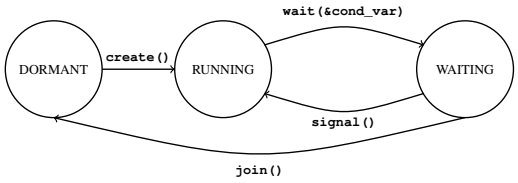

Fig. 3. Executor thread states

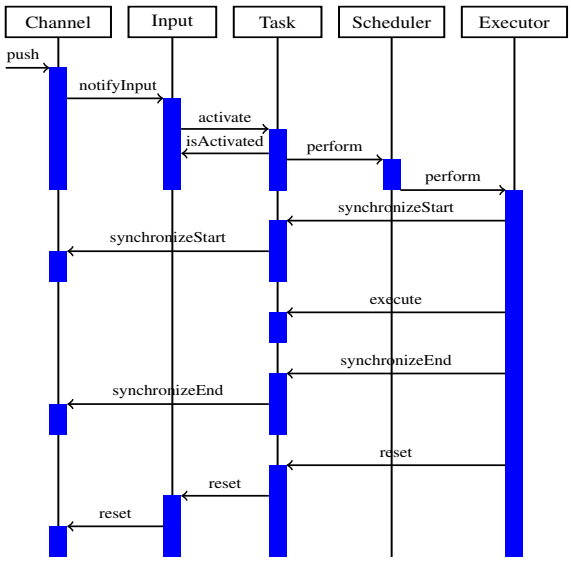

Fig. 4. The sequence of method calls in Tasking Framework to execute a task

The executor that executes the task τ_i activates the successor tasks and queue them in the ready queue, and it will signal a free executor, which is in *WAITING* state, if there is any. That is to say, Tasking Framework balances the load on the available executors. Even in case of one executor, the executor returns first from the `execute()` method of τ_i before checking the ready queue and executing the successor tasks of τ_i. The sequence of method calls that are performed by Tasking Framework to execute a task by an executor thread is shown in Figure 4. Because we have multiple threads that may try to access the data stored in the channel, a protection mechanism is implemented to synchronize the access to this shared data by different threads. The protection mechanism is implemented by the means of two virtual methods: `Channel::synchronizeStart` and `Channel::synchronizeEnd`.

In the implementation for Linux, the clock is implemented as a thread with the real-time clock provided by POSIX. The clock thread goes to sleep for a timeout equal to, e.g., the period of a periodic task. Then it signals a free executor if there is any, and it computes the next timeout.

A. Scheduling and priority handling

As has been mentioned, each instance of the `Tasking::Scheduler` is assigned a set of tasks,

inputs, channels, executors and a scheduling policy. With one instance for the application, the scheduling approach follows the global scheduling, i.e. all tasks can be assigned to any executor. However, it is possible to have multiple instances in one application. Considering the RTOS, we assign priorities to the executors (threads). Hence, we can handle priorities in groups (each group represents one `Tasking::Scheduler` instance).

IV. Tasking Modeling Language (TML)

We designed the API to be as usable as possible considering the high performance requirements of real-time onboard software systems. However, as the Tasking Framework is used in scientific missions with experts of different domains working on the system, the framework users might not be experts in implementing real-time software. To further improve applicability, we developed a model-driven tool environment that can be used to generate calls to the Tasking Framework API and its communication code to transfer data between different tasks. The tool is integrated into Virtual Satellite[1], a tool for model-based systems engineering. As Figure 6 shows, the TML development environment uses different types of description methods to model the software. Atomic data types are defined in tables, whereas data type classes and software components can be specified in textual languages. Because our focus is on data and event-driven communication, the connection of different components is modeled graphically. Each of the languages is specifically designed to describe software based on the Tasking Framework and, thus, further simplifies creating Tasking code.

A. Tasking-Specific Languages

Modeling languages specifically designed for a project or tool provide the benefit of introducing only few project-specific elements. The fewer elements in a language, the less effort is necessary to learn it. An early prototype of the modeling environment used the unified modeling language (UML) and the systems modeling language (SysML) to describe the Tasking-based software for the ATON project [11]. While the project clearly profited from modeling and its code generation, usage of the modeling tool required to understand UML, SysML and the Tasking Framework.

To improve modeling of software based on the Tasking Framework, we developed a tool suite including several domain-specific languages (DSLs) that contain all tasking relevant information. Figure 5 shows the editors of the different languages. The basic work flow is to define atomic data types first, then employ these to specify more complex data types, which are later generated as classes. As shown in the figure, atomic data types can be listed in a table. Additional attributes, such as the size of these types, allows running analyses about exchanged data and performance of components. After data types have been specified, it is possible to model software

[1]Virtual Satellite: a model-based tool for space system development; web page: https://www.dlr.de/sc/en/desktopdefault.aspx/tabid-5135/8645_read-8374/

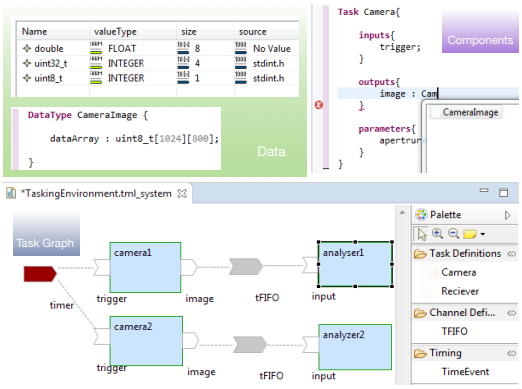

Fig. 5. Modeling tool environment for TML

components. Components can be modeled either as plain classes or tasks and can have inputs, outputs and parameters.

After the description of data types and components, they can be instantiated in the main part of TML, the task graph. Instantiating components in this graphical description automatically adds inputs and outputs of the components. Connecting components validates data types and allows only compatible elements to be connected. Tasking-specific event parameters as well as timing and priorities of the components can be configured in this diagram.

Besides modeling of these main components of the Tasking Framework, it is possible to model custom communication channels, storage types, scheduling policies and units within the model.

B. Increased Development Productivity

The model-driven tool does not only simplifies the application of the Tasking Framework, it also increases the efficiency of developing software based on the framework. Projects with model-driven software development benefit from higher short-term productivity because users can generate new features from the model; long-term productivity increases because changing requirements can be handled by simply updating the model [12]. Thus, in context of the Tasking Framework, adding components to the diagram and connecting them generates their execution containers and communication code. This code does not have to be implemented manually. Furthermore, if project requirements change and the components have to be connected differently, reconnecting the elements in the diagram automatically updates the software's source code and documentation.

As Figure 6 shows, generation from the model not only generates source code but also build files and tests. SCons scripts or CMake files allow to build the generated code after generation immediately. This way, it is possible to start the development of a project from a running system and iteratively add new features.

