PROCEEDINGS OF

OSPERT 2016

the 12th Annual Workshop on
Operating Systems Platforms for Embedded Real-Time Applications

July 5th, 2016 in Toulouse, France

in conjunction with

ECRTS

the 27th Euromicro Conference on Real-Time Systems
July 6–8, 2016, Toulouse, France

Editors:
Robert KAISER
Marcus VÖLP

Contents

Message from the Chairs **3**

Program Committee **3**

Keynote Talk **5**

Session 0: Short Paper **7**

Towards versatile Models for Contemporary Hardware Platforms
Hendrik Borghorst, Karen Bieling, Olaf Spinczyk 7

Session 1: Multicore and Parallel Systems **11**

A communication framework for distributed access control in microkernel-based systems
Mohammad Hamad, Johannes Schlatow, Vassilis Prevelakis, Rolf Ernst 11

Tightening Critical Section Bounds in Mixed-Criticality Systems through Preemptible Hardware Transactional Memory
Benjamin Engel 17

GPU Sharing for Image Processing in Embedded Real-Time Systems
Nathan Otterness, Vance Miller, Ming Yang, James H. Anderson, F. Donelson Smith, Shige Wang 23

Session 2: Real-Time and Predictability **31**

Combining Predictable Execution with Full-Featured Commodity Systems
Adam Lackorzynski, Carsten Weinhold, Hermann Härtig 31

Timeliness Runtime Verification and Adaptation in Avionic Systems
José Rufino and Inês Gouveia 37

Session 3: OS and System Modelling **43**

Effective Source Code Analysis with Minimization
Geet Tapan Telang, Kotaro Hashimoto, Krishnaji Desai 43

Towards Real-Time Operating Systems for Heterogeneous Reconfigurable Platforms
Marco Pagani, Mauro Marinoni, Alessandro Biondi, Alessio Balsini, Giorgio Buttazzo 49

An implementation of the flexible spin-lock model in ERIKA Enterprise on a multi-core platform
Sara Afshar, Maikel P.W. Verwielen, Paolo Gai, Moris Behnam, Reinder J. Bril 55

Program **62**

Message from the Chairs

Welcome to Toulouse, France and welcome to OSPERT'16, the 12th 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 Adam Lackorzynski of Kernkonzept GmbH, Germany. He will present his views and experience in transitioning concepts that originate from research into a corporate setting. We are delighted that Adam volunteered to share his experience and perspective, as the exchchange between academics and industry has always been one of OSPERT's goals.

As a new feature this year, we will try to initiate an open discussion among workshop participants about important open research challenges in real-time operating systems. Contributors to the workshop have been asked in advance to suggest topics they would like to see discussed. Their suggestions have been collated and will be presented for discussion at the workshop.

The workshop received a total of fifteen submissions, three of which were in the short-paper format. All papers were peer-reviewed and nine papers were finally accepted. Each paper received at least three individual reviews.

The papers will be presented in three sessions. The first session includes three papers that explore approaches to real-time multicore and parallel systems design. Following the lunch break, we expect to have a lively discussion on open research challenges in real-time operating systems. Two interesting papers from the context of real-time and predictabilty will be presented in Session 2, and, last but not least, the third session will present three compelling papers addressing topics from the broad range of system modelling.

OSPERT'16 would not have been possible without the support of many people. The first thanks are due to Gerhard Fohler, Rob Davis and the ECRTS steering committee for entrusting us with organizing OSPERT'16, 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.

Our special thanks go to the program committee, a team of ten experts from seven different countries, 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.

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,

Robert Kaiser
RheinMain University of Applied Sciences
Wiesbaden, Germany

Marcus Völp
University of Luxembourg
Luxembourg

Program Committee

Andrea Bastoni, *SYSGO AG*
Michael Engel, *multicores.org*
Paolo Gai, *Evidence Srl*
Shinya Honda, *Nagoya University*
Adam Lackorzynski, *Kernkonzept / TU Dresden*
Daniel Lohmann, *FAU Erlangen-Nuernberg*
Chanik Park, *Pohang University of Science and Technology*
Pavel Pisa, *Czech Technical University Prague*
Linh Thi Xuan Phan, *University of Pennsylvania*
Richard West, *Boston University*

Keynote Talk

From Research to Reality: Releasing System Software to the Masses

Adam Lackorzynski
Kernkonzept GmbH

In this talk I will share some of our experiences made so far while transitioning a university-founded research project into real life. Work on L4 systems started about two decades ago and has gone through various ups and downs. With our latest version, the L4Re system, we finally had the opportunity to be part of technologically challenging projects that gave us the possibility to found our company, Kernkonzept, and move on beyond university's life. Our projects since then include a multi-VM secure smartphone and world's first open-source bare-metal hypervisor platform for the MIPS architecture. The ride might be bumpy from time to time, but it is well worth pursuing.

Dr. Adam Lackorzynski is a post-doctoral researcher with Prof. Hermann Härtig and the Operating-Systems Group of Technische Universität Dresden. He is a co-founder of Kernkonzept, the technology company behind L4Re, a microkernel-based operating system framework. Adam received his Ph.D. in Computer Science from Technische Universität Dresden. His research interests focus on secure operating systems for real-time and virtualization use-cases in a wide range of application fields. The L4Re system universally targets these fields and is being applied, for example, in secure communication devices such as mobile phones and routers. Adam is a main architect of the L4Re system, which originated from TU Dresden and is now being developed by Kernkonzept.

Towards versatile Models for Contemporary Hardware Platforms

Hendrik Borghorst, Karen Bieling and Olaf Spinczyk
Department of Computer Science 12
Technische Universität Dortmund, Germany
e-mail: {hendrik.borghorst, karen.bieling, olaf.spinczyk}@tu-dortmund.de

***Abstract*—The demand for computationally intensive workloads in the domain of real-time systems is growing which needs to be satisfied with more capable hardware. Cheap but powerful multi-core hardware seems to be a good solution but these processors often lack a good predictability. An operating system can hide measures to regain the predictability, like cache management but to do so a good knowledge of the hardware is required. A problem with these measures is that they are usually not portable and require a lot of work to adapt them to new platforms. It is desirable to generate the platform-specific code from abstract architecture descriptions to get a portable operating system that adapts itself to the specific hardware properties of modern hardware to provide a predictable execution environment.**

I. Introduction

Workloads within the real-time domain are getting more and more computationally intensive with the automotive industry pushing for autonomous cars, real-time face detection systems in security systems or power supply line monitoring for smart-grids. To meet this demand and at the same time reduce the cost of the hardware it is preferred to use cheap standard hardware with capable multi-core processors. But the low price for high performance computing power comes at the cost of loss of predictability.

These multi-core processors are usually designed to share resources to keep both the energy consumption and the price low. As a result of this, the timing behavior of these processors is not predictable and therefore they are not directly suitable for the use in real-time systems. The primary sources of unpredictability are caches [1], buses [2] and the main memory [3]. These sources of unpredictability have been in the focus of research for some time. Software-based control of the content of the shared caches has been proven to be an effective instrument to reduce the unpredictability of caches [4]. Rescheduling of memory accesses also was shown to be a viable mean to improve the memory access behavior [3].

These approaches can be used to reduce the unpredictability of modern hardware but often require special knowledge of the hardware and software a system uses. This could lead to an additional complexity for the system developers because they have to take into account on what hardware their code runs. One example is the alignment of data structures to cache line sizes. Instead these platform-specific measures should be handled by the operating system. In the past we presented an operating system concept that explicitly manages what data is in the cache, to get a more predictable system[5]. A problem with an approach like this is that it adds even more platform-specific code to the operating system which should be avoided. Instead we would like to write code that is normally platform-specific, in a new generic way to reuse it for all platforms. To do so we present an approach that uses an domain-specific language to describe an hardware architecture that can be used to generate code for low-level operating system functions like context switching, cache management, memory protection and other low-level functions that need to be written for every new hardware platform.

On the other hand an operating system also needs good information about the hardware it uses to fully utilize all the resources as good as possible. To achieve this the system needs a comprehensive model about the available resources and the timing behavior of a platform. An empirical approach to generate such a model can be used and the methods to do so can also be generated by the code generation. Profiling of a hardware architecture is used as a case study for this paper as it is usually a complex task because the profiling code has to be written in a low-level assembly language [6]. The model that is generated should provide essential information to the operating system at runtime and during the compilation to optimize it as much as possible.

In the following section we present an approach that utilizes a generic domain-specific language to describe hardware architectures with all their details needed to generate platform-specific code that can be used instead of manually written hardware-adaption code.

II. Approach

To specify an architecture we chose an approach with a domain-specific language (*DSL*). A language to model a hardware architecture, needs to be flexible enough to be able to specify current and upcoming architectures. This means that it should not have limitations, how the memory system of a architecture is structured. The language should be able to model a processor with multiple scratchpad memories for one processor core and a *NUMA*-based architecture just as well. To completely model a memory hierarchy it is also important to represent the interconnects between components like memories or processor cores correctly to ensure that the operating system can later take full advantage of measures to increase the performance and predictability of the hardware.

```
architecture ExampleArch {
 Memory RAM {}
 Memory L2Cache : RAM {}
 Memory Cache0 : L2Cache {}
 Memory Cache1 : L2Cache {}
 Processor CPU0 : Cache0 {}
 Processor CPU1 : Cache1 {}

 ISA {
  registers { R%[0..15] }
  instructions {
      add_const ADD: dest , arg , #arg const
      add_reg ADD: dest , arg , arg
    }
  }
}
```

Fig. 1: Example of an architecture description

Besides the memories, interconnects and computing units the architecture description needs to specify the instruction set architecture (*ISA*) of the available computing units. This ISA description is used for the code generator and contains details about the available registers, with the names used by the assembler for the architecture, and a basic set of assembly instructions. For heterogeneous architectures it is also possible to specify multiple ISAs for one architecture. So that different processor cores could use different ISAs.

An example representation of an architecture is shown by Figure 1. It consists of two processors which are each connected to a private cache, that is connected to a shared level-2 cache. The last item in the memory hierarchy is a main memory called `RAM`. The example architecture also included the register specification for the registers R0 to R15. The interconnects between multiple components are directly derived from the inheritances, for example in Figure 1 level-2 cache is connected to both the private `Cache0` and `Cache1`.

The `instructions` block includes all platform-specific assembly instructions needed for the abstract assembly language for the code generator. As an example Figure 1 only shows two instructions to add two values. Once with a constant and once with a value residing in another register. This language also allows to model a *NUMA*-based architecture by specifying multiple `RAM`-components that are only connected to one processor unit.

Figure 2 lists an example of a memory component. It describes a exemplary memory of the example architecture. To generate low-level operating system code it is necessary to specify some parameters that the operating system can use to optimize itself to the target architecture. These parameters include properties like the cache-line length (`minAccessSize`) or where a memory is mapped to in the address space.

In addition to the architecture description an abstract low-level development language needs to be defined. This language

```
RAM {
        wordLength: 4 // Bytes
        minAccessSize: 16 // Bytes
        startAddress: 0x40000000
        size: 2G
}
```

Fig. 2: Example of a memory component description

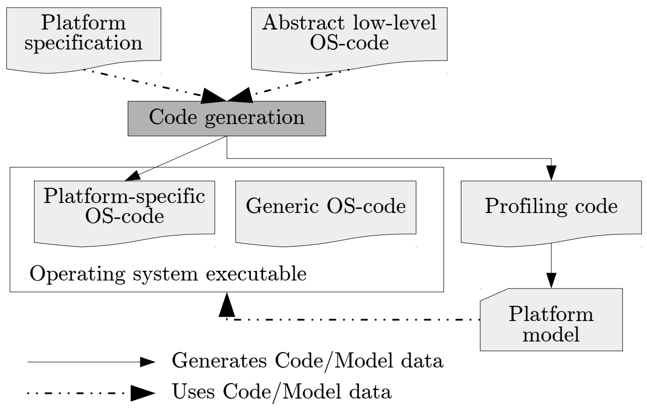
```
ram_benchmark {
  move(dest reg:0,arg %[bmStart_<BM>])
  move(dest reg:1,arg %[bmEnd_<BM>])

  jmp_mark(arg "loop_begin:")
  measure_start
  load(dest reg:3,src *reg:0)
  measure_end

  add_const(dest reg:1,arg reg:1,
  arg <WordLength>)
  cmp(arg reg:0,arg reg:1)
  cond_jump_lt(arg "loop_begin")
}
```

Fig. 3: Simple memory benchmark in abstract assembly code

is an abstract form of an assembly language that can be translated to platform-specific assembly code via the code generator. To do so the architecture description has to specify a minimal set of assembly instructions that are necessary for the code generator. The abstract assembly language can then be used to write low-level operating system code like context switching, cache flushing and time measurements in an abstract way so that it has to be done only once.

An example how to use the abstract assembly language is given with Figure 3. It depicts a memory read performance profiler. The profiling starts with the preparation of several constant values that are necessary to run the code like such as limits of the benchmark range. The next step is the creation of a label to create a loop over a certain benchmark range. Inside this loop is an abstract load instruction surrounded with two abstract methods that handle the measurement of elapsed clock cycles. The content of these functions is omitted here to keep the listing short. Each assembler instruction needs certain arguments. Some of them are register values and some of them constants which has to be annotated at the moment. Also the registers need to be allocated manually but we like to improve this in the feature with register allocation techniques borrowed from compiler research.

With the architecture description and the abstract assembly code it is possible to develop a code generator that creates the operating system code for a specific hardware platform. A simplified overview of the process is given with Figure 4. The



Platform specification
Abstract low-level OS-code
Code generation
Platform-specific OS-code
Generic OS-code
Profiling code
Operating system executable
Platform model
Generates Code/Model data
Uses Code/Model data

Fig. 4: Concept of operating system with abstract code

code generation combines one specific platform architecture with the abstract code and generates the assembler code for the architecture. This is then integrated with the generic program code of the operating system. The code generator can also be used to create comprehensive profiling code for the creation of a timing behavior description for the platform, that can also be used by the operating system as a base for optimizations like cache management to get a predictable system.

III. Evaluation

As a proof of concept we chose to develop a memory read performance profiler with the abstract language, because it is essential for the operating system to have information on the platforms memory performance to get predictable execution times. We want to use the generated information within our prototype operating system for the cache management [5].

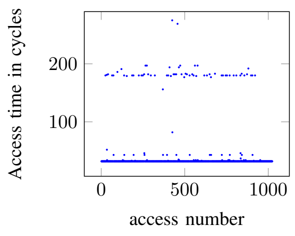
We implemented the presented languages with the Eclipse Modeling Framework (*EMF*) and *Xtext* [7] as this allows rapid prototyping of our domain specific languages and code generation which is helpful to quickly adapt the language to the changing demand as we developed our requirements to develop an operating system with abstract low-level code. We evaluated our implementation of the code generation with a *Samsung Exynos 4412* ARM-processor on a prototype operating system where no other load is simultaneously active.

The results of two generated benchmarks are shown on Figure 5. We evaluated two abstract benchmarks. One benchmark warms up the private cache of a processor by iterating over a memory range with the size of the private cache and finally iterates over the same range and measure the access times. The results are shown in Figure 5a. Another test is shown in Figure 5b where the main memory is tested without warming up so that we get many more cache misses.

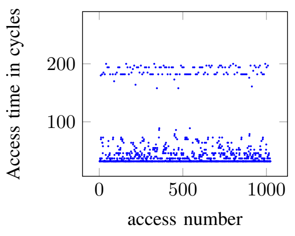
Although the profiling code for now was only generated for an *ARM* processor, it is possible to adapt it to other processors in the future.

IV. Conclusion & Future Work

We demonstrated that it is possible to create abstract low-level code that can be transformed to architecture-specific



(a) L1-Cache profiling



(b) RAM accesses

Fig. 5: Memory read benchmarks

assembly code by providing a simple architecture description. Although we could only present some preliminary results for now we intend to improve on this in the future.

One use case for the code generation process can be to write abstract profiling code once and then run it on many hardware platforms. To do so would require a good execution base to get reliable results. We intend to run profiling code on our prototype operating system [5]. But it would be interesting to see if it is possible to generate code that could be run on a operating system like Linux to get much better hardware support right away. A possible solution would be to generate Linux kernel modules that take control over the system and run the profiler code exclusively for a limited time. This would allow a broad range of hardware architectures to be analyzed. These models could be used by real-time operating systems to adapt them on specific hardware properties.

As hardware platforms are getting more difficult to develop for it would be handy to write low-level operating system code for hardware features like memory management, memory address translation and other things only once. To achieve this our abstract languages need to evolve to provide the necessary means.

A distant goal we would like to aim for is to generate an open source database with hardware models that describe the hardware in a way that is especially useful for the design and implementation of operating systems.

References

[1] J. M. Calandrino and J. H. Anderson, "Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study," in *20th Euromicro Conf. on Real-Time Sys. (ECRTS '08)*, Jul. 2008, pp. 299–308.

[2] D. Dasari, B. Akesson, V. Nelis, M. Awan, and S. Petters, "Identifying the sources of unpredictability in COTS-based multicore systems," in *08th IEEE Int. Symp. on Industrial Embedded Systems (SIES 2013)*, Jun. 2013, pp. 39–48.

[3] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *27th Int. Symp. on Comp. Arch. (ISCA '00)*. New York, NY, USA: ACM, 2000, pp. 128–138.

[4] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Making shared caches more predictable on multicore platforms," in *25th Euromicro Conf. on Real-Time Sys. (ECRTS '13)*. IEEE, Jul. 2013, pp. 157–167.

[5] H. Borghorst and O. Spinczyk, "Increasing the predictability of modern COTS hardware through cache-aware OS-design," in *11th W'shop on OS Platf. for Emb. Real-Time App. (OSPERT '15)*, Jul. 2015.

[6] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system," in *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, Sep. 2009, pp. 261–270.

[7] "Xtext," https://eclipse.org/Xtext/, accessed: 2016-05-23.

A communication framework for distributed access control in microkernel-based systems

Mohammad Hamad, Johannes Schlatow, Vassilis Prevelakis and Rolf Ernst
Institute of Computer and Network Engineering, TU Braunschweig
{mhamad,schlatow,prevelakis,ernst}@ida.ing.tu-bs.de

Abstract—**Microkernel-based architectures have gained an increasing interest and relevance for embedded systems. These can not only provide real-time guarantees but also offer strong security properties which become increasingly significant in certain application domains such as automotive systems. Nevertheless, the functionality of those complex systems often needs to be distributed across a network of control units for various reasons (e.g. physical location, scalability, separation). Although microkernels have been commercially established, distributed systems like these have not been a major focus. This is basically originated by the fact that – in the microkernel world – policy, device drivers and protocol stacks are userspace concerns and rather left to be solved by the particular application domain. Following the principle of least privilege, we therefore developed a distributed access-control framework for all network-based communication in microkernel-based systems that can be generically deployed. Our design not only enforces security properties such as integrity but is also scalable without adding too much overhead in terms of run time or code.**

I. Introduction

Nowadays embedded systems are ubiquitous, i.e. in most of the electronic devices in our life; from simple devices such as microwaves to sophisticated ones such as cars. The latter is a driving motor when it comes to safety concerns of complex embedded systems, which is typically approached by a deliberate system design that uses separation and "safety nets". A contemporary car contains from 70 to 100 microcontroller-based computers [1], known as electronic control units (ECUs). These ECUs control many functions within the car, ranging from the mundane such as controlling courtesy lights to the highly critical such as engine control. These ECUs are distributed around the vehicle and interconnected using different bus systems such as CAN, MOST or FlexRay in a rather static setup. The need of exchanging bigger and more expressive messages is pushing towards using Internet Protocol (IP) standards for both on-board and vehicle-to-X communications [2], [3], which is also driven by the desire of better modifiability and updateability in the automotive domain. However, one main reason that connected ECUs are becoming increasingly vulnerable is the use of unprotected wireless and wired communication [4].

Increasing the flexibility of vehicular (software) platforms while not neglecting the safety and security therefore is a major challenge. Moreover, in contrast to traditional computing systems, embedded systems not only come with limited resources concerning memory and CPU power but also have slightly different demands on the system's security.

Modern sedans run a huge number of applications (functions) with millions of lines of code (LOC) [1]. These applications come from several vendors with various levels of code quality. Safety-relevant functions, such as anti-lock braking systems, are typically well-engineered and heavily tested, while others, such as the entertainment systems, could be implemented with security and reliability not as prime factor. The uncontrolled interference within shared buses between applications with a mixed level of safety, security and criticality may create vulnerabilities [5]. Compromising uncritical components by an adversary could be sufficient to control critical components across the entire car, which must be dealt with by appropriate protection mechanisms. Using microkernels could be the first step towards providing a secure environment for such systems [6], which will benefit from the small amount of privileged code, the minimal trusted computing base (TCB), and the memory protection between the different components. However, providing a comprehensive framework for controlling the communication between the various platform subsystems is a crucial complementary task to the microkernel's security services.

We have created a distributed access-control framework which allows only authorized components to interact with each other inside the vehicle and with external entities. Our framework ensures the ability to define the type of security provided for each communication link (e.g. integrity, confidentiality), and other connection properties (e.g. priority). We defined a secure communication policy, which determines all permitted paths between different components, centrally and gradually. Later on, the policy was enforced by each ECU on the vehicle isolatedly (i.e. without any need for additional interactions).

The rest of the paper is organized as follows. Section II describes the communication framework and the objectives of our work. It also explains the security approach of local and remote communication between the different components. In Section III, we describe the main parts of the communication module and its implementation. Finally, related work is presented in Section IV before we evaluate our implementation in Section V and discuss our findings in Section VI.

II. Communication Framework

Fig. 1 depicts an exemplary distributed system and the different scenarios of inter-component communication. In our idealistic world, each ECU is running a microkernel-based operating system that hosts multiple (interacting) software

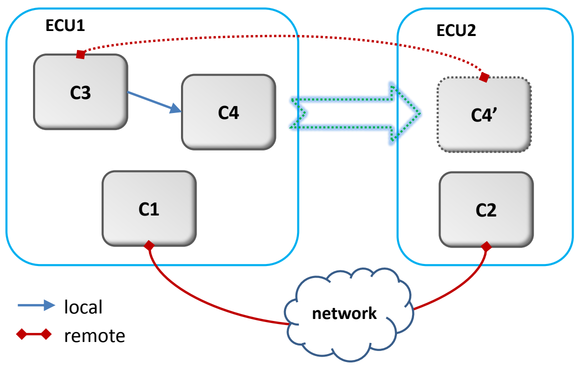


Fig. 1. Communication scenarios in distributed automotive systems

components. The ECUs are then inter-connected by a network. Here, we identify two types of inter-component communication: *Local* communication, which refers to the communication between two components on the same ECU (e.g. C3 and C4), and *remote* communication, which denotes the communication between components on two different ECUs (e.g. C1 and C2).

Our goal is to build a framework that ensures secure communication between the components in this scenario while still maintaining a high flexibility. For this purpose, we first state our objectives before we discuss local communication mechanisms w.r.t. their application in our scenario and finally derive our networking architecture approach.

A. (Security) objectives

When it comes to designing a communication framework for distributed systems in critical application domains such as automotive systems, we identified the following objectives:

1) Fine-grained access control: Our primary goal is to control who should talk to whom in an efficient manner without the need for static access-control mechanisms. Components should only communicate with other components who are specified by the policy. One main advantage of this is that even a compromised component will only be able to interact with authorized components and will be unable to attack other components indiscriminately.

2) Secure communication: Providing security services for authorized connections is a fundamental requirement. The required security services are varying from one application to another. However, since most security issues in vehicle communications are related to the lack of authentication mechanisms, providing integrity and mutual authentication is necessary to prevent unauthorized parties from sending false data or injecting them in established connections.

3) Composability and migratability: In a component-based system, the desired functionality is integrated by composing several interacting components. In a distributed scenario, we also gain the freedom of choice where to execute each component. Fig. 1 illustrates this on component C4 which could alternatively be executed on ECU2 but then requires a remote communication mechanism to C3. Hence, composability and migratability are important values whose lack would quickly restrict the design space. Note that we consider migration in terms of a (partial) system reconfiguration that must undergo several admission tests before being applied.

4) Minimum (application-specific) TCB: A common goal when building secure systems is the minimization of the TCB, i.e. the subset of hardware and software that must be relied upon. Microkernels already do their share w.r.t. minimizing the TCB. Yet, in userspace, a flawed design can easily bloat the TCB, e.g. by adding a middleware for all applications. Instead, each application should only rely upon a minimum set of software components with minimum complexity and therefore have its specific TCB [7].

5) Legacy application support: Another concern is the ability to integrate legacy applications into a component-based system. Note that legacy APIs might need to be monitored and restricted so that they do not conflict with the above objectives. In the scope of this work, we demonstrate the feasibility of this by providing a socket API to conventional network applications.

B. From IPC towards networked communication

Any microkernel architecture provides strong isolation of application components in order to minimize the TCB. Therefore, any communication between isolated components, i.e. inter-process communication (IPC), needs to be mediated by the kernel. On the one hand, this introduces additional overhead which was historically one of the main drawbacks of the microkernel approach but was weakened by the optimization of (synchronous) IPC mechanisms and the evolution of microkernels [8], [9]. On the other hand, this has the benefit of making any communication explicit. This property was further strengthened by introducing capability-based access control that enables a fine-grained and unforgeable control of a component's communication channels. As a result, today's microkernel architectures allow us to apply the principle of least privilege and enforcing security policies when integrating application components from different, potentially untrusted, parties [10]. In summary, all these properties helped establishing microkernels as sophisticated, and also commercialized [11]–[14], implementation vehicles for critical application domains.

When it comes to more dynamic scenarios like distributed systems, a service-oriented approach is commonly taken to equip the system with the required flexibility. Typically, a communication middleware then takes care of routing the messages to the communication partner that registered under a certain service name thereby deploying a communication mechanism (API) that is agnostic of the actual communication partners and their location. Yet a major drawback of such a middleware is that enforcing security policies and providing isolation (e.g. local namespaces) while not adding too much overhead is a non-trivial obligation. This approach clearly trades ease-of-use against simplicity and efficiency.

We therefore believe that the strong architectural guarantees already provided by microkernels can and should be utilized in such scenarios. That means local communication shall still benefit from the existing efficient and secure implementations

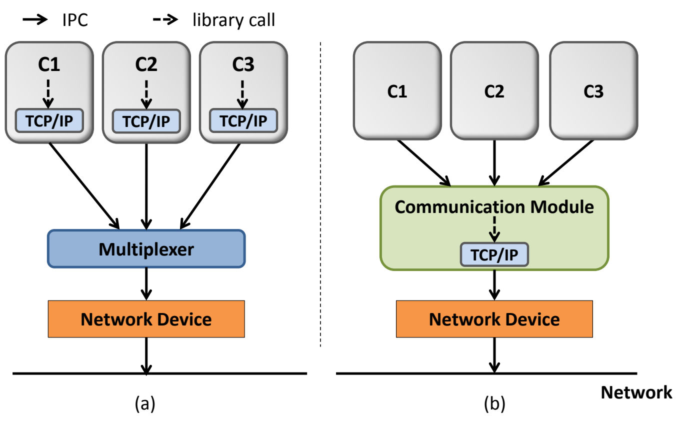


Fig. 2. Architecture with multiple TCP/IP stacks and a shared multiplexer (a) compared to a shared communication module with an integrated TCP/IP stack (b).

while we transparently transform between the local and remote communication mechanisms where necessary. The challenge here is to provide similar guarantees for remote communication, i.e. the fine-grained access control and integrity of remote communication channels. Yet there is a mismatch between the fine-grained access control for local IPC and the socket API typically used for network applications that gives full access to any attached network. We therefore need to provide the infrastructure with which we can control network accesses in order to establish unforgeable network communication between application components as we are used to on when using local IPC.

C. From user-level networking towards a distributed firewall

Microkernel philosophy is based on moving all policy, including device drivers and protocol stacks, from the kernel to the userspace. Therefore, implementing the network stack in the userspace was a hot topic for many years; it was proposed for different motivations, including increasing the performance and flexibility of the network layer [15].

Providing maximum isolation between different applications, a straight-forward approach of executing several network applications on a microkernel consists in using dedicated network stacks for each application (cf. Fig. 2a). Here, the low-level Network Interface Controller (NIC) must be multiplexed/virtualized, e.g. by a network bridge. As a result each application is linked to its network-stack library and requires its own MAC and IP addresses. The drawback of this approach is that unless some sort of packet filtering is deployed, each application also gets full access to the shared network, which contradicts our objective of a fine-grained access control. More precisely, this approach is susceptible to the following communication threats:

a) Spoofing: An application, which has a full access to the network stack, can emit a frame with fake IP or MAC addresses. Such an application may imitate other applications, eavesdrop on their communication, or collect relevant information about the platform. It could also change the transmitted data and inject false values.

```
Authorizer:   Integrator_public_key
Licensees:    Platform_public_key
Conditions:  (Vendor_id =="ACME_INSTRUMENTS"
             && Src_device_name == "headlight_control"
             && Dst_device_name == "ambient_light_sensor"
             && Src_device_type == CONTROL_PLATFORM
             && Dst_device_type == LIGHT_SENSOR
             && Security_level >= SL_INTEGRITY && Priority_level == HIGH
             && Bitrate_limit == X Kbitd/sec ) -> "ALLOW"
Signature:    Integrator signature
```

Fig. 3. An example of KeyNote credential which enables an ambient light sensor to communicate with headlight control. The credential ensures integrity of the communication line with high priority.

b) Denial of service (DoS): One of the primary results of the IP spoofing can be a DoS attack. I.e. a malicious application could spoof a target service's IP address and send many packets to different receivers. All responses to the spoofed packets will be directed to the services IP, which will be flooded. Sometimes, an attack cannot cause a disruption of the service, but can cause a degradation of its quality (e.g. by increasing its response time). The DoS could lead to serious issues if that service is responsible for the users' safety.

In order to combat these threats, adequate access-control mechanisms should be implemented to control the interaction between different applications and to prevent unauthorized parties from processing foreign data. However, packet filtering is more network-centric and typically too abstract for fine-grained application-level access control. Moreover, there is a consistency challenge when it comes to updating static filtering rules in a distributed system in case of a dynamically changing environment.

Integrating other traditional network protection methods such as firewalls in vehicle communication networks was shown to be inadequate too [16], especially if the integrator keeps the assumption that all insider nodes are trusted. Moreover, using a single ECU as a firewall to control the whole communications within the vehicle is also not an optimal solution. Such an ECU will create congestion, become a single point of failure and jeopardize the scalability.

Hence, adopting the distributed firewall technique [17] seems to be a favorable solution in order to remove any performance bottleneck. We applied this method by providing a single communication module for each ECU as shown in Fig. 2b. This module is playing the role of a firewall by controlling all incoming and outgoing communications on a single ECU and by enforcing the security policy locally. The security policy is managed centrally and then distributed to all ECUs. Note that the communication module implements a shared network stack and multiplexes the network device. It is therefore a potentially complex component that might compromise the isolation of the application components. We believe, however, that this design choice can actually simplify the policy enforcement and multiplexing task in contrast to solutions that implement these on other layers of abstraction.

In our previous work [18], we presented a mechanism to integrate the evaluation of communication policy into the

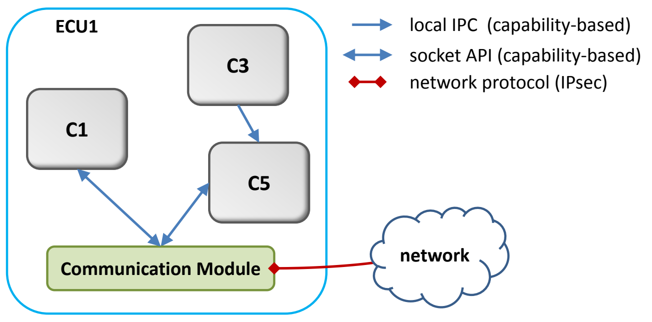


Fig. 4. Architecture with a shared communication module on each ECU

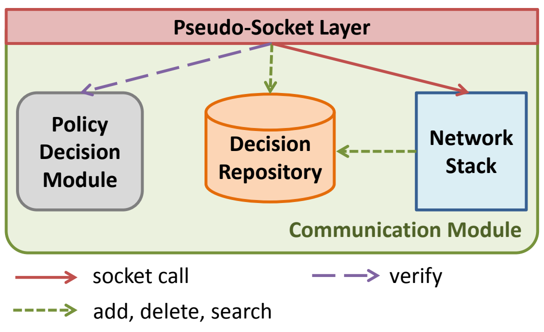


Fig. 5. Architecture of the communication module

components' development flow; from the design process until the final integration stage of the component with the platform. Such integration will ensure that the defined policy will fulfill all the operational component requirements. Each component is identified by its own credential, which gives it the ability to communicate with other components regardless its topological location in the network.

We used the KeyNote policy definition language [19] to formulate the communication policy as shown in Fig. 3. The application-independent design of KeyNote allows for the support of a variety of different applications. KeyNote furthermore enables the delegation of the policy by allowing principals to delegate authorization to other principals (e.g. in Fig. 3 where the integrator delegates rights to the platform). Consequently, the delegation capability allows to decentralize the administration of policies.

III. Implementation of the Communication Module

We implemented our communication module for the Genode OS Framework as distinct userspace server which provides and monitors network accesses. Hence any network application acts as a client that connects to this server using a socket-like API. Fig. 4 illustrates the different communication scenarios (cf. Fig. 1): While C1 represents a network application component directly using the socket API, C3's local IPC is translated by C5 and the communication module into a network communication. An important detail here is the capability-based access control used to manage the access to the communication module. As a system integrator, we can thus perfectly control the local inter-component communication and thus guarantee that no application component has direct access to the network interface. Moreover, the communication module is able to distinguish its clients by their capabilities and can therefore select and enforce different (pre-defined) policies for the network communication. In this way, all network accesses are securely mediated by the communication module. Note that this is based on the assumption that capabilities cannot be arbitrarily delegated between application components.

