PROCEEDINGS OF

OSPERT 2015

the 11th Annual Workshop on
Operating Systems Platforms for
Embedded Real-Time Applications

July 7th, 2015 in Lund, Sweden

in conjunction with

the 27th Euromicro Conference on Real-Time Systems
July 8–10, 2015, Lund, Sweden

Editors:
Björn B. Brandenburg
Robert Kaiser

Contents

Message from the Chairs

Welcome to Lund in Skåne Län, Sweden and welcome to OSPERT'15, the 11th annual workshop on Operating Systems Platforms for Embedded Real-Time Applications. As we are entering the second decade of this unique venue, 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 Robert Leibinger, Solution Manager Innovations at Elektrobit Automotive GmbH, Germany. He will present his views on software architectures for advanced driver assistance systems. We are delighted that Robert volunteered to share his experience and perspective, as a healthy mix of academics and industry experts among its participants has always been one of OSPERT's key strengths.

In addition to the traditional full workshop paper format, OSPERT'15 also solicited short papers this time. The workshop received a total of fourteen submissions, five of which were in the short-paper format. All papers were peer-reviewed and eleven 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 compelling papers that explore unconventional approaches to real-time systems design. The short papers, which cover a diverse and interesting range of current topics, will be presented in Session 2. Last but not least, the day will close with an interesting session on integration, isolation, and scheduling issues in the context of shared (multicore) platforms.

OSPERT'15 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'15, 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 eleven experts from four different continents, 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,

Björn B. Brandenburg
Max Planck Institute for Software Systems
Kaiserslautern, Germany

Robert Kaiser
RheinMain University of Applied Sciences
Wiesbaden, Germany

Program Committee

OSPERT 2015 Keynote Talk

Software Architectures for Advanced Driver Assistance Systems (ADAS)

Robert Leibinger
Elektrobit Automotive GmbH

In recent years, the demand for electronic control units (ECUs) has been rapidly growing along with the number and complexity of functions these ECUs help to realize. The criticality of functions from a functional safety point of view is also increasing, which has lead to a demand for standards for safety-critical systems such as IEC 61508 or ISO 26262. At the same time, the underlying software architecture has also been standardized by committees such as AUTOSAR.

Traditionally, most automotive systems have been constructed as fail-safe systems, i.e., the failure of a system is detected with high confidence and the system degrades or is simply shut down. Every driver is nowadays aware of such diagnostic functions in the form of yellow or red warning signs telling them to visit the garage or even to stop the car immediately.

However, with the advent of supported, assisted, or even autonomous driving, the focus shifts from fail-safe systems to fail-operational systems. Such systems need to detect an error or even the fault leading to an error before the error leads to the failure of the system. Such systems are well established in other domains such as nuclear, where failure is not an option, or avionics, where many systems simply can't be shut down during flight operation.

As the automotive market is cost sensitive, different patterns need to be applied depending on the functionality, criticality, and reliability requirements of the system. To identify which pattern needs to be implemented, error scenarios as well as their effect on the reliability of the system need to be analyzed.

Reliability engineering shows that these solutions must use some form of redundancy, e.g., a degraded function on a different core of a multi-core processor or a different ECU or even a fully redundant function on a different ECU. Real-time requirements of the system as well as network latency and bandwidth are important factors for the selection of the optimal pattern. Such constraints often have a large impact on the implementation and can even influence the selection of algorithms that are used in advanced driver assistance systems, e.g., object recognition.

The keynote will show with examples how established concepts can be integrated into the automotive domain using both well-known approaches such as AUTOSAR or standard diagnostic functions, as well as new approaches such as service-oriented architectures based on automotive Ethernet.

Robert Leibinger studied Communication Electronics at Georg Simon Ohm University of Applied Sciences in Nuremberg, Germany. After graduating as Diplom-Ingenieur he started at 3SOFT (now Elektrobit, EB) in 2001 as Software Engineer for medical systems. In 2002, he switched to the automotive team, working on the OSEK operating