C. Extensibility of the TML Model and Generated Code

To be applicable in as many projects as possible, the modeling environment is highly extensible. Besides defining

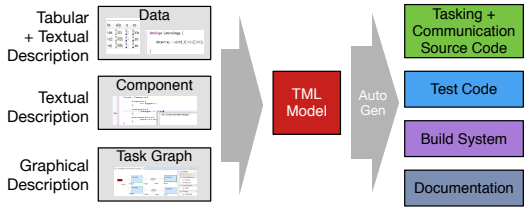

Fig. 6. Overview of the modeling environment and its generated atrifacts

custom data types and components, it is also possible to dynamically add new channel types with custom parameters. For such new channel types, the generator creates base classes with templates or constructor parameters depending on the parameters definition as static or dynamic. Instances of these custom types in the task graph are also generated as instances of the generated class and configured with parameter values from the model. Thus, even with a dynamic definition of new types and parameters within the model, the generated code remains executable.

In addition to the extensibility of the model language, the generated code can be customized by keeping code updates through regeneration possible. A combination of the decorator and generation gap pattern allows customizing the generated code by subclassing [13] [14]. The generated abstract class is regenerated, the concrete class is generated only once and then kept to not overwrite customizations.

V. Case Study

To demonstrate the benefits of the Tasking Framework in a simple real-time software, we recall our example in Figure 1. Figure 7 shows the architecture of the software as TML task graph diagram.

With the Tasking Framework, software components are implemented as tasks, data is stored in Channels and Events are used for periodic activation of the components. For the execution on the prototype flight computer, we assigned four threads to execute the software. As soon as the camera driver task pushes an image into the subsequent channel, the feature tracking task is notified and activated. The crater navigation task is configured to be activated only for every second image and, thus, runs with a reduced frequency. As the IMU driver does not have an external trigger, it is implemented as a thread that runs continuously and produces acceleration rate data with a frequency of 100Hz. Because the navigation filter has to update the output position with every IMU value, the final flag of its input data from the IMU is set. Therefore, if the input data from the crater navigation and feature tracking are available they are used, otherwise they are ignored.

To model this setup with TML for generation of the necessary tasking code, the first step is to define data types that can be used in the software. After a definition of the atomic types, such `double` and `uint8_t`, we can model the data types for `CameraImage`, `EstimatedPosition`, `AccelerationRate` and `NavigationState`. As next step, we have to model the actual components, which are generated as tasks. To create a model element for the `CameraDriver` task, we specify an input that does not have any data type and an output of type `CameraImage`. As last step of the element definition, we specify two different channel types: one LIFO channel with a parameter for its size and a channel with two buffers that switch every time data is added. In the task graph diagram we can then instantiate and connect all the previously defined elements. As we have two cameras, the camera driver task is instantiated twice. In the diagram we can then configure the timing and event parameters. As the crater navigation should run only every second image, we configure its input with a threshold of two. With the selected scheduling policy of priority-based, we can configure priorities for each task.

After we described the system in a TML model, we can generate its source code. All task definitions are generated with their in- and output interface; their instances in the diagram are created as objects in the software. Both cameras can be instances of the same camera driver task. For task definitions and custom channels, the generator creates base classes, which can be customized by subclassing.

VI. Related Work

Many platforms have been proposed for developing and testing embedded systems. Sadvandi, Corbier and Mevel presented in [15] a real-time interactive co-execution platform designed at Dassault Systèmes. The objective is to provide integration, co-execution and validation of heterogeneous models using model-based testing process, which comprises In-the-Loop testing, namely, Model-In-the-Loop (MIL), Software-In-the-Loop (SIL) and Hardware-In-the-Loop.

The Embedded Multicore Building Blocks (EMB^2) [16] is an open source C/C++ framework for the development of parallel applications. EMB^2 is developed by Siemens AG to efficiently exploit symmetric and asymmetric multicore processors. EMB^2 provides different scheduling strategies for both hard and soft real-time systems. Although Tasking Framework supports multithreading, it is not specifically dedicated for multicore systems.

OUTPOST is an open source mission and platform independent library developed in the German Aerospace Center (DLR) to design and implement reusable embedded software as early as possible and hence to be independent from the operating system and the underlying hardware. OUTPOST is originally called libCOBC, and it has been used in the Eu:CROPIS project [17], and in the ScOSA project [8]. Tasking Framework runs on the top of OUTPOST, and makes use of the services provided by it. One implementation of the execution model of the Tasking Framework is dedicated for OUTPOST as we have mentioned in Section III.

RODOS (Real-time On-board Dependable Operating System) [18], [19] is a real-time operating system developed at the German Aerospace Center (DLR) for network-centric core avionics [20]. Currently, RODOS is developed at University of Würzburg. The main goal of RODOS developers was to make it simple and dependable. The publisher-subscriber messaging

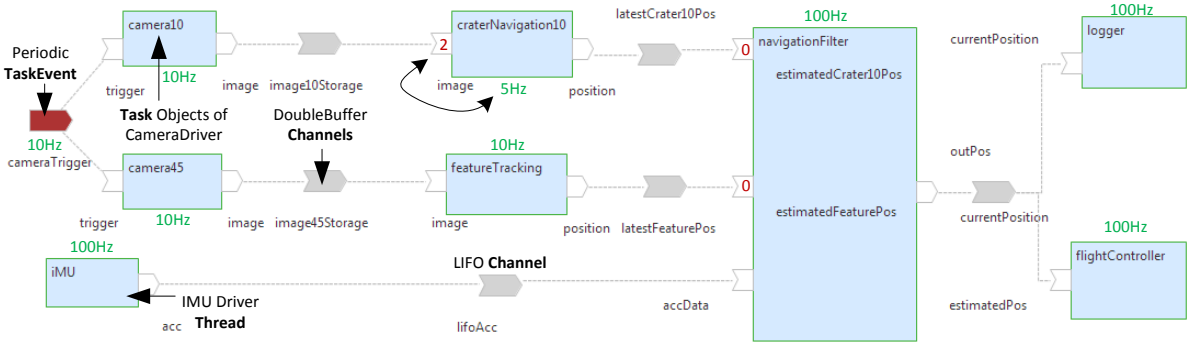

Fig. 7. TML system diagram of an application of the Tasking Framework in a real-time system for optical navigation in space.

pattern is considered in RODOS. In this pattern, publishers label messages according to predefined topics; one or more subscribers to a given topic receive all messages that are published under this topic. Unlike RODOS, a task in Tasking Framework pushes its output data into the associated channel and notifies the input of the next task/s with no call of the execute method of that task. However, Taking Frameworks is not an operating system.

VII. Conclusion

In this paper we presented our event-driven multithreading execution platform and software development library: Tasking Framework. It is dedicated to develop space applications which perform on-board data processing and sophisticated control algorithms, and have high computational demand. Tasking Framework has been used in already flying satellites, e.g., Eu:CROPIS.

Tasking Framework is neither a testing platform nor an operating system. It is a set of abstract classes with virtual methods to develop and execute data-driven on-board software systems on single-core as well as parallel architectures. It is compatible with the POSIX-based real-time operating systems, mainly RTEMS and FreeRTOS. Tasking Framework is supported with a model-driven tool environment (TML) that can be used to generate the API and its communication code.

Our plan is to make Tasking Framework open source. A bare-metal implementation is also on our agenda.