The communication module is composed of four cooperating submodules as depicted in Fig. 5. The pseudo-socket layer plays a central role by providing a suitable interface to the applications as well as by coordinating the other submodules. In the remainder of this section, we elaborate on these submodules in more detail:

a) Pseudo-socket layer: A lot of conventional (legacy) applications use a socket-like API (as in the standard C library) to access the network stack. Similarly, the pseudo-socket layer represents the interface which the applications use to interact with the communication module. In contrast to the conventional function/library calls, the pseudo-socket layer uses local IPC and must therefore take care of the memory management between the different address spaces of the communication module and its clients before a call can be handed over to the actual network stack. More precisely, we used shared memory to transfer the data between the address space of the application and the communication module to avoid imposing extra overhead by copying data multiple times. All this is transparently taken care of by this layer, so that the clients can still use the typical socket API functions. Legacy applications can be supported by linking against a slightly modified version of the standard C library that forwards the socket API calls to the communication module. Additionally, this layer checks the parameter validity, invokes the policy decision component with the relevant information related to the connection, and reacts to the received decision. If an affirmative decision (i.e. allow) is received, a new rule will be added to the repository. This rule contains many runtime specified selectors such as IP addresses and port numbers of the two ends of the connection. It also includes the required security level (i.e. integrity or confidentiality), the maximum allowed bit rate, and the priority level of the connection. Associating this rule with the opened socket gives us the ability to enforce this rule in two different layers. The first one is at the socket layer when an application uses the socket to send or receive data while the second one is placed at the IP layer whenever a new packet is received. The rule will be removed from the repository as soon as the socket is closed.

b) Policy Decision Module: This submodule is responsible for monitoring, i.e. granting or denying, the main requests of an application such as initiating a connection, sending, or receiving data. In order to make a decision, it determines whether a proposed request is consistent with the local policy and whether the conditions specified in the credentials were met. For this purpose, we use the KeyNote library.

c) Network Stack: As mentioned before, the network stack was integrated into the communication module to provide the basic network access. In our implementation, we used the lightweight TCP/IP stack (lwIP). In addition, we integrated embedded IPsec [20] into this network stack (as proposed in [21]) in order to provide basic security services (e.g. integrity, confidentiality) to the clients. Furthermore, we consider implementing traffic monitoring in a later phase of our work to keep tracking of the bit rate of a connection in order to prevent DoS attacks, like proposed in [22].

d) Decision Repository: Providing a repository for saving the policy decisions is an essential technique in our design to spare the run-time costs of the request evaluation. By doing this, the evaluation only occurs when an application initiates a connection (i.e. accept() and connect() for TCP-based communication). For this purpose, the decision repository stores the decided rules for any opened connection.

IV. Related Work

Many authors have addressed deficiencies of vehicle communication and the need for a mechanism to control the interaction between the components within the vehicle and between the vehicle and the outside world [23]. However, only a few proposals have appeared to provide such a mechanism.

Based on legacy network solutions, Chutorash [16] proposes an approach for using a firewall to control the interaction between applications on the one side and vehicle bus and vehicle components on the other side. His approach was restricted to monitor the interaction between HMI systems and other vehicle components, which ignored controlling the interaction between the vehicle's components. We extended this approach by using a firewall for each ECU in order to build a distributed firewall that is concerned about all communication inside the vehicle.

Concerning multiple network stacks, a userspace port switch was proposed in [24] that controls interconnecting independent network applications which run together. Swarm assigns the same IP, MAC address to all different stacks and uses port number to distinguish them. Yet using port numbers to control the communication flows is not sufficient when the applications use dynamic port assignment.

The Genode OS Framework [25] proposed the use of a NIC bridge which implements the Proxy-ARP protocol [26] to multiplex and monitor the communications of different applications that run on the same host. Neither solutions use filtering mechanisms which identify the application properly.

QNX Neutrino RTOS is a commercial microkernel-based real-time operating system that uses a networking architecture [27] very similar to what we propose. The dominant (local) IPC mechanism is synchronous message passing. Network communication is enabled via a socket API by a central network manager which implements device drivers and the network stack. In addition, the so-called Qnet protocol transparently extends the message-passing paradigm over a distributed system [28]. However, as Qnet is designed to be deployed for a group of trusted machines, it does not perform any authentication. Moreover, policy enforcement is only done in terms of packet filtering.

TABLE I
Communication module code size

Part	SLOC
Pseudo-socket layer	500
Policy Decision Module interface	300
IPsec extension of the Network Stack	2000
Decision Repository	600

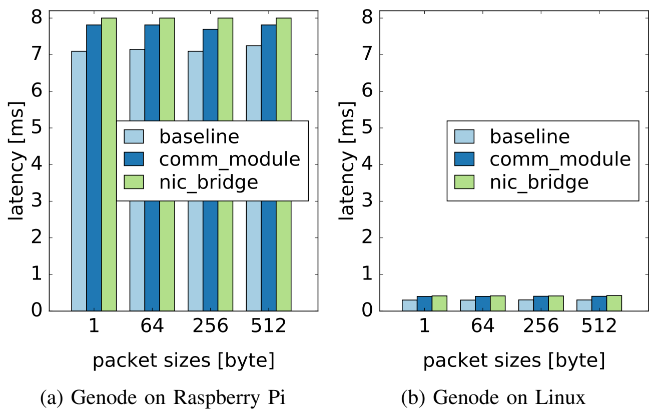


Fig. 6. Average round-trip latency results for our two test platforms.

V. Evaluation

We evaluated our implementation of the communication module w.r.t. its overhead in terms of source lines of code (SLOC) and latency. Table I summarizes the SLOC values that have been acquired by the *cloc* tool. Note that we only evaluated the part of the policy decision module which interfaces the unmodified KeyNote library. It is also worth mentioning, that by our approaches saves about 750 SLOC by superseding the multiplexing component (i.e. nic_bridge).

Regarding the latency, we used the TCP_RR test of the netperf tool in order to measure the average round-trip latency. As a matter of course, the communication module must perform worse than a scenario where a single application directly accesses the network device. We therefore compared our approach against the scenario illustrated in Fig. 2a. More precisely, the multiplexer we used is the nic_bridge of the Genode OS Framework that implements the Proxy-ARP protocol. Hence the scenarios we compared both include additional copying and context switching caused by the nic_bridge and the communication module. The netserver component was executed on the system under test while the netperf binary was run from a standard linux machine. In particular, we added the parameters `-i 10,3 -I 99,5` in order to perform multiple iterations and achieve a confidence level of 99 %.

The average round-trip latency results are shown for different package sizes and two different platforms in Fig. 6. As a baseline, we also included the results for a setup in which the netserver directly accesses the network device. One of the test platforms was running Genode on a Raspberry Pi whereas the other platform was running Genode directly on the same linux machine as the netperf binary. Note that we used the latter to bypass any physical network devices and drivers as it only utilizes rather simple virtual network interfaces. Interestingly, we can observe a slight improvement of the latency for our approach in comparison to the nic_bridge.

Since the TCP_RR benchmark does not measure the TCP connection setup, we accounted the additional latency imposed by the policy decision module separately. This overhead only occurs once for every TCP connection and is thus amortized over the lifetime of the connection. For this, we measured a maximum of 30 milliseconds. Note that our implementation is still in a proof-of-concept stage so that various optimization techniques could be applied to improve the performance.

VI. Conclusion

From the microkernel-perspective, using a dedicated network stack for each application is a common and reasonable design decision in order to achieve a high level of isolation. However, this approach either enables full and uncontrolled network access to potentially malicious applications or complicates the process of controlling the network accesses on a rather low abstraction level. In the scope of this work, we presented and implemented an alternative approach that consists in providing a single communication module that efficiently mediates and controls all network accesses. By deploying this communication module in a distributed system like an automotive system that feature rather complex networks of many ECUs, we can equip those systems with a distributed firewall that enforces the integrity of all network communication. As this communication module authorizes both, incoming and outgoing, connection requests, by invoking a policy engine, it protects the network from malicious processes on the ECU and the ECU from unauthorized network connections.

Nevertheless, as security often has a price, it is clear that our access-control mechanism imposes some overhead. For our approach, this consists in processing overhead by the communication module and protocol overhead required to provide secure network communication (i.e. IPsec). For the latter we have already shown in our preliminary work [21] that the overhead is typically very low. By introducing the communication module, we could marginally improve the average round-trip latency once a connection is established. However, this also requires some additional (but amortized) cost for establishing a connection.

In summary, communication integrity is an essential prerequisite for functional safety, a major requirement for automotive systems. We therefore believe that our approach enables the use of microkernels in such demanding distributed systems.

Acknowledgement

This work was supported by the DFG Research Unit Controlling Concurrent Change (CCC), funding number FOR 1800. We thank the members of CCC for their support.

References

[1] R. Charette, "This car runs on code," feb 2009. [Online]. Available: http://www.spectrum.ieee.org/feb09/7649

[2] A. Bouard, B. Glas, A. Jentzsch, A. Kiening, T. Kittel, F. Stadler, and B. Weyl, "Driving automotive middleware towards a secure IP-based future," in *10th conference for Embedded Security in Cars (Escar'12)*, Berlin, Germany, Nov. 2012.

[3] RTI Conntext DDS. [Online]. Available: http://www.rti.com

[4] M. Wolf, A. Weimerskirch, and C. Paar, "Security in automotive bus systems," in *Workshop on Embedded Security in Cars (ESCAR)*, 2004.

[5] Y. Laarouchi, Y. Deswarte, D. Powell, J. Arlat, and E. De Nadai, "Ensuring Safety and Security for Avionics: A Case Study," in *DAta Systems in Aerospace (DASIA)*, ser. ESA Special Publication, vol. 669, May 2009, p. 28.

[6] G. Heiser, "Secure embedded systems need microkernels," *USENIX ;login:*, vol. 30, no. 6, pp. 9–13, dec 2005.

[7] H. Härtig, "Security architectures revisited," in *10th ACM SIGOPS European Workshop*. New York, NY, USA: ACM, 2002, pp. 16–23.

[8] J. Liedtke, "Improving IPC by kernel design," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5, pp. 175–188, Dec. 1993.

[9] K. Elphinstone and G. Heiser, "From L3 to seL4 – what have we learnt in 20 years of L4 microkernels?" in *ACM Symposium on Operating Systems Principles*, Farmington, PA, USA, nov 2013, pp. 133–150.

[10] A. Lackorzynski and A. Warg, "Taming Subsystems: Capabilities As Universal Resource Access Control in L4," in *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems (IIES)*. New York, NY, USA: ACM, 2009, pp. 25–30.

[11] GenodeLabs. [Online]. Available: http://genode-labs.com

[12] Kernkonzept. [Online]. Available: http://www.kernkonzept.com

[13] Cog Systems: OKL4 Microvisor. [Online]. Available: http://cog.systems/products/okl4-microvisor.shtml

[14] QNX Neutrino RTOS. [Online]. Available: http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html

[15] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A user-level network interface for parallel and distributed computing," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA: ACM, 1995, pp. 40–53.

[16] R. Chutorash, "Firewall for vehicle communication bus," Feb. 24 2000, wO Patent App. PCT/US1999/017,852. [Online]. Available: http://www.google.de/patents/WO2000009363A1?cl=en

[17] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith, "Implementing a distributed firewall," in *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS)*. New York, NY, USA: ACM, 2000, pp. 190–199.

[18] V. Prevelakis and M. Hamad, "A policy-based communications architecture for vehicles," in *1st International Conference on Information Systems Security and Privacy (ICISSP)*, Feb. 2015, pp. 155–162.

[19] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis, "The keynote trust-management system version 2," RFC 2704, September 1999, http://www.rfc-editor.org/rfc/rfc2704.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2704.txt

[20] S. Kent and K. Seo, "Security architecture for the internet protocol," RFC 4301, December 2005.

[21] M. Hamad and V. Prevelakis, "Implementation and performance evaluation of embedded ipsec on microkernel os," in *The 2nd World Symposium On Computer Networks and Information Security*, September 2015.

[22] A. Garg and A. N. Reddy, "Mitigation of DoS attacks through QoS regulation," *Microprocessors and Microsystems*, vol. 28, no. 10, 2004.

[23] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. Mccoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *In Proceedings of IEEE Symposium on Security and Privacy in*, 2010.

[24] M. Unzner, "A split TCP/IP stack implementation for GNU/Linux," Diploma thesis, Technische Universität Dresden, 2014.

[25] Genode OS framework. [Online]. Available: https://genode.org/

[26] S. Carl-Mitchell and J. S. Quarterman, "Using arp to implement transparent subnet gateways," RFC 1027, October 1987, http://www.rfc-editor.org/rfc/rfc1027.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1027.txt

[27] (2014, feb) QNX Neutrino RTOS System Architecture. [Online]. Available: http://support7.qnx.com/download/download/26183/QNX_Neutrino_RTOS_System_Architecture.pdf

[28] (2014, mar) QNX Core Networking Stack User's Guide. [Online]. Available: http://support7.qnx.com/download/download/26171/Core_Networking_with_io-pkt_Users_Guide.pdf

Tightening Critical Section Bounds in Mixed-Criticality Systems through Preemptible Hardware Transactional Memory

Benjamin Engel
Operating-Systems Group
Department of Computer Science
Technische Universität Dresden
Email: ⟨name⟩.⟨surname⟩@tu-dresden.de

***Abstract*—Ideally, mixed criticality systems should allow architects to consolidate separately certified tasks with differing safety requirements into a single system. Consolidated, they are able to share resources (even across criticality levels) and reduce the system's size, weight and power demand. To achieve this, higher criticality tasks are also subjected to the analysis methods suitable for lower criticality tasks and the system is prepared to relocate resources from lower to higher criticality tasks in case the latter risk missing their deadlines. However, non-preemptible shared resources defy separate certification because higher criticality tasks may become dependent not only on the functional behavior of lower criticality tasks but also on their timing behavior. For shared memory resources, hardware transactional memory (HTM) allows to discard changes made to the resource and roll back to a previous state. But instead of using HTM for conflict detection and synchronization, we use this hardware feature to abort low critical shared resource accesses in case they *overrun their time budget*.**

In this paper, we present the results from extending HTM to allow transactions to become preemptible in order to support mixed criticality real-time shared resource access protocols. We implemented a lightweight cache-based HTM implementation suitable for embedded systems in the cycle accurate model of an out-of-order CPU in the Gem5 simulation framework. The software implementation using this extension in a priority-ceiling shared resource access protocol complements our work and demonstrates how transactional memory can be used to protect higher criticality tasks from untimely lower criticality tasks despite shared resources. Our simulation with synthetically generated tasksets show a reduction in system load of up to 22 % compared to scheduling LO resource accesses with HI bounds and a schedulability improvement of up to 54 % for state-of-the art real-time locking protocols. We used a LO-to-HI ratio of 1:1.2 – 1:2 and loaded the system between 50 % – 75 %.

I. Introduction

Announced in 2007, but later cancelled, Sun's Rock processor [1] was supposed to be the first production-ready CPU to include hardware transactional memory (HTM) [2]. Four years later, IBM's 3rd generation BlueGene/Q [3] fulfilled this promise by providing HTM functionality to high-performance computing, followed in 2014 with Intel's implementation [4] for general-purpose desktop and server systems. We expect cache-based HTM implementations to soon make their way into embedded processor architectures. For example, the open-source RISC-V ISA [5] already contains a placeholder for transactional memory instructions. Ferri et al. [6] identified energy and throughput improvements for accessing contented resources in a simulated ARM multiprocessor system-on-a-chip of 30 % and 60 %, respectively.

In its simplest version, cache-based HTM implementations keep transactional data stored in the cache until the transaction is committed. The cache will continue to respond normally to coherence requests, but accesses from other CPUs (writes to cached transactional data and reads to dirty transactional state) will cause an local abort and the invalidation of all cached transactional state. The result is either that the complete transaction becomes visible (in case the cache returns to normal operation) or the core reacts as if the transaction did not happen (by invalidating all transactional state).

In this paper, we exploit this all-or-nothing effect of transactions in mixed criticality systems to protect resources that are shared across criticality levels.

Mixed criticality is about consolidating tasks with different certification requirements into a single system. In his seminal work, Vestal [7] observes that independent tasks can be integrated in such a way by ensuring that higher criticality tasks can still meet their deadlines, even if they have failed to do so when they were scheduled with more optimistic scheduling parameters of lower criticality levels. Baruah et al. [8] calls this interpretation of mixed criticality systems *certification cognisant* as it maintains the increasing pessimism that is imposed by evaluation criteria to assert correct and timely operation of more safety critical tasks. In this paper, we adopt this certification cognisant interpretation of mixed criticality systems.

Unfortunately, the independence assumption is not very realistic in practical systems because in general tasks share resources that are not as easily preemptible as the CPU. For single criticality systems, a wealth of resource access protocols have been proposed following the early works of Baker [9] and Sha et al. [10] to bound priority inversion [1] and minimise blocking times. Priority inversion occurs if a lower prioritised job prevents a higher prioritised job from running because it is holding a resource that the latter needs or because the resource is otherwise inaccessible due to the mechanics of the resource access protocol.

[1]For ease of presentation, we use a priority based formulation for all preemption conditions and leave it as future work to adjust this formulation to preemption levels for EDF-based locking protocols.

In mixed criticality systems arises a second problem, which has led to a debate whether resources should actually be shared across criticality levels: *the trustworthiness of the resource after a lower criticality access*. For example, in [11], Burns takes the view that with the exception of some cryptographic protocols, resources should not be shared across criticality levels. He introduces MC-PCP to prevent unbounded priority inversion among jobs of the same criticality level. Brandenburg [12] on the other hand takes a much more radical approach and requires all resource accesses to be executed in a server, which assumes the criticality level of the highest criticality resource accessing task.

We take the view that resource sharing across criticality levels should be possible without having to subject resource accesses to a timing analysis at this highest criticality level. Instead we use available hardware features, namely transactional memory, to enforce timely bounds on shared resource accessed from low criticality tasks. Unfortunately, IBM Blue-Gene, although successful in high-performance computing is typically not widely used in real-time and mixed criticality systems. We therefore extend a simple x86 cache-based HTM implementation with support for a single preempted transaction and report in Section III about the implementation of this HTM variant in the cycle accurate model of an out-of-order CPU in the Gem5 hardware simulator. In Section IV, we evaluate the performance of our approach before we draw conclusions in Section V.

We are confident that it is much easier to establish partial correctness (i.e., that if the resource access terminates, the resource will be in a good state) than establishing the timeliness of such accesses. In particular, establishing partial correctness with sufficient confidence is still possible if the code is incompatible with sophisticated timing analysis tools. Our main contribution of this paper is to provide a means to ensure the timeliness of lower criticality accesses by executing them transactionally. We use the hardware feature of transactional memory not for synchronizing access to shared resources (the usual locks are still in place), but to quickly abort low critical shared resource accesses that violate their time bounds.

II. Background and Related Work

In this section we describe the foundations our research builds upon, namely hardware transactional memory (HTM) as a feature of modern processors and real-time locking protocols like immediate-ceiling or inheritance based protocols for controlling the access to a shared resource. We combine both in a mixed criticality system, where low critical tasks can be aborted if they overstep their temporal bounds or if higher critical tasks overstep their optimistic scheduling parameters and actually need to be scheduled with more pessimistic ones.

A. Hardware Transactional Memory

As of today, IBM Blue Gene/Q [3] has the most elaborate HTM implementation. By versioning data in the shared L2 cache, Blue Gene/Q is able to maintain multiple transactional states in parallel, which allows them to roll back later transactions if they conflict with earlier ones. Both, IBM's and Intel's HTM, have dedicated instructions to start and end a transaction. Within a transactional region, updates to memory are kept local to the CPU and are not visible to other processors. When the transaction finishes, it tries to *commit* all changes atomically and thereby makes them visible to other CPUs. If this commit fails, no changes are written back at all, the transaction is said to be *aborted* and the CPU state is rolled back to the state before the transaction was started to do proper error handling. We use this all-or-nothing approach when accessing shared resources within temporal bounds.

Cain et al. [13] give a very detailed description of the transactional memory system, its hardware implementation and suggested OS, and application programming models for the IBM Power architecture. Interestingly, this paper also explains in detail how and why they allow suspending and resuming transactions. Rather than aborting transactions, interrupts preempt transactions. In addition, transaction preemption and resuming is made available to developers through explicit instructions: `tsuspend` and `tresume`. The authors thoroughly evaluate the costs and benefits and show that transaction suspension is a valuable feature when building robust and reliable systems.

In this work, we propose a more lightweight implementation of transaction suspension for x86 that advances Intel's Transactional Synchronization Extensions (TSX). Although most implementation details of TSX [4] remain confidential, some parts may be inferred from released information in the Intel developer and optimisation manuals, which indicate a L1D cache-based implementation.

B. Real-Time Locking Protocols

In this paper, we consider both single and mixed criticality resource protocols, which we classify by the mechanism used to guarantee bounded priority inversion:

- **immediate-ceiling based protocols**, such as the stack resource [9] (or ceiling priority [14]) protocol (SRP), immediately raise the priority of resource acquiring threads to a resource dependent ceiling priority. By preventing released threads at a lower priority from executing, they seek to ensure that all resources are readily available once the thread starts executing.
- **inheritance based protocols**, such as the priority inheritance protocol (PI) and the original priority ceiling protocol (OPCP) by Sha et al. [10], allow preemptions of resource holders by higher prioritised threads but *help out* the resource holder in case a thread requests a resource by raising its priority to the priority of the higher prioritised, blocked thread. We distinguish between *local helping* (i.e., helping out a resource holder on the same CPU) and *global helping* (i.e., pulling the resource access from a remote CPU to the local CPU) and restrict ourselves to local helping protocols only. The rationale is that global helping would require transferring transactional state from one CPU to another, a complexity we are not willing to take into account when extending our cache-based HTM implementation.

Single criticality protocols of the first class are the multiprocessor variants MRSP by Gai et al. [15] and FMLP by Brandenburg et al. [16]. Both execute global resource accesses (i.e., resources accessed from threads on multiple cores) non-preemptively, which corresponds to raising the priority of

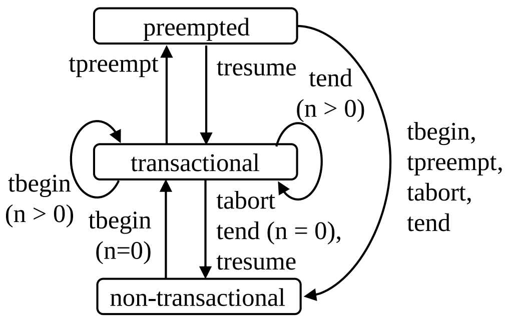


Fig. 1: States of cache controller for preemptible transactions.

the resource accessing thread to the maximum priority of threads on its core. Zhao et al. [17] extend the stack resource protocol to work with EDF schemes in which threads have more than one deadline to accommodate mode changes. As a member of the second class, Burns [11] extends the analysis of OPCP to consider criticality dependent blocking terms. Avoiding resource sharing across criticality levels, Burns allows local helping only between tasks of the same criticality. Single criticality protocols with local helping include the partitioned multiprocessor priority inheritance protocol [18] and similar variants for EDF [19]. The clustered O(m) locking protocol [20] and Brandenburg's inter-process communication scheme [12] apply global helping and are therefore not considered in this work in their original form. However, it is possible to modify the former to apply local helping (i.e., inheritance) only and we address this variant. Lakshmanan et al. [21] integrate ceiling (PCCP) and inheritance (PCIP) in their slack based scheduling approach to allow resource sharing across criticality levels. In addition to inheriting priority, they propose to also inherit criticality to prevent tasks from being suspended by low criticality tasks. Both PCIP and PCCP are single processor variants with local helping and ceiling, respectively.

III. Preemptible Transactions in the Gem5 Out-of-Order Model

Gem5 is a modular simulation framework with various CPU, memory, device and cache models. At the time of writing, there was already an HTM implementation [22] in Gem5, which is based on LogTM [23]. However, it was not built for the cycle accurate Out-of-Order CPU model (O3CPU) but for a simpler, less timing precise model. Moreover, its implementation was based on an undo log (like PARs [24]) whereas we focus on cache-based implementations, since available hardware (IBM, Intel) most likely implements transactions in the cache. We therefore started a new implementation in the O3CPU model, which we will introduce shortly in the following before we return to our implementation in Section III-B. Like most modern simulators, Gem5 decouples the internal architecture from the instruction set architecture (ISA) exposed to the user. In this way, Gem5 unifies different CPU models, like AtomicSimple, TimingSimple, and the 5-stage Out-of-Order model we use. Internally, Gem5's O3CPU makes use of a RISC like ISA, called M5, whereas user ISAs can be x86, ARM and others.

A. *Out-of-Order CPU Model*

Currently the most advanced CPU model in Gem5 is the 5 stage pipelined Out-of-Order CPU model, which loosely resembles an Alpha 21264. It implements the following usual pipeline stages: fetch, decode, rename, issue + execute + writeback, and commit. Issue forwards instructions to specific queues where they are processed by the execution units and the memory subsystem in the order in which their parameters become ready. Relevant for this work is the load/store queue and the ordering enforced by the memory barrier instruction.

The CPU model is event-driven and timing costs are attached and accumulated at each individual step. An external clock drives the CPU and creates 'ticks' for each of its stages to advance the model in a cycle-precise fashion. The number of instructions that can be fetched, decoded, issued and sent to the execution units is configurable. The delay and the bandwidth in each step, the delay of caches, the traversing of multiple ports, and the accumulating lookup-, forward- or data-copying delay are also subject to configuration. For our evaluation in Section IV, we use the default configuration for the Out-of-Order CPU, with a L1 instruction and L1 data cache of 32KB each and a 256 kB unified L2 cache. Cachelines store 64 bytes. The associativity of L1D is 4, 8 for L1I, and 16 for the L2 cache. Although modern CPUs have shared L3 cache, we did not add it, since transactional data will solely be placed in the L1 data cache. The cache one level beneath is important for the simulation, but multiple levels do not add any further detail.

B. *Preemptible transactions in O3CPU*

Based on publicly available information, we recreated part of the restricted transactional memory (RTM) implementation proposed by Intel [4][2]. More precisely, we augmented the L1 data cache with additional state —the **T** bit— to distinguish transactional from non-transactional data and extended the logic for the MOESI cache coherence protocol to react accordingly.

We chose to implement basic RTM functionality on top of MOESI although Intel CPUs implement MESIF because a MOESI protocol implementation was already present in Gem5. Common to both protocols are the cacheline states **I**nvalid for empty cachelines, **E**xclusive for data that has not yet been modified and that is present only in this cache, **M**odified for exclusive data that has been modified and **S**hared for data that may exist with the same value in multiple caches. **O**wned cachelines allow sharing of dirty data by delaying the write back to the time of eviction. The data in memory might be stale, but the cacheline is shared. **F**orward is a similar variant of **S**, which allows the forwarding cache to respond, instead of the underlying memory.

We first describe the modifications required to put the CPU and the caches in transactional and transaction preempted state before returning to the coherence protocol and how transactions change the state machine of the cache controller.

[2]Notice, while we added the full user functionality of RTM, including nested transactions, we leave the triggering of transaction aborts in all kind of exceptional cases as a future engineering task. For example, we added all instructions to begin, end and abort a transaction but do not trigger the abort mechanism when the page-table walker experiences a page fault.

Figure 1 illustrates the transaction states and the transitions assuming aborts are eager. To implement these state changes, we added three control signals to the CPU —HTM-ENABLE, HTM-COMMIT and HTM-ABORT— and interpret them in the load/store unit and in the cache controller. The outermost TBEGIN instruction transitions the CPU into transactional mode and informs the cache to start a new transaction. From now on, until the outermost TEND commits or aborts the transaction, all memory accesses will be stored transactionally in the L1 cache with the **T** bit set. Subsequent execution of TBEGIN stays in this state but increases the transaction nesting level, which TEND decreases. The outermost TEND with nesting level $n = 0$ sends a HTM-COMMIT-request to the underlying cache. The TEND instruction will retire not before the cache responds, either with commit or abort. TABORT triggers the abort directly through HTM-ABORT. In all three cases, the cache and the CPU return to non-transactional operation.

To add transaction preemption, we implemented two further instructions TPREEMPT and TRESUME and introduced one additional control signal HTM-PREEMPT to signal that the cache and the CPU are not in preempted transaction mode. TPREEMPT sets this signal, so that further memory-requests are no longer transactional and TRESUME clears it, returning to the previous transaction. Depending on the desired abort behavior, TRESUME will return an error if the transaction was aborted and immediate aborts should be supported. For lazy aborts, TRESUME returns normally but transactions will no longer commit. All other variants (including TBEGIN while a transaction is preempted) map to an abort. Aborts always affect all transactions up to the outermost one.

Special attention needs to be payed on in-flight memory operations and outstanding cache misses, since we cannot commit or abort a transaction that has pending memory requests. For this reason, all transaction instructions have to behave like a full memory barrier, which ensures that earlier memory accesses (including outstanding cache misses) are completed before a mode change is triggered and that later instructions are not started before the instruction is commited by the processor pipeline. In particular, we cannot execute these instructions speculatively because they change the behaviour of the CPU and the cache.

What remains is to ensure that the cache controller reacts appropriately depending on the state it is in and, in particular, that it detects all conflicts that lead to transaction aborts. For that we augment the cache with a vector of **T**ransaction bits (one for each cacheline). To enable HTM, the snoop logic changes its behavior depending on the state of the **T** bit of the affected cacheline. While the cache executes in transactional mode, the snoop logic responds normally to external reads to exclusive (**E**), shared (**S**, **O**) or modified **M** cachelines. However, if the read origins from the local core and targets an **E**xclusive, **S**hared or **O**wned cacheline, it sets the **T** bit to mark these lines as belonging to the read set. Writes put cachelines to the write set by setting the **T** bit in the **M** state. When an external snoop request hits a cacheline that is transactional (i.e. belongs to the read or write set and thus has its **T** bit set), the transaction will be aborted if the snoop request signals an external write (rfo) or a read (busrd) of modified data. An abort unconditionally invalidates all modified cachelines and returns to non-transactional operation.

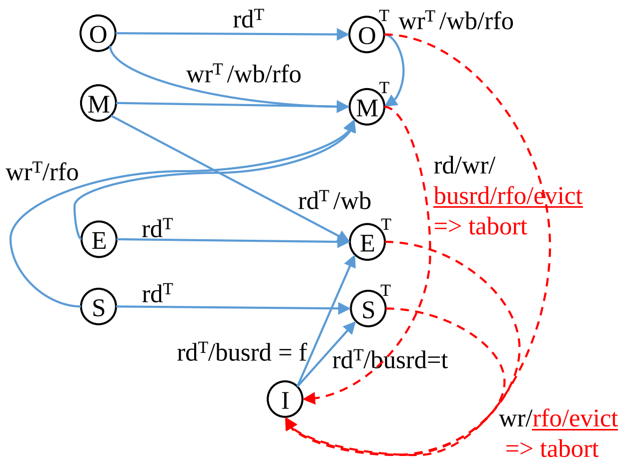


Fig. 2: Augmented coherence protocol for preemptible transactions. We omit the standard MOESI transitions and present in blue (solid) the behavior or transactional reads and writes and in red (dashed) the effect of non-transactional reads and writes on ntransactional data while the transaction is preempted. $.^T$ denotes transactional operations and state, wb indicates a required write back, evict a cache eviction and rfo and busrd are events indicating external writes and reads.

Special care must be taken for non-transactional dirty cachelines that become transactional. Because aborts will unconditionally discard cachelines, we first have to write back dirty cachelines to not lose the old data when aborting the transaction. More precisely, we have to write back **O**wned cachelines before they are written in a transaction and **M**odified lines before they are read or written.

Now, if the cache controller enters preempted transaction mode, the controller has to react to local accesses as if they were external. That is, if a local read hits the write set or if a local write hits this transactional data in the read or write set, the transaction is aborted prior to executing this request. Figure 2 shows the modified transitions of the resulting cache coherence protocol. For better readability, we omitted the transitions of the normal MOESI protocol and only show the transitions due to transactional reads and writes accessing non-transactional data and of non-transactional reads and writes hitting a preempted transaction. All other transitions among the non-transactional MOESI states and among their transactional counterparts are like in the standard MOESI protocol except that evictions of the latter trigger aborts.

To evaluate the costs of transactions, we annotate all steps in this execution with the costs we found for similar instructions (i.e., memory barriers and the signal propagation delay to the cache).

IV. EVALUATION

For the experimental evaluation we generated 1,000 random tasksets with up to 10 tasks and a given maximum utilisation (0.5, 0.75, and 1.0) using the uunifast algorithm [25]. We use a periodic task model with implicit deadlines, in which periods are the product of two randomly chosen factors from the set $[2, 3, 4, 6, 8, 9, 12]$, resulting in a maximum hyper period of 5184. Randomly choosing arbitrary periods from a given range typically results in extremely long hyper periods that can no longer be simulated in a reasonable amount of time. These tasks access shared resources and split their execution time in such a way that the first half in each period is spent outside of the critical section and the second half within. Furthermore we selected 83%, 67%, and 50% of them to be high-critical and increased their high-critical WCET by a factor of 1.2, 1.5, or 2.0 respectively. Thus we roughly have the same utilisation for the low and the high criticality mode. Fig. 3, 4, and 5 show the histogram of 1,000 tasksets. The solid three plots are almost overlapping and depict the distribution of tasks when using preemptible transactional memory, so that low critical tasks accessing their shared resource can be scheduled with their low-WCET. The transactional semantic of the cache allows us to use the more optimistic low criticality bounds when accessing the shared resource. In the case of overrunning the time budget, the timer will fire, the resource access will be aborted and the system changes into its high criticality mode, dropping all low criticality tasks. If the resource access finishes within time, the transaction will commit, the job will finish and the next job will be scheduled.