References

[1] G. Ortega and R. Jansson, "GNC application cases needing multi-core processors." https://indico.esa.int/event/62/contributions/2787, October 2011. 5th ESA Workshop on Avionics Data, Control and Software Systems (ADCSS).

[2] German Aerospace Center (DLR), "Open modUlar sofTware PlatfOrm for SpacecrafT (OUTPOST)." https://github.com/DLR-RY/outpost-core. accessed 2019-04-14.

[3] K. Brie, W. Bärwald, F. Lura, S. Montenegro, D. Oertel, H. Studemund, and G. Schlotzhauer, "The BIRD mission is completed for launch with the PSLV-C3 in 2001," in *3th IAA Symposium on Small Satellites for Earth Observation* (R. Sandau, H.-P. Röser, and A. Valenzuela, eds.), pp. 323–326, Wissenschaft und Technik Verlag Berlin, April 2001.

[4] R. Olfati-Saber, "Distributed Kalman filtering for sensor networks," in *2007 46th IEEE Conference on Decision and Control*, pp. 5492–5498, Dec 2007.

[5] S. Theil, N. A. Ammann, F. Andert, T. Franz, H. Krüger, H. Lehner, M. Lingenauber, D. Lüdtke, B. Maass, C. Paproth, and J. Wohlfeil, "ATON (autonomous terrain-based optical navigation) for exploration missions: recent flight test results," *CEAS Space Journal*, March 2018.

[6] O. Maibaum and A. Heidecker, "Software evolution from TET-1 to Eu:CROPIS," in *10th International Symposium on Small Satellites for Earth Observation* (R. Sandau, H.-P. Röser, and A. Valenzuela, eds.), pp. 195–198, Wissenschaft & Technik Verlag, April 2015.

[7] B. Weps, D. Lüdtke, T. Franz, O. Maibaum, T. Wendrich, H. Müntinga, and A. Gerndt, "A model-driven software architecture for ultra-cold gas experiments in space," in *Proceedings of the 69th International Astronautical Congress*, pp. 1–10, 2018.

[8] C. J. Treudler, H. Benninghoff, K. Borchers, B. Brunner, J. Cremer, M. Dumke, T. Gärtner, K. J. Höflinger, D. Lüdtke, T. Peng, E.-A. Risse, K. Schwenk, M. Stelzer, M. Ulmer, S. Vellas, and K. Westerdorff, "ScOSA - scalable on-board computing for space avionics," in *IAC 2018*, October 2018.

[9] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, D. de Niz, and G. von der Brüggen, "Many suspensions, many problems: a review of self-suspending tasks in real-time systems," *Real-Time Systems*, vol. 55, pp. 144–207, Jan 2019.

[10] Google, "Googletest - Google testing and mocking framework." https://github.com/google/googletest. accessed 2019-04-05.

[11] T. Franz, D. Lüdtke, O. Maibaum, and A. Gerndt, "Model-based software engineering for an optical navigation system for spacecraft," *CEAS Space Journal*, no. 0123456789, 2017.

[12] C. Atkinson and T. Kühne, "Model-driven development: A metamodeling foundation," *IEEE Computer Society*, 2003.

[13] E. Gamma, R. Helm, J. Ralph, and J. Vlissides, "Structural patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software*, p. 196208, Addison-Wesley Professional, 1 ed., 1994.

[14] M. Fowler, "Generation gap," in *Domain-Specific Languages*, p. 571573, Addison-Wesley Signature, 2010.

[15] S. Sadvandi, F. Corbier, and E. Mevel, "Real time and interactive co-execution platform for the validation of embedded systems," in *9th European congress on embedded real-time software and systems*, 2018.

[16] Siemens AG, "Embedded Multicore Building Blocks ." https://embb.io/get_started.htm. accessed 2019-04-14.

[17] F. Dannemann and F. Greif, "Software platform of the DLR compact satellite series," in *Small Satellite Systems and Services Symposium*, 2014.

[18] S. Montenegro, "Network centric core avionics," in *2009 First International Conference on Advances in Satellite and Space Communications*, pp. 197–201, July 2009.

[19] S. Montenegro and F. Dannemann, "RODOS real time kernel design for dependability," in *Proceedings of DASIA 2009 Data Systems in Aerospace*, 2009.

[20] S. Montenegro, V. Petrovic, and G. Schoof, "Network centric systems for space applications," in *2010 Second International Conference on Advances in Satellite and Space Communications*, pp. 146–150, June 2010.

Towards Real-Time Checkpoint/Restore for Migration in L4 Microkernel based Operating Systems

Sebastian Eckl, David Werner, Alexander Weidinger, Uwe Baumgarten
Chair of Operating Systems
Technische Universität München
sebastian.eckl@tum.de, david.werner@tum.de, alexander.weidinger@tum.de, baumgaru@tum.de

***Abstract*—Future automotive systems will rely on multi-core hardware support and will be gradually exposed to mixed-criticality demands. Support for different kinds of context sensitive reaction behavior (e.g. fail-operational behavior) will be required, wherefore the concept of dynamic reconfiguration will extend existing well-established static system configurations. A promising approach is offered by migration of software components and processes, based on the operating system (OS) layer. Efficient snapshot creation within an embedded real-time environment is thereby critical. The suggested concept extends a real-time operating system (RTOS), based on L4 Fiasco.OC and the Genode OS Framework, by the Real-Time Checkpoint/Restore (RTCR) component, providing a solid base for checkpoint/restore in L4 microkernel based OSs. Additional optimizations regarding checkpointing were introduced and could partially be tested. The results demonstrate current shortcomings of the purely software-based design and underscore the assumption that a real-time capable C/R mechanism will have to rely on the support of dedicated hardware components.**

***Index Terms*—checkpoint/restore, microkernel, L4, migration**

I. Introduction & Motivation

Future development in Cooperative Intelligent Transport Systems (C-ITS) and Autonomous Driving will require distributed embedded real-time systems (DERTS) to cope with an ever increasing amount of software-based functionality. Hardware consolidation and system virtualization will enable future automotive systems to provide both required computing power and guarantee safety and security within a mixed-criticality multi-core environment. Separation kernels (e.g. PikeOS) apply microkernel design considerations to comparably combine software of different criticality on the same underlying hardware platform. Unfortunately, existing partitioning approaches lack an adequate degree of flexibility regarding adaptation at runtime. Considering required future automotive context-sensitive reaction behavior (e.g. fail-operational behavior) [1], existing techniques will reach their limits [2].

Aside from real-time systems, virtual machine migration techniques already provide dynamic reconfiguration for fault tolerance/availability or efficient resource management. By also building upon existing L4 microkernel-based principles, a conceptual foundation addressing the migration of software components and processes in DERTS was proposed within [3]. An L4 microkernel based operating system (OS) provides a minimalist base for a flexible homogeneous run-time environment (RTE), allowing for mixed-critical partitioning on multi-core hardware. Regarding migration, the OS was enhanced by two mechanisms: migration planning and migration execution. Execution hereby addresses functionality to support snapshot creation, transfer and restoration of a process to be migrated. This paper is focusing on snapshot creation, providing a design of a high-performance checkpoint/restore (C/R) mechanism for L4 microkernel based OS: the Real-Time Checkpoint/Restore (RTCR) open source software component[1].

II. Related Work

There already exist multiple C/R mechanisms, both implemented as kernel-space and user-space solutions, and mainly focusing on Linux or UNIX systems. A well established example for an implementation in kernel-space is *BLCR* [4]; *libckpt* [5], *CRIU* [6] and *Zap* [7] mark user-space implementations. To improve performance of such mechanisms, different ideas were developed over time. This varies from compiler-based solutions, like *porch* [8], the checkpointing mechanism by Bronevetsky et al. [9] or *lightweight memory checkpointing* [10] over VM migration optimizations [11] to hardware based optimizations like the ability to detect write access to memory pages [12], as implemented in [13] and [14]. Other possibilities are the ability to concurrently copy memory as described in [15] or the usage of sheaved memory as described in [16]. Additionally to the mostly Unix-based C/R solutions there already exist implementations specifically for microkernel-based operating systems like *OSIRIS* [17] or an implementation from Luan et al. [18]. Some solutions also address specific L4-based microkernels, like the kernel-space implementation *L4ReAnimator* [19] and the user-space solution *PointStart* [20], both for L4Re. Luan et al. [21] designed a solution for the seL4, which is implemented in kernel-space. Unfortunately, existing Linux/UNIX-based solutions mostly lack real-time focus and are not meant to be transferred to microkernel-based OSs. Both L4Re-based solutions are not targeting the migration aspect, i.e. the restoration on a different machine. By placing C/R policy inside the kernel, the seL4 based solution [21] violates the microkernel principle.

[1]https://github.com/argos-research/genode-CheckpointRestore-SharedMemory

III. Design Assumptions

RTCR is realized as a component in a RTOS based on the L4 Fiasco.OC microkernel[2] and the Genode OS Framework[3]. Genode systems consist of multiple components organized in a recursive, hierarchical tree structure, leading to a minimal trusted computing base (TCB) and a parent-child relationship between components, whereas the parent is responsible for the childrens' execution and has control over their lifecycles. A component is a program that runs in a dedicated sandbox, equipped with all access rights and resources the corresponding program needs. As root component, Genode *Core* has direct access to the kernel and thus to the hardware resources, which it hands out to other components in the form of services (e.g. CPUs are represented by the CPU service). One important type of resource are dataspaces (DS), which represent allocated physical memory. Genode and L4 both provide a capability-based security mechanism. RTCR's design properties align with L4 and Genode, by sticking to the capability-based security concept and upholding a minimal TCB by designing RTCR as a user-space component that keeps any kind of policy outside of the kernel. Designed as a stop/start mechanism, it guarantees the creation of a consistent snapshot, trying to keep system interruption as minimal as possible. By being transparent, RTCR ensures that components will be unaware of being checkpointed. Keeping up high performance and a system's real-time capability is of highest priority. RTCR has to be made fast enough to cope with timings expected in automotive systems (e.g. a switchover-time of max. 50ms) [1] and has to be made deterministic, to provide worst-case timing guarantees. As a comparable L4 microkernel based C/R mechanism has not been existing beforehand, this paper is focusing on the former aspect and its potential optimizations. The latter aspect will be subject to future work.

IV. Real-Time Checkpoint/Restore (RTCR)

RTCR exists in two distinct variants, differing in the way in which memory content is checkpointed: the shared memory (SM) approach (post-copy) and the redundant memory (RM) approach (pre-copy). The SM approach is able to checkpoint relevant (memory) data at snapshot creation time by explicitly stopping the system for a certain period of time, in order to create a consistent snapshot. The RM approach instead intercepts each memory write access call at runtime in order to copy relevant data to a backup area in parallel to program execution. Only non-memory related data (e.g. register data) has to be copied at snapshot creation time.

1) Shared Memory: As an user-space component, RTCR can acquire relevant knowledge only by observing the communication of a target component to the *Core* component and other server components. RTCR itself consists of the following four modules: *online storage*, *offline storage*, *interception* and *checkpointer*. Observation is realized by using custom implementations of all relevant services (*Core* services as well as user-space services), wrapping the original services and intercepting them. The *interception* ensures that desired functionality is still provided to target components, by handing out sessions of the custom services instead of delivering regular ones. Methods of the custom services fulfill two tasks: all relevant information that can be extracted from a method call is saved to distinct data structures called *online storage*, then the call is routed to the original service. The following services are hereby intercepted: *RAM*, *PD*, *CPU*, *RM* and *LOG* all from *Core* and *TIMER*. Further explanation can be found in the official Genode documentation [22]. The interception logic is paired with a checkpoint and a restore logic. At the time of the checkpoint creation, the *checkpointer* 1) stores a target component's capability space, 2) copies session state information from *online storage* to *offline storage*, which makes the data persistent, and 3) copies the memory content of the target component. Figure 1 depicts all components of RTCR. The restore process of RTCR is based on the *offline storage*. RTCR creates a new child component which is then filled with the information from it. This way, the original state of the component is recreated and it can continue its execution.

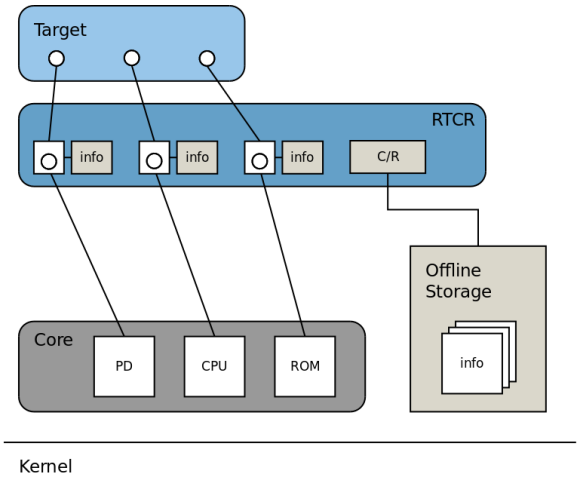

Fig. 1. Shared Memory based C/R

2) Redundant Memory: In contrast to SM, the RM version creates a copy of the target's memory during its runtime. As a proof-of-concept implementation the RM version uses instruction emulation. Apart from the different way of memory checkpointing, the workflow of SM and RM is identical. Genode offers the possibility to create managed DSs which, in contrast to regular DSs, do not represent a contiguous range of physical addresses but a range of virtual ones. Accessing a managed DS not backed by a regular DS leads to a page fault. Using the custom RAM service of RTCR, it is possible to hand out managed instead of regular DSs to the checkpointing target. Internally, RTCR also creates a regular DS but does not attach it to the managed DS. This way, the target produces a page fault each time it tries to write to its memory. The page fault is received by a fault handler in RTCR. Having access to the regular DS that is intended to store the target's memory and the DS that represents a redundant copy, the