The dashed three plots show the same taskset when no transactional memory is used to bound the low critical WCET. Hence, we have to use their high critical counterpart for low critical jobs, resulting in a higher overall system load. The low critical WCET to high critical WCET ratios are 1.2, 1.5, and 2.0, i.e. a ratio of $1 : 1.2$ means high-critical WCETs are 20% higher then their low criticality counterpart, reflecting the higher trust and associated higher costs.

In the first experiment we chose an utilisation target of 50%, so that the load of all low-critical execution times sums up to about 50%. Since low-critical tasks share resources with high-critical ones, their WCET to access the resource needs the highest confidence of all sharing tasks. Although a task is low-critical, the resource access has to use its high-critical WCET, which leads to a higher load on the CPU. In this setup we observed up to 72% load, compared to the 50% when not sharing resources between low and high tasks. Even at an assumed very moderate LO to HI ratio of 1.2, already 1% of the tasksets were no longer schedulable, due to missed deadlines. With higher low critical to high critical ratios (1.5 and 2.0 respectively) the deadline misses increased to 6% and 21% of all tasksets. With our proposed hardware extension, we are able to use transactions for low-critical tasks in their critical section and therefore use the lower but less trustworthy low criticality bounds and abort jobs if they overrun their budget. Fig. 4 and Fig 5 show the results when increasing the initial load in the system to 75% and 100%. At a system load of 75%, already 4% of all tasksets (at ratio 1.2), 14% (at ratio 1.5), and 54% (at ratio 2.0) cause deadline misses. The very extreme is at a maximum utilisation of 100%. Due to pessimistic WCET for low critical tasks only 55% of all tasksets were schedulable when assuming a LO to HI ratio of 1.2. At 1.5 or 2.0 virtually 100% were no longer schedulable. This is not surprising, since adding even minor additional load to a very loaded system very likely causes deadlines to be missed. Therefore, we did not plot the actual load, but rather the theoretical load this system would have to handle, if we ignore all occurring deadline misses.

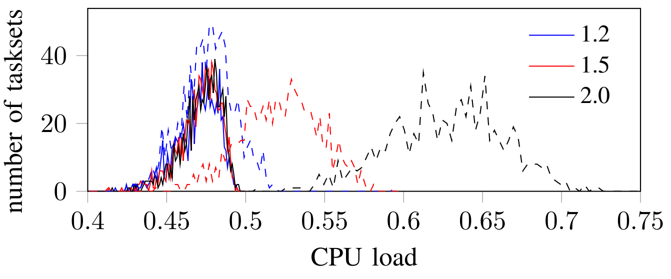


Fig. 3: Taskset with a maximum CPU load of 0.5 and a ratio of 1:1.2, 1:1.5, and 1:2.0 for low-critical to high-critical WCET. Although all tasksets are schedulable with EDF on a uniprocessor, it is clear that the additional pessimism for low critical tasks sharing a resource with a high critical task significantly increases their WCET and thus leads to a higher utilisation, i.e. higher resource demands (or less slack).

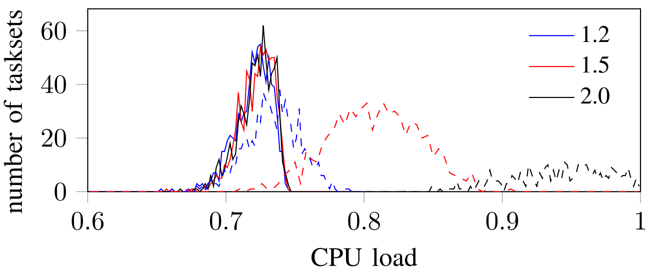


Fig. 4: Taskset with a maximum CPU load of 0.75 and a ratio of 1:1.2, 1:1.5, and 1:2.0 for low-critical to high-critical WCET. At a ratio of 2.0, 54% of the tasksets are no longer schedulable.

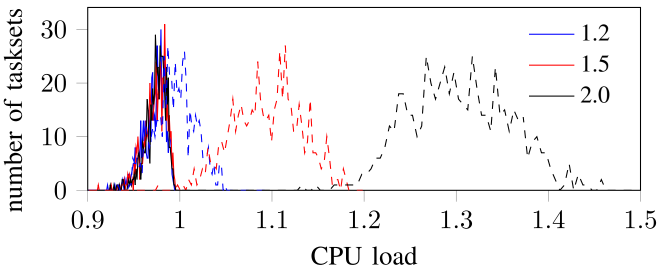


Fig. 5: Taskset with a maximum CPU load of 1.0 and a ratio of 1:1.2, 1:1.5, and 1:2.0 for low-critical to high-critical WCET. The three solid plots show the actual load when using transactions to bind low-critical resource access, whereas the three dashed plots depict the *theoretical* system load, since with a ratio of 1.5 and 2.0 virtually no tasksets were schedulable any longer. So we ignored the deadline misses and report the load the system *would* have to handle.

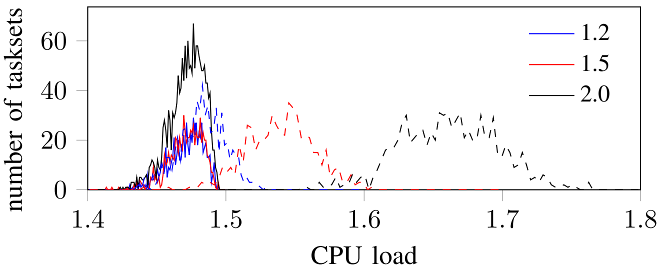


Fig. 6: Taskset with a CPU load of about 1.5, 20% of the WCET is spent in a critical section and in high criticality mode tasks require 1.2/1.5/2.0 times their low-WCET.

To substantiate the feasibility of our approach and to quantify the benefits of using transactional memory in mixed criticality systems, we evaluate a very simple multiprocessor setup. We use the same task model as in the uniprocessor case, generate the tasksets in the same fashion, and use partitioned EDF with one synchronisation processor for accessing global resources according to [26]. To generate tasksets that are still schedulable, the task's first 80% of its execution time is spent outside critical sections, the remaining 20% within. Of all tasks, about 83%, 67%, and 50% of them are classified as high critical and their high-WCET is 1.2, 1.5, and 2.0 times of their low-WCET, respectively. We removed all tasksets which caused deadline misses either in low or high critical mode, Fig. 6 shows the results. As in the uniprocessor case, reliably enforcing low-critical WCETs for shared resource accesses reduces the overall load in the system. Moreover, at a ratio of 1.2, 5% of the tasksets caused deadline misses when *not* using transactional memory to enforce timely bounds on critical sections. At 1.5 this number raises to 26% and with high-WCETs being twice as long as their low-WCETs counterparts 50% of the tasksets were no longer schedulable. This means that approximately one half is plotted, the other half was schedulable with hardware transactional memory enforcing lower WCET bounds, but could not be scheduled without it. This clearly shows the benefit of using preemptible transactional memory in combination with mixed criticality systems to improve schedulability and reduce system utilisation.

V. Conclusions

In this work, we investigated the use of hardware transactional memory (HTM) in real-time locking protocols to make low criticality resource access bounds trustworthy at higher criticality levels. We have seen that although existing HTM implementations are quite limiting or too complex to integrate in embedded systems, a lightweight implementation supporting preemptible transactions significantly broadens the applicability of our approach.

Future work includes extending our HTM implementation to an L2 victim cache to increase the amount of data that can be accessed within a resource access. Also, we did not yet exploit the optimistic locking behavior of transactions when a thread finds a resource blocked. To preserve the real-time guarantees of the legitimate lock holder, support for optimistic locking requires control over which transaction gets aborted (the optimistic) and which will be continued (the lockholder's).

References

[1] M. Tremblay and S. Chaudhry, "A third-generation 65nm 16-core 32-thread plus 32-scout-thread sparc processor," in *International Solid-State Circuits Conference Digest of Technical Papers (ISSCC 08)*. IEEE, 2008, pp. 82–83.

[2] M. Herlihy, J. Eliot, and B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Computer Architecture, 1993., Proceedings of the 20th Annual International Symposium on*, May 1993, pp. 289–300.

[3] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim., "The IBM blue gene/q compute chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, April 2012.

[4] I. Corp., "Web resources about intel transactional synchronization extension," www.intel.com/software/tsx, July 2014.

[5] A. Waterman, Y. Lee, D. Patterson, and K. Asanović, "The RISC-V instruction set manual volume i: User-level ISA - version 2.0," CS Division, EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2014-54, May 2014.

[6] C. Ferri, A. Viescas, T. Moreshet, I. R. Bahar, and M. Herlihy, "Energy implications of transactional memory for embedded architectures," in *Workshop on exploiting parallelism with transactional memory and other hardwre assisted methods*, April 2008.

[7] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Real-Time Systems Symposium*, December 2007, pp. 239–243.

[8] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *16th IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS. IEEE, April 2010, pp. 13–22.

[9] T. P. Baker, "A stack-based resource allocation policy for real-time processes," in *Real-Time Systems Symposium*. IEEE, 1991.

[10] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronisation," in *IEEE Transaction on Computers, 39*, 1990.

[11] A. Burns, "The application of the original priority ceiling protocol to mixed criticality systems," in *L. George and G. Lipari, editors, Proc. ReTiMiCS, RTCSA*, 2013, pp. 7–11.

[12] B. Brandenburg, "A synchronous IPC protocol for predictable access to shared resources in mixed-criticality systems," in *35th IEEE Real-Time Systems Symposium (RTSS 2014)*, 2014, pp. 196–206.

[13] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, "Robust architectural support for transactional memory in the power architecture," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 225–236.

[14] N. H. Cohen, "Ada as a second language, chapter real-time systems annex." McGraw-Hill, 1996.

[15] P. Gai, G. Lipari, and M. D. Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip." in *Real-Time Systems Symposium*. IEEE, 2001, pp. 73–83.

[16] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A flexible real-time locking protocol for multiprocessors," in *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2007.

[17] Q. Zhao, Z. Gu, and H. Zeng, "Integration of resource synchronization and preemption-thresholds into EDF-based mixed-criticality scheduling algorithm," in *RTCSA*, 2013.

[18] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," in *Real-Time Systems Symposium*. IEEE, 1988, pp. 259–269.

[19] C.-M. Chen and S. K. Tripathi, "Multiprocessor priority ceiling based protocols," College Park, MD, USA, Tech. Rep., 1994.

[20] B. B. Brandenburg and J. H. Anderson, "Real-time resource-sharing under clustered scheduling: mutex, reader-writer, and k-exclusion locks," in *EMSOFT*, 2011.

[21] K. Lakshmanan, D. de Niz, and R. Rajkumar, "Mixed-criticality task synchronization in zero-slack scheduling," in *IEEE RTAS*, 2011, pp. 47–56.

[22] G. Blake and T. Mudge, "Duplicating and verifying LogTM with os support in the M5 simulator ABSTRACT."

[23] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based transactional memory," in *in HPCA*, 2006, pp. 254–265.

[24] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek, "Preemptible atomic regions for real-time java," in *In 26th IEEE Real-Time Systems Symposium*, 2005.

[25] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Syst.*, vol. 30, pp. 129–154, 2005.

[26] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," in *Real-Time Systems Symposium*. IEEE, 1988, pp. 259–269.

GPU Sharing for Image Processing in Embedded Real-Time Systems*

Nathan Otterness[1], Vance Miller[1], Ming Yang[1], James H. Anderson[1], F. Donelson Smith[1], and Shige Wang[2]
[1]Department of Computer Science, University of North Carolina at Chapel Hill
[2]General Motors Research

Abstract

To more efficiently utilize graphics processing units (GPUs) when supporting real-time workloads, it may be beneficial to allow multiple tasks to issue GPU computations without blocking one another. For such an option to be viable, it is necessary to know the extent to which concurrent GPU computations interfere with each other when accessing hardware resources. In this paper, measurement data is presented regarding such interference for several image processing routines motivated by automotive use cases. These measurements were taken on NVIDIA Jetson TK1 and TX1 boards. The presented data suggests that currently available real-time GPU management frameworks should evolve to enable the option of co-scheduling GPU computations.

1 Introduction

Vision-based sensing through cameras is being widely used in automobiles today to support advanced driver assistance systems (ADASs). Common capabilities of current ADASs include forward collision detection with automatic braking, lane departure warnings, and adaptive cruise control. Envisioned capabilities include advanced obstacle-tracking features, sign recognition, and 360-degree sensing.

Such capabilities give rise to workloads that can be challenging to support for three reasons. First, individual tasks may be subject to real-time constraints. Second, such tasks may be computationally intensive. Third, the overall workload must be supported on a hardware platform that operates within an acceptable size, weight, and power (SWaP) envelope and also is not too expensive.[1] In light of these needs, multicore+GPU platforms have been suggested as a promising way forward. Such a platform consists of several general-purpose CPUs augmented with one or more graphics processing units (GPUs) that can accelerate computations typically required in automotive settings.

Prior foundational work: GPUSync. Unfortunately, efficiently utilizing GPUs in contexts where real-time constraints exist requires sifting through many tradeoffs involving how GPUs are allocated at runtime and how GPU computations and related overheads are analyzed when checking real-time schedulability. To enable such tradeoffs to be systematically studied, our research group developed a real-time GPU allocation framework called GPUSync [15]. In GPUSync, the management of GPU-related hardware resources is viewed as a *synchronization* problem and thus real-time multiprocessor locking protocols are used to acquire and release such resources. GPUSync is highly configurable: options exist to control how tasks are scheduled on CPUs, how data is copied to and from GPUs, how GPU-related computations are queued and prioritized, *etc.*

Beyond GPUSync. In recent work, we have been attempting to evolve our work on GPUSync to more directly meet the needs of automotive use cases. The consideration of such use cases has caused the nature of our work to change in two significant ways. First, GPUSync is implemented primarily in LITMUS$^{\mathrm{RT}}$, and the code base is large, approximately 15,000 lines. Automotive manufacturers would likely be highly resistant to allowing such extensive operating system (OS) modifications. Due to this, we have shifted our attention to a simplified variant of GPUSync called GPUSyncLite that implements only a few GPUSync configurations (one currently) and requires only minimal OS modifications (none currently). Second, our prior GPUSync-related experimental work was conducted on an Intel platform that provides 12 CPU cores augmented with eight high-end GPUs. At present, it is hard to imagine such an expensive, energy-hungry platform being used in a production automobile. As a result, we have shifted our attention to less-expensive ARM-based platforms that provide a single less-costly, less-capable GPU.

Efficient GPU utilization through co-scheduling. This shift in hardware platform has created a new dilemma: when using a single, less-capable GPU, any waste of the GPU's capacity becomes untenable. Unfortunately, when using most previously proposed real-time GPU management frameworks [7, 8, 9, 12, 16, 25, 26, 50, 48, 49, 55, 56], including GPUSync, such under-utilization may be common. In particular, these frameworks disallow concurrent GPU execution by different tasks, so a task that under-utilizes the GPU's hardware resources can waste much of its capacity. Other prior work [10, 11] has considered co-scheduling GPU workloads, but in this work, several simplifying assumptions are made that preclude applicability on real-world GPUs. Notably, GPU instructions are assumed to always require only

*Work supported by NSF grants CPS 1239135, CNS 1409175, and CPS 1446631, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and funding from General Motors.

[1]In contrast to various "one-off" implementations of autonomous or semi-autonomous features, as seen for example in the Google car [1] and various DARPA challenge vehicles [46], affordability is a serious limitation with respect to production automobiles.

a single clock cycle, and cache misses and memory latency are not considered. Furthermore, this prior work includes no evaluation using real hardware.

To combat GPU under-utilization, we are beginning to investigate a new variant of GPUSyncLite that allows GPU computations issued by different tasks to be concurrently co-scheduled. When considering multi-threaded workloads scheduled on conventional multicore platforms, Jain *et al.* [22] observed that some co-scheduling choices are constructive and some are destructive. This is true in our context as well. In particular, it is *constructive* to co-schedule GPU computations issued by different tasks if the resulting GPU execution times and blocking times (*i.e.*, times spent waiting to access a GPU) yield real-time schedulability improvements. In contrast, such co-scheduling is clearly *destructive* if it causes a large inflation in GPU execution times or blocking times. Any such inflation is a sign that the co-scheduled GPU computations are adversely interfering with each other with respect to the hardware resources they access.

Contributions of this paper. To get a sense of the nature of such interference, we conducted experiments involving several common image-processing routines motivated by automotive use cases. For each of the considered routines, we obtained execution-time data via a measurement process under various co-scheduling scenarios. These measurements were taken on NVIDIA Jetson TK1 and TX1 boards. The obtained data suggests that certain co-scheduling choices are indeed constructive, while others are clearly destructive. The main contribution of this paper lies in presenting this data and discussing its implications as far as the future evolution of real-time GPU management frameworks is concerned.

Organization. In the rest of the paper, we provide needed background on GPUs (Sec. 2), describe the image-processing benchmarks under consideration (Sec. 3), present our experimental data (Sec. 4), and conclude (Sec. 5).

2 Background on GPUs

In this section, we provide a brief introduction to GPU hardware and programming fundamentals.

GPU hardware. GPUs may be either discrete or integrated. *Discrete GPUs* are packaged on adapter cards that plug into a host computer bus. Such a GPU has its own local DRAM memory that is completely independent from the DRAM memory used by the host processor. Discrete GPUs commonly draw between 150 and 250 watts, need active cooling, and occupy substantial space. *Integrated GPUs* are commonly found in system-on-chip (SOC) designs. The SOC typically combines a multicore machine with a GPU and uses DRAM memory that is tightly shared between the GPU and CPU cores. Integrated GPUs commonly draw between 5 and 15 watts, require minimal cooling, and add virtually no space requirements. These attributes make integrated GPUs the *de facto* choice in many embedded computing domains.

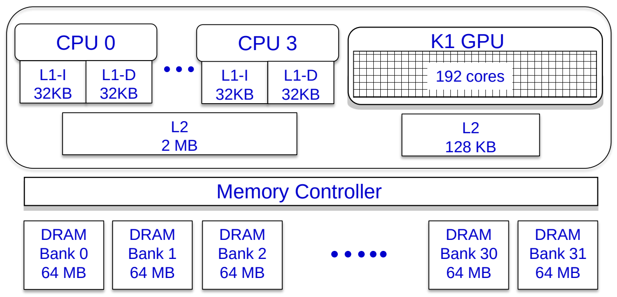


Figure 1: Jetson TK1 architecture.

Several SOC implementations with integrated GPUs capable of running sophisticated image-processing programs are on the market, including options from AMD [5], Intel [21], NXP [41] and NVIDIA [38]. In this work, we are using NVIDIA Jetson TK1 [39] and TX1 [40] boards, which retail for $200 and $600, respectively. These are likely acceptable price points in many automotive settings.

As illustrated in Fig. 1, the TK1 employs an SOC design that incorporates a quad-core 2.32 GHz 32-bit ARM machine and an integrated Kepler GK20a GPU. The CPUs share a 2-MB L2 cache. The GPU has 192 cores and a 128-KB L2 cache and provides up to 365 32-bit GFLOPS. The TK1 is a "big-little" platform in which an additional low power, low performance ARM CPU (not shown in Fig. 1) is provided on chip. The ARM CPUs and the GPU share 2 GB of 930 MHz DRAM memory partitioned into 32 banks.

The TX1 is a higher-end platform with a similar design. It consists of a quad-core 1.91 GHz 64-bit ARM machine, a 2-MB L2 cache shared by all CPUs, 4 GB of 1600 MHz DRAM, and an integrated Maxwell GM20B GPU. The GPU has 256 cores and a 256-KB L2 cache, and provides up to 512 32-bit GFLOPS. The TX1 is also a "big-little" platform.

As Fig. 1 suggests, GPU-using tasks may compete for many hardware resources. These resources include caches, DRAM memory banks, the memory bus and memory controller, and GPU cores. In prior work on real-time multicore computing, issues related to shared-hardware interference have received considerable attention [2, 3, 4, 6, 13, 14, 17, 18, 19, 20, 23, 27, 29, 28, 30, 31, 33, 35, 43, 45, 47, 51, 52, 53, 54]. However, we are aware of no such work that considers hardware interference with respect to GPU computations.

Obviously, concurrent GPU computations by different tasks may directly interfere with each other. Additionally, such computations can also interfere with programs running on CPU cores. For example, on both the TK1 and TX1, requests to load new lines into the GPU's L2 cache require accesses to the DRAM banks and may interfere with accesses by CPU cores. Further, so that GPU programs may be easily ported between discrete and integrated GPUs, CUDA (see below) explicitly treats memory as being either CPU-local (host memory) or GPU-local (device memory) and provides operations for copying data between the two. Such copy operations run concurrently with programs running on both the GPU cores and the CPU cores, potentially creating additional DRAM interference.[2] With integrated GPUs, explicit data copying can be avoided by using the *zero-copy* functions of CUDA (see below).

GPU programming in CUDA. The following is a high-level description of GPU programming in CUDA [37]. A GPU is fundamentally a co-processor that performs operations requested by CPU programs. CUDA programs use a set of C or C++ library routines to request GPU operations that are implemented by a combination of hardware and device-driver software. The typical structure of a CUDA program is as follows: **(i)** allocate GPU-local (device) memory for data; **(ii)** use the GPU to copy data from host memory to GPU device memory; **(iii)** launch a program—called a *kernel*—to run on the GPU cores to compute some function on the data; **(iv)** use the GPU to copy output data from the device memory back to the host memory; **(v)** free the device memory. On integrated GPUs, CUDA provides a zero-copy option where programs can simply pass a pointer to shared memory where data used for a kernel is located—that is, explicit copying from CPU-local memory to GPU-local memory is avoided.

By default, copy operations are synchronous with respect to the CPU program: they do not return until the copy is complete and will not start until any prior kernels have finished. However, kernel launches are always asynchronous, and asynchronous copy operations are also available. These operations require the CPU process to explicitly wait for GPU operations to complete, using a configurable synchronization mechanism. We configured our experiments to block the CPU process while synchronizing.

CUDA operations pertaining to a given GPU are ordered by associating them with a *stream*. By default, there is a single stream for all programs that share a GPU, but multiple streams can be optionally created. Operations in a given stream are executed in FIFO order, but the order of execution across different streams is determined by the GPU scheduling in the device driver. They may execute concurrently (or out of request order with respect to other streams).

Each GPU operation from a CUDA program is represented internally by a command string that is written to a command buffer (queue) managed by the device driver. The driver then schedules these commands for execution on the GPU. Programmers can think of a GPU as being abstractly composed of one or more *copy engines (CEs)* that implement transfers of data between device memory and host memory, and an *execution engine (EE)* that executes GPU kernels. The TK1 has one CE that moves data both ways. The TX1 has two CEs, one for each direction of transfer.

EEs and CEs operate concurrently. When there are multiple streams, multiple kernels and one or two copy operations can operate concurrently depending on the GPU hardware. When a kernel is scheduled, it may not require all EE resources, in which case the GPU scheduler may co-schedule more than one kernel (from different streams only) to execute concurrently and increase GPU occupancy. Concurrent kernel execution can create more interference in the GPU L2 cache and for DRAM accesses. To the best of our knowledge, complete details of kernel attributes and policies used by NVIDIA to co-schedule kernels are not available.

3 Benchmark Programs

In the study presented herein, we considered both GPU programs and CPU-only benchmark programs.

GPU programs. We chose three CUDA programs as representative of typical image-processing computations, and a fourth to represent a general class of programs that create stress on GPU resources:

- **stereoDisparity (SD)**: Extracts 3D depth information from 2D images taken with a stereo camera. The input consists of left and right 640×533 color images; the output is a 640×533 grayscale image.
- **fastHOG (HOG)**: Detects objects in an image using histograms of oriented gradients. The input is a 640×480 color image; the output is a matrix of bounding-box coordinates and object-detection probabilities.
- **Convbench (CONV)**: Executes convolutional neural-network layers as used in image recognition. The input is a 227×227 color image; the output is a matrix of neural-network parameters.
- **matrixMul (MMUL)**: Multiplies two square matrices of 32-bit floats (16 MB each).

SD and MMUL were taken from CUDA samples distributed by NVIDIA [36], HOG was downloaded from Oxford University [44], and CONV was constructed using code from AlexNet [32], implemented in Caffe [24]. All programs were adapted to run as iterative tasks, with a short random sleep between iterations. Each iteration corresponds to processing one image (SD, HOG, and CONV) or performing one matrix multiplication (MMUL). The programs were instrumented to log total execution time and the time required for performing data copies and executing kernels in every iteration. Even though our experiments were conducted using fixed images as inputs, we still verified that none of the benchmarks exhibited different runtime characteristics based on the content of the input images.

Each CUDA program was executed in a stream of its own, with all memory copies performed asynchronously and placed in the stream along with kernel launches in the intended FIFO order. After each kernel launch or group of memory copies, the CPU execution of each program was blocked while waiting to synchronize with the GPU. Each program was structured to ensure that all memory allocation

[2]With discrete GPUs, only the GPU data-copy operations may cause DRAM interference with respect to CPU usage and then typically in the form of DMA operations over a bus.

and freeing operations were done outside the iteration loop and all memory accesses within each iteration were to pinned memory, as is common practice in real-time systems. Display operations for the visualization of input or output images were removed. Image input data was read from memory buffers as would happen with camera-driven input. Two versions of each program were constructed, one with zero-copy memory and one without.

CPU-only benchmark. We used this program as a CPU-only workload:

- **vectorAdd (VADD)**: Adds two vectors of 32-bit floats (16 MB each).

VADD was based on the CUDA samples [36] and instrumented in an identical fashion as the GPU programs, but launches no GPU kernels.

4 Experiments

We are interested in supporting automotive image-processing workloads on a multicore+GPU platform such as the TK1 or the TX1. We assume that such workloads have soft real-time constraints: missing a deadline (occasionally) does not have catastrophic consequences, as long as an incomplete frame can be dropped and the system as a whole can use redundant or historical data processed by hard real-time components as a fail-safe mechanism. Given this assumption, our tasks can be provisioned by determining their execution times via measurement. Such a provisioning could be based on a task's average-case execution time, its worst-case execution time, or some intermediate value between the two. A measurement-based approach is further justified by the lack of adequate static timing analysis tools for multicore+GPU platforms. Even if such tools did exist, they would probably produce execution-time estimates that are so pessimistic that virtually no interesting workload could be supported.

The issue being considered in this paper is whether allowing GPU co-scheduling might have schedulability benefits. To get a sense of any potential benefits, we conducted experiments on both the TK1 and TX1 in which the various benchmark programs described in Sec. 3 were used as surrogates for real application code. These experiments were designed to assess whether GPU co-scheduling can be constructive from a schedulability point of view. We assessed this by running different combinations of the benchmark programs and recording execution-time data. We call each experiment involving such a combination of programs a *scenario*. In each scenario, execution-time data was recorded for a set amount of time (typically 10–15 minutes) under the default Linux scheduler with the considered programs pinned to separate CPUs. We present our obtained execution-time data by plotting cumulative distribution functions (CDFs), as such functions provide a sense of the best-case, average-case, and worst-case recorded times. We denote a given scenario by simply listing the combination of programs that were run.

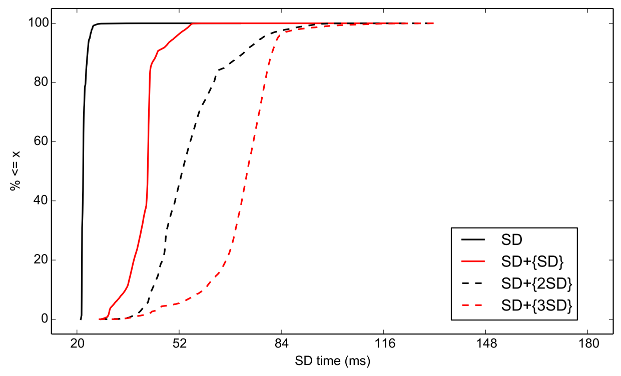


Figure 2: CDF of execution times of SD in scenarios only involving multiple SD instances.

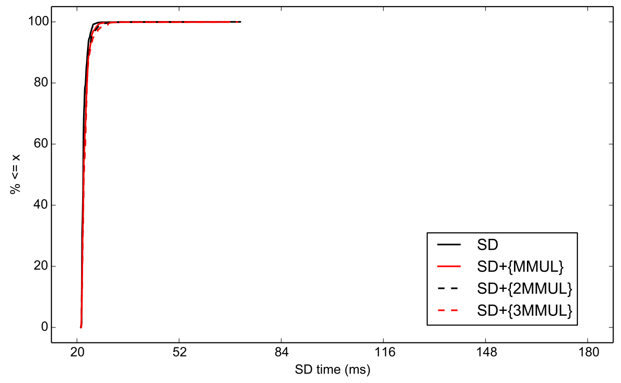


Figure 3: CDF of execution times of SD in scenarios involving MMUL competitors.

For example, in the scenario HOG+{2SD,HOG}, execution-time data was obtained on one CPU for the HOG program in the presence of two instances of SD and another instance of HOG running on the other three CPUs.

In total, we tested 52 scenarios, each both with and without the zero copy feature of CUDA and on both the TK1 and TX1. Unless otherwise noted, the scenarios presented here were measured on the TK1 and did not use the zero-copy feature. Data for all considered scenarios can be found in an online appendix [42].

Typical observed trends. We begin by commenting on general trends seen in our collected data.

Obs. 1. GPU co-scheduling was always constructive in scenarios consisting of multiple instances of a single benchmark.

Fig. 2 supports this observation for the case of the SD benchmark. In this case, GPU co-scheduling is mildly constructive. While SD execution times do increase with more competition, they do not increase to the point of eliminating any benefit due to co-scheduling. In particular, the addition of one competitor yields execution times that are somewhat better than simply doubling the execution time of a single instance, and this trend continues to apply as more competition is introduced.

Obs. 2. GPU co-scheduling was so constructive in some

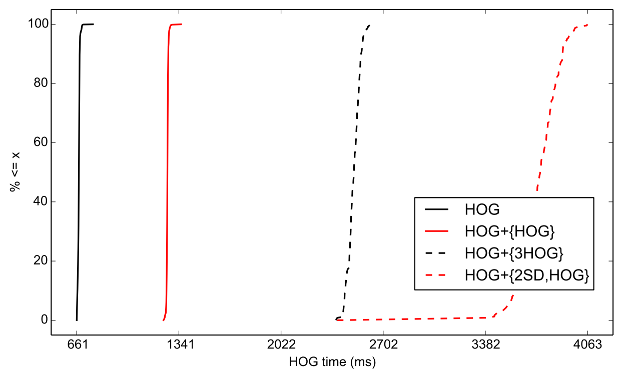


Figure 4: CDF of execution times of HOG in scenarios involving multiple instances of other benchmarks.

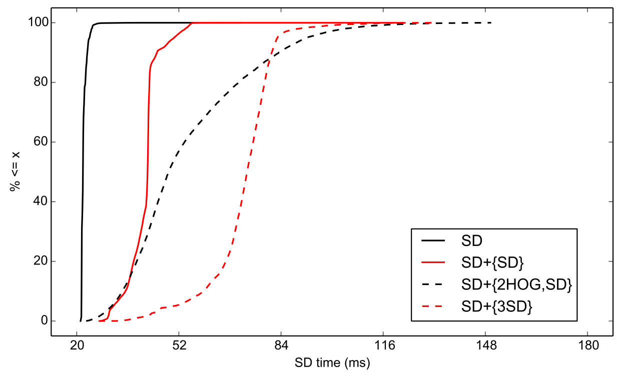


Figure 5: CDF of execution times of SD in scenarios involving multiple instances of other benchmarks.

scenarios that any introduced interference was practically negligible.

Fig. 3 supports this observation. Note that the execution times for SD remain virtually unaffected when instances of MMUL are introduced. This low impact is probably due to MMUL having short kernel execution times (approx. 1ms), which would rarely prevent SD from accessing the GPU.

Obs. 3. In some scenarios, particularly those involving HOG, GPU co-scheduling proved to be rather destructive.

Fig. 4 supports this observation. Note that the most destructive interference occurs when two instances of HOG and two instances of SD are run together, given by the curve for the scenario HOG+{2SD,HOG}. Fig. 5 presents execution-time data for SD that allows us to examine this same scenario from the perspective of SD. In particular, note the curve labeled SD+{2HOG,SD} in Fig. 5.

In our TK1 experiments, the worst-case execution time of SD running in isolation was 71.1ms, and the worst-case execution time of HOG running in isolation was 768.9 ms. However, the *median* execution time of HOG in the HOG+{2SD,HOG} scenario was 3747.0ms. Had the scheduler simply treated the GPU as an exclusive resource when running two instances of HOG and two instances of SD, we could expect HOG's worst-case execution time to be closer to 1680.0ms, which is the sum of each instance's worst-case execution time in isolation. By examining the curve for SD+{2HOG,SD} in Fig. 5, we see that SD in this scenario has a median execution time only approximately 30ms worse than its execution time in isolation. Since a single iteration of HOG performs over 180 kernel invocations of varying sizes, and an iteration of SD performs only one, the plots support the hypothesis that a large portion of the effect on HOG is due to HOG's multiple kernels being interleaved with SD's single kernel at multiple points in each HOG iteration. While one may argue that this scheduling in SD's favor is beneficial in some applications, the significantly increased execution time for HOG may result in an overall net loss in terms of schedulability.

Obs. 4. The TX1 platform exhibited similar trends to those observed on the TK1.

The TX1, with greater resources, unsurprisingly exhibited improved execution times. Most interference patterns, however, applied to both platforms. This is shown in Fig. 7, which shows similar patterns to Fig. 4, and Fig. 8, which is analogous to Fig. 6 (discussed next).

An anomalous result. We conclude this section by discussing an anomalous result that suggests that further study of sources of interference among GPU-using tasks is needed.

Obs. 5. In rare cases, a benchmark program exhibited *better* performance when executing in the presence of a competing workload rather than in isolation.