[2]https://os.inf.tu-dresden.de/fiasco/
[3]https://genode.org/

fault handler can emulate the write call to *DS_orig* and the currently valid *DS_copy* instance. Therefore, the fault handler determines the corresponding instruction by inspecting the instruction pointer of the thread performing the access. The retrieval of the instruction from the binary is then followed by its decoding. After decoding the instruction, the memory access is performed on both DSs. Within RM checkpointing, an incremental approach is utilized. For the first checkpointing iteration, (*DS_copy*) marks a complete memory snapshot. For consecutive checkpointing iterations, only modifications since the last memory checkpoint are captured within instances of *DS_inc*, e.g. *DS_inc_1*. All other areas that don't have to be checkpointed again are referencing their precursor snapshot. At checkpoint time, the target is halted, all non-memory data is stored and redundant memory writing is directed to a new instance of *DS_inc*, e.g. *DS_inc_2*. At this point, target execution is resumed and snapshot creation is started in parallel. A flattener thread is hereby merging the content of the old *DS_inc*, e.g. *DS_inc_1*, into *DS_copy*, leading to a consistent memory snapshot. The behavior of the instruction emulation and the redundant memory copying is depicted in figure 2. In the future, the inefficient instruction emulation will be replaced by a dedicated hardware tracing (and copying) component (see chapter V-B).

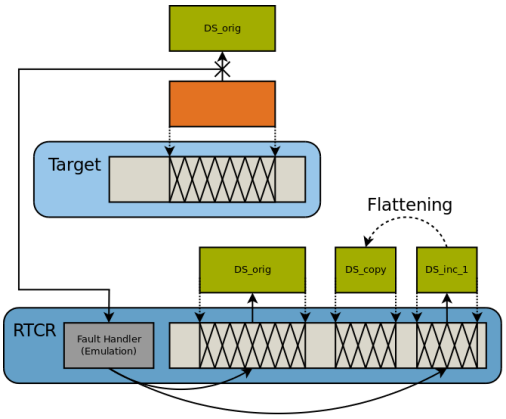

Fig. 2. Redundant Memory based C/R, with managed DSs (orange) and regular DSs (green)

V. RTCR OPTIMIZATIONS

As can be seen from figure 6, main bottlenecks of the SM approach are the strict sequential workflow of RTCR and the long-lasting duration of memory-related checkpointing aspects (see methods *_prepare_ram_sessions()* and *_checkpoint_dataspaces()* and their corresponding sub-methods). The former is part of the online-to-offline copying and sets up the DS translations while the latter is responsible for memory checkpointing. As during a regular workload there will be a lot more checkpoints created than in the end will have to be restored, our proposed optimizations only target the former process. Main focus of optimization methods hereby lies on reducing the overall snapshot creation time. Considering as well the RM approach, a current bottleneck is the inefficient emulation-based tracing of memory accesses. As SM and RM version of RTCR share the same code for most of their implementations, optimizations regarding these shared parts can be applied to both.

A. Software-based optimizations

Based on the mechanisms described in chapter IV, several software-based optimizations have been designed and implemented.

1) Incremental checkpointing: Already applied in the basic RM approach, incremental checkpointing is a technique that enables the checkpointing process to only store memory areas which were modified since the last time a checkpoint was created, resulting in reduced amount of memory that needs to be copied. The mechanism requires additional information about write accesses to DSs, wherefore managed DSs are utilized (see chapter IV-2). The managed DSs are hereby backed by multiple smaller regular DSs. The number of regular DSs per managed DS depends on the desired granularity that shall be achieved. Initially the designated DSs are detached from the managed DS. When the memory area is first accessed, a page fault is triggered. A fault handler then attaches the regular DS that corresponds to the accessed address and marks it as dirty. During the next snapshot creation only the content of regular DSs that have been marked dirty needs to be copied. After the checkpointer is finished, it detaches all regular DSs from all managed DSs, so that the dirty flagging can be repeated for the next iteration of the checkpointing process.

2) Read-Only Memory: Both the emulation mechanism of the RM approach and the incremental optimization mechanism exploit nondeterministic page faults for memory access detection. Hereby, both read and write accesses trigger page faults although only the detection of write accesses is required. Albeit the system is already able to distinguish the type of access responsible for a page fault, the regular DSs need to be attached either way, to be able to read from/write to them. The concept of read-only memory (ROM) describes memory areas which can be read but not written to. Hereby, read-only attachments can be realized by modifying the attachment mechanism of Genode region maps [22]. Each time a DS is attached to an address space, a mapping is created. By enabling these mappings to be read-only, any type of DS that is attached can be made non-writeable implicitly. As part of the address resolution it can be determined whether a write to a DS corresponding to a certain mapping is allowed or not. The read-only attachments thus allow writable DSs to become read-only depending on the way they were attached to the address space.

3) Copy-on-write: Copy-on-write (COW) is an optimization for the SM approach that targets the decoupling of the memory copying process from the rest of the checkpointing, trying to mitigate the halting of the target component. As components constantly alter their state during execution, there is no other way than interrupting to create a consistent snapshot. Applying COW, a target is allowed to continue its execution before the memory copying is started. Should the target access

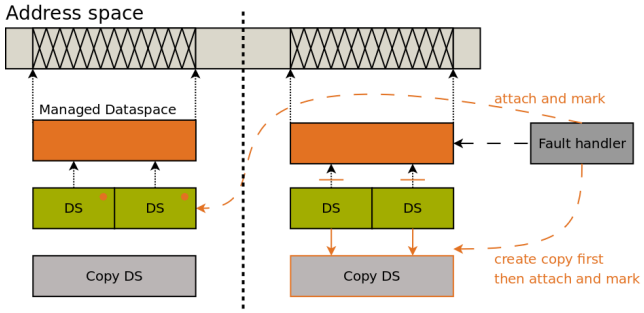

Fig. 3. Copy-on-write mechanism

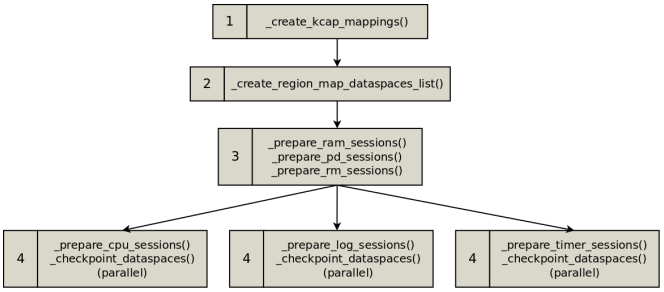

Fig. 4. Dependency graph showing the dependencies between individual RTCR methods and the applied parallelization strategy

a memory area that is not yet checkpointed, the COW logic performs the copying on demand. The COW functionality is hereby integrated in the incremental mode of RTCR, by piggybacking the corresponding logic on the fault handler that is responsible for marking the regular DSs as dirty. As the fault handler of the incremental approach already detects the first access to each DS after every checkpointing procedure, it is easy to combine this detection with a copying process if the affected DS is not yet checkpointed. Figure 3 shows the additional COW behavior of the fault handler. The left side marks the case, in which the DS is already copied by the checkpointer. Hereby, the fault handler acts like before the COW logic was added and simply attaches and marks the regular DS. On the right side, the DS is not yet copied and the fault handler executes the COW logic before it attaches the regular DS and marks it as dirty.

4) Parallelization: As multiple CPU cores allow for the parallel execution of threads, a multi-threaded implementation of RTCR might accelerate the strict sequential basic workflow. To be able to choose a fitting parallelization technique, first of all the dependencies between all checkpointing tasks need to be analyzed. As a parallel execution of dependent methods is impossible, dependencies represent a limitation to the degree of parallelism. The parallelization scheme depicted in figure 4 shows a combination of both horizontal and vertical parallelization [23]. Based on the identified dependencies and conflicting accesses to data structures, a horizontal approach clusters RTCR in one strict sequential phase in the beginning and one parallel phase in the end of the checkpointing process. Parallelization only affects parts of the capturing of the session state (online-to-offline copying, see *_prepare_XX_sessions()*) and the copying of memory contents (see *_checkpoint_dataspaces()*). As most of the checkpointing tasks require very little amounts of time, vertical parallelization is only applied to the memory copying checkpointing methods (see several instances of *_checkpoint_dataspaces()*). Multiple threads can operate on the DS translation list and copy DS contents in parallel.

B. Hardware-based optimization

As main bottleneck of the pure software-based implementations, the duration of tracing memory access and checkpointing memory could be identified. In order to accelerate the existing solutions, a combination between the RTOS and reconfigurable hardware seemed promising. Addressing the aspects efficient memory tracing and copying, a custom memory interceptor was developed with the help of a FPGA. This hardware component redirects all memory accesses done by the OS to a FPGA based peripheral and intercepts memory access in order to distinguish between read and write access calls, trace write calls and write respective data to a redundant backup during runtime (see figure 5). The work is based on the concept of an AXI-based memory bridge done by Li et al. [24].

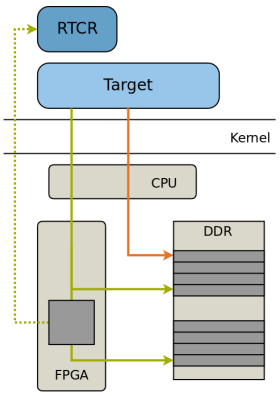

Fig. 5. Hardware-based memory interceptor: traditional memory access (orange) vs. redirected memory access and redundant copying (green) and gathering tracing information (green dotted)

VI. Implementation

The proposed OS components have been implemented prototypically based on a modified version of the L4 Fiasco.OC and the Genode OS Framework (version 16.08). The underlying hardware platform is based on the ARMv7-A architecture, represented in form of an emulated Cortex-A9 quad-core processor (QEMU PBX-A9; QEMU in version $\geq$2.9). FPGA-based implementation was done on physical hardware, utilizing the Xilinx Zynq-7000 SoC, based on the Digilent Zybo and Avnet Zedboard.

VII. Evaluation

Tests were conducted based on an unoptimized SM version of RTCR to investigate RTCR's basic performance and based on a modified version, containing the incremental optimization mechanism. The tests were executed on QEMU PBX-A9 and

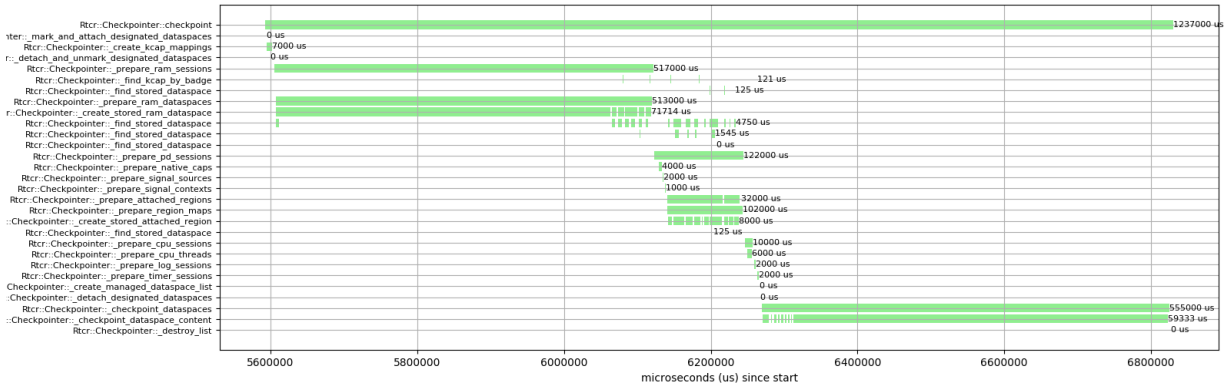

Fig. 6. Base Case: Shared Memory - Full Mode

consist of a component that can be configured to allocate a certain amount of memory and modifies allocated memory within an endless loop. Variations were done regarding scaling the amount of memory and the number of DSs to be checkpointed. Multiple iterations of the checkpointing process with changing amounts of target memory were done.

A. Base case: Full Mode

Within the base case, RTCR is using the full checkpointing mode, meaning that all memory allocated by a target component has to be saved. Several scenarios were tested, whereas varying amount of memory (up to 128MB in total) was checkpointed either represented by a single DS or split into a varying number of DSs, exemplarily demonstrated by the granularity levels 8, 32 and 64. Allocated memory was thereby distributed equally over the DSs. As can be observed in figure 7, the duration of the checkpointing process scales linearly with the amount of memory that needs to be checkpointed. Reason for that is the explicit copying of memory contents using a software-based *memcpy* function. With the same amount of memory distributed over more DSs the duration increases even more. So, in general, the performance of the checkpointer gets worse with an increasing number of DSs. This is caused by the management overhead that the handling of multiple DSs entails. However, if the respective DS size exceeds 1MB, the DS induced performance loss decreases.

B. Incremental Mode

In the best case no modifications were done, so using the incremental checkpointing mode means no memory has to be saved at all. Therefore, worst case runtime shall be evaluated, which is equivalent to full mode (see chapter VII-A). The target component hereby allocates 32MB of memory in a single managed DS. RTCR fills the managed DS according to the chosen granularity with a varying number of regular DSs. The size of a regular DS is thereby made up by granularity * page size (which is 4KB, the smallest possible unit for a regular DS). As depicted in figure 8, the range of tested sizes reaches from 128*4KB = 512KB to 1024*4KB = 4MB per regular DS. A greater number of pages means coarser granularity regarding checkpointing, so less regular DSs per managed DS are used (e.g. 32MB / 4MB = 8 DSs), which leads to a better performance of the checkpointer. Nevertheless, incremental checkpointing of the full managed DS performs drastically worse than the full mode in chapter VII-A. If the target component allocates large amounts of memory while the checkpointer is configured to use incremental checkpointing, the mechanism will trigger the creation of a vast amount of regular DSs. The kernel is unable to handle that many capabilities, resulting in an overflow of the capability space in *Core*. As capability spaces can not be arbitrarily large, the only way to prevent this scenario is to choose a reasonable size for the granularity. However, this limits the effectiveness of the incremental checkpointing as a coarser granularity generally leads to the copying of more unmodified memory areas. Considering the test case, creation of more than 32MB / (128*4KB) = 64 DSs leads to a system crash.