We were very surprised to find that in some cases, increasing the concurrent workload unintuitively led to slight *improvements* in observed benchmark execution times. We observed such improvements in two sets of scenarios, shown in Figs. 6 and 8, where instances of HOG exhibited execution-time improvements with additional competition. This behavior was noticed in HOG with one or two VADD competitors on the TK1, and with up to 3 VADD competitors on the TX1. The only other scenarios where we observed such behavior involved the CONV benchmark competing against additional CONV instances.

Our current hypothesis is that this behavior is due to DRAM or CPU L2 cache activity. This hypothesis is based on the observation that, in Fig. 6, the VADD benchmark runs solely on the CPU. This fact eliminates GPU contention as the source of the anomaly in Fig. 6, leaving only hardware resources shared by the two benchmarks as potential causes: the CPU, its L2 cache, and the DRAM banks. We still, however, do not have a concrete explanation of this anomalous behavior, and plan to continue investigating it in hopes of identifying specific causes.

5 Conclusion

In order to effectively use GPUs in automotive settings, it is imperative to not waste GPU capacity. Such waste can lead to the necessity of introducing additional hardware, which can have a detrimental impact with respect

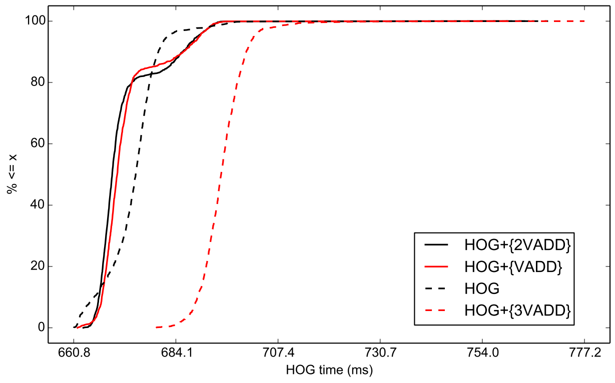


Figure 6: CDF of execution times of HOG in scenarios involving VADD competitors (which are CPU-only).

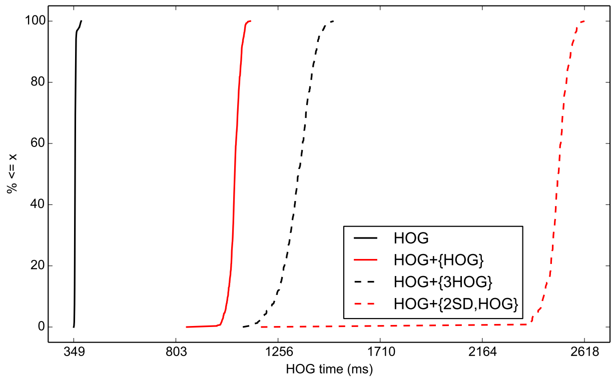


Figure 7: The same scenarios as Fig. 4 running on the TX1.

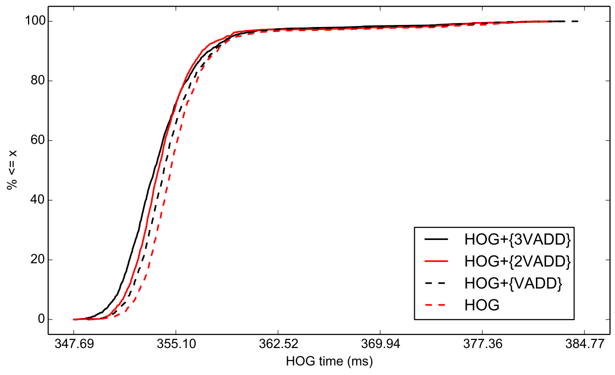


Figure 8: The same scenarios as Fig. 6 running on the TX1.

to SWaP and monetary cost. Unfortunately, most prior GPU management frameworks proposed for real-time systems [7, 8, 9, 12, 16, 25, 26, 50, 48, 49, 55, 56] preclude multiple tasks from executing GPU kernels concurrently. If such a kernel requires only a relatively small fraction of a GPU's processing cores, then much of that GPU's capacity will be wasted. In this paper, we have explored the possibility of allowing multiple kernels to be co-scheduled in the context of image-processing applications. Our results suggest that, in some cases, allowing multiple kernels to be co-scheduled can have a positive impact on real-time schedulability. Allowing such functionality will require new extensions to prior real-time GPU management frameworks.

In future work, we plan to introduce such extensions to the frameworks developed by our group, GPUSync and GPUSyncLite. These extensions will require the use of real-time locking protocols that sometimes allow multiple tasks to hold locks simultaneously. Blocking analysis will be required for these protocols as well. We believe that the needed protocols can be obtained by using ideas found in recently proposed multiprocessor real-time locking protocols for managing replicated resources [34]. Our idea here is to abstractly view a single GPU as a replicated resource and require a task to lock only the replicas it needs. In other future work, we intend to conduct more in-depth experimental studies to try to discern the root sources of interference that cause some kernels to perform poorly when co-scheduled. Additionally, we plan to consider other GPU-based hardware platforms that might be viable in automotive use cases.

References

[1] Google self-driving car project. Online at `https://www.google.com/selfdrivingcar/`, 2016.

[2] A. Alhammad and R. Pellizzoni. Trading cores for memory bandwidth in real-time systems. In *RTAS '16*.

[3] A Alhammad, S. Wasly, and R. Pellizzoni. Memory efficient global scheduling of real-time tasks. In *RTAS '15*.

[4] S. Altmeyer, R. Douma, W. Lunniss, and R.I. Davis. Evaluation of cache partitioning for hard real-time systems. In *ECRTS '14*.

[5] AMD. Amd embedded g-series system-on-chip product brief. Online at `https://www.amd.com/Documents/AMDGSeriesSOCProductBrief.pdf`.

[6] N. Audsley. Memory architecture for NoC-based real-time mixed criticality systems. In *WMC '13*.

[7] J. Aumiller, S. Brandt, S. Kato, and N. Rath. Supporting low-latency CPS using GPUs and direct I/O schemes. In *RTCSA '12*.

[8] C. Basaran and K. Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *ECRTS '12*.

[9] K. Berezovskyi, , L. Santinelli, K. Bletsas, and E. Tovar. WCET measurement-based and extreme value theory characterisation of CUDA kernels. In *RTNS '14*.

[10] K. Berezovskyi, K. Bletsas, and B. Andersson. Makespan computation for GPU threads running on a single streaming multiprocessor. In *ECRTS '12*.

[11] K. Berezovskyi, K. Bletsas, and S. Petters. Faster makespan estimation for GPU threads on a single streaming multiprocessor. In *ETFA '13*.

[12] A. Betts and A. Donaldson. Estimating the WCET of GPU-accelerated

applications using hybrid analysis. In *ECRTS '13*.

[13] M. Campoy, A. Ivars, and J. Mataix. Static use of locking caches in multitask preemptive real-time systems. In *IEEE/IEE Real-Time Embedded Sys. Workshop '01*.

[14] M. Chisholm, B. Ward, N. Kim, and J. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *RTSS '15*.

[15] G. Elliott. *Real-Time Scheduling of GPUs, with Applications in Advanced Automotive Systems*. PhD thesis, University of North Carolina at Chapel Hill, 2015.

[16] G. Elliott, B. Ward, and J. Anderson. GPUSync: A framework for real-time GPU management. In *RTSS '13*.

[17] G. Giannopoulou, N. Stoimenov, P. Huang, and L.Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *EMSOFT '13*.

[18] M. Hassan and H. Patel. Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems. In *RTAS '16*.

[19] M. Hassan, H. Patel, and R. Pellizzoni. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *RTAS '15*.

[20] J. Herter, P. Backes, F. Haupenthal, and J. Reineke. CAMA: A predictable cache-aware memory allocator. In *ECRTS '11*.

[21] Intel. Intel atom processor series product brief. Online at `http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/atom-x3-c3000-brief.pdf`.

[22] R. Jain, C. Hughs, and S. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *RTSS '02*.

[23] J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello, and P. Cazorla. A dual-criticality memory controller (DCmc) proposal and evaluation of a space case study. In *RTSS '14*.

[24] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *ACMMM '14*.

[25] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *RTSS '11*.

[26] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX Annual Technical Conference '11*.

[27] H. Kim, D. Broman, E. Lee, M. Zimmer, A. Shrivastava, and J. Oh. A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In *RTAS '15*.

[28] H. Kim, D. de Niz, B. Anderson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *RTAS '14*.

[29] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *ECRTS '13*.

[30] N. Kim, B. Ward, M. Chisholm, C.-Y. Fu, J. Anderson, and F.D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *RTAS '16*.

[31] Y. Krishnapillai, Z. Wu, and R. Pellizzoni. ROC: A rank-switching, open-row DRAM controller for time-predictable systems. In *ECRTS '14*.

[32] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*.

[33] R. Mancuso, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time colored lockdown for cache-based multi-core architectures. In *RTAS '13*.

[34] C. Nemitz, K. Yang, M. Yang, P. Ekberg, and J. Anderson. Multiprocessor real-time locking protocols for replicated resources. In *ECRTS '16*.

[35] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity environment. In *ECRTS '14*.

[36] NVIDIA. Cuda sample programs. Online at `http://docs.nvidia.com/cuda/cuda-samples`.

[37] NVIDIA. Cuda zone. Online at `http://www.nvidia.com/object/cuda_home_new.html`.

[38] NVIDIA. Jetson tx1 system-on-module data sheet. Online at `https://developer.nvidia.com/embedded/downloads`.

[39] NVIDIA. Whitepaper: NVIDIA Tegra K1. Online at `http://www.nvidia.com/content/pdf/tegra_white_papers/tegra-k1-whitepaper.pdf`.

[40] NVIDIA. Whitepaper: NVIDIA Tegra X1. Online at `http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf`.

[41] NXP. i.mx 6dual/6quad automotive and infotainment applications processors data sheet. Online at `http://cache.freescale.com/files/32bit/doc/data_sheet/IMX6DQAEC.pdf`.

[42] N. Otterness, V. Miller, M. Yang, J. Anderson, F.D. Smith, and S. Wang. GPU sharing for image processing in embedded real-time systems. Full version of this paper available at `http://www.cs.unc.edu/~anderson/papers.html`, 2016.

[43] R. Pellizzoni, A. Schranzhofer, J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *DATE '10*.

[44] V. Prisacariu and I. Reid. fastHOG–a real-time GPU implementation of HOG. *Department of Engineering Science*, 2310, 2009.

[45] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. Phatak, R. Pellizzoni, and M. Caccamo. A real-time scratchpad-centric OS for multi-core. In *RTAS '16*.

[46] S. Thrun. Toward robotic cars. *Communications of the ACM*, 53:99–106, 2010.

[47] P. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS '16*.

[48] U. Verner, A. Mendelson, and A. Schuster. Batch method for efficient resource sharing in real-time multi-GPU systems. In *ICDCN '14*.

[49] U. Verner, A. Mendelson, and A. Schuster. Scheduling periodic real-time communication in multi-GPU systems. In *ICCCN '14*.

[50] U. Verner, A. Mendelson, and A. Schuster. Scheduling processing of real-time data streams on heterogeneous multi-GPU systems. In *SYSTOR '12*.

[51] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *ECRTS '13*.

[52] M. Xu, S. Mohan, C. Chen, and L. Sha. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *RTAS '16*.

[53] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicoore platforms. In *RTAS '14*.

[54] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *ECRTS '12*.

[55] J. Zhong and B. He. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *TIEEE Transactions on Parallel and Distributed Systems*, 25:15221532, 2014.

[56] H. Zhou, G. Tong, and C. Liu. GPES: A preemptive execution system for GPGPU computing. In *RTAS '15*.

Combining Predictable Execution with Full-Featured Commodity Systems

Adam Lackorzynski, Carsten Weinhold, Hermann Härtig
Operating Systems Group, Technische Universität Dresden
{adam.lackorzynski,carsten.weinhold,hermann.haertig}@tu-dresden.de

***Abstract*—Predictable execution of programs is required to satisfy real-time constraints in many use cases, including automation and controlling tasks. Unfortunately, background activities of the operating system may influence execution of such workloads in unpredictable ways, as do other applications running on the system concurrently. Separating time-critical workloads from unrelated activities is thus a common approach to ensure predictable execution.**

Different strategies are used to achieve this separation. On multi-core systems, developers typically assign work loads to dedicated cores, which then run completely separate software stacks. They often do not provide fault isolation nor security guarantees. Another approach is to co-locate a commodity operating system with a real-time executive, which hooks into the interrupt subsystem of the standard kernel to run real-time code at the highest priority in the system. There are also ongoing activities to modify commodity kernels such as Linux to enable more predictable execution. This pairing of the rich and versatile feature set of Linux with a real-time execution is very compelling, but it requires significant developer effort to ensure that the huge monolithic code base does not violate real-time requirements.

In this paper, we present a mechanism that combines predictable execution and all of Linux' functionality with much less effort. It allows unmodified programs to be started on top of a virtualized Linux kernel and then "pull them out" of the virtual machine to let them run undisturbed on the microkernel that hosts Linux. Whenever the program performs a system call or catches an exception, those are forwarded to Linux transparently. Experimental results show that execution-time variation is reduced by two orders of magnitude.

I. Introduction

Predictable execution, often also called real-time execution, is a required functionality for a broad range of uses cases. Common real-time operating systems (RTOS) are simple, and thus predictable, but lack features commonly offered by a full-featured commodity operating system (OS), such as Linux. Unfortunately, full-featured OSes typically cannot ensure predictable execution. Still, there is ongoing work on full-featured operating systems aiming to run real-time workloads. An example are the real-time extensions for Linux (Linux-RT [1]) which are merged step-by-step into mainline Linux. However, it is a constant challenge to keep the huge code base of the kernel preemptible, while hundreds of developers add new features all the time or rework entire subsystems; preemptibility and real-time capabilities are typically not the main concerns for most of these developers.

Another common approach is to use multi-core systems and to run a commodity OS and an RTOS on the same system. This provides good temporal isolation but lacks spatial isolation. Both OSes exist side by side and interaction between them is usually coarse grained, for example, through mailbox systems. As each of the two OSes runs with full system privileges, such setups do not offer effective fault containment as required for security and safety-critical usage scenarios. A hypervisor (or microkernel) that virtualizes the platform can contain faults within each of the two software stacks by deprivileging their OSes. Adding the virtualization layer may cause a slight performance degradation, but run-time overheads are not prohibitive in most cases.

The contribution of this work is a mechanism for combining the flexibility and feature set of a full-featured commodity OS with the real-time characteristics of an RTOS. By means of virtualization, we enable threads of a program to detach from the unpredictable commodity OS and run in the real-time capable environment. Whenever such a detached thread needs to call services of the feature-rich OS (e.g., system calls), those requests will be forwarded to the commodity OS transparently. In our prototype, we use the L4Re system, a microkernel-based operating system framework [2] for building customized systems. The L4Re microkernel serves both as a hypervisor and in the role of an RTOS to runs detached threads. We use L^{4}Linux, a paravirtualized variant of the Linux kernel that comes with L4Re. It has been adapted to run on the L4Re system as a deprivileged user-space application.

We are not the first to combine a real-time executive and a feature-rich commodity operating. However, our approach represents a new way of building this kind of split OS platform. We reach this goal without "reinventing the wheel" by enhancing existing microkernel technology with a simple mechanism. We believe that our approach is low-effort, maintainable, and it provides continuous access to the latest releases of the feature-rich OS. This paper builds on our previous work in the context of high-performance computing [3].

In the remainder of the paper, we will describe our system in more detail (Section II) and then discuss the detaching mechanisms we added in Section III. We evaluate our work in Section IV before we conclude.

II. Virtualization System

We build an OS for predictable execution based on the L4Re microkernel system, which hosts a virtualized Linux kernel called L^{4}Linux. To get an understanding of L4Re's capabilities and the detaching mechanism described in Section III, we will now introduce the L4Re system architecture.

A. L4Re Microkernel and User Land

The L4Re microkernel is a small and highly portable kernel. It is the only component of the system running in the most privileged mode of the CPU. Its main task is to provide isolation among the programs it runs, in both the spatial and temporal domains. To do so, the kernel needs to provide mechanisms for all security-relevant operations, such as building up virtual memory for programs and scheduling them. To support virtual machines (VMs), the kernel also provides abstractions for those virtualization-related CPU instructions that can only be executed in the most privileged processor mode. Thus it also takes the role of a hypervisor. Functionality that is not required to enforce isolation of applications and virtual machines is built on top of the kernel in user-level components.

The L4Re system is a component-based operating system framework that provides a set of components on top of the microkernel. It can be tailored to the needs of applications. The set of components includes services and libraries for memory management, application loading, virtual machines, device drivers, and more. As the L4Re system provides functionality as components, applications need to rely only on those services they use, thereby minimizing their Trusted Computing Base (TCB). The TCB is also application-specific, as different application to may depend on different services. This is in contrast to monolithic designs, where, for example, a malfunction in a file-system leads to a kernel panic that concerns every application, including those that do not use that file-system at all.

The L4Re microkernel supports hardware-assisted virtualization such as Intel's VT and ARM's VE, as well as a pure software approach to hosting VMs. The latter only relies on the memory management unit, which also provides address spaces for isolating ordinary applications. The kernel provides interfaces specifically designed so that OS developers can port their kernels to L4Re with little effort. This paravirtualization support includes support for mapping guest processes and threads to the L4 tasks and L4 vCPUs that the microkernel provides: An L4 task encapsulates address spaces both for memory and kernel objects such as capabilities; a vCPU is a thread and thus a unit of execution, however, enriched with features beneficial for virtualization.

Besides providing address spaces through L4 tasks and execution through L4 thread and vCPUs, the microkernel provides a few more mechanisms. Interrupts are abstracted using Irq objects. Irqs are used for both physical device interrupts as well as for software-triggered interrupts. The microkernel also schedules the threads on the system and offers multiple, compile-time selectable scheduling algorithms.

The whole L4Re system is built around an object capability model. Any operation on an object outside the current L4 task must be invoked through a capability; this includes the objects that provide inter-process communication (IPC) and Irqs. Thus one can state that IPC is used to invoke capabilities. L4Re uses the same invocation method for all objects in the system, whether they are implemented in the microkernel itself or provided by user-level applications.

The L4Re microkernel always runs on all cores of the system and address spaces span all cores; threads can be migrated. The microkernel itself will never migrate a thread between cores on its own; however, user-level applications can request migrations.

B. L4Linux

In our work, we use L^4Linux, a paravirtualized variant of the Linux kernel that has been adapted to run on the L4Re system. L^4Linux is binary compatible to normal Linux and runs nearly any Linux binary [4]. We chose L^4Linux instead of a fully-virtualized Linux because L^4Linux is integrated more tightly with the underlying L4Re system and thus allows our approach to be implemented much more easily. In the following we will describe L^4Linux in sufficient detail to understand our approach to detaching thread execution.

The L^4Linux kernel runs in an L4 task and each Linux user process is assigned its very own L4 task, too. Thus, the L^4Linux kernel is protected from misbehaving applications like native Linux is, where user processes run in another privilege level. There are no dedicated L4 threads for the user processes as those are provided by the vCPU. A vCPU is like a thread, but provides additional functionality useful for virtualization. Such features include an asynchronous execution model with a virtual interrupt flag and also the ability of a vCPU to migrate between address spaces which is used to implement user processes. Thus, from the host's point of view, an L^4Linux VM comprises multiple vCPUs (one for each virtual CPU in the guest) and L4 tasks that provide address spaces for the L^4Linux kernel and each user process.

During operation, a vCPU executes both guest kernel code and the code of the user processes. When the L^4Linux kernel performs a *return-to-user* operation, the vCPU state is loaded with the register state of the user process as well as the L4 task of the user process. The vCPU will then continue execution in that task. For any exception that occurs during execution (e.g., system call invocations or page faults), the vCPU migrates back to the guest kernel task and resumes execution at a predefined entry vector, where the exception is analyzed and handled appropriately. Interrupts are handled similarly: After having bound a vCPU to an interrupt object, firing the interrupt will halt current execution and transfer the vCPU to the entry point of the L^4Linux kernel where the interrupt will be processed.

Memory for user processes is exclusively managed by the Linux kernel. To populate the address spaces of user processes, L^4Linux maps memory from the Linux kernel task into the respective L4 tasks using L4 system calls to *map* and *unmap* memory pages. When resolving page faults for user processes, L^4Linux traverses the page tables that Linux builds up internally to look up guest-kernel to user address translations. Note that those shadow page tables are not used by the CPU. Only the L4 microkernel manages the hardware page tables; the only way to establish mappings in Linux user processes (or any other L4 task) is to use the microkernel's map functionality.

III. Detaching Work

Now we want to pursue how we can separate a thread of a Linux user program so that it can run undisturbed from the rest of the L^{4}Linux system. As described in the previous section, L^{4}Linux does not use separate L4 threads for user processes, but it multiplexes user threads onto a single vCPU. However, to isolate execution of a user thread from the unpredictable L^{4}Linux, we must create a dedicated L4 thread that is not managed by the Linux scheduler. This detached thread will run Linux user code, be scheduled by the L4Re microkernel independently from L^{4}Linux's scheduler. As a separate L4 thread, we can also move it to a different core, preferably one that does not share caches with L^{4}Linux. A schematic view of our architecture is depicted in Figure 1.

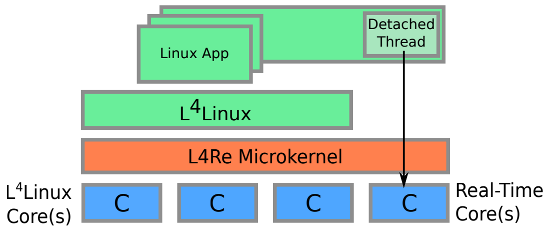


Fig. 1. A thread detached from L^{4}Linux running on a separate core.

To implement the creation of separate threads we can leverage infrastructure developed in previous versions of L^{4}Linux: the thread-based execution model [5], in which Linux threads are mapped one-to-one to L4 threads. This approach to threading in L^{4}Linux predates the superior vCPU execution model, but it is still being maintained. We build upon this older implementation to add creation of separate L4 threads to the vCPU model that is now used. Detached processes start as normal processes, for which a new L4 host thread is created and placed in the L4 task of the user process. Then, instead of resuming execution through the vCPU, the execution is resumed to the L4 thread by using L4 exception IPC. Exception IPC is a special type of IPC carrying the thread's register state and that is used to transfer the exception state between the causing thread and a handler thread, which is the L^{4}Linux kernel.

After launching the thread, L^{4}Linux puts the kernel-part of the user thread into *uninterruptible* state and calls `schedule()` so that another context is chosen. While a context is in state *uninterruptible* it is not chosen to be dispatched by the Linux scheduler. Thus, in L^{4}Linux's view, the context is blocked, however, it is running outside and independent of the virtual machine provided by L^{4}Linux.

While the detached program is running, it will eventually cause an exception, such as triggered by issuing a system call, or causing a page fault. In both cases the thread's state will be transferred to the L^{4}Linux kernel using L4 exception IPC. However, the context that will be active at that time in L^{4}Linux's kernel will not be the one of the detached thread as this one is in *uninterruptible* state. Thus the L^{4}Linux kernel will just save the transmitted state in the thread's corresponding kernel context and bring the thread out of the *uninterruptible* state via a *wakeup* operation. When L^{4}Linux's scheduler has chosen the thread again, the request will be handled. When done, execution is resumed by replying to the incoming exception IPC and setting the thread to *uninterruptible* again.

By using exception IPC, any request made by the detached user-level thread is transparently forwarded to the L^{4}Linux kernel. One may also describe that in a way that the user thread is being reattached while executing requests to the L^{4}Linux kernel.

A. L4 Interactions

When a thread is running detached, it is not restrained to run code only but it can also interact with other L4 components or the microkernel. For example, a control loop can be implemented using absolute timeouts of the L4 system or the thread can wait on other messages or interrupts, including device interrupts. Waiting directly for device interrupts in detached threads might be beneficial to avoid interaction with the Linux kernel and thus to achieve lower interrupt response latency.

For doing L4 IPC, the L^{4}Linux kernel needs to provide the thread information where its User Thread Control Block (UTCB) is located. The UTCB is a kernel provided memory area that is used to exchange data with the kernel and communication partners. The way of retrieving the address of the UTCB, as used in native L4 programs, does not work within an L^{4}Linux environment as the segment, as used on x86, registers are managed by L^{4}Linux and might be used by the libc. Thus an alternative approach must be provided, for example, by a specifically provided extra system call. As the UTCB address is fixed for a thread, it can be cached. When just using one thread in the application, the UTCB address is always the same and a well-known constant can be used as a shortcut.

For the program to communicate with other L4 services, the L^{4}Linux kernel needs to map a base set of capabilities into the task of the user process. L^{4}Linux must have been setup accordingly to receive those capabilities itself beforehand. Further the user program must be able to get information on where which capabilities have been mapped. In L4Re, this information is provided through the environment when the application is started. As application starting is done by the L^{4}Linux kernel, an alternative approach is required, such as a dedicated system call or a `sysfs` interface.

B. Implementation Details

In the following we will shortly describe interesting aspects of the implementation.

1) Signal Handling: As threads run detached from the L^{4}Linux kernel they are blocked by being in the *uninterruptible* state. This affects signal delivery, such as `SIGKILL`, to take effect, as the signal will just be processed when the thread is in the kernel or enters it. When the detached thread never enters the L^4Linux kernel again ("attaches" again), any posted signal will have no effect. For that reason, we added a mechanism that periodically scans detached threads for pending signals, and if it finds any, the detached thread is forced to enter the L^4Linux kernel to have the signal processed eventually.

2) Memory: As already described, all memory of a detached application is managed by L^4Linux. Linux may do page replacement on the pages given to the application which in turn affect the undisturbed execution. Thus it is advised that applications instruct the L^4Linux kernel to avoid page replacement by means of `mlock` and `mlockall` system calls. Generally, using large pages to reduce TLB pressure is also recommended. L^4Linux and the L4Re microkernel support large pages.

With the possibility of a detached thread to call out to other L4 services, it could also acquire memory pages. This is possible, given the application is provided with appropriate service capabilities, however, care must be taken as the address space is managed by the L^4Linux kernel and Linux is unaware of other mappings in the address space. Reservations of regions of the address space can be done via `mmap`, and given no page faults are generated in those regions, the pages can be used. Using memory from elsewhere is useful, for example, to use shared memory with other L4Re applications.

3) Floating Point Unit: vCPUs also multiplex the state of the Floating Point Unit (FPU) on behalf of the virtualized OS kernel. FPU handling for vCPUs is built in a way that it matches the hardware's behavior and thus aligns well with how operating systems handle the FPU natively. Although a vCPU can handle multiple FPU states, only one at a time can be active per vCPU. However, with detached threads, there are additional L4 threads, and thus active FPU states, that need to be handled.

The FPU-state multiplexing is built in a way that an FPU state travels between different threads, that is, the set of L4 threads building up a virtual CPU just use one single FPU state. Additionally, the state is handled lazily so that an FPU state transfer must only be done when the FPU is actually used. Thus, when a detached L4 thread enters the L^4Linux kernel, its FPU state cannot be transferred automatically to the L^4Linux kernel because another FPU state might be active there. To resolve this situation, we extended the L4Re microkernel with an operation for explicitly retrieving a thread's FPU state. This way L^4Linux can save the FPU state of a thread to L^4Linux kernel's internal FPU state for other Linux activities to access it. An operation for setting the FPU state of an L4 thread is not required because the FPU state is transferred with the exception IPC upon the resume operation. This is possible because resumption is done out of the thread's context, contrary to the incoming operation, that is done on a different context.

4) Sysfs Interface: As already described, we use a `sysfs`-based interface to control detaching of threads. Contrary to using an extra system call, this gives use the possibility to easily use it in wrapper scripts without requiring to modify the application itself. Noteworthy characteristics is that the detached state is retained through the `execve` system call, allowing to build wrapper scripts that detach an application:

```
#! /bin/sh
SYSFS_PATH=/sys/kernel/l4/detach
echo $$ > $SYSFS_PATH/detach
echo $HOST_CORE_ID > $SYSFS_PATH/$$/cpu
exec "$@"
```

As seen, specifying the target host CPU of the detached thread is also possible via the `sysfs` interface. The `sysfs` interface will only detach the first thread of an application, thus multi-threaded programs will need to take care of detached threads themselves.

IV. Evaluation

In the following we will evaluate our detaching mechanism regarding undisturbed execution. First, we use the FWQ benchmark, which is famous in the high performance computing (HPC) area for measuring OS noise. Then we will implement a control loop and monitor results for timing deviations. With both experiments we will generate load in the L^4Linux VM.

For all benchmarks, we use the same x86 system, running an Intel® Core™ i7-4770 quad-core CPU clocked at nominally 3.4GHz, reported with 2993MHz.

A. FWQ Benchmark

First, we run the fixed-work quantum (FWQ) benchmark [6]. The benchmark measures a fixed amount of work multiple times. Ideally the time it takes to run the work loop is the same for all runs, however, due to preemptions and other activities in the OS and the hardware, the measured times fluctuate. Thus the degree of deviation shows the impact of those other activities. The benchmark executes the work 10,000 times.

Figure 2 shows a run of FWQ on native Linux-4.6 built with preemption enabled (`CONFIG_PREEMPT=y`) and run with `chrt -f 10` while I/O intensive work is running as well, comprising network and disk load.

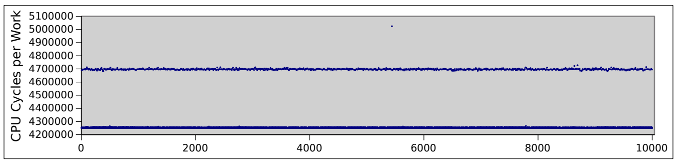


Fig. 2. FWQ results for Linux-4.6 PREEMPT with I/O load in Linux.

We see, although the FWQ benchmark is running as a real-time program and the Linux kernel uses its full preemption mode, deviation goes up to about 18%.

When running the same FWQ benchmark in L^4Linux using our presented mechanism, we measure results as seen in Figure 3. The maximum deviation is 1152 CPU cycles, or 0.027%.

When we run a Linux kernel compile instead of I/O load in L^4Linux, we see a pattern as in Figure 4 that has larger deviations: 6500 cycles, or 0.15%. When the L^4Linux is idle,

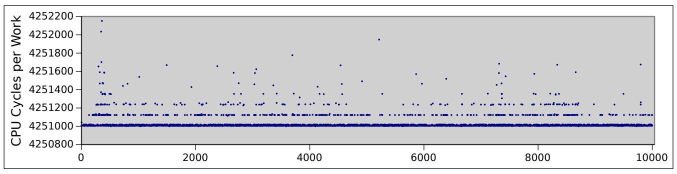


Fig. 3. FWQ results for detached mode with a I/O load in L^{4}Linux-4.6.

we see a behavior as seen in Figure 5 with just 21 cycles difference.

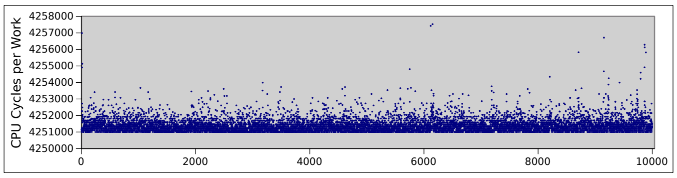


Fig. 4. FWQ results for detached mode with a build load in L^{4}Linux-4.6.

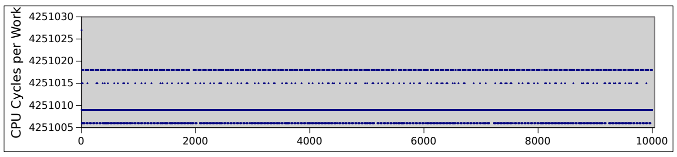


Fig. 5. FWQ results for detached mode with an idle L^{4}Linux-4.6.

Although the FWQ benchmark is so small that it is running out of L1 cache, effects can be seen in the results. Our speculation is that due to the inclusiveness of the caches in Intel's multi-level cache architecture, cache content can be evicted due to aliasing. However, whether this explains the different levels in Figure 3 is unclear and requires further investigations that are out of scope for this paper.

In summary, the results show for the FWQ benchmark that our detaching mechanism significantly improves the execution predictability of programs. It effectively isolates activities of the Linux kernel and unrelated background load from the detached real-time program, such that execution-time jitter is reduced by more than two orders of magnitudes.

B. Host-driven Control Loop

In our second experiment, we emulate a control loop that blocks repeatedly until an absolute time in the future to execute some task. In each iteration of the loop, we increment the programmed wake-up time by 1,000μs (`delta = 1,000`μs) as illustrated in the following code:

```
next = now() + delta;
while (1) {
  wait_for_time(next);
  /* do work */
  next += delta;
}
```

While the loop is running, we capture the time-stamp counter (TSC) and plot the delta of each consecutive loop iteration. Ideally, the measured delta between TSC-read operations should be constant, meaning that the `wait_for_time` call unblocks at precisely the specified time. The target for the delta is 2,993,000 cycles, as determined by CPU clock speed of 2,993MHz. We run the loop for 10,000 iterations so that the benchmark runs for 10 seconds.

On Linux, we implement the blocking using `clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, ...)`. We see results as depicted in Figure 6 for I/O load and in Figure 7 for a build load. The way to generate the load has been the same as in the previous experiment. All Linux programs are pinned to a core and run with real-time priority (using `chrt -f 10`).

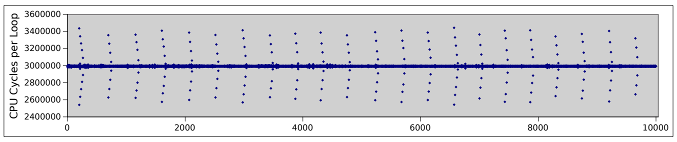


Fig. 6. Control loop results on native Linux with I/O load.