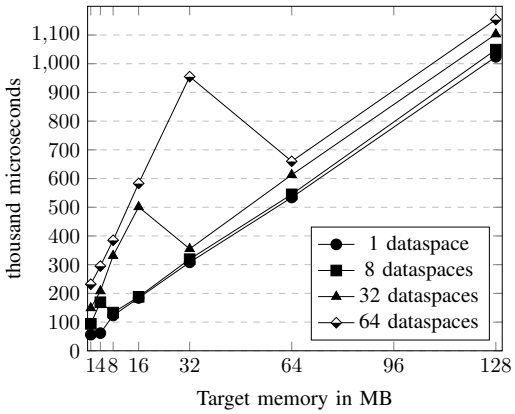

Fig. 7. Base Case: Shared Memory - Full Mode

VIII. Future Work

Current shortcomings affect the RTCR restoration process, as RTCR currently is limited to creating a memory dump of components, due to being unable to C/R all required Fiasco.OC capabilities. Ongoing development targets the porting of RTCR from outdated Genode 16.08 to the most recent Genode version. For performance comparison, RTCR shall

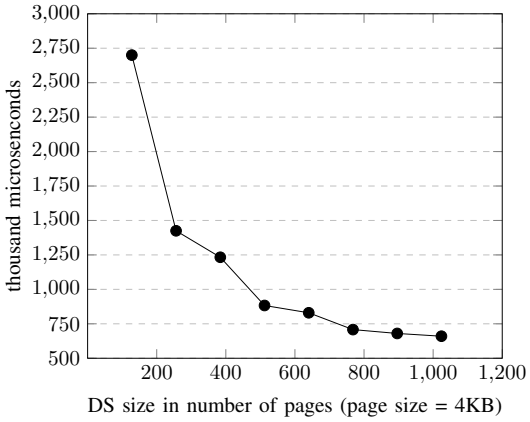

Fig. 8. Incremental Case: Shared Memory - Worst Case

also be ported to the combination of seL4 and Genode. Further software-based optimizations include the porting of RTCR to *Core* to reduce IPC, and extending parallelization to multiple instances of RTCR running simultaneously to increase the throughput of snapshot creation. Further hardware optimization approaches target the utilization of existing co-processors (e.g. ARM Cortex-M) or the development of additional hardware components, like custom FPGA-based co-processors (e.g. based on AXI CDMA) or the modification of existing CPU design (e.g. MMU adaption based on RISC-V), for acceleration of memory tracing and copying. Further evaluation shall target the COW and parallelization optimization mechanisms. Coupling RTCR with the hardware-based memory interceptor will also allow for tests of the RM version. As testing based on emulation is insufficient, physical hardware platforms shall be utilized (e.g. NXP i.MX6 based).

IX. Conclusion

The design of RTCR provides a solid base for C/R in L4 microkernel based OSs. Additional optimizations for increasing performance were introduced and could partially be tested. Software-based optimizations alone most likely won't lead to a real-time capable C/R mechanism. Future work will therefore concentrate on the development of dedicated hardware components for further acceleration.

Acknowledgment

This work is partially funded by the German Federal Ministry of Education and Research under grant no. 01IS12057 through the Software Campus (project HaCRoM).

References

[1] K. Becker, B. Schätz, M. Armbruster, and C. Buckl, "A Formal Model for Constraint-Based Deployment Calculation and Analysis for Fault-Tolerant Systems," in *Software Engineering and Formal Methods* (D. Giannakopoulou and G. Salaün, eds.), vol. 8702, pp. 205–219, Cham: Springer International Publishing, 2014.

[2] G. Weiss, M. Zeller, D. Eilers, and R. Knorr, "Towards Self-organization in Automotive Embedded Systems," in *Autonomic and Trusted Computing* (J. González Nieto, W. Reif, G. Wang, and J. Indulska, eds.), vol. 5586, pp. 32–46, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[3] S. Eckl, D. Krefft, and U. Baumgarten, "Migration of components and processes as means for dynamic reconfiguration in distributed embedded real-time operating systems," in *OSPERT 2017*, 2017.

[4] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, p. 494, IOP Publishing, 2006.

[5] J. S. Plank, M. Beck, G. Kingsley, and K. Li, *Libckpt: Transparent checkpointing under unix*. Computer Science Department, 1994.

[6] "Criu." https://www.criu.org/Main_Page. Accessed: 2019-04-20.

[7] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: A system for migrating computing environments," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 361–376, 2002.

[8] V. Strumpen *et al.*, "Compiler technology for portable checkpoints," *submitted for publication (http://theory. lcs. mit. edu/strumpen/porch. ps. gz)*, vol. 477, pp. 481–484, 1998.

[9] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz, "Application-level checkpointing for shared memory programs," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 5, pp. 235–247, 2004.

[10] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum, "Lightweight memory checkpointing," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 474–484, IEEE, 2015.

[11] D. Kapil, E. S. Pilli, and R. C. Joshi, "Live virtual machine migration techniques: Survey and research challenges," in *2013 3rd IEEE International Advance Computing Conference (IACC)*, pp. 963–969, IEEE, 2013.

[12] M. Vasavada, F. Mueller, P. H. Hargrove, and E. Roman, "Comparing different approaches for incremental checkpointing: The showdown," in *Linux Symposium*, p. 69, 2011.

[13] D. Vogt, A. Miraglia, G. Portokalidis, H. Bos, A. Tanenbaum, and C. Giuffrida, "Speculative memory checkpointing," in *Proceedings of the 16th Annual Middleware Conference*, pp. 197–209, ACM, 2015.

[14] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, p. 9, IEEE Computer Society, 2005.

[15] K. Li, J. F. Naughton, and J. S. Plank, *Real-time, concurrent checkpoint for parallel programs*, vol. 25. ACM, 1990.

[16] M. E. Staknis, "Sheaved memory: architectural support for state saving and restoration in pages systems," in *ACM SIGARCH Computer Architecture News*, vol. 17, pp. 96–102, ACM, 1989.

[17] K. Bhat, D. Vogt, E. van der Kouwe, B. Gras, L. Sambuc, A. S. Tanenbaum, H. Bos, and C. Giuffrida, "Osiris: Efficient and consistent recovery of compartmentalized operating systems," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 25–36, IEEE, 2016.

[18] G. Luan, Y. Bai, C. Wang, J. Zeng, and Q. Chen, "An efficient checkpoint and recovery mechanism for real-time embedded systems," in *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pp. 824–831, IEEE, 2018.

[19] D. Vogt, B. Döbel, and A. Lackorzynski, "Stay strong, stay safe: Enhancing reliability of a secure operating system," in *Proceedings of the Workshop on Isolation and Integration for Dependable Systems (IIDS 2010), Paris, France, April 2010*, 2010.

[20] J. Kulik, "Client-independent checkpoint/restart of l4re-server-applications," 2015.

[21] G. Luan, Y. Bai, L. Xu, C. Yu, C. Wang, J. Zeng, Q. Chen, and W. Wang, "Towards fault-tolerant task backup and recovery in the sel4 microkernel," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 721–726, IEEE, 2018.