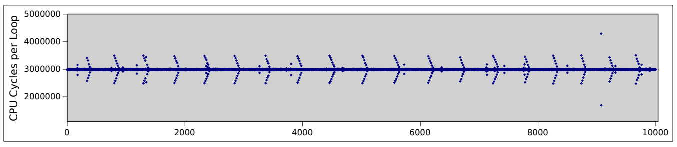


Fig. 7. Control loop results on native Linux with build load.

The two graphs show an interesting arrow-style pattern. With about 20 of such outlier events, one each half second, we suspect an internal activity in the Linux kernel that induces this result. We see a deviation from the target of about 500,000 CPU cycles in each direction, translating to about 167μs. The results for the I/O-load experiment look similar to the build-load case, however, there is an even larger outlier with about 1,300,000 cycles deviation (430μs).

With L^{4}Linux, using our detaching mechanism, the control loop uses L4 system calls to block until a specified point in time (absolute timeout). Thus, the blocking and unblocking is directly done by the microkernel and does not use or depend on Linux. We use the same background load as before; the results are shown in Figures 8 and 9. Note the change of range in the y-axis.

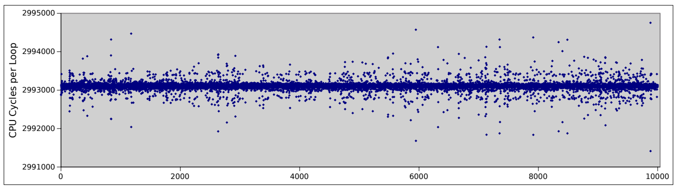


Fig. 8. Control loop results on L^{4}Linux with I/O load.

The major difference between Linux and L^{4}Linux is the significantly reduced deviation. With I/O load, we observe that the biggest outlier is about 1700 cycles away from the target

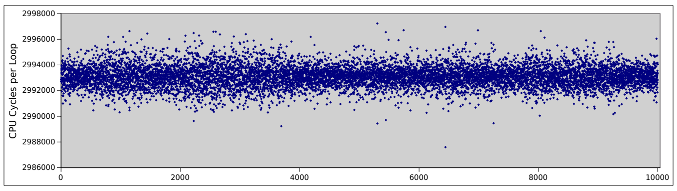


Fig. 9. Control loop results on L^{4}Linux with build load.

while the biggest outlier of the build load is about 4700 cycles away, translating to 600ns and 1.6μs deviation. This is a 2-fold improvement over the Linux results.

V. Related Work

There is plenty of work regarding the combination of real-time and general purpose operating systems (GPOS), using virtualization or co-location approaches. There are also efforts for enhancing the real-time capabilities of Linux itself [1].

In co-location approaches, a real-time executive is added to the GPOS that hooks into low-level functions to execute real-time tasks. Examples are Xenomai [7] and RTAI [8].

Xen-RT [9] adds real-time support to the Xen Hypervisor [10] by adding real-time schedulers. Jailhouse [11] is a recent development that uses the Jailhouse hypervisor to partition Linux and an RTOS to different cores on a multi-core system. Other hypervisors for real-time are Xtratum [12] and SPUMONE [13], and there are also commercial offerings, such as Greenhill's Integrity.

Similar work is also done in the HPC community. Although the real-time and HPC communities are typically disjunctive, they strive for similar goals. The focus in HPC is to minimize the disturbance caused by other software, such as the OS, and hardware, that is experienced while executing HPC applications. Uninterrupted execution is required because HPC application communicate over many nodes where a delay on a single node also has influences on other nodes. Thus disturbance must be minimized [14]. Proposed solutions are similar to what is done in the real-time area: Multi-core systems are partitioned into "OS Cores" and "Compute Cores". The OS core(s) typically run Linux to provide functionality that applications running on the compute cores require, but that the jitter-free "light-weight kernel" (LWK) does not implement. Several implementations of this approach exist, such as mOS [15] and McKernel/IHK [16], as well as our own work [3].

VI. Conclusion and Future Work

Our experiments show that our detaching mechanism is capable of improving the predictability of execution by at least two orders of magnitude compared to using a standard Linux. As the real-time programs on our system are unmodified Linux programs, existing development environments and tool can be used. This allows for an efficient use of developer's time when implementing timing sensitive functionality.

Implementing this or a similar mechanism using hardware-assisted virtualization promises to use any available Linux version, giving a broader access to platforms. We also plan evaluation on other architectures than Intel x86.

Acknowledgment

The research and work presented in this paper is supported by the German priority program 1500 "Dependable Embedded Software" and the German priority program 1648 "Software for Exascale Computing" via the research project FFMK [17]. We also thank the cluster of excellence "Center for Advancing Electronics Dresden" (*cfaed*) [18].

References

[1] Real-Time Linux Project. Real-Time Linux Wiki. https://rt.wiki.kernel.org.
[2] Alexander Warg and Adam Lackorzynski. The Fiasco.OC Kernel and the L4 Runtime Environment (L4Re). avail. at https://l4re.org/.
[3] Adam Lackorzynski, Carsten Weinhold, and Hermann Härtig. Decoupled: Low-Effort Noise-Free Execution on Commodity System. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '16, New York, NY, USA, 2016. ACM.
[4] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A Study of Modern Linux API Usage and Compatibility: What to Support when You'Re Supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 16:1–16:16, New York, NY, USA, 2016. ACM.
[5] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.
[6] Lawrence Livermore National Laboratory. The FTQ/FWQ Benchmark.
[7] Xenomai Project. https://xenomai.org.
[8] RTAI – Real Time Application Interface. https://www.rtai.org/.
[9] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. Rt-xen: towards real-time hypervisor scheduling in xen. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, pages 39–48. ACM, 2011.
[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177. ACM, 2003.
[11] Jan Kiszka and Team. Jailhouse: Linux-based partitioning hypervisor . http://www.jailhouse-project.org/.
[12] A. Crespo, I. Ripoll, and M. Masmano. Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach. In *Dependable Computing Conference (EDCC), 2010 European*, pages 67–72, April 2010.
[13] Tatsuo Nakajima, Yuki Kinebuchi, Hiromasa Shimada, Alexandre Courbot, and Tsung-Han Lin. Temporal and spatial isolation in a virtualization layer for multi-core processor based information appliances. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, ASPDAC '11, pages 645–652. IEEE Press, 2011.
[14] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010.
[15] R.W. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen. mOS: An Architecture for Extreme-scale Operating Systems. In *Proc. ROSS '14*, pages 2:1–2:8. ACM, 2014.
[16] T. Shimosawa, B. Gerofi, M. Takagi, G. Nakamura, T. Shirasawa, Y. Saeki, M. Shimizu, A. Hori, and Y. Ishikawa. Interface for heterogeneous kernels: A framework to enable hybrid os designs targeting high performance computing on manycore architectures. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–10, Dec 2014.
[17] FFMK Website. https://ffmk.tudos.org. Accessed 17 Jun 2016.
[18] cfaed Website. https://www.cfaed.tu-dresden.de/. Accessed 17 Jun 2016.

Timeliness Runtime Verification and Adaptation in Avionic Systems

José Rufino and Inês Gouveia
LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
jmrufino@ciencias.ulisboa.pt, igouveia@lasige.di.fc.ul.pt

***Abstract*—Unmanned autonomous systems (UAS) avionics call for advanced computing system architectures fulfilling strict size, weight and power consumption (SWaP) requisites, decreasing the vehicle cost and ensuring the overall system dependability. The AIR (ARINC 653 in Space Real-Time Operating System) architecture defines a partitioned environment for aerospace applications, following the notion of time and space partitioning (TSP), aiming to preserve the highly demanding application timing and safety requirements.**

In addition to expected changes in the vehicle configuration, which may naturally vary according to the mission's progress and its phases, a vehicle may be exposed to unforeseeable events (e.g., environmental) and to failures. Thus, vehicle survivability requires advanced adaptability and reconfigurability features, to be supported in the AIR architecture. Adaptation in the presence of hazards may largely benefit from the potential of non-intrusive runtime verification (RV) mechanisms, currently being included in AIR. Although this paper focuses on system level (timeliness) monitoring and adaptation, similar approaches and methods may be taken with respect to application/mission adaptation.

I. Introduction and Motivation

Avionic systems have strict safety and timeliness requirements as well as strong size, weight and power consumption (SWaP) constraints. Modern unmanned autonomous systems (UAS) avionics follow the civil aviation trend of transitioning from federated architectures to Integrated Modular Avionics (IMA) [1] and resort to the use of partitioning.

Partitioned architectures implement the logical separation of applications in criticality domains, named partitions, and allow hosting both avionic and payload functions in the same computational infrastructure, thus fulfilling both SWaP and safety/timeliness requirements [2]. Avionic functions are related with vehicle control and typically include: Attitude and Orbit Control Subsystem (AOCS) or Guidance, Navigation and Control (GNC); Onboard Data Handling (OBDH); Telemetry, Tracking and Command (TTC); Fault Detection, Isolation and Recovery (FDIR). On the other hand, payload functions are strictly related with the mission's purpose. Partitioning implies that each one of those functions is hosted in a different partition.

The notion of temporal and spatial partitioning (TSP) means that the execution of functions in one partition does not affect other partitions' timeliness and that dedicated addressing spaces are assigned to different partitions [3]. The design and development of AIR (ARINC 653 in Space Real-Time Operating System) has been motivated by the interest in applying TSP concepts to the aerospace domain [4]. However, TSP concepts can be applied to a broader set of applications such as, planetary exploration, automotive, underwater rovers, and aquatic/aerial drones. The case for low-cost drones, commonly available as radio-controlled gadgets, with little or no provisions of safety guarantees, is specially sensitive.

This work was partially supported by FCT, through LaSIGE Strategic Project 2015-2017, UID/CEC/00408/2013. This work integrates the activities of COST Action IC1402 - Runtime Verification beyond Monitoring (ARVI).

Usually, an UAS mission goes through multiple phases (e.g., takeoff, flight, approach, exploration, flight back, landing). Adaptation to changing temporal requirements as the mission progresses, throughout its phases, is of great importance. Furthermore, adaptation to unplanned circumstances, such as unforeseeable external events and internal failures, is mandatory for vehicle and mission's survivability [5]. The design of AIR Technology already includes mechanisms of support for adaptation and reconfiguration [6]. Nevertheless, given the high complexity of UAS functions, modern avionic systems may largely benefit from the verification in runtime whether or not the system/mission parameters are in conformity with the planned specification.

This paper addresses how innovative non-intrusive runtime verification (RV) capabilities, specially designed for time- and space-partitioned systems, may enable the design and implementation of advanced timeliness adaptation mechanisms, which allow to reduce the temporal overhead of such mechanisms in the operation of onboard systems.

The paper is organized as follows. Section II introduces the AIR architecture for TSP systems. Section III describes the non-intrusive RV features being introduced in the AIR architecture. Section IV details the new adaptability features of AIR. Section V discusses the integration of those features in the AIR architecture and performs its analysis. Section VI describes the related work and, finally, Section VII presents some concluding remarks and future research directions.

II. AIR Technology for TSP systems

The AIR Technology evolved from a proof of feasibility for adding ARINC 653 functional support to the Real-Time Executive for Multiprocessor Systems (RTEMS) to a multi-OS (operating system) TSP architecture [4]. The AIR modular design aims at high levels of flexibility, hardware- and OS-independence (through encapsulation), easy integration and independent component verification, validation and certification.

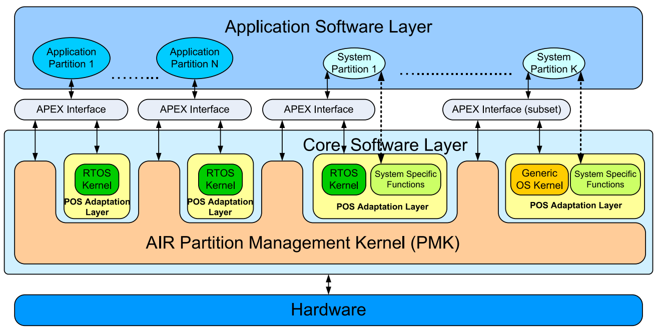


Fig. 1. AIR architecture for TSP systems

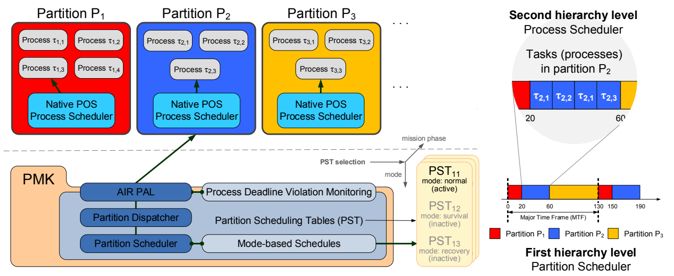

Partition P1
Partition P2
Partition P3
Native POS Process Scheduler
Native POS Process Scheduler
Native POS Process Scheduler
Second hierarchy level
Process Scheduler
Tasks (processes) in partition P2
PMK
AIR PAL
Process Deadline Violation Monitoring
Partition Dispatcher
Partition Scheduling Tables (PST)
Partition Scheduler
Mode-based Schedules
mission phase
PST selection
mode
PST11 mode: normal (active)
PST12 mode: survival (inactive)
PST13 mode: recovery (inactive)
Major Time Frame (MTF)
Partition P1
Partition P2
Partition P3
First hierarchy level
Partition Scheduler

Fig. 2. Two-level hierarchical scheduling with partition scheduling featuring timeliness adaptation through mode-based schedules

A. System architecture

The AIR modular architecture is pictured in Figure 1. The *AIR Partition Management Kernel (PMK)* is the basis of a core software layer, enforcing robust TSP properties and providing support to fundamental mechanisms such as partition scheduling and dispatching, low-level interrupt management, interpartition communication facilities and encapsulation of special-purpose hardware resources. Temporal partitioning ensures that the real-time requisites of the different functions executing in each partition are guaranteed. Spatial partitioning relies on having dedicated addressing spaces for the functions executing on different partitions.

Each partition can host a different OS (the partition operating system, POS), which, in turn, can be either a real-time operating system (RTOS) or a generic non-real-time one. The *AIR POS Adaptation Layer* (PAL) encapsulates the POS of each partition, providing an adequate POS-independent interface to the surrounding components.

The *Portable Application Executive* (APEX) *interface* [7] provides a standard programming interface derived from the ARINC 653 specification [1], with the possibility of being subsetted and/or adding specific functional extensions, on a system-level and/or on a per-partition basis [8].

The organization of vehicle functions in different partitions requires interpartition communication services, since a function hosted in a partition may need to exchange information with other partitions. Interpartition communication consists of the authorized transfer of information between partitions without violating neither spatial separation constrains nor information security properties [3], [4], [9].

B. Two-level scheduling

The AIR technology employs a two-level scheduling scheme, as illustrated in Figure 2. The first level corresponds to partition scheduling and the second level to process scheduling. Partitions are scheduled on a cyclic basis, through the partition scheduling and dispatching components (Figure 2), according to a partition scheduling table (PST) repeating over a major time frame (MTF). The PST assigns execution time windows to partitions. Inside each partition's time windows, its processes compete for processing resources according to the POS's native process scheduler.

This paper proposes an evolution of the AIR two-level hierarchical scheduling towards a highly effective short-term adaptation of timeliness parameters to the mission phase and/or to environmental changes, through a proficient use of mode-based schedules, as highlighted in Figure 2.

C. Health monitoring and event handling

The AIR architecture incorporates *Health Monitor* (HM) functions that spread throughout virtually all of the AIR architectural components, aiming to contain faults within their domains of occurrence.

At system-level, HM functions monitor the correctness of fundamental AIR system components. In the event of an error, handling is performed through fully integrated HM event handlers (Figure 3). For example: system-level timeliness (e.g., partition scheduling) is verified at runtime, with a contingency signalling of timing errors through low-level event handlers.

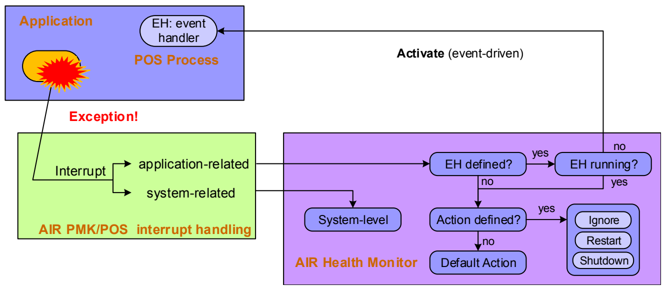


Fig. 3. System-level and application-level health monitoring

At application-level (Figure 1), which comprises both avionics and payload functions, HM functions aim to enforce overall correctness and to prevent the ill-effects of process and/or partition level errors, occurring at one partition, from propagating to the remaining partitions.

The runtime verification of application correctness is deeply dependent on the application itself. Detection of deviations from a given application/mission specification and handling of abnormal situations must be performed by special-purpose event handlers, provided by the application programmer and/or by the system integrator, as shown in the diagram of Figure 3. Only some specific aspects of application correctness may be verified at system-level (e.g., the monitoring of violations to registered process deadlines, pointed out in Figure 2).

In any case, the actions to be performed in the event of errors are cast into appropriate event handlers. These may comprise adaptability features such as the redefinition of timing and control parameters. If no handler is provided, a response action defined by the partition's HM ARINC 653 configuration table is executed, as shown in Figure 3. The design of AIR allows HM handlers to simply replace existing exception handlers or to be added to existing ones, in pre- and/or post-processing modes.

III. TSP-oriented Non-intrusive Runtime Verification

Runtime verification (RV) obtains and analyses data from the execution of a system to detect and possibly react to behaviours, either satisfying or violating a given specification. The classical approach to RV implies the instrumentation of system components. Small components, which are not part of the functional system, acting as *observers*, are added to monitor and assess the state of the system in runtime.

The usage of reconfigurable logic supporting versatile platform designs (e.g., soft-processors) enables innovative approaches to RV [10]. In particular, in the context of TSP systems, a design for TSP-oriented observers was proposed [11]. The *AIR Observer* (AO) features: *non-intrusiveness*, meaning system operation is not adversely affected; *flexibility*, meaning code instrumentation with RV probes is not required, although it may be used; *configurability*, being able to accommodate a set of different system-level, application-related and even mission-specific event observations.

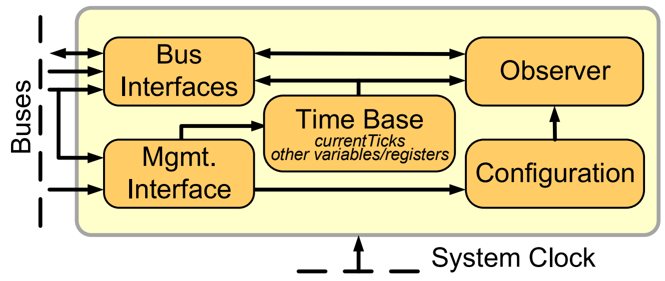


Fig. 4. AIR Observer architecture

The AO is plugged to the platform where the AIR software components execute, and comprises the hardware modules depicted in Figure 4: Bus Interfaces, capturing all physical bus activity, such as bus transfers or interrupts; Management Interface, enabling AO configuration; Configuration, storing the patterns of the events to be detected; Observer, detecting events of interest based on the registered configurations.

A **robust time base**[1] accounts for, in the AO hardware (Figure 4), the number of system-level clock ticks elapsed so far, to which AIR components have access, through the read only $currentTicks$ variable/register. For optimization purposes, other relevant read/write variables/registers may be available from the AO.

[1]The design and engineering of AIR robust timers is out of the scope of this paper. It will be addressed in future work.

The AO continuously monitors the timeliness of AIR components and applications, functionally assuming a dual role: it detects when a given temporal bound is reached and/or if a given deadline was violated; it signals that it is time to perform a given (check) action, in order to verify/enforce timeliness.

IV. Mission's Timeliness Adaptation

The adaptation to changing environmental or operating conditions is crucial for unmanned space and aerial missions survivability, which can be significantly improved through software reconfigurability, as reported in [5].

The design of AIR integrates special-purpose mechanisms to address specific adaptation requirements, as thoroughly described in [6]. Aiming to improve its time domain behaviour, the *mode-based schedules* mechanism is reviewed.

A. Mode-based schedules

The original ARINC 653 notion of a single fixed PST [1], defined offline, hinder adaptation to changes in application requirements, according to the mission's phase, given certain functions may be required to execute only during some phases.

To address this primary limitation, AIR uses the notion of *mode-based partition schedules* [4], [6], inspired by the optional service defined within the scope of ARINC 653 Part 2 specification [12].

B. AIR mode-based schedules: original design and limitations

Instead of using one fixed PST, AIR-based systems can be configured with multiple PSTs, which may differ in terms of the MTF duration, of which partitions are scheduled, and of how much processor time is assigned to them, as shown in Figure 2. The system can switch between different PSTs; selection of the active PST is performed through a service call issued by an authorized and/or dedicated partition.

In the original definition of AIR *mode-based schedules*, a PST switch request is only effectively granted at the end of the ongoing MTF. This simple approach ensures that every process in all partitions have executed completely, upon a PST switch. Thus: applications are in a coherent state; PST switching is in conformity with the specified application timing. This model is adequate for long-term stable adaptation, such as entering a different mission's phase.

C. Redesigning AIR mode-based scheduling

During the ongoing MTF, a response to sudden and unexpected events (such as, a warning of an imminent collision) may be adversely delayed by the execution of functions, defined in the active PST, which do not have the capability of reacting to those (critical) events.

To extend the number/duration of periods where the executing functions have the capability of responding to critical events, a different schedule is required. The selection/activation of a new schedule is enhanced in two ways:

- by design, the new schedule is activated as soon as no critical activity is executing;
- by PST definition and configuration, the new schedule assigns execution time windows only to critical activities.

The first condition implies that, each partition needs to be classified as having its execution as critical or non-critical and that the time boundaries delimiting the execution of the critical execution periods need to be registered in the AO, both for monitoring purposes and to avoid ill-timed mode changes.

Secondly, for each mission phase, three schedules should be provided, each corresponding to a mode (see Figure 5), as follows:

- **normal** - corresponding to the normal execution of the activities defined for the mission;
- **survival** - meaning some severe external/internal condition that puts the vehicle and/or the mission in risk has been detected. This state is entered in response to the issuing of a SET_MODE_SCHEDULE primitive (Table I), by some system/application event handler (Figure 3). The schedule for this phase/mode shall allocate processor time only to fundamental avionic functions, in order to ensure safe and secure operation.
- **recovery** - the operation of the vehicle is no longer in risk, as confirmed, at all levels, by the RV mechanisms. A relevant system/application component issues a further SET_MODE_SCHEDULE primitive. Processing could now include full FDIR activities that, once accomplished, may allow the return to the normal mode.

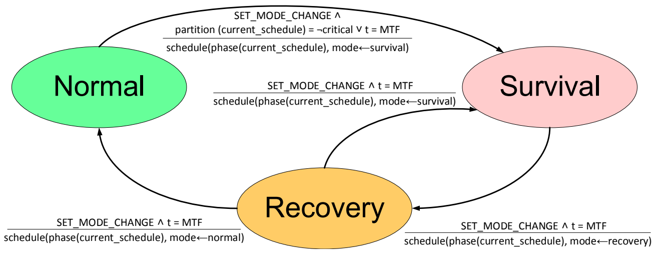

SET_MODE_CHANGE ∧ partition (current_schedule) = ¬critical ∨ t = MTF
schedule(phase(current_schedule), mode←survival)
Normal
Survival
SET_MODE_CHANGE ∧ t = MTF
schedule(phase(current_schedule), mode←survival)
Recovery
SET_MODE_CHANGE ∧ t = MTF
schedule(phase(current_schedule), mode←normal)
SET_MODE_CHANGE ∧ t = MTF
schedule(phase(current_schedule), mode←recovery)

Fig. 5. Function schedule modes and allowed transitions

Hosting multiple PSTs aboard autonomous vehicles opens room for the (self-)adaptability of unmanned missions, in function of passage of time and of changing environmental and operational conditions. The use of full-fledged mode-based schedules contributes to a timely response to sudden changes in the operational conditions.

Pre-generation of different partition schedules can be aided by a tool that applies rules and formulas to the temporal requirements of processes/partitions, taking into account the functions' needs in different anticipated conditions [4], [13]. Unforeseeable conditions can be handled thorough the mechanisms for remote update of onboard software and PSTs [8].

V. Implementing Mission's Timeliness Adaptation

The proposed integration of non-intrusive RV and timeliness adaptation features follows the hardware-assisted approach described in [11]. This hardware/software co-design allows to maintain some degree of AIR architectural flexibility with advantages in terms of improved safety and timeliness, being specially interesting for running AIR in platforms integrating processor cores (e.g., dual-core ARM) and FPGA logic [14].

Algorithm 1 AIR Partition Scheduler with runtime verification featuring adaptation through mode-based schedules

```
1: ▷ Entered upon exception: partition preemption point signalled by the AO
2: ▷ Runtime verification actions
3: if $(mode(currentSchedule) = mode(nextSchedule) \wedge schedules_{currentSchedule}.table_{tableIterator}.tick \neq (currentTicks - lastScheduleSwitch) \bmod schedules_{currentSchedule}.mtf) \vee (mode(currentSchedule) \neq mode(nextSchedule) \wedge schedules_{currentSchedule}.table_{tableIterator}.critical > (currentTicks - lastScheduleSwitch) \bmod schedules_{currentSchedule}.mtf)$ then
4:     HealthMonitor($activePartition$)
5: else ▷ Partition Scheduling Table (PST) and partition switch actions
6:     if $currentSchedule \neq nextSchedule \wedge ((mode(currentSchedule) \neq mode(nextSchedule) \wedge (currentTicks - lastScheduleSwitch) \bmod schedules_{currentSchedule}.mtf \geq schedules_{currentSchedule}.table_{tableIterator}.critical) \vee ((currentTicks - lastScheduleSwitch) \bmod schedules_{currentSchedule}.mtf = 0))$ then
7:         ▷ PST switch actions
8:         $currentSchedule \leftarrow nextSchedule$
9:         $lastScheduleSwitch \leftarrow currentTicks$
10:        $tableIterator \leftarrow 0$
11:    end if
12:    ▷ Partition switch actions
13:    $heirPartition \leftarrow schedules_{currentSchedule}.table_{tableIterator}.partition$
14:    $tableIterator \leftarrow (tableIterator + 1) \bmod schedules_{currentSchedule}.numberPartitionPreemptionPoints$
15: end if
```

A. AIR full-fledged mode-based scheduling

In a hardware-assisted approach to the implementation of AIR full-fledged mode-based scheduling, partition scheduling switch decisions from the AO hardware are complemented with software RV and partition switch actions: when a partition is dispatched, the absolute value (in POS-level clock ticks) of its *partition preemption point* is inserted in the AO configuration; when this instant is reached, an AO's hardware exception triggers the execution of Algorithm 1.

In **Algorithm 1**, if no mode change is claimed, the RV actions check (line 3), from the active PST, if the current instant is a partition preemption point. If a mode change is pending, the RV actions ensure (line 3) that no critical activity is executing at this instant. If none of these conditions apply, a severe system level error has occurred and the HM is notified (line 4) to handle the situation. Otherwise, the conditions for a **full-fledged mode-based** partition switch are checked in line 6: a PST schedule switch request is pending; a mode change switch is claimed and no critical activities are executing at the current instant or the current instant is the end of the MTF. If these conditions apply, the PST switching actions specified in [4], [11] are applied (lines 7-10) and a different PST will be used henceforth (line 8). The remaining lines of Algorithm 1 (lines 13-14) implement the conventional partition switch actions of [4], [11]. The processing resources to be assigned to the heir partition, until the next partition preemption point, are obtained from the PST in use (line 13). The AIR Partition Scheduler is set (line 14) to access the heir partition parameters.

Algorithm 2 AIR Partition Dispatcher

```
1: ▷ Entered from the AIR Partition Scheduler after partition switch actions
2: SAVECONTEXT($activePartition.context$)
3: $activePartition.lastTick \leftarrow currentTicks - 1$
4: $elapsedTicks \leftarrow currentTicks - heirPartition.lastTick$
5: ~~$activePartition \leftarrow heirPartition$~~
6: REPLACEPREEMPTIONPOINT($heirPartition.tick$)
7: REPLACECRITICALPOINT($heirPartition.critical$)
8: RESTORECONTEXT($heirPartition.context$)
9: PENDINGSCHEDULECHANGEACTION($heirPartition$)
```

B. AIR hardware-assisted partition dispatching

The partition switch actions are followed by the execution of the AIR Partition Dispatcher specified in **Algorithm 2**. The hardware-assisted optimizations of [11] are maintained with respect to the software-based approach [4]: suppression of specific elapsed clock ticks setting, which are not required because the partition dispatcher is always invoked after a partition switch; insertion of the next partition preemption point in the AO configuration (line 6). However, to allow the runtime verification of mode change requests, the value of the next time critical schedule bound is now also inserted in the AO. The remaining actions in Algorithm 2 are related to saving and restoring the execution context (lines 2 and 8) and evaluation of the elapsed clock ticks (line 4). Line 9 enforces the execution of pending actions the first time the partition is executed after a PST change [4]. This last point is specially sensitive, since abrupt mode changes may leave some partitions in an inconsistent state.

C. Extending the APEX interface

The implementation of AIR full-fledged mode-based scheduling implies the addition of new primitives to the APEX interface, summarized in Table I. The AIR PAL component provides the adequate encapsulation with respect to the registering of the schedule timing information in the AO. The primitives listed in Table I can only be issued from an authorized and/or dedicated partition.

TABLE I
EXTENDING APEX PRIMITIVES TO SUPPORT FULL-FLEDGED MODE-BASED SCHEDULES

Primitive	Short description
Need to register/update critical execution period bounds in the AO	
SET_MODE_SCHEDULE	Requests a mode change for a new schedule Served if/when no critical activities
SET_PHASE_SCHEDULE	Requests a new mission phase schedule Served in normal mode, at the end of a MTF
No need to register/update critical execution period bounds in the AO	
GET_MODE_SCHEDULE_ID	Obtains the current schedule identifier
GET_MODE_SCHEDULE_STATUS	Obtains the current schedule status

Although semantically different APEX primitives are listed in Table I for the long-term adaptation of mission phases and for a (fast) short-term (self-)adaptation, through mode changes, both primitives share the same method (i.e., the activation of a new schedule), thus being optimal with respect to fitting the previous design and implementation of AIR components.

D. Analysis and discussion

Critical software, namely that developed to go aboard an aerial or space vehicle, goes through a strict process of verification, validation and certification. Code complexity affects the effort required for that process.

The AIR hardware-assisted approach translates to a significant reduction of AIR software code complexity. Most AIR software-based components have constant time complexity, $\mathcal{O}(1)$: accesses to multielement structures are made by index, being independent of the number and position of the elements. Nevertheless, some components exhibit a linear time complexity. That is the case associated with the *Pending Schedule Change Actions* procedure (Algorithm 2 - line 9), which in the worst case wields $\mathcal{O}(n)$, being n the number of processes in the partition.

Similar considerations apply to timing issues. However, due to the highly effective (i.e., $\mathcal{O}(1)$) implementation of the AIR software-based approach, the analysis in [11] for the *AIR Partition Scheduler* and *AIR Partition Dispatcher* components has shown only a moderate improvement in time overheads.

The expected reduction in the *mode change response delay*, with the corresponding increase in the ability of AIR-based systems to timely respond to sudden and unexpected changes in operational conditions, is heavily dependent of the structure of the active PST.

The exact value of the normalized *mode change response delay*, $\mathcal{T}_{mcd}$, in general depends on the instant, t, a *mode change primitive* is issued. That value is given by:

$$\mathcal{T}_{mcd}(\mathcal{T}) = \sum_{i=1}^{ncp} \big(H(\mathcal{T} - \mathcal{T}_{cs,i}) - H(\mathcal{T} - \mathcal{T}_{ce,i})\big) \times (\mathcal{T}_{ce,i} - \mathcal{T}_{cs,i}) \tag{1}$$

where:

$$\mathcal{T} = mod\left(\frac{t}{\mathcal{T}_{MTF}}\right) \tag{2}$$

is the current instant t normalized with the major time frame duration, $\mathcal{T}_{MTF}$, through the module function, $mod()$. $H()$ is the Heaviside function, defined as:

$$H(\mathcal{T} - \mathcal{T}_0) = \begin{cases} 0 & \mathcal{T} < \mathcal{T}_0 \\ 1 & \mathcal{T} \geq \mathcal{T}_0 \end{cases} \tag{3}$$

Furthermore, the following parameters are defined:

- ncp - the number of periods executing critical activities;
- $\mathcal{T}_{cs,i}$ - the instant, normalized by $\mathcal{T}_{MTF}$, where the execution of the critical period i starts;
- $\mathcal{T}_{ce,i}$ - the instant, normalized by $\mathcal{T}_{MTF}$, where the execution of the critical period i ends.

The results obtained for the *mode change response delay* with different types of schedules is illustrated in Figure 6. In the first case, only critical activities are scheduled for execution and therefore the mode change can only be performed by the end of the MTF. The second schedule includes an initial period of critical activities followed by a (small) period

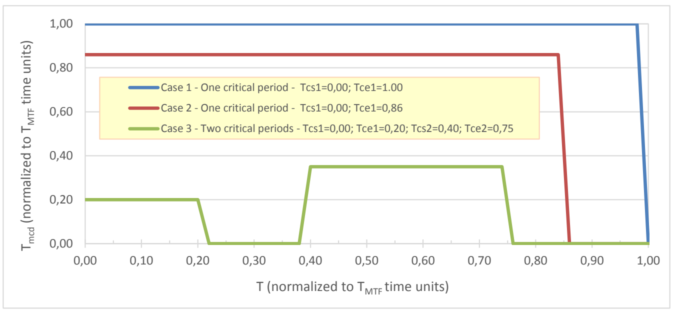


Fig. 6. Analysis of the *mode change response delay* obtained under different schedule scenarios

where mode change is allowed without delay. In the third and last schedule, illustrated in Figure 6, critical and non-critical activities are intermixed along the MTF, resulting in an overall decrease in the duration of the maximum and average periods where it is necessary to wait for a mode change to occur.