[22] N. Feske, *Genode Operating System Framework Foundations*. 2018.

[23] K.-C. Kim, *Vertical and Horizontal Program Parallelization Techniques*. PhD Thesis, University of California at Irvine, Irvine, CA, USA, 1992.

[24] L. W. Li, G. Duc, and R. Pacalet, "Hardware-assisted Memory Tracing on New SoCs Embedding FPGA Fabrics," in *Proceedings of the 31st Annual Computer Security Applications Conference on - ACSAC 2015*, (Los Angeles, CA, USA), pp. 461–470, ACM Press, 2015.

Analyzable and Practical Real-Time Gang Scheduling on Multicore Using *RT-Gang*

Waqar Ali, Michael Bechtel, Heechul Yun
University of Kansas
{wali, mbechtel, heechul.yun}@ku.edu

In this *presentation*, we will introduce a new real-time gang scheduling framework in Linux, called RT-Gang [2], and provide a brief tutorial and a demo of using the framework in a self-driving car application [3].

Emerging safety-critical real-time control systems in automotive and aviation applications often consist of a number of highly computationally expensive and data intensive workloads (e.g., deep neural networks) with strict real-time requirements for agile control (e.g., 50Hz). Guaranteeing timely execution of these real-time tasks is an important requirement for safety of the system. However, it is challenging to provide such a guarantee on today's highly integrated embedded computing platforms because they often show unpredictable and extremely poor worst-case timing behaviors that are hard to understand, analyze, and control [4], [8]—chiefly due to interference in shared memory hierarchies. Broadly, timing unpredictability is a serious problem especially in automotive and aviation industries. For example, Bosch, a major automotive supplier, reported "predictability on high-performance platforms" as a major industrial challenge for which the industry is actively seeking solutions from the research community [6]. In aviation, the problem was dubbed as "one-out-of-m" problem [7] because the current best practice for certification, which requires *evidence of bounded interference*, is to disable all but one core of a multicore processor [5].

RT-Gang [2] is a new real-time gang scheduling framework implemented in Linux to address the timing unpredictability problem on COTS multicore platforms for safety-critical real-time applications. In RT-Gang, all threads of a parallel real-time task form a real-time gang and the scheduler globally enforces a *one-gang-at-a-time* scheduling policy. When a real-time task is released, all of its threads are scheduled simultaneously if it is the highest priority real-time task, or none at all if a higher priority real-time task is currently in execution. Any idle cores, if exist, can be used to schedule best-effort tasks but their shared memory access rates are strictly regulated by a memory throttling mechanism to bound their impact to the real-time task. Specifically, each real-time task defines its tolerable maximum memory bandwidth budget, which is strictly enforced by a kernel level regulator for any co-scheduled best-effort tasks. (see Figure 1.)

RT-Gang eliminates the problem of contention in the shared memory hierarchy between real-time tasks by executing only one real-time task at any given time, which effectively transforms parallel real-time task scheduling on a multicore into the well-understood uni-core real-time scheduling problem. Because of the strong temporal isolation guarantee offered by RT-Gang, a real-time task's worst-case execution time (WCET) can be tightly bounded without making strong assumptions about the underlying hardware. Thus, RT-Gang can improve system schedulability while providing a mechanism to safely utilize all cores of a multicore platform.

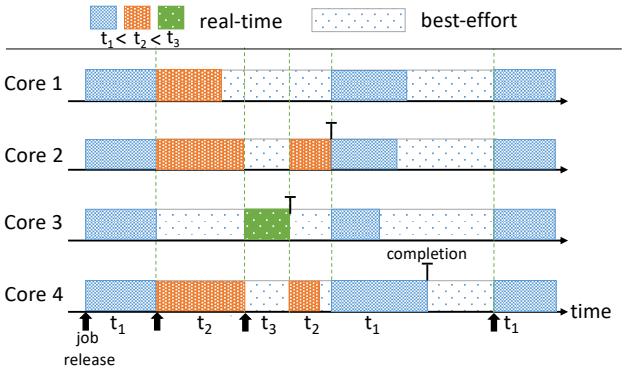

Fig. 1. Illustration of RT-Gang

RT-Gang is currently implemented as a "feature" of the standard Linux SCHED_FIFO real-time scheduler (kernel/sched/rt.c), which can be enabled or disabled dynamically at run-time [1]. In this presentation, we will provide a quick tutorial on how to use the feature, and demonstrate its effects on a real self-driving car application [3], which uses deep neural networks (processed by TensorFlow).

References

[1] RT-Gang code repository. https://github.com/CSL-KU/RT-Gang.
[2] W. Ali and H. Yun. RT-Gang: Real-Time Gang Scheduling Framework for Safety-Critical Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.
[3] M. G. Bechtel, E. McEllhiney, and H. Yun. DeepPicar: A Low-cost Deep Neural Network-based Autonomous Car. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2018.
[4] M. G. Bechtel and H. Yun. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.
[5] Certification Authorities Software Team. CAST-32A: Multi-core Processors. Technical report, Federal Aviation Administration (FAA), 2016.
[6] A. Hamann. Industrial challenges: Moving from classical to high performance real-time systems. In *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2018.
[7] N. Kim, B. C. Ward, M. Chisholm, J. H. Anderson, and F. D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. *Real-Time Systems*, 53(5):709–759, 2017.
[8] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

Notes

OSPERT 2019 Program

	Tuesday, July 9 2019
8:00 – 9:00	Registration
9:00 – 10:30	Welcome Session 1: Static Analysis ARA: Automatic Instance-Level Analysis in Real-Time Systems *Gerion Entrup, Benedikt Steinmeier, Christian Dietrich* Boosting Job-Level Migration by Static Analysis *Tobias Klaus, Peter Ulbrich, Phillip Raffeck, Benjamin Frank, Lisa Wernet, Maxim Ritter von Onciul, Wolfgang Schröder-Preikschat*
10:30 – 11:00	Coffee Break
11:00 – 12:30	Session 2: The Wild World Keynote talk: *Time-domain determinism using modern SoCs* *David Haworth, Elektrobit Automotive GmbH* Experiments for Predictable Execution of GPU Kernels *Flavio Kreiliger, Joel Matejka, Michal Sojka and Zdenek Hanzalek*
12:30 – 14:00	Lunch
14:00 – 15:30	Session 3: Platforms Event-Driven Multithreading Execution Platform for Real-Time On-Board Software Systems *Zain A. H. Hammadeh, Tobias Franz, Olaf Maibaum, Andreas Gerndt, Daniel Lüdtke* Towards Real-Time Checkpoint/Restore for Migration in L4 Microkernel based Operating Systems *Sebastian Eckl, David Werner, Alexander Weidinger, Uwe Baumgarten* Analyzable and Practical Real-Time Gang Scheduling on Multicore Using RT-Gang *Waqar Ali, Michael Bechtel, Heechul Yun* Closing
15:30 – 16:00	Coffee Break

	Wednesday, July 10th – Friday, July 12th 2019
	ECRTS main conference.