VI. Related Work

Reconfiguration and adaptation approaches have been applied in the realm of TSP systems and tested in avionic demonstrators [15]. Furthermore, non-instrusive runtime monitoring has been applied in embedded systems [16], [17] and, more specifically, in safety critical environments [18]. Configurable non-intrusive event-based frameworks for runtime monitoring have been developed within the embeddedd systems' scope [19], employing a minimally intrusive method for dynamic monitoring. Additionally, the RV concept has been applied to autonomous systems [20] and to a AUTOSAR-like RTOS, aiming the automotive domain [21]. [22] describes a runtime monitoring approach for autonomous vehicle systems requiring no code instrumentation by observing the network state. A unified framework for the specification, analysis and description of mode-change semantics applicable to real-time systems is presented in [23]. However, to the extent of our knowledge, no such techniques have been applied to TSP systems, specially if targeting avionic applications.

VII. Conclusion

This paper addressed fundamental mechanisms providing support for adaptive and self-adaptive behaviour to applications based on the AIR architecture for time- and space-partitioned systems. The usage of hybrid platforms combining processor cores and programmable logic makes advantageous the use of a hardware-assisted design complemented with some simple software-based components.

The introduction of full-fledged mode-base scheduling contributes for achieving a timely response to sudden and/or unexpected environmental and internal conditions, and enables improvements in both safety and timeliness properties. These mechanisms benefit from the use of non-intrusive runtime verification.

Non-intrusive runtime verification is a relevant contribution with respect to verification, validation and certification efforts of TSP systems that will be extended in future research. Additional work aims to take full advantage of multicore platforms in AIR, which include adaptation/reconfiguration features and, in the near future, extended RV capabilities.

References

[1] AEEC (Airlines Electronic Engineering Committee), *Avionics Application Software Standard Interface, Part 1 - Required Services*, Mar. 2006.
[2] TSP Working Group, "Avionics time and space partitioning user needs," ESA, Technical Note TEC-SW/09-247/JW, Aug. 2009.
[3] J. Rushby, "Partitioning in avionics architectures: Requirements, mechanisms and assurance," SRI International, California, USA, Tech. Rep. NASA CR-1999-209347, Jun. 1999.
[4] J. Rufino, J. Craveiro, and P. Verissimo, "Architecting robustness and timeliness in a new generation of aerospace systems," in *Architecting Dependable Systems VII*, ser. LNCS, A. Casimiro, R. de Lemos, and C. Gacek, Eds., vol. 6420. Springer, 2010.
[5] M. Tafazoli, "A study of on-orbit spacecraft failures," *Acta Astronautica*, vol. 64, no. 2-3, pp. 195–205, 2009.
[6] J. P. Craveiro and J. Rufino, "Adaptability support in time- and space-partitioned aerospace systems," in *Proc. 2nd Int. Conf. on Adaptive and Self-adaptive Systems and Applications*, Lisbon, Portugal, Nov. 2010.
[7] S. Santos, J. Rufino, T. Schoofs, C. Tatibana, and J. Windsor, "A portable ARINC 653 standard interface," in *Proc. 27th Digital Avionics Systems Conference*, St. Paul, MN, USA, Oct. 2008.
[8] J. Rosa, J. P. Craveiro, and J. Rufino, "Safe online reconfiguration of time- and space-partitioned systems," in *Proceedings 9th IEEE Int. Conf. on Industrial Informatics (INDIN 2011)*, Caparica, Portugal, Jul. 2011.
[9] J. Carraca, R. C. Pinto, J. P. Craveiro, and J. Rufino, "Information security in time- and space-partitioned architectures for aerospace systems," in *Atas 6th Simpósio de Informática (INForum 2014)*, Porto, Portugal, Sep. 2014, pp. 457–472.
[10] R. C. Pinto and J. Rufino, "Towards non-invasive run-time verification of real-time systems," in *26th Euromicro Conf. on Real-Time Systems - WIP Session*, Madrid, Spain, Jul. 2014, pp. 25–28.
[11] J. Rufino, "Towards integration of adaptability and non-intrusive runtime verification in avionic systems," *SIGBED Review*, vol. 13, no. 1, Jan. 2016, (Special Issue on 5th Embedded Operating Systems Workshop).
[12] AEEC (Airlines Electronic Engineering Committee), *Avionics Application Software Standard Interface, Part 2 - Extended Services*, Dec. 2008.
[13] J. P. Craveiro and J. Rufino, "Schedulability analysis in partitioned systems for aerospace avionics," in *Proceedings 15th IEEE International Conference on Emerging Technologies and Factory Automation*, Bilbao, Spain, Sep. 2010.
[14] *ZYBO Reference Manual*, DILIGENT, Feb. 2014.
[15] G. Durrieu, G. Fohler, G. Gala, S. Girbal, D. G. Pérez, E. Noulard, C. Pagetti, and S. Pérez, "DREAMS about reconfiguration and adaptation in avionics," in *Proc. 8th Congress on Embedded and Real-Time Software and Systems (ERTS2016)*, Toulouse, France, Jan. 2016.
[16] C. Watterson and D. Heffernan, "Runtime verification and monitoring of embedded systems," *Software, IET*, vol. 1, no. 5, Oct. 2007.
[17] T. Reinbacher, M. Fugger, and J. Brauer, "Runtime verification of embedded real-time systems," *Formal Methods in System Design*, vol. 24, no. 3, pp. 203–239, 2014.
[18] A. Kane, "Runtime monitoring for safety-critical embedded systems," Ph.D. dissertation, Carnegie Mellon University, USA, Feb. 2015.
[19] J. C. Lee and R. Lysecky, "System-level observation framework for non-intrusive runtime monitoring of embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 20, no. 42, 2015.
[20] G. Callow, G. Watson, and R. Kalawsky, "System modelling for run-time verification and validation of autonomous systems," in *Proc. 5th Int. Conference on System of Systems Engineering*, Loughborough, UK, Jun. 2010, pp. 1–7.
[21] S. Cotard, S. Faucou, J.-L. Bechennec, A. Queudet, and Y. Trinquet, "A data flow monitoring service based on runtime verification for AUTOSAR," in *Proceedings of the 14th Int. Conf. on High Performance Computing and Communications*. Liverpol, UK: IEEE, Jun. 2012.
[22] A. Kane, O. Chowdhury, A. Datta, and P. Koopman, "A case study on runtime monitoring of an autonomous research vehicle (ARV) system," in *Proc. 15th Int. Conf. on Runtime Verification*, Vienna, Austria, Sep. 2015, pp. 102–117.
[23] L. T. X. Phan, I. Lee, and O. Sokolsky, "A semantic framework for mode change protocols," in *Proceedings of the 17th Real-Time and Embedded Technology and Applications Symposium*. IEEE, Apr. 2011.

Effective Source Code Analysis with Minimization

Geet Tapan Telang
Hitachi India Pvt. Ltd.
Research Engineer
Bangalore, India
Email: geet@hitachi.co.in

Kotaro Hashimoto
Hitachi India Pvt. Ltd.
Software Engineer
Bangalore, India
Email: kotaro.hashimoto.jv@hitachi.com

Krishnaji Desai
Hitachi India Pvt. Ltd.
Researcher
Bangalore, India
Email: krishnaji@hitachi.co.in

***Abstract*—During embedded software development using open source, there exists substantial amount of code that is ineffective which reduces the debugging efficiency, readability for human inspection and increase in search space for analysis. In domains like real-time embedded system and mission critical systems, this may result in inefficiency and inconsistencies affecting lower quality of service, enhanced readability, increased verification and validation efforts. To mitigate these shortcomings, we propose the method of minimization with an easy to use tool support that leverages preprocessor directives with GCC for cutting out ifdef blocks. Case studies of Linux kernel tree, Busybox and industry-strength OSS projects are evaluated indicating average reduction in lines of code roughly around 5%-22% in base kernel using minimization technique. This results in increased efficiency for analysis, testing and human inspections that may help in assuring dependability of systems.**

I. Introduction

The outgrowth of Linux operating system and other real-time embedded systems in dependable computing has raised concern in possible areas such as mission critical system. The size of code has grown to approximately 20 million lines of code (Linux) and with this scale and complexity, it becomes impossible to follow traditional methods for meeting the safety and real-time requirements.

Since tools for analysis and testing are getting advanced, the safety and time requirements can be verified and validated by capturing evidence and justifications. Most of these tools are interested in meeting the coverage expectations in terms of code, execution times, resources, throughput and fault tolerance that define the overall dependability of systems.

Code coverage with different analysis and test tools becomes the major part of verification and validation, in order to perform this effectively, we propose method of minimization devised for keeping functional safety and real-timeliness into consideration for narrowed search space verification, false positive reduction, easier human inspection and shorter verification time. The term minimization as shown in figure 1 signifies removal of unused piece of code comprising of `#ifdef` and `#if` blocks. The target code along with configuration file when executed using minimization process produces compilable code without `#ifdef` and `#if` block and other unused lines of code, which is different from execution with GCC preprocessor where compiled code comprises of `#ifdefs`.

The evaluation is exercised on targets such as Linux Kernel source tree, BusyBox [1] tree and similar quantification of other OSS projects.

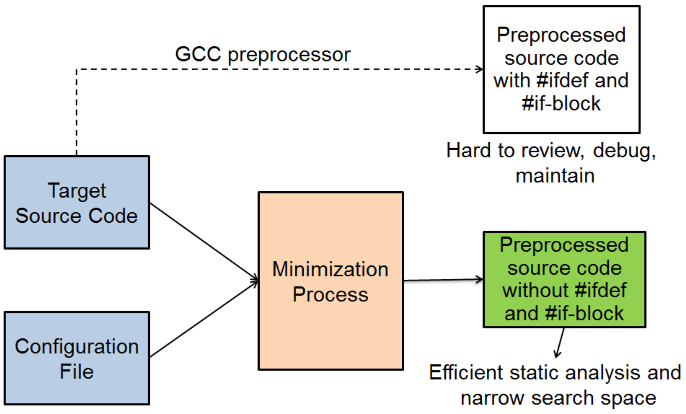

GCC preprocessor
Preprocessed source code with #ifdef and #if-block
Hard to review, debug, maintain
Target Source Code
Configuration File
Minimization Process
Preprocessed source code without #ifdef and #if-block
Efficient static analysis and narrow search space

Fig. 1: Overview of minimization.

II. Background

In OSS software domain, there are many developers contributing towards the common goal such as real-time, safety-mitigation etc. because of which, the source code developed lacks strict guidelines. Although, there are checks made with semantic patches [2] and other utilities before the source code is committed, no clear coding guidelines are followed that will make the source code easy to inspect and analyze.

Primary problem with the OSS code in embedded domain is the usage of pre-processor directives and conditional code compilations that are used due to varying configuration options. In case of Linux kernel alone, there are more than 10,000 different configuration flags that have to enabled/disabled and thereby used as part of the pre-processor directive in the source code [3].

The configurations of OSS code is easy to enumerate and apply depending on the configuration flags and configure command. However, the source code is still having all of conditional compilation code with pre-processor directives. Too much of conditional compilation code based on configuration, is difficult to inspect and analyze for different static analysis tools. As the configuration options increase, the usage of same in code also increases significantly resulting in analysis complexity.

To solve this problem, there are few tools that are built with pre-processor directives awareness so that during the source code analysis these tools read system configuration and directives to selectively analyze relevant code and skip the disabled code automatically. Some of these tools are GNU cflow, Coccinelle etc. Problem here is that dead code elimination is not the main purpose of these tools. Pre-processor handling is best done with its corresponding compiler in place. Standalone tools cannot do a good job with this as they do not have required constructs configured for effective application of conditional compilation. Hence, a standardized method needs to be made available that can suffice minimization agnostic to other static analysis tools and human inspection. However, the proposed method needs to leverage compiler technology that can best apply the pre-processor directives.

For this purpose, the GCC [4] compiler based pre-processor is selected for realizing the minimization technique. GCC is the choice due to its immense usage in OSS community including Linux, Busybox etc.

A. Linux kernel configuration

Configuration plays a very important role in building any software. In case of OSS, it becomes an essential pre-requisite as the software is developed by different developers with multiple configurations concurrently. To illustrate the same, Linux kernel is a classic example for showing the varied configuration it supports.
The Linux kernel configurations are available in `arch/*/configs`. To alter configuration, integrated options such as `make config`, `make menuconfig` and `make xconfig` are available. Once configuration is done, it gets saved in `.config` file. The indication available in `.config` file has option `=y` illustrating driver is built into the kernel, `=m` for built as a module or it is not selected [5]. The `.config` file appears as below:

```
#General setup
CONFIG_INIT_ENV_ARG_LIMIT=32
CONFIG_CROSS_COMPILE=""
# CONFIG_COMPILE_TEST is not set
CONFIG_LOCALVERSION=""
# CONFIG_LOCALVERSION_AUTO is not set
CONFIG_HAVE_KERNEL_GZIP=y
```

B. GCC preprocessor

GCC [4] is the compiler choice that is used from utils to operating system level and rigorously tested for several years using tools such as CSMITH [6]. Hence, configuration based pre-processor is best applied with GCC and is choice for realizing our methodology.
The GCC preprocessor implements macro language that is utilized to change C programs before they are compiled. The output is similar to input however, the preprocessor directive lines are replaced with blank lines and spaces are appended instead of comments based on the configuration. Certain time some directives may be duplicated in output of the preprocessor, majority of these are `#define` and `#undef` that contains certain debugging options [7].

III. Related work

The proposed minimization methodology improves static analysis efficiency and easier code inspection. As per prior work regarding source code stripping [8], the GCC options are used to tweak pre-processor directives such that conditional compilation code is stripped as per enabled configurations. This helps in generating .c code that has conditional directives applied to remove the redundant code. Limitation here is that, the approach is not generalized for complete source code tree.

One alternative can be Cflow [9], [10] a GNU based tool, which can preprocess input files before analyzing them and it is integrated with pre-processing option itself, however it renders difficulty because all the required preprocess options needs to be copied and pasted for execution of Cflow leading to incorrectness.

Other alternative can have GCC compilation log and then tweak the log options for each file with required pre-processor options using GREP. Based on which the required .c and .h files with the stripped code based on the pre-processor options can be generated. In subsequent sections, an approach is proposed with Makefile integration and subsequent post processing for easier code inspection and narrowed down search space.

IV. Minimization Approach

A. Minimization Process

Minimization approach emphasizes on a collection of processes which tweaks integrated MakeFile options to produce compilable minimized code.

1) Definition: The term minimization signifies an efficient way to get a set of stripped source code, where all the code which is not required according to `.config` file is left out. Often it is observed that it becomes hard to debug, maintain and verify the code because of macros and preprocessor directives that are expanded during compilation through GCC. This difficulty is subdued with our approach where target source code is free from selected preprocessor options and macros expansion, thereby reducing source code. The approach that has been used for minimization of source code follows the use of GREP command. This filters GCC (used compiler commands) and generates the source tree consisting of limited or useful internal components which are available in shortlisted configuration `.config` file as dedicated output.

The GCC compiler: In general scenario the source code modules are compiled with certain GCC options to make them work [11]. Typical GCC option looks like given snippet of kernel modules:

```
gcc -Wp,-MD,
arch/x86/tools/.relocs_32.o.d
-Wall -Wmissing-prototypes
-Wstrict-prototypes -O2
-fomit-frame-pointer -std=gnu89
-I./tools/include -c -o
arch/x86/tools/relocs_32.o
arch/x86/tools/relocs_32.c
```

For reduction of unused code the above illustrated options are further added with `-E -fdirectives-only` to produce human readable output for easier review, debug, maintain and verify.

2) Problem: The major difficulties with GREP based approach is illustrated below:

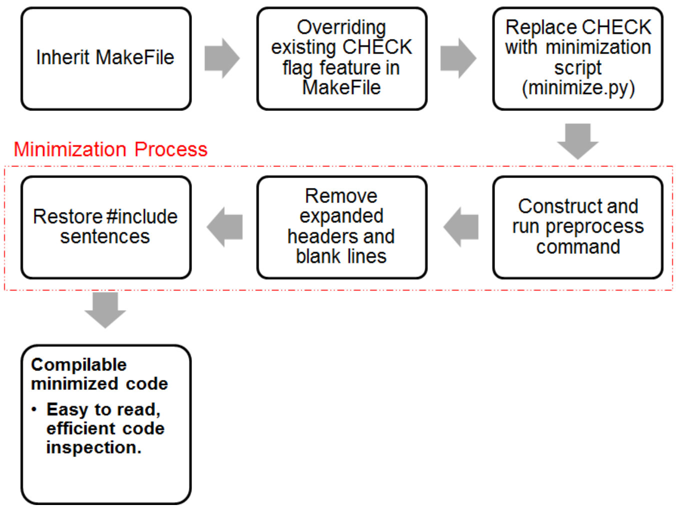
- It requires a complete build in advance to obtain full set of used GCC commands written in build log.
- The text parsing (grep and gcc commands) is required and has to be acquired from the build log.
- Finally source code needs to be modified to remove `#include` lines.

However with the help of minimization approach code reduction can be achieved by executing `minimize.py` script which requires no pre-build, no build log parsing and no code modification.

The minimization approach is implemented using python script with a stripping technique [12] as below:

- Elimination of configuration conditionals such as `#ifdef #if #endif`.
- Preservation of `#define` macros.
- Preservation of `#include` sentences.

The stripping is initiated and exercised by initially focusing on Linux kernel source code through tweaking the GCC preprocessor options for complete kernel source tree.



Inherit MakeFile
Overriding existing CHECK flag feature in MakeFile
Replace CHECK with minimization script (minimize.py)
Minimization Process
Construct and run preprocess command
Remove expanded headers and blank lines
Restore #include sentences
Compilable minimized code
• Easy to read, efficient code inspection.

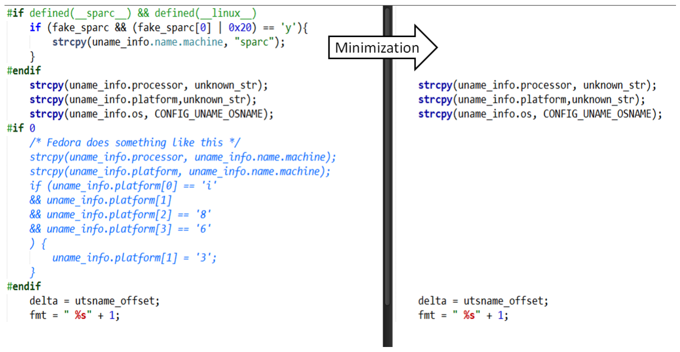
Fig. 2: Minimization technique process flow.

3) Solution: As depicted in figure 2, the MakeFile is inherited and CHECK option is tweaked, where existing CHECK feature in kernel MakeFile is replaced with `minimize.py` script, which processes the minimization on the fly with single execution pass. In `make` process, `minimize.py` receives options that are completely similar as the compile flag of each source file along with `$CHECKFLAGS` variable. Below snippet shows on the fly approach:

```
$ make C=1 CHECK=minimize.py
CF="-mindir ../minimized-tree/"
```

The pre-process tweaks the source files with the gcc options `gcc -E -fdirectives-only`. This command allows removal of `#ifdef`, followed by expansion of `#include` but preserving `#define` macros.

The `preprocess()` function available in minimization technique, takes gcc options that are passed via `Makefile` as inputs, which then appends `gcc -E -fdirectives-only` flags and performs preprocess for target C files.
Next is identification and deletion of the expanded header contents that is present in used compiler commands once `make` command is executed. To remove header content, line-markers are used as clues that exists in the preprocessed file of kernel source. For example: `#30 "/usr/include/sys/stsname.h" 2`.
The `stripHeaders()` function in minimization script acquires the preprocessed C file and then search for preprocessor output which is relevant to `#include` lines and is accompanied by deletion of `#include` contents guided by line-markers. `#include` content file name and line number information is conveyed in preprocessor output, for example: In following syntax `#30 "/usr/include/sys/stsname.h" 2`, 30 signifies that this line originates in line 30 of file `utsname.h` after having included in another file which is signified by flag 2. The flag which in this example is indicated by 2 represents returning to the file. However flag 1 signifies start of the file.
This `stripHeader()` algorithm finds the line-markers that starts with `# number` file name and if file name is the target C file then it copies the line and searches for flag. If flag in the line marker is 2 the algorithm marks it `"TO BE REPLACED"` which illustrates "there is `#include` line". Finally the `#include` sentences are restored from the original source code by copying relevant `#include` lines.
The `restoreHeaderInclude()` function in minimization technique carry out header-stripped preprocessed files and searches for `"TO BE REPLACED"` mark, followed by comparing with the original C file and copy original `#include` lines. Once the above steps are accomplished the diff result is only deletion of the unused code without changing `#include` and `#define` lines.



```
#if defined(__sparc__) && defined(__linux__)
    if (fake_sparc && (fake_sparc[0] | 0x20) == 'y'){
        strcpy(uname_info.name.machine, "sparc");
    }
#endif
    strcpy(uname_info.processor, unknown_str);
    strcpy(uname_info.platform,unknown_str);
    strcpy(uname_info.os, CONFIG_UNAME_OSNAME);
#if 0
    /* Fedora does something like this */
    strcpy(uname_info.processor, uname_info.name.machine);
    strcpy(uname_info.platform, uname_info.name.machine);
    if (uname_info.platform[0] == 'i'
    && uname_info.platform[1]
    && uname_info.platform[2] == '8'
    && uname_info.platform[3] == '6'
    ) {
        uname_info.platform[1] = '3';
    }
#endif
    delta = utsname_offset;
    fmt = " %s" + 1;
```

```
    strcpy(uname_info.processor, unknown_str);
    strcpy(uname_info.platform,unknown_str);
    strcpy(uname_info.os, CONFIG_UNAME_OSNAME);

    delta = utsname_offset;
    fmt = " %s" + 1;
```

Fig. 3: Code reduction through minimization technique.

Figure 3 illustrates the minimized code after the `make` process where the `#ifdef` and `#if` blocks are removed. The minimization script `minimize.py` does not support minimization of include files. The main motive behind this exclusion is that, a single include file is referred from multiple C files and resulting minimized include file is not identical for all C files referring the include file. Consequently, if compile option for each C file differ, effective definitions at compile time shall differ too and this differentiate `#ifdef` blocks in the included file.

B. *Minimization methodology*

1) Prerequisites: The script which is developed to exercise minimization approach requires following commands executing in the host machine:

- `diffstat`
- `diff`
- `echo`
- `file`
- `gcc` (Other options that are required to build Linux Kernel or BusyBox)
- `python` (2.x and 3.x compatibility is supported by minimization technique)

2) Usage: Proposed minimization technique needs following points for execution of script:

1) Navigate to source directory. Example:
   `$ cd linux-4.4.9`
2) Copy `minimize.py` to kernel directory.
3) Prepare configuration file by tuning the `.config` file and storing it in kernel tree directory. The `.config` can also be generated by executing `make` command. Example:
   `$ make allnoconfig`
4) Add the script directory path. For example:
   `` $ export PATH=$PATH:`pwd` ``
5) Execute `make` with the following `CHECK` options:
   ```
   $ make C=1 CHECK=minimize.py
   CF="-mindir ../minimized-tree/"
   ```
   Parameter value `C=1` signifies minimization only for (re)compilation target files. `C=2` is used to perform minimization for all the source files regardless of whether they are compilation target or not. Similarly to specify output directory `-mindir` option in `CF` flag is used.

On other hand minimization is also applicable for sub target sources. For example: `$ make drivers C=1 CHECK=minimize.py CF="-mindir ../minimized-tree/"`.
In addition, the script has been modified in such a way that on successful execution, compilation and minimization will be performed at the same time and minimized source tree will be generated under directory `../minimized-tree/`. One thing that needs to be known is that only the target C source files will be minimized. The other file contents(included header etc) remain as they are.

V. Results

Minimization technique [12] has been experimented and evaluated on platforms such as Linux kernel and BusyBox Tree. The experiment is basically conducted to check reduction metrics after executing minimization technique on original code base.

The evaluation of minimization technique has been implemented on hardware specifications: Processor: 3600MHz, width-64bits, cores-8. Memory: size-7891MiB. Architecture: x86_64.

It has been performed by comparing different configurations of target source, particularly `"allnoconfig"` and `"defconfig"`. Main motivation for using different configuration is to comply minimization with expectations, as follows:

- In case of `"allnoconfig"` most features are disabled. This signifies substantial amount of disabled `#ifdef` causing large amount of code reduction. Eventually, it leads to higher minimization ratio.
- Similarly, in case of `"defconfig"`, only a part of features are disabled which leads to less number of disabled `#ifdef` resulting in less amount of code reduction. Hence in case of `"defconfig"` reduction is expected to be lower than `"allnoconfig"`.

A. *Linux Kernel*

Implementation on Linux kernel with `"allnoconfig"` and `"defconfig"` option results in substantial reduction of unnecessary code has been achieved as shown in figure 4. The metrics are as follows:

- `allnoconfig`: 64684 unused lines were removed from kernel source which constitutes around 22% of original C code in kernel source.
- `defconfig`: With this option 103144 unused lines were removed from kernel source that comprises about 5% of original C code.

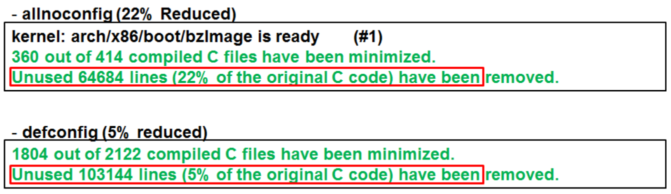

```
- allnoconfig (22% Reduced)
kernel: arch/x86/boot/bzImage is ready    (#1)
360 out of 414 compiled C files have been minimized.
Unused 64684 lines (22% of the original C code) have been removed.

- defconfig (5% reduced)
1804 out of 2122 compiled C files have been minimized.
Unused 103144 lines (5% of the original C code) have been removed.
```

Fig. 4: Minimization technique execution on Linux Kernel.

The minimization script `minimize.py` executes not only for limited configurations, but also other customized ones including PREEMPT_RT patch.

B. *BusyBox Tree*

On executing minimization technique in BusyBox tree having `"allnoconfig"` and `"defconfig"` configuration options, the reduction metrics obtained are as follows:

- `allnoconfig`: 51 out of 112 compiled C files have been minimized. 5945 lines (34% of original C code) unused lines were removed.
- `defconfig`: 296 out of 505 compiled C files have been minimized. 20453 lines (11% of original C code) unused lines were removed.

C. *Quantification of other OSS projects*

Apart from Linux Kernel and BusyBox tree, quantification of `#ifdef` and `#if-blocks` that could potentially be removed from open-source project ARCTIC Core source code [13] as compared to Linux Kernel has been exercised.
The motive is to quantify how much beneficial can Minimization approach be for OSS projects such as ARCTIC Core. The quantification is carried out by finding total number of `#ifdef` and `#if-block` and calculating the ratio with total lines of code as below:

Complexity Metrics	Linux Kernel			BusyBox Tree			PREEMPT_RT	
	Original Source	Minimized(x86_defconfig)	Minimized(allnoconfig)	Original Source	Minimized(x86_defconfig)	Minimized(allnoconfig)	Original	Minimized
Average Line Score	23	7	5	22	21	19	10	7
50%-ile score	4	3	2	9	9	5	4	3
Highest Score	1846	194	158	283	283	283	530	194

TABLE I: Complexity metrics in original and minimized targets.

```
Total number of lines in all C files
of Arctic Core source code = 407994 lines.
Total number of #ifdef existing = 12744.
Number of lines that can
be reduced = 12744/407994*100 = 3.12%
```

Similarly, in Linux Kernel,

```
Total number of lines in all
C files = 15086494 lines.
Total number of #ifdef existing = 85728.
Number of lines that can be reduced =
85728/15086494*100 = 0.568%
```

The statistics above indicates that there are more (approximately 5.5 times higher) chances in Arctic Core of eliminating unused `#ifdef` switches. This can be stated as a possible advantage of Minimization technique, however port implementation is yet to be realised.

VI. Evaluation

A. Complexity statistics

To analyze the complexity of "C" program function, Linux with PREEMPT_RT patch, Linux Kernel source and BusyBox tree has been evaluated by comparing complexities of C program functions of minimized and original source code of these targets respectively. The statistics have been acquired using "Complexity" tool [14].
The complexity tool has been used because it helps extensively in getting an idea of how much effort may be required to understand and maintain the code. Higher the score, more complex is the procedure, and minimization shows comparably lower complexity score which signifies it is easy to read and maintain [14].

Table I illustrates the measured complexities of original and minimized targets (Linux kernel, BusyBox tree and PREEMPT_RT Kernel) respectively. For Linux kernel and BusyBox `allnoconfig` and `x86_defconfig` configurations has been evaluated for minimized code. The minimized code demonstrate decreased complexity in terms of average line score, 50%-ile score and highest score in all three targets.

B. Verification for the minimized built binary

The disassembled code ("objdump -d") matches the binaries that are built from minimized and original source code. Also the configuration and target has been confirmed based on Busybox and Linux kernel as below:

- BusyBox-1.24.1: Checked configuration options include `defconfig` and `allnoconfig`.
- Linux kernel-4.4.1: Configuration options verified `allnoconfig`.

VII. Benefits

A. Verification time and cost improvement

For verification time improvement static analysis has been implemented by comparing results of original and minimized kernel source tree using Coccinelle which is a program matching and transformation engine for C code and has many semantic patches to the new submissions to the mainline kernel repository [15], [16]. The verification has been implemented by executing a semantic patch [2] which detects functions whose declared return value type and actually returned type differs by scanning source files (*.c and *.h) that are referred from `init/main.c` in kernel tree. Results of the static verification in terms of time parameter are illustrated below:
Average spatch execution time:

```
Original Kernel Source: 12.37[s]
Minimized Kernel Source: 2.24[s]
```

The minimized technique provide around 5.5 times faster analysis as compared to original kernel source tree.

B. False Positive reduction

False positive is a test result which wrongly indicates that a particular condition or attribute is present. To mitigate such situation static analysis [15] was conducted on original and minimized kernel source tree. The number of meaningless detection were mitigated as follows in the minimized kernel source. Number of detection using Coccinelle:

```
Original Kernel Source: 126
Minimized Kernel Source: 82
```

C. Easy Code Inspection

The minimization technique generates easy to read source code by implementing following assimilation:

- Unused `#ifdef`, `#if` blocks are removed.
- `#include` and `#define` lines are preserved.
- Producing same binary file as that of original source tree.

D. Pruning function call graph:

During analysis, it is required to identify every possible call path to establish and trace relationship between program and subroutines, callgraph is a directed graph that represents this relationship [17]. The call graph displays every function call regardless of `#ifdef` switches which results in substantially complex graph which is difficult to trace. With minimization technique, call graph display illustrates only used function calls

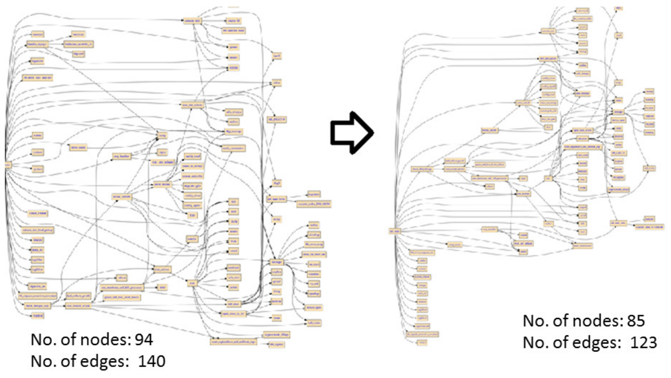


Fig. 5: Call graph for Linux kernel before (left) and after (right) minimization.

thereby providing minimized search space. Figure 5 illustrates call graph transformation before and after minimization.

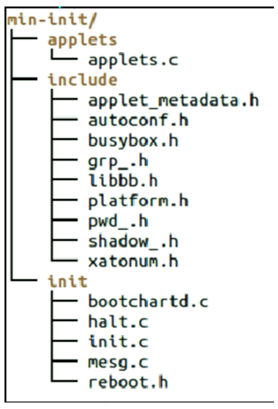
With minimization the number of nodes reduced from 94 to 85 followed by edges which are from 140 to 123 hence a narrow search space.

E. Extracting minimal subtarget sources:

To easily identify which files are used in source tree for efficient software walk-through, subtarget can be specified in the minimized command in result of which minimization will extract only the used source files. The following snippet shows addition of subtarget *init* in minimized command:

```
$ make init C=2 CHECK=minimize.py
CF="-mindir ../min-init"
```

This results in extraction of only used source files when subtarget is defined and is shown in figure 6.



```
min-init/
├── applets
│   └── applets.c
├── include
│   ├── applet_metadata.h
│   ├── autoconf.h
│   ├── busybox.h
│   ├── grp_.h
│   ├── libbb.h
│   ├── platform.h
│   ├── pwd_.h
│   ├── shadow_.h
│   └── xatonum.h
└── init
    ├── bootchartd.c
    ├── halt.c
    ├── init.c
    ├── mesg.c
    └── reboot.h
```

Fig. 6: Depended *.c files of Linux kernel in minimized form. Actually included *.h files.

VIII. Conclusion

The minimization technique helps substantially in improving the readability of source code which results in efficient code review and inspection. It helps in narrowing down search space by giving evidence for unused code. The evaluation of this technique has been performed on target platform such as Linux Kernel, BusyBox Tree and PREEMPT_RT Kernel. Minimization reduction of approximately 5% is achieved across the PREEMPT_RT Linux kernel, Linux kernel and the Busybox software. From analysis stand-point, this provide essential benefits such as reduction in verification time (spatch execution) from 12.37[s] in original kernel source to 2.24[s] in minimized kernel, false positive reduction where the number of detection relating to bugs using Coccinelle (static analysis) reduces from 126 to 82.
This helps in application domains such as automotive, railways, industry etc. The future work for minimization technique includes extension to other compilers such as LLVM [18] followed by adaption with architecture such as ARM; various build system e.g. CMake, automake. Binary equivalence is checked, however formal equivalence between the original and minimized source code tree is still a future work. The source code is available at GitHub [12].

References

[1] E. Andersen. Busybox. [Online]. Available: https://www.busybox.net

[2] W. Sang. Evolutionary development of a semantic patch using coccinelle. [Online]. Available: http://lwn.net/Articles/380835/

[3] S. Zhou, J. Al-Kofahi, T. N. Nguyen, C. Kästner, and S. Nadi, "Extracting configuration knowledge from build files with symbolic analysis," in *Proceedings of the Third International Workshop on Release Engineering*, ser. RELENG '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 20–23. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820690.2820700

[4] GCC-Team. Gcc, the gnu compiler collection. Free Software Foundation, Inc. [Online]. Available: https://gcc.gnu.org/

[5] D. Gilbert. (2003, August) The linux 2.4 scsi subsystem howto. Linux Document Project. [Online]. Available: http://www.tldp.org/HOWTO/SCSI-2.4-HOWTO/kconfig.html

[6] Y. Xuejun, C. Yang, E. Eric, and R. John. Csmith. [Online]. Available: https://embed.cs.utah.edu/csmith/

[7] GNU-Manual, *The C preprocessor*, GNU. [Online]. Available: https://gcc.gnu.org/onlinedocs/cpp/index.html

[8] StackOverflow. Strip linux kernel sources according to .config. [Online]. Available: http://stackoverflow.com/questions/7353640/strip-linux-kernel-sources-according-to-config

[9] S. Poznyakoff. Gnu cflow. GNU. [Online]. Available: http://www.gnu.org/software/cflow/

[10] A. Younis, Y. K. Malaiya, and I. Ray, "Assessing vulnerability exploitability risk using software properties," *Software Quality Journal*, vol. 24, no. 1, pp. 159–202, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11219-015-9274-6

[11] P. J. Salzman. The linux kernel module programming guide. [Online]. Available: http://www.tldp.org/LDP/lkmpg/2.4/html/x208.html

[12] K. Hashimoto, *The Minimization script*. [Online]. Available: https://github.com/Hitachi-India-Pvt-Ltd-RD/minimization

[13] ArcticCore. Arctic core autosar 3.1 repositories. [Online]. Available: www.arccore.com/resources/repositories

[14] B. Korb. Measure complexity of c source. [Online]. Available: https://www.gnu.org/software/complexity/manual/complexity.html

[15] G. Muller. (2015, October) Coccinelle. INRIA; LIP6; IRILL. [Online]. Available: http://coccinelle.lip6.fr/documentation.php

[16] V. Nossum. Impact on the linux kernel. [Online]. Available: http://coccinelle.lip6.fr/impact_linux.php

[17] G. Kaszuba. Python call graph. [Online]. Available: http://pycallgraph.slowchop.com/en/master/guide/intro.html

[18] The llvm compiler infrastructure. [Online]. Available: http://www.llvm.org

Towards Real-Time Operating Systems for Heterogeneous Reconfigurable Platforms

Marco Pagani, Mauro Marinoni, Alessandro Biondi, Alessio Balsini, Giorgio Buttazzo
Scuola Superiore Sant'Anna, Pisa, Italy
Email: {name.surname}@sssup.it

Abstract—**Heterogeneous platforms equipped with processors and field programmable gate arrays (FPGA) can be exploited to accelerate specific functions triggered by software activities. Thanks to dynamic partial reconfiguration (DPR) capabilities of modern FPGAs, such functions can be programmed at run-time, thus opening a new dimension in the resource management problems for such platforms. To properly exploit the DPR feature, novel operating system supports are needed. With the aim of investigating this direction, we developed a prototype implementation of a timesharing mechanism that can be used to dynamically reconfigure predefined FPGA areas for accelerating different functions associated with real-time recurrent tasks.**

This work reports some preliminary experimental studies conducted to evaluate the feasibility of the proposed approach, profile the temporal parameters involved in such systems (e.g., reconfiguration and execution times) and identify possible bottlenecks. The achieved results are encouraging and clearly show that, in spite of the relatively high reconfiguration times of FPGAs, a timesharing mechanism can significantly improve the performance of real-time applications with respect to fully static approaches.

I. Introduction

Modern computing architectures integrate heterogeneous components, like different types of processors and field programmable gate array (FPGA) modules that can be exploited to accelerate specific functions to improve the application performance. FPGAs with dynamic partial reconfiguration (DPR) capabilities allow the user to reconfigure a portion of the FPGA at runtime, while the rest of the device continues to operate [1]. This is especially valuable in mission-critical systems that cannot be disrupted while some subsystems are being redefined [2].

Such a DPR feature opens a new scheduling dimension for systems running on such heterogeneous platforms, giving the possibility of virtualizing the FPGA, using timesharing techniques, so that it can be used to accelerate a number of hardware functions that is higher than that allowed by static partitioning, thus further improving the application performance.

Today, however, reconfiguration times are about three orders of magnitude higher than context switch times in multitasking, therefore FPGA virtualization can only be used for a limited set of applications. As shown in the next section, reconfiguration times significantly reduced in the recent years and are expected to further decrease in the near future. This enables the development of a new generation of operating systems that can manage the FPGA module, handling both software tasks (SW-tasks) and hardware tasks (HW-tasks) in a uniform fashion.

To investigate this issue, this paper presents a prototype implementation of a timesharing mechanism that can be used to dynamically reconfigure predefined FPGA areas for accelerating different functions associated with real-time periodic tasks. The results achieved on such a prototype are encouraging and clearly show that, in spite of the relatively high reconfiguration times, a timesharing mechanism on the FPGA can significantly improve the performance of real-time applications with respect to a fully static approach.

A. Trend of Partial Reconfiguration Performance

During a partial reconfiguration process, different hardware modules are involved, such as the memory, the bus, and the FPGA reconfiguration port. As a reconfiguration bitstream traverses such series of modules, the performance of the reconfiguration processes is limited by the slowest element, which represents the DPR bottleneck. Since the DPR feature was introduced in FPGAs, all such elements were improved during the years. In the early 2001, Xilinx developed the Virtex-II FPGA device, which was able to store data on 64x8 bits DDR memory at 294 MHz and write the configuration to the logic elements with a peripheral (denoted as Slave SelectMAP) running at 50 MHz with a data size of 8 bits. The DPR throughput of this device was measured as 60 Mbps.

Nowadays, one of the top gamma products is represented by the Xilinx Zynq Ultrascale+, compatible with DDR4 memory and able to reach a maximum transfer rate of 2400 Mbps. It is connected with the ARM AMBA AXI4 and its logic elements are configured by an evolution of the SelectMAP reconfiguration port, called ICAP, running at a maximum frequency of 200 MHz with a data size of 32 bits.

In addition to the improvements achieved on the memory and the communication bus, a performance boost from the memory storage side has also been obtained through a bitstreams compression [3], moving the actual bottleneck to the reconfiguration interface.

Estimating the throughput of the reconfiguration process is not trivial, as it requires a precise ad-hoc orchestration of each hardware module involved in the process and also requires the availability of all the hardware devices that are intended to be compared. Figure 1 shows the evolution of the FPGA reconfiguration performance during the last years, obtained by comparing the theoretical maximum throughput estimations calculated from the device's datasheets.

Since a higher throughput corresponds to smaller reconfiguration times (for a given bitstream size), the positive trend shown in Figure 1 enables a more dynamic management of the FPGA, allowing the implementation of virtualization mechanisms that can provide great advantages to real-time applications, with respect to fully static approaches.

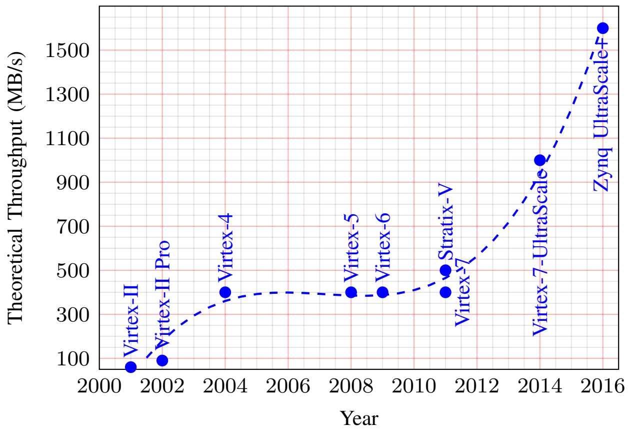


Figure 1: Reconfiguration interface throughput evolution.

II. Related Work

The reduction of reconfiguration times resulting from the FPGA technology evolution allowed exploiting the advantages of DPR for handling applications with a dynamic behavior. For example, a HW-task that could only be statically allocated in the earlier platforms, can now be reconfigured at runtime to implement mode changes in the application. More recently, some authors proposed methods for supporting a reconfiguration that can be periodically requested by SW-task at every job execution. This approach is referred to as *job-level reconfiguration*.

A few approaches have been proposed to provide an operating system support for DPR in platforms including an FPGA. The common adopted solution for exchanging data between SW-task and HW-tasks is through proper software stubs interacting with the kernel scheduler and handling the HW-tasks using a dedicated library.

For instance, Lübbers and Platzner [4] proposed the ReconOS operating system, which extends the classic multi-threading programming model to hardware activities executed on an FPGA. HW-tasks interact with SW-tasks threads trough a custom developed POSIX-style API, using the same operating system mechanisms, like semaphores, condition variables, and message queues. Originally designed for fully-reconfigurable FPGAs, this solution has then been extended by the same authors to support partial reconfiguration [5], with a cooperative multitasking approach dealing with the contentions on a set of predefined reconfiguration slots. More recently, Happe et. al. [6] extended the ReconOS execution environment to provide HW-tasks preemptability. However, the focus of this work is on hardware enabling technologies, rather than kernel support mechanisms.

Iturbe et al. [7] presented the R3TOS operating system to support dynamic task allocation on an FPGA without relying on predefined slot partitioning and static communication channels. In their solution, scheduling and allocation of HW-tasks are performed by a module, called HWuK, which is also in charge of controlling the programming interface in an exclusive manner. The authors proposed a HW-task model, as well as algorithms for scheduling and allocation. However, a worst-case analysis is not provided and nothing is said on the schedulability of SW-tasks. Such a dynamic slot partitioning increases flexibility in the FPGA allocation at the cost of a higher complexity of the reconfiguration algorithms, reflecting in higher worst-case reconfiguration times.

The major problem in such kernel extensions is that they have been designed to improve the *average* system performance, without providing tight worst-case response times bounds. As a consequence, a model of the FPGA runtime behavior based on these methods leads to huge pessimism if used for a real-time scheduling analysis.

In the context of real-time systems, Di Natale and Bini [8] proposed an optimization method to partition the FPGA area between slots allocated to HW-tasks and softcores in charge of executing the remaining tasks. Pellizzoni and Caccamo [9] considered a more dynamic scenario proposing an allocation scheme coupled with an admission test to provide real-time guarantees of applications supporting mode changes. Other authors [10], [11] presented scheduling algorithms to manage job-level reconfiguration of the FPGA, but assuming reconfiguration times negligible or fixed. Dittmann and Frank [12] addressed the issue of scheduling reconfiguration requests as a uniprocessor scheduling problem. However, their model can manage only HW-tasks and it is not suitable for platforms that also integrate softcores or processors. Although these works were aimed at providing real-time bounds, the models used for the reconfiguration infrastructure are too simplistic to describe the complexity of real platforms, hence the corresponding approaches cannot be used for analyzing real implementations with DPR features.

This paper. In summary, none of the presented papers addressed the problem of modelling the timing behavior of the reconfiguration interface and the interaction between SW-tasks and HW-tasks in such a way that they can be used for a tight real-time analysis. To address this issue, a prototype implementation of a job-level FPGA management has been developed to **(i)** profile the timing behavior of the reconfiguration port with the purpose of deriving such a model, **(ii)** investigate the practical feasibility of the job-level approach for real-time applications, and **(iii)** identify possible bottlenecks. Section V reports the results of some experimental studies conducted on such a prototype implementation.

III. System description

This work considers a heterogeneous computing system consisting of *one* processor and a DPR-enabled FPGA fabric, both sharing a common DRAM memory. A representative block diagram of the considered system is illustrated in Figure 2.

Possible representative platforms compatible with the considered system include the Zynq-7000 family by Xilinx, which provides ARM Cortex A9 processors and a FPGA fabric ranging from 28K up to 444K logic cells. Two types of computational activities can run on such a system:

- *software tasks* (SW-tasks): they are computational activities running on the processor; and
- *hardware tasks* (HW-tasks): they are functions implemented in programmable logic and executed on the FPGA fabric.

SW-tasks can speedup parts of their computation by requesting the execution of HW-tasks, which can be considered as *hardware accelerated functions*.

The area of the FPGA fabric is divided into a reconfigurable region and a static region. The reconfigurable region hosts the HW-tasks while the static region includes support modules for the HW-tasks, such as communication devices. The reconfigurable region is partitioned into *slots*, each including the same number of logic blocks. A HW-task can execute only if it has been programmed into a slot. Each slot can be reconfigured at run-time by means of a *FPGA reconfiguration interface* (FRI) and can accommodate at most one HW-task.

As typical for most real-world platforms (e.g., [13], [14]), the FRI

(i) can reconfigure a slot without affecting the execution of the HW-tasks currently programmed in other slots;
(ii) is a peripheral device external to the processor (e.g., like a DMA [15]) and hence does not consume processor cycles to reconfigure slots; and
(iii) can program at most one slot at a time.

To program a given HW-task into a slot, the FRI has to program all the logic blocks of the slot. This is because unused logic blocks have to be disabled to "clean" possible previous configurations. The FRI is characterized by a *throughput* ρ, meaning that a time $r = b^S/\rho$ is needed to reconfigure a slot, where b^S is the number of logic blocks in each slot.

Each SW-task uses a set of HW-tasks by alternating execution phases with *suspension* phases where the SW-task is descheduled to wait for the completion of the requested HW-task. The same HW-task cannot be used by more than one SW-task. Each SW-task is periodically (or sporadically) released, thus generating an infinite sequence of execution instances (denoted as jobs). SW-tasks are also subject to timing constraints, meaning that each of its jobs must complete its execution within a *deadline* relative to its activation. Figure 3 reports the pseudo-code defining the implementation skeleton of a SW-task that calls a single HW-task.

The HW-task is initialized at line 7, where the label `sample_hw_task` is used to refer its implementation stored in memory. At line 15, the SW-task configures the HW-task by specifying two memory locations: (i) `input_ptr`, that contains the *input data* for the HW-task and (ii) `output_ptr`, prepared to contain the *output data* produced by the HW-task. Finally, at line 18, the SW-task executes a *blocking* call that triggers the reconfiguration and executes the HW-task. The SW-task correspondingly suspends its execution until the completion of the HW-task. The inter-task communication mechanism is discussed in the following section.

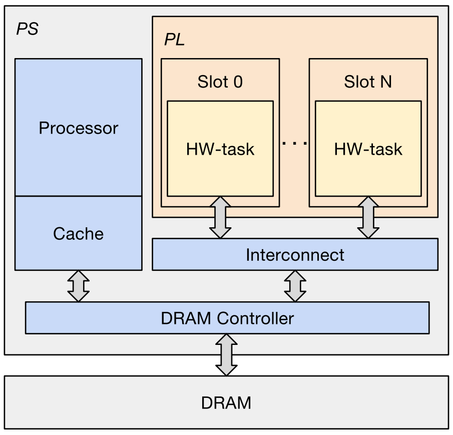


Figure 2: Block diagram of the considered system.

```
1  void sample_software_task()
2  {
3   // Task initialization (executed only once)
4   << Initialization part >>
5
6   // Define an instance of an HW-task
7   Hw_Task hw_task = hw_task_init(sample_hw_task);
8
9   // Task body
10  while (1)
11  {
12    << Software elaborations chunk >>
13
14    // Configure input and output data for the HW-task
15    hw_task_set_args(hw_task, input_ptr, output_ptr);
16
17    // Reconfigure and execute the HW-task
18    rcfg_manager_execute_hw_task(hw_task);
19
20    << Software elaborations chunk >>
21
22    // Wait for the next job
23    suspend_until(period);
24  }
25 }
```

Figure 3: Pseudocode of a SW-task calling a HW-task.

IV. System prototype

This section presents the implementation of a system prototype to handle HW-tasks under DPR on a real platform. The prototype has been used to conduct some preliminary experiments to evaluate the feasibility and the performance of the proposed approach.

A. Reference platform

The Zynq-7000 SoC family has been chosen as a reference platform for developing a working prototype of the system. It includes a dual-core ARM Cortex-A9 processor and a DPR-enabled FPGA fabric integrated on the same die.

The internal structure of a Zynq SoC comprises two main functional blocks referred to as processing system (PS) and programmable logic (PL) [15]. The PS block includes the ARM Cortex-A9 MPCore, the memory interfaces and the I/O peripherals, while the PL block includes the FPGA fabric. The subsystems in the PS are interconnected among themselves, and to the PL side, through an *ARM AMBA AXI Interconnect*.

The Interconnect can be accessed by custom logic modules (configured on the PL side) through a set of *master* and *slave* AXI interfaces exported by the PS to the PL side. In particular, the slave interfaces allow hardware modules hosted on the PL to access the global memory space where the physical RAM memory is mapped. This is achieved by implementing an AXI master interface inside the module logic. Such a master interface can be connected to the corresponding slave interfaces offered by the PS. In this way it is possible to implement a *shared-memory* infrastructure between the processor and the custom modules deployed on the PL.

The SoCs of the Zynq family supports dynamic partial reconfiguration under the control of the software running on the PS. The FPGA fabric included in the PL can be fully or partially reconfigured via the device configuration interface (DevC) subsystem. The DevC includes a DMA engine that can be programmed to transfer bitstreams (i.e., images of custom modules to be configured onto the FPGA) from the main memory to the PL. This is achieved by means of the the processor configuration access port (PCAP).

B. Prototype architecture

In the system prototype, the area of the FPGA fabric included in the PL is divided into a static region and a reconfigurable region. The static region contains the static portion of the communication infrastructure (consisting in interconnection blocks similar to switches) and other support modules, while the reconfigurable region hosts the hardware modules that implement the HW-tasks and a common communication interface.

Such a common interface is similar to the one adopted by Sadri et al. [16] and includes (i) an *AXI master interface* for accessing the system memory, (ii) an *AXI slave interface* through which the HW-task can be controlled by the PS, and (iii) an *interrupt signal* to notify the PS when the computation has been completed. In the current setup, the AXI master interfaces included in the HW-tasks are attached to high-performance (HP) ports exported by the PS, while the AXI slave control interfaces are attached to the PS AXI master general purpose ports.

The reconfigurable region is partitioned into a fixed number of slots, each containing an equal number of logic resources. Each slot can accommodate a single HW-task. Since bitstreams relocation is not supported by the Xilinx's standard tools [13] [14] (i.e., the same bitstream cannot be used for multiple slots), each HW-task is synthesized as a *set of bitstreams*, one for each slot defined in the PL.

C. Software support

The software part of the system prototype has been developed as a user-level library for the FreeRTOS [17] operating system. The library facilitates the reconfiguration and the execution of HW-tasks by providing a simple API that enables the client programmer to exploit hardware acceleration.

From the client programmer perspective, the library models the concept of hardware acceleration with a set of HW-task objects and a software module named reconfiguration service. The interface of the reconfiguration service offers a single function to request the execution of a HW-task (as shown in Figure 3, line 18). Each HW-task object includes the following information: (i) a set of bistreams, one for each slot; (ii) the input parameters (memory pointers or data); (iii) two optional callbacks (linked to the start and the completion of the HW-task) that can be used to ensure memory coherence. The library has been build on top of the Xilinx software support library [18].

Before executing a HW-task, our implementation flushes the portion of cache containing the input data prepared by the SW-task, thus ensuring that the HW-task can access coherent data from the RAM memory.

Once the input data have been prepared, the SW-task checks for a vacant slot performing a *wait* operation on a FreeRTOS counting semaphore (initialized with the number of available slots). If all the slots are busy, the calling task is suspended until one of the slots will be released. When at least one slot is available, the function searches if any of the vacant slots already contains the requested HW-task. If none of the vacant slots contains the required HW-task, one of the vacant slots is reconfigured with the corresponding bistream. The calling task is suspended until the reconfiguration has been completed.

As soon as the requested HW-task is configured, it starts executing. The calling SW-task suspends its execution until the completion of the HW-task. When the HW-task completes, the calling SW-task is resumed and performs a *signal* operation on the slots counting semaphore. The completion is notified to the PS with the interrupt signal predisposed in the common interface described in Section IV-B. Once the SW-task is resumed, our implementation invalidates the cache portion corresponding to the output data produced by the HW-task, thus ensuring that the processor can access coherent data.

D. Experimental setup

To perform a set of experiments, the system prototype has been deployed on a ZYBO board that includes the Z-7010 Zynq SoC and 512 MB of DDR3 memory. The ARM core included in the PS of the Z-7010 runs at 650 MHz, while the clock frequency for the PL is set to 100 MHz.

In the experimental setup, 50% of the logic resources of the PL are allocated to the reconfigurable partition, while the remaining 50% are allocated to the static part. The reconfigurable partition is divided into two slots of equal size. Each slot contains half of the resources available in the reconfigurable partition. Since both slots contain an equal number of resource, the corresponding bitstreams (resulting from the logic synthesis of HW-task in each slot) have the same size, equal to 338 KB. Considering the size of the RAM memory available on the platform (512 MB), a large number of partial bitstreams can be stored without any relevant impact on the available memory.

V. EXPERIMENTAL RESULTS

This section reports the results of a set of experiments that have been conducted to evaluate the proposed approach on a case study application.

To test the system, four standard algorithms have been implemented as both HW-tasks and equivalent software procedures. The test set includes tree simple implementations of image convolution filters (*Sobel*, *Sharp* and *Blur*) and an integer matrix multiplier (referred to as *Mult*). The HW-tasks have been designed with the Vivado high-level synthesis tool, while the software versions have been implemented in the C language.

The Blur and the Sharp filters have been configured to process images of size 800×600 pixels, while the Sobel filter has been configured to process images of size 640×480 pixels. All the three filters process images with 24-bit color depth. The matrix multiplier processes matrices of size 64×64 elements.

A. Speed-up evaluation

A first experiment has been carried out to measure the speed-up factors achievable by the HW-task implementation of the four algorithms used in the case study. For each of such algorithms, the execution time of the corresponding

HW-task has been compared with the equivalent full software implementation for more 1000 runs. The results of this test are reported in Table I. The minimum speedup has been computed as the ratio between the minimum observed execution time of the software implementation and the maximum observed execution time for the HW-task.

As can be seen from the table, even though the FPGA is running at a lower clock frequency (100 MHz) compared to the processor (650 MHz), HW-tasks provide a consistent speed-up ranging from 2.5 to 15.2. The small differences between average and worst-case execution times can be explained by the fact that the functions are essentially stream processing operations with no branches depending on the input data.

Algorithm		Mult	Sobel	Sharp	Blur
Observed HW execution times	Average [ms]	0.785	12.710	24.631	24.628
	Longest [ms]	0.785	12.712	24.633	24.629
Observed SW execution times	Average [ms]	1.980	115.518	304.975	374.785
	Longest [ms]	2.017	115.521	304.994	374.811
Speedup	Average	2.523	9.089	12.381	15.217
	Minimum	2.515	9.087	12.380	15.216

Table I: Speed-up evaluation.

B. *Response-time evaluation*

A second experiment has been performed to evaluate the system behavior in a scenario where the number of HW-tasks to be executed exceeds the number of slots available on the FPGA fabric. Please note that such a scenario *is only possible by exploiting DPR*. The task set used for this experiment consists of four periodic SW-tasks with implicit deadline (i.e., deadlines equal to task periods). Each SW-task requests the execution of the HW-task corresponding to the algorithm of the case study (Section V). SW-tasks priorities are assigned according to the Rate-Monotonic algorithm. As mentioned in Section IV-C, each SW-task executes a flush operation (denoted as *cache flush*) before calling the HW-task and invalidates the cache when the HW-task completes (*cache invalidate* operation).

Table II reports the periods of the SW-tasks, the execution times of the cache flush and cache invalidate operations, and the response-times of the SW-tasks observed in 8 hours of execution.

Based on the collected data, it is worth observing that the considered application *cannot be scheduled without DPR* for the following reasons:

- due to the large execution times (see Table I), the application cannot be scheduled with a full software implementation;
- since the FPGA fabric has only two slots, it is not possible to statically configure all the four HW-tasks of the application;
- if the algorithms that cannot be allocated on the FPGA as HW-tasks are executed on the processor as pure software implementation, *any* possible combination of HW-tasks and software implementations leads to a non schedulable system.

This example shows that virtualizing the FPGA by the proposed timesharing mechanism can effectively improve the schedulability of applications on current heterogenous platforms.

The longest observed response time for the *Mult* SW-task shows that, even if this task has the highest priority in the system, it may experience high delays due to slot contention with other HW-tasks issued by lower-priority SW-tasks.

This happens because of the FIFO ordering of the semaphores used in the implementation. The execution of HW-tasks can hence be delayed by the reconfiguration and the execution of all the HW-tasks requested by other SW-tasks (independently of their priority). The analysis of such a delay is beyond the scope of this paper.

For some applications, the response-times can be improved by adopting different scheduling policies (i.e., different from FIFO) to manage HW-tasks. However, since HW-tasks execute in a non-preemptive manner, the largest execution time of the HW-tasks will always impose a lower-bound for the slot contention delay.

SW-task		Mult	Sobel	Sharp	Blur
Period [ms]		30	50	80	100
Cache flush [ms]		0.030	1.123	1.754	1.754
Cache invalidate [ms]		0.017	1.240	1.939	1.939
Observed Response time	Average [ms]	3.829	17.603	31.416	35.624
	Longest [ms]	24.017	20.418	33.086	43.160

Table II: Hardware accelerated task-set.

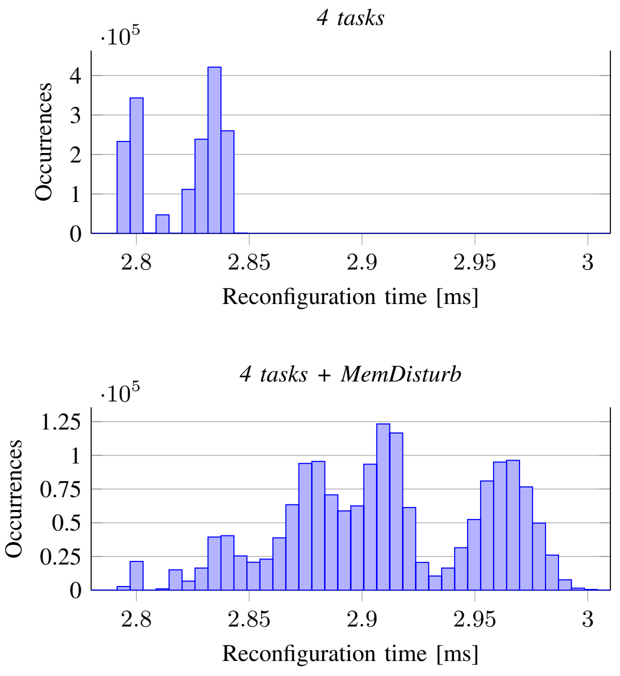


Figure 4: Distribution of reconfiguration times.

C. *Reconfiguration times profiling*

Finally, a third experiment has been conducted to profile reconfiguration times. The reconfiguration of the FPGA fabric is performed by the DevC subsystem described in Section IV-A. Such a module transfers bitstreams from the main memory to the PL configuration memory trough the PCAP port, which exploits the DevC DMA engine. The DMA accesses the system memory (where bistreams are stored) through an AXI master interface connected to the internal AXI Interconnect. Unlike the processor and the HW-tasks connected to the AXI slave ports, the DevC subsystem is not directly connected to the DRAM controller. In fact, it contends the access to the DRAM controller with other peripherals in the PS side.

In general, the throughput achievable by the DevC DMA depends on the traffic conditions on the AXI Interconnect, and the load on the DRAM controller. Modeling the bus contention on the AXI Interconnect and evaluating its performance goes beyond the scope of this paper. However, a first test was carried out to evaluate how a memory intensive SW-task interferes with the DevC, and hence affects reconfiguration times.

The task set used for this test includes the four tasks described in the experiment of Section V-B, and an additional *memory intensive* software activity (referred to as *MemDisturb*) continuously running in background without invoking HW-tasks. The MemDisturb software activity performs memory transfers between two memory buffers of 32 MB. The sizes of the buffers exceed the size of the processor L2 cache. Therefore, such a memory transfers generate a continuous stream of request to the DRAM controller that simulates a memory intensive SW-task.

Table III compares the reconfiguration times with and without the MemDisturb activity. Figure 4 illustrates the reconfiguration times distribution in both cases. The results of this experiment show that, despite a memory intensive software activity can affect reconfiguration times, its impact is very small and in the order of 0.1 ms. We believe that this result, although preliminary and far from being complete, is encouraging for exploiting partial reconfiguration in real-time systems, where bounded reconfiguration delays are essential to guarantee the system predictability. Given the size of the partial bistreams (338 KB), the average observed throughput for the DevC amounts to 117 MB/s without MemDisturb and to 113 MB/s with MemDisturb.

Experiment	Reconfiguration time [ms]		
	Min	Avg	Max
4 tasks (Section V-B)	2.791	2.820	2.846
4 tasks + MemDisturb	2.795	2.910	3.012

Table III: Observed reconfiguration times.

VI. Conclusions

This work presented an experimental study aimed at evaluating the use of dynamic partial reconfiguration for implementing a timesharing mechanism to virtualize the FPGA resource in heterogeneous platforms that also include a processor. Hence, an application consists of both software computational activities (running on the processor) and hardware modules implemented in programmable logic to be dynamically allocated on the FPGA, as requested by the software tasks. The temporal parameters involved in such a system (e.g., reconfiguration and execution times) have been profiled for a case study application. The achieved results are encouraging and clearly show that, in spite of the relatively high reconfiguration times of FPGAs, a timesharing mechanism can significantly improve the performance of real-time applications with respect to a fully static approach.

Besides the encouraging results, the experimental studies highlighted two major bottlenecks of today's platforms. First, all the evaluated FPGA platforms provide only a single reconfiguration interface, which is then contended by all the HW-tasks. Second, when the main memory is used to store both data and bistreams, an additional contention there exists on the Interconnect and the DRAM controller, which introduces further complications in the timing analysis. As a consequence, the presence of memories dedicated to bitstream storage would significantly improve both performance and predictability.

Future challenges include **(i)** the design and the analysis of scheduling algorithms for HW-tasks, **(ii)** the investigation of partitioning approaches for the FPGA area to limit contention on the reconfiguration interface, **(iii)** the implementation of improved inter-task communication mechanisms, and **(iv)** the design of real-time operating system mechanisms to support such a dynamic approach.

References

[1] M. Goosman, N. Dorairaj, and E. Shiflet. (2006) How to take advantage of partial reconfiguration in fpga designs. [Online]. Available: www.eetimes.com/document.asp?doc_id=1274489

[2] S. Altmeyer and G. Gebhard, "WCET analysis for preemptive scheduling," in *Proceedings of the 8th Int. Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2008.

[3] R. Stefan and S. D. Cotofana, "Bitstream compression techniques for virtex 4 FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL 2008)*, 2008.

[4] E. Lübbers and M. Platzner, "Reconos: Multithreaded programming for reconfigurable computers," *ACM Transactions on Embedded Computing Systems*, vol. 9, no. 1, pp. 8:1–8:33, October 2009.

[5] ——, "Cooperative multithreading in dynamically reconfigurable systems." in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, August 2009.

[6] M. Happe, A. Traber, and A. Keller, *Proceedings of the 11th International Symposium on Applied Reconfigurable Computing (ARC)*. Springer International Publishing, April 2015, ch. in Preemptive Hardware Multitasking in ReconOS, pp. 79–90.

[7] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, R. Torrego, and T. Arslan, "Microkernel architecture and hardware abstraction layer of a reliable reconfigurable real-time operating system (r3tos)," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 8, no. 1, pp. 5:1–5:35, March 2015.

[8] M. D. Natale and E. Bini, "Optimizing the fpga implementation of hrt systems," in *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, April 2007.

[9] R. Pellizzoni and M. Caccamo, "Real-time management of hardware and software tasks for fpga-based embedded systems," *IEEE Transactions on Computers*, vol. 56, no. 12, pp. 1666–1680, December 2007.

[10] K. Danne and M. Platzner, "Periodic real-time scheduling for fpga computers," in *Proceedings of the 3rd International Workshop on Intelligent Solutions in Embedded System*, May 2005.

[11] S. Saha, A. Sarkar, and A. Chakrabarti, "Scheduling dynamic hard real-time task sets on fully and partially reconfigurable platforms," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 23–26, March 2015.

[12] F. Dittmann and S. Frank, "Hard real-time reconfiguration port scheduling," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, April 2007.

[13] D. Koch, *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*. Springer-Verlag New York, February 2012.

[14] *Vivado Design Suite User Guide: Partial Reconfiguration*, Xilinx, 2015, v2015.4.

[15] *Zynq-7000 AP SoC Technical Reference Manual*, Xilinx, 2015, v1.10.

[16] M. Sadri, C. Weis, N. Wehn, and L. Benini, "Energy and performance exploration of accelerator coherency port using xilinx zynq," in *Proceedings of the 10th FPGAworld Conference*, September 2013.

[17] R. T. E. Ltd. Freertos real-time operating system. [Online]. Available: http://www.freertos.org/

[18] *OS and Libraries Document Collection*, Xilinx, 2015, v2015.3.

An implementation of the flexible spin-lock model in ERIKA Enterprise on a multi-core platform

Sara Afshar[1], Maikel P.W. Verwielen[2], Paolo Gai[3], Moris Behnam[1], Reinder J. Bril[1,2]
[1]Mälardalen University, Västerås, Sweden
[2]Technische Universiteit Eindhoven, Eindhoven, Netherlands
[3] Evidence Srl, Pisa, Italy
Email: {sara.afshar, moris.behnam}@mdh.se, pj@evidence.eu.com, r.j.bril@tue.nl

***Abstract*—Recently, the flexible spin-lock model (FSLM) has been introduced, unifying spin-based and suspension-based resource sharing protocols for real-time multiprocessor platforms by explicitly identifying the spin-lock priority as a parameter. Earlier work focused on the definition of a protocol for FSLM and its corresponding analysis under the assumption that various types of implementation overhead could be ignored.**

In this paper, we briefly describe an implementation of the FSLM for a selected range of spin-lock priorities in the ERIKA Enterprise RTOS as instantiated on an Altera Nios II platform using 4 soft-core processors. Moreover, we present measurement results for the protocol specific overhead of FSLM as well as the natively provided multiprocessor stack resource policy (MSRP). Given these results, we are now in a position to judge when it is advantageous to use either MSRP or FMLP for our system set-up for given global resource access times of tasks.

I. Introduction

In traditional lock-based resource-sharing protocols for real-time multiprocessor platforms, a task that is blocked on a global resource either performs a non-preemptive busy wait, i.e. *spins*, or releases the processor, i.e. *suspends*. The flexible spin-lock model (FSLM) [1] unifies these two traditional approaches. By viewing suspension on a core as spinning on a priority lower than any other priority on a core, the *spin-lock priority* can be treated as a parameter. Spin-based protocols, such as the multiprocessor stack resource policy (MSRP) [15], can be viewed to use the highest priority (HP) as spin-lock priority, and suspension-based protocols, such as the multiprocessor priority ceiling protocol (MPCP) [21], to use the lowest priority (LP). By being able to use an arbitrary priority for spinning rather than the two extremes, the FSLM is expected to improve schedulability.

The resource sharing rules for the FSLM have been defined in [1], assuming partitioned, fixed-priority preemptive scheduling, FIFO-based global-resource queues and both non-nested as well as non-preemptive global resource access, similar to MSRP and MPCP. These rules are complemented with schedulability analysis for specific spin-lock priorities, such as the HP, the LP, and the highest resource ceiling of global resources on a core, also called the ceiling priority (CP). Initial simulation results based on the developed theory [2] confirm the expectations with respect to improved schedulability. In particular, CP turned out to significantly improve schedulability compared to HP. The schedulability analysis developed in [1] does not take implementation overhead into account, however. The simulation results may therefore be biased.

This work is supported by the Swedish Foundation for Strategic Research via the research program PRESS, the Swedish Knowledge Foundation and ARTEMIS Joint Undertaking project EMC2 (grant agreement 621429).

In this paper, we present an implementation of the FSLM for a selected range of spin-lock priorities [26], in particular the range from CP until HP in Erika Enterprise [13], as instantiated on an Altera DE0 board from Terasic [24] using 4 soft-core processors. Erika Enterprise is a free of charge, open-source real-time operating system (RTOS) implementation, which was originally developed for small-scale OSEK/VDX [18] compatible embedded systems for the automotive market. Erika Enterprise has been ported to the Altera Nios II environment [11], supporting multiple soft-cores. We have ported Erika Enterprise to the Altera DE0 board. Based on our implementation, we compare the overhead of HP, as originally implemented in Erika Enterprise, and CP.

The remainder of this paper is organized as follows. In Section II, we briefly present related work. Next, in Section III, we present our real-time scheduling model and system. Sections IV and V describe the design and implementation of FSLM in Erika Enterprise. Section VI describes the experiments performed and briefly presents the measurement results. We conclude the paper in Section VII.

II. Related Work

In [17], two-phase waiting algorithms [19] are investigated through analysis and experiments, with the aim to minimize the cost of synchronization in large-scale multiprocessors. A two-phase waiting protocol is a combination of a spin-based and a suspension-based protocol. A task first spins for a statically determined amount of time, and subsequently blocks if further waiting is required. The MIT Alewife distributed-memory multiprocessor [3], which supports a shared-memory programming model, has been used for experimental measurements. The paper suggests to use knowledge about wait-time characteristics and the cost of blocking (i.e. the context-switching overhead) to set the maximum spinning time.

In [14], an experimental evaluation of MPCP and MSRP is presented based on a Janus dual-processor architecture. For random period generation of tasks the results show MSRP to be better than MPCP, although the results are not conclusive. For a more application-specific architecture representing a typical automotive application, MSRP has shown to clearly perform better. Moreover, they observed that MSRP is significantly simpler to implement, has lower overhead, and can achieve RAM memory optimization. Similar to this work, interrupt-based inter-processor mechanisms have been used for communication among tasks on different processors and atomic test-and-set mechanisms have been used for shared memory.

A first implementation of the PCP [22], SRP [5], M-PCP (an extension of PCP for multiprocessors), D-PCP [20] (a variant of MPCP used for distributed systems) and FMLP [6] synchronization protocols has been discussed in [7]. FMLP uses suspension-based mechanism for access to long resources and spin-based mechanism for access to short resources. A LITMUS^{RT} [10] platform has been selected for implementation which is a real-time extension of Linux operating system. In [8] a schedulability comparison has been made among MPCP, D-PCP and FMLP considering runtime overheads on LITMUS^{RT}. The experiments showed that the spin-based FMLP variant always had the best performance. The results confirmed their earlier results in [9] regarding preferability of spin-based approach to suspension-based approach under EDF scheduling.

This work complements earlier work by evaluating preemptable spinning, as supported by FSLM, through experimental measurements.

III. Scheduling model and system

In this section we describe our real-time scheduling model, the Altera DE0 board and development environment, and the Erika Enterprise and accompanying tool-suite RT-Druid.

A. Real-time scheduling model

We assume a set $\mathcal{P}$ of m identical cores $P_0, \ldots, P_{m-1}$, a set $\mathcal{T}$ of n sporadic tasks $\tau_0, \ldots, \tau_{n-1}$, and a set $\mathcal{R}$ of resources other than cores used by tasks. Tasks are statically allocated to cores, assigned unique priorities on each core, and scheduled using fixed-priority pre-emptive scheduling. Tasks do not suspend themselves.

Resources are categorized as *private*, *local*, or *global* based on task usage and task allocation. Private resources are used by a single task. Local resources are used by multiple tasks, and all those tasks are allocated to the same core. Global resources are also used by multiple tasks, but that set of tasks is allocated to at least two different cores. In this paper, the focus will be on global resources. Example 1 illustrates a configuration with a global resource.

Example 1. *Consider a set $\mathcal{P}$ of two cores P_0 and P_1, a set $\mathcal{T}$ of 4 tasks $\tau_0, \ldots \tau_3$, and a singleton set $\mathcal{R}$ of one resource R. As also indicated in Table I, R is used by tasks τ_0, τ_1, and τ_3. Task τ_3 is allocated to core P_0 and tasks τ_0, τ_1, and τ_2 to P_1. As a result, R becomes a global resource.*

	resource usage	allocation
τ_3	R	P_0
τ_2		P_1
τ_1	R	P_1
τ_0	R	P_1

TABLE I: Resource usage and allocation of tasks of $\mathcal{T}$.

Moreover, we assume that the priority π_i of task τ_i is higher than the priority π_j of task τ_j if and only if $i > j$. An activation of a task is also called a *job*. We assume constrained deadlines, i.e. deadlines of tasks equal or smaller than their periods.

For FSLM, we assume FIFO-based resource queues and both non-nested as well as non-preemptive global resource access, similar to MSRP and MPCP. When a task is blocked on a global resource, it will perform a busy-wait on a core-specific spin-lock priority. That spin-lock priority is determined statically, and may range from the lowest to the highest priority on the core. In this paper, we assume the spin-lock priority is taken from the range $[CP, HP]$, where HP represents the highest priority on the core and CP represents the highest resource ceiling of the global resources used on that core. Example 2 illustrates FSLM for the configuration described in Example 1.

Example 2. *For the configuration of Example 1, the highest resource ceiling of the global resources used on core P_0 is equal to the priority π_3 of task τ_3. Similarly, the highest resource ceiling on P_1 is equal to the priority π_1 of task τ_1.*

For the same arrival pattern of tasks, Figure 1 illustrates FSLM for two different spin-lock priority assignments; one conform MSRP (Figure 1(a)), i.e. using HP, and one using CP on each core (Figure 1(b)). Because task τ_3 accesses the global resource R in the time interval $[1, 8)$, task τ_0 starts spinning upon its resource request to R at time 3 for both cases. Spinning is performed non-preemptively for MSRP (Figure 1(a)), i.e. using HP, and preemptively when using CP (Figure 1(b)). Using HP, τ_2 is blocked from its arrival at time 6 until task τ_0 releases the global resource R at time 12. Conversely, using CP, task τ_2 can preempt τ_0 at time 6 during spinning. Task τ_2 can execute till time 8, when τ_3 releases R, τ_0 is granted R, and τ_0 subsequently accesses R till time 12. When task τ_0 releases R at time 12, τ_2 is resumed.

This example shows that tasks with a priority higher than the spin-lock priority, e.g. τ_2 on P_1, experience less blocking due to global resource arbitration under FSLM using CP than using HP as spin-lock priority.

By restricting the range to $[CP, HP]$, the protocol maintains two attractive properties of MSRP. Firstly, at any moment in time, at most one job on a core can have a pending request for or access to a global resource. As a result, a job that is spinning on a global resource will have to wait for at most $m - 1$ jobs on remote cores. Consequently, the length of any global resource queue, even the sum of the length of all global resource queues, is at most $m - 1$. Secondly, any job of a task

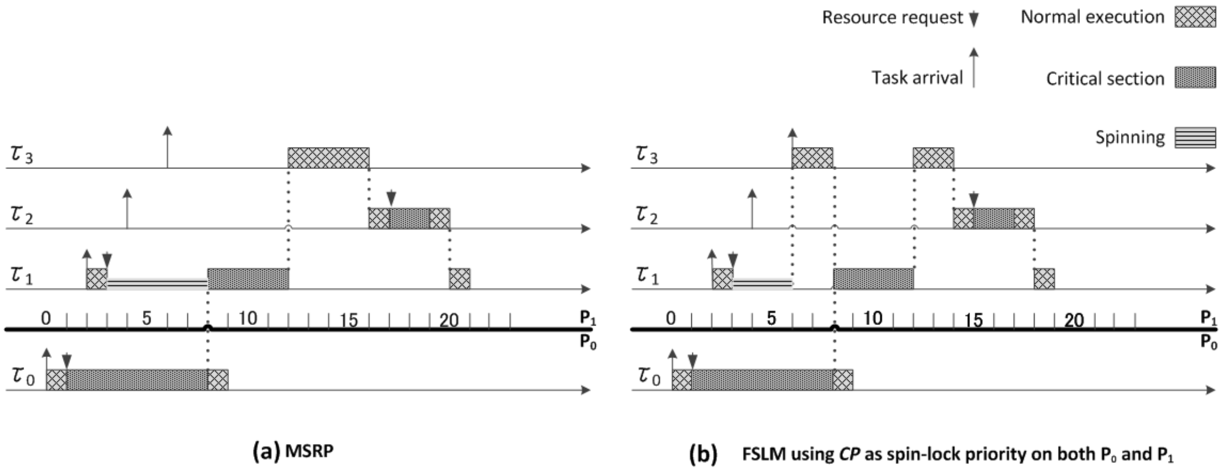


Fig. 1: Timelines for the same arrival pattern of tasks of $\mathcal{T}$, illustrating the FSLM for an assignment of (a) HP (conform MSRP) and (b) CP to the spin-lock priorities of each core.

on a core can be blocked at most once due to global resource requests of lower priority tasks on that core. Another attractive property of MSRP, i.e. the ability to use a single stack for all tasks on a core, is no longer maintained, however, as illustrated by the preemption of task τ_2 by τ_0 in Figure 1(b) at time 8.

B. Altera DE0 board and development environment

The Altera DE0 development and education board is equipped with the Altera Cyclone III 3C16 field-programmable gate array (FPGA) device, which offers 15,408 logical elements (LEs). The FPGA device can be configured by means of Altera's Quartus II Web Edition Software and Altera's Nios II Embedded design suite.

Using Altera's tools, we created a hardware design consisting of 4 Nios II processors (cores) and added internal (RAM) and external (SDRAM) memory, a mutex (to support mutual exclusive access), inter-core interrupt communication between every pair of cores, and performance counters (to enable high-resolution measurements) to the design, amongst others.

The resulting multi-core platform can communicate through a shared memory interconnect [23] and via inter-core interrupts. The connections for the inter-core interrupts are illustrated in Figure 2.

C. Erika Enterprise and RT-Druid

As mentioned above, Erika Enterprise was originally developed for OSEK/VDX-based systems. We used the multi-core extension [11] of the so-called "multistack" configuration of the "BCC2" conformance class of the OO (OSEK OS) kernel [13] of Erika Enterprise. RT-Druid [12] is a tool-suite developed for Erika Enterprise providing a system modeler, code-generator plugins for the open-source Eclipse framework [25] and schedulability analysis plug-ins. The RT-Druid Modeler is used for configuring both the application as well as Erika Enterprise, using the OSEK Implementation Language (OIL).

Both Erika Enterprise as well as RT-Druid have been extended for multiprocessor systems. To that end, the standard

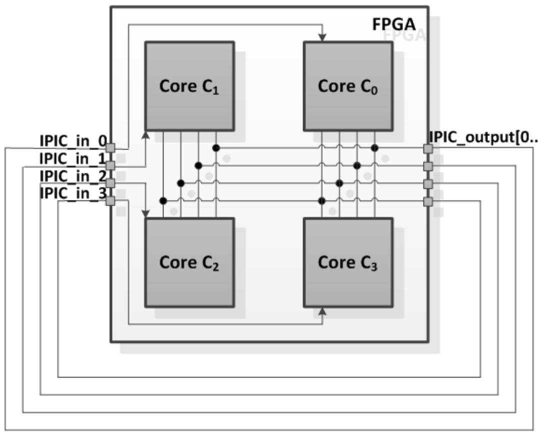


Fig. 2: The connections for inter-core interrupts

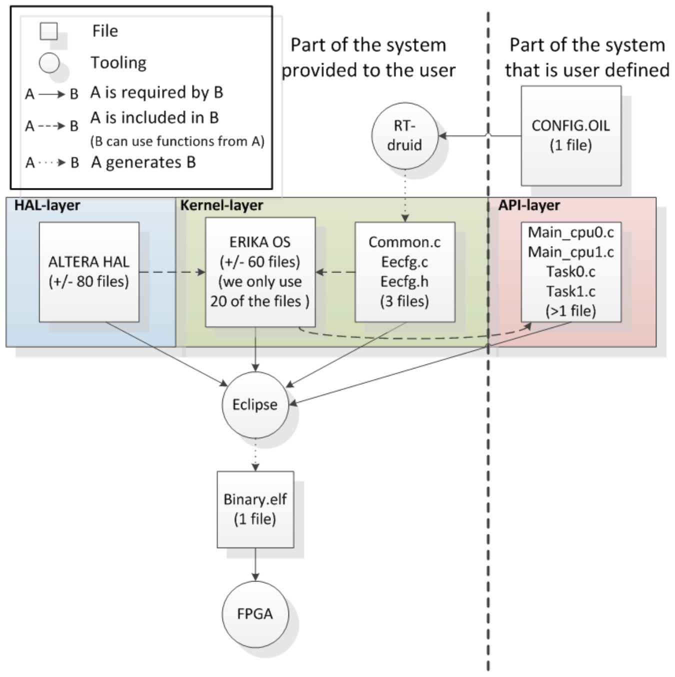
OIL has been extended to facilitate allocation of tasks to cores, amongst others.

Below, we first briefly describe the structure of a multi-core Erika Enterprise and its mapping on the Altera DE0 board. Next, we describe some key characteristics of Erika Enterprise.

1) Structure and mapping: The multi-core Erika Enterprise is a kernel-layer on top of Altera's hardware abstraction layer (HAL); see Figure 3. For our instantiation, the kernel-layer consists of approximately 20 standard files and 3 files generated per core by RT-Druid. The input for RT-Druid is a `CONFIG.OIL` file. The actual application is described by a set op files in the API-layer next to the `CONFIG.OIL` file.

2) Characteristics of Erika Enterprise: Erika Enterprise supports MSRP. To that end, it maintains a data structure in shared memory. When a task is busy waiting for a global resource, it spins on, i.e. polls, data in shared memory using the G-T algorithm [16].

Erika Enterprise also support event-based communication between cores using inter-core interrupts. As an example, a remote activation of a task can be accomplished through a



File
Tooling
A is required by B
A is included in B
(B can use functions from A)
A generates B
Part of the system provided to the user
Part of the system that is user defined
RT-druid
CONFIG.OIL (1 file)
HAL-layer
Kernel-layer
API-layer
ALTERA HAL (+/- 80 files)
ERIKA OS (+/- 60 files) (we only use 20 of the files)
Common.c Eecfg.c Eecfg.h (3 files)
Main_cpu0.c Main_cpu1.c Task0.c Task1.c (>1 file)
Eclipse
Binary.elf (1 file)
FPGA

Fig. 3: Erika Enterprise, Altera's HAL, application and mapping

so-called *remote notification* (RN). The sending core builds an RN message in shared memory and subsequently raises an interrupt at the receiving core. The interrupt handler of the receiving core inspects and processes the RN message asynchronously. Message buffers for RN require mutual exclusion.

IV. Design of the Flexible Spin-Lock Model

For the implementation of FSLM, five main aspects need to be considered:

1) Static selection of the spin-lock priority per core;
2) Dynamic change of the system ceiling to the spin-lock priority when a task blocks on a global resource;
3) Notification of a (blocked) task on a remote core that a global resource became available, when applicable;
4) Preemption of a higher priority task upon global resource access, when applicable.
5) Resumption of the preempted, higher priority task, when applicable.

We consider each of these aspects in more detail below.

A. Selection of spin-lock priorities

On each core where one or more tasks use a global resource, a spin-lock priority must be selected. In this paper, we only consider spin-lock priorities from the range $[CP, HP]$, and we therefore need to derive CP and HP from the system configuration. RT-Druid therefore needs to be extended with means (*i*) to determine CP and HP from the `CONFIG.OIL` file, (*ii*) to interact with a user to allow selection of spin-lock priorities per core, and (*iii*) to configure the kernel-layer of the RTOS with the spin-lock priorities.

B. Blocking on a global resource

When a task blocks on a global resource, the system ceiling on that core is raised to the spin-lock priority and the task starts spinning. This is illustrated in Figure 1(b) at time 3. Raising the system ceiling upon blocking is similar to the regular behavior upon a local resource access, which is based on the stack resource policy (SRP) [5].

C. Notification of a (blocked) task

Unlike MSRP, a task may be preempted during spinning, as illustrated in Figure 1(b) at time 6. As a result, the blocked task may not be aware that the global resource is released and becomes available. The design therefore has to be adapted from a polling-approach by the spinning task to a notification-approach by the releasing task. The existing remote notification mechanism present in Erika Enterprise can be used for FSLM as well. The first blocked job in the FIFO-queue of a global resource R, if any, will therefore be notified upon release of R by means of an interrupt, as illustrated in Figure 1(b) at time 8.

D. Preemption of the preempting task

Whenever a task τ_s spinning on a global resource is preempted by a task τ_p with a higher priority than the spin-lock priority, the preempting task τ_p must be preempted when the τ_s is granted the resource, as illustrated in Figure 1(b) at time 8. Although this gives rise to a preemption that disallows tasks to use a single stack, this behavior is supported by Erika Enterprise.

E. Resumption of the preempting task

When the task releases a global resource, it is checked whether or not the task preempted a task with a higher priority than the spin-lock was executing at the moment the resource was granted. In the former case, the preempted task is resumed, as illustrated in Figure 1(b) at time 12. In any case, the system ceiling is adapted, removing the traces of the request and access to the global resource.

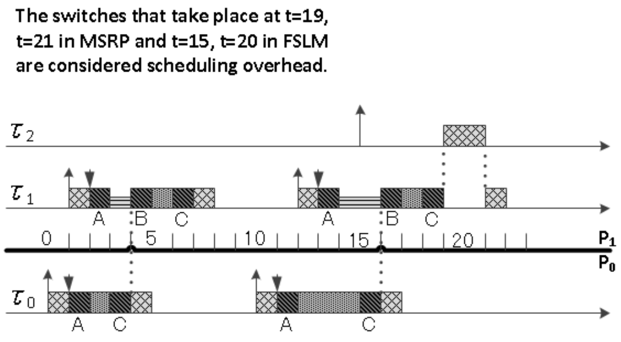
V. Implementation of the Flexible Spin-Lock Model

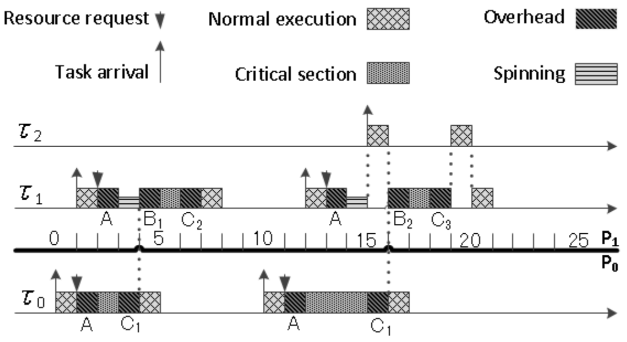
In this section, we first briefly present the implementation of MSRP in Erika Enterprise. Next, we will present the necessary changes for the generalization of MSRP to FSLM for the restricted range $[CP, HP]$ of spin-lock priorities.

A. Existing Implementation of MSRP in Erika Enterprise

In MSRP, a task requiring access to a global resource busy waits non-preemptively until (*i*) it is the first in line (first-in-first-out) waiting for the resource and (*ii*) the resource is free. Because spinning in MSRP is non-preemptive, at most one task per core can spin on a global resource. It is therefore also possible to associate a FIFO-queue of cores with every global resource.

To implement MSRP, Erika Enterprise essentially maintains a distributed polling-bit queue for each global resource (G-T algorithm [16]), i.e. a (non-empty) FIFO queue of polling


The switches that take place at t=19, t=21 in MSRP and t=15, t=20 in FSLM are considered scheduling overhead.
τ2
τ1
A B C A B C
0 5 10 15 20
P1
P0
τ0
A C A C
(a) MSRP



(b) FSLM using CP as spin-lock priority on both P0 and P1

Fig. 4: Synchronization protocol specific overhead under MSRP and FSLM

bits used by cores that want to access that global resource. The polling bits are stored as global data and the addresses of queue elements are stored in local data. To enable access to the queue, the tail of the queue is also stored as global data, containing the address of the global polling bit that needs to be inspected by the next core that requires access to the global resource. Access to the tail of the queue requires mutual exclusion. Releasing a global resource requires toggling the related polling bit only.

B. From MSRP to FSLM in Erika Enterprise

In the MSRP implementation of Erika Enterprise, a task that is accessing a global resource is unaware of the fact that another task on a remote core may (or may not) have requested the same resource, i.e. it is unaware of successors in the polling-bit queue. A task that is waiting for a global resource to become available is aware of the task in front of it in the polling-bit queue.

To facilitate notification for FSLM, the releasing task must know which task/core needs to be granted the global resource. The "knowledge" of the order in the queue must therefore become bi-directional. Rather than using a global polling-bit, we therefore used a global field representing both locked and unlocked as well as the task to be notified upon release of the global resource, if any. This global field requires mutual exclusive access. To reduce contention on shared data, we implemented an additional local bit for spinning.

Upon a global resource release, a task first checks whether or not tasks are blocked on that resource. The resource is subsequently released. In case tasks are blocked, the first in line, i.e. the successor of the releasing task, is notified through a dedicated remote notification (RN).

When an RN is received, it is first checked if the task blocked on the global resource is still spinning or has been preempted by a (or actually one or more) task(s) with a higher priority than the spin-lock priority. In the former case, the system ceiling is raised to reflect non-preemptive execution and the local bit is toggled, enabling the spinning task to access the global resource. In the latter case, the currently executing task must be preempted in addition, and the blocked task allowed to continue.

When a task has released a global resource, it has to check whether or not its access to the global resource induced the preemption of a task. In the former case, the preempted task (or any other task with a yet higher priority), is allowed to be resumed (or started).

For the original implementation of MSRP, a single low-level spin-lock is used for both the access to the shared data structures for global resources as well the RN message buffers. For the implementation of FSLM we added a low-level spin-lock, allowing parallel access to these two types of shared data.

VI. Experimental Evaluations

We performed a comparative evaluation of the implementation of MSRP and FSLM by measuring the overhead of both protocols. Overhead occurs at three specific moments during the protocols (see also Figure 4), i.e.

A) upon global resource request,
B) when the access to a global resource is granted, and
C) when a global resource is released.

A global resource request requires mutual exclusive access to the shared data structures for global resources for both MSRP and FSLM. Because all m cores may simultaneously perform a request to that data, a core may have to wait on $m-1$ other cores before it is granted access. Under FSLM, the release of a global resource also requires mutual exclusive access to that shared data. Under MSRP, releasing a resource only requires toggling a bit.

Under FSLM, releasing a resource may require the submission of a RN, and therefor mutual exclusive access to the RN message buffers. Similarly, access to the RN message buffers is required when a global resource is granted to a task while it is waiting. Upon release, all cores, except the waiting core(s), may require access to the RN message buffers, i.e. at most $m-1$. Upon access, all cores may require access to the RN message buffers.

Measurements using performance counter cores [4] were performed for two scenarios, one without preemption during spinning (from time 0 until time 8 in Figures 4(a) and 4(b)) and one with preemption during spinning (from time 10 onwards in Figures 4(a) and 4(b)). We have repeated the experiments 100 times. The measurement results are given in Table II.

	MSRP		FSLM	
Request	A	$160 + 79^{(a)}$	A	$189 + 146^{(a)}$
Access	B	18	B_1	$127 + 538^{(b)}$
			B_2	$140 + 538^{(b)} + 700^{(c)}$
Release	C	255	C_1	$322 + 94^{(a)} + 560^{(b)}$
			C_2	$255 + 94^{(a)}$
			C_3	$255 + 94^{(a)} + 700^{(c)}$

TABLE II: Measurement results in cycles. The superscripts (a) and (b) are added to the values of the worst-case critical section length for access to shared data structures for global resources and to the RN message buffers, respectively. The superscript (c) denotes context-switching overhead.

From these results, we conclude that the overhead for a global resource request is roughly the same for MSRP and FSLM. Compared to MSRP, the overhead for a global resource access and a global resource release is significantly higher for FSLM, however. As indicated in the table, this is mainly due to additional logic, reading and writing RN message buffers, and the additional context switches.

As described in Section III, FSLM reduces the blocking time due to spinning for tasks with a higher priority than the spin-lock priority. Based on our measurements, we are now in a position to determine when to use MSRP or FSLM for those tasks. The sum of the additional overheads for MSRP is 512 cycles, whereas this sum for FSLM is $2,762$ cycles, which corresponds to $10\mu s$ and $55\mu s$ on our $50MHz$ platform. The break-even is therefore when the sum of the remote global resource access times of tasks on our multi-core platform exceeds $2,250$ cycles, or $45\mu s$.

VII. Conclusions

In this paper, we presented an implementation of FSLM in Erika Enterprise on an Altera Nios II platform and a comparative evaluation of the protocol specific overheads of the native MSRP supported by Erika Enterprise and FSLM. Our experiments reveal that the overhead of global resource access and global resource release is significantly increased for FSLM. Based on these results, we are now in a position to judge when it is advantageous to use either MSRP or FSLM for such a system set-up for given resource access times.

References

[1] S. Afshar, M. Behnam, R. Bril, and T. Nolte. Flexible spin-lock model for resource sharing in multiprocessor real-time systems. In 9th *IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 41–51, June 2014.

[2] S. Afshar, M. Behnam, R. J. Bril, and T. Nolte. On per processor spin-lock priority for partitioned multiprocessor real-time systems. Technical Report 3874, Available: http://www.es.mdh.se/publications/3874-, Mälardalen University Sweden, 2014.

[3] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, and D. Nussbaum. The MIT Alewife Machine: A large-scale distributed-memory multiprocessor. In M. Dubois and S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*, pages 239–261. Springer US, Boston, MA, 1992.

[4] Altera. Quartus II 8.1 handbook, Volume 5: Embedded peripherals, Chapter 29: Performance counter core. Technical Report https://www.altera.com/en_US/pdfs/literature/hb/qts/archives/quartusii_handbook_8.1.pdf, November 2008.

[5] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, March 1991.

[6] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In 13th *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 47–56, Aug. 2007.

[7] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS^{RT}. In 14th *IEEE Intl. Conf. on Embedded and Real-Time Computing Sys. and Applications (RTCSA)*, pages 185–194, Aug. 2008.

[8] B. B. Brandenburg and J. H. Anderson. A comparison of the M-PCP, D-PCP, and FMLP on LITMUS^{RT}. In 12th *International Conference On Principles of Distributed Systems (OPODIS)*, pages 105–124, Dec. 2008.

[9] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In 14th *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 342–353, April 2008.

[10] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In 27th *IEEE International Real-Time Systems Symposium (RTSS)*, pages 111–126, Dec 2006.

[11] Evidence S.r.l. Erika Enterprise Manual for the Altera Nios II target - the multicore RTOS on FPGAs (version 1.2.3). Technical Report http://download.tuxfamily.org/erika/webdownload/manuals_pdf/arch_nios2_1_2_3.pdf, Evidence S.r.l., Pisa, Italy, December 2012.

[12] Evidence S.r.l. RT-Druid reference manual - A tool for the design of embedded real-time systems (version: 1.5.0). Technical Report http://download.tuxfamily.org/erika/webdownload/manuals_pdf/rtdruid_refman_1_5.0.pdf, Evidence S.r.l., Pisa, Italy, December 2012.

[13] P. Gai, E. Bini, G. Lipari, M. D. Natale, and L. Abeni. Architecture for a portable open source real time kernel environment. In 2nd *Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, 2000.

[14] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In 9th *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 189–198, May 2003.

[15] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In 22nd *IEEE Real-Time Systems Symposium (RTSS)*, pages 73–83, Dec. 2001.

[16] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

[17] B.-H. Lim and A. Agarwal. Waiting algorithms for synchronization in large-scale multiprocessors. *ACM Transactions on Computer Systems*, 11(3):253–294, August 1993.

[18] OSEK group. OSEK/VDX operating system. Technical report, February 2005. [Online], Available: http://portal.osek-vdx.org/files/pdf/specs/os223.pdf.

[19] J. Ousterhout. Scheduling techniques for concurrent systems. In 3rd *IEEE International Conference on Distributed Computing Systems*, Sep. 2014.

[20] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[21] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In 19th *Real-Time Systems Symposium (RTSS)*, pages 259–269, Dec. 1988.

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

[23] A. S. Tanenbaum. *Structured Computer Organization (5th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[24] Terasic16. Altera DE0 board. http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=364, Last visited: May 2016.

[25] The Eclipse Foundation. eclipse. Technical Report http://www.eclipse.org/.

[26] M. P. Verwielen. Performance of resource access protocols. Eindhoven University of Technology (TU/e), MSc-thesis, June 2016.

Notes

OSPERT 2016 Program

	Tuesday, July 5th 2016
8:30 – 9:30	Registration
9:30 – 10:45	Keynote talk: *From Research to Reality: Releasing System Software to the Masses* *Adam Lackorzynski*
10:45 – 11:00	Short Paper Towards versatile Models for Contemporary Hardware Platforms *Hendrik Borghorst, Karen Bieling, Olaf Spinczyk*
11:00 – 11:30	Coffee Break
11:30 – 13:00	Session 1: Multicore and Parallel Systems A communication framework for distributed access control in microkernel-based systems *Mohammad Hamad, Johannes Schlatow, Vassilis Prevelakis, Rolf Ernst* Tightening Critical Section Bounds in Mixed-Criticality Systems through Preemptible Hardware Transactional Memory *Benjamin Engel* GPU Sharing for Image Processing in Embedded Real-Time Systems *Nathan Otterness, Vance Miller, Ming Yang, James H. Anderson, F. Donelson Smith, Shige Wang*
13:00 – 14:30	Lunch
14:30 – 15:00	Discussion on Open Research Challenges in Real-Time Operating Systems
15:00 – 16:00	Session 2: Real-Time and Predictability Combining Predictable Execution with Full-Featured Commodity Systems *Adam Lackorzynski, Carsten Weinhold, Hermann Härtig* Timeliness Runtime Verification and Adaptation in Avionic Systems *José Rufino and Inês Gouveia*
16:00 – 16:30	Coffee Break
16:30 – 18:00	Session 3: OS and System Modelling Effective Source Code Analysis with Minimization *Geet Tapan Telang, Kotaro Hashimoto, Krishnaji Desai* Towards Real-Time Operating Systems for Heterogeneous Reconfigurable Platforms *Marco Pagani, Mauro Marinoni, Alessandro Biondi, Alessio Balsini, Giorgio Buttazzo* An implementation of the flexible spin-lock model in ERIKA Enterprise on a multi-core platform *Sara Afshar, Maikel P.W. Verwielen, Paolo Gai, Moris Behnam, Reinder J. Bril*
18:00 – 18:15	Closing Remarks

	Wednesday, July 6th – Friday, July 8th 2016
	ECRTS main conference.

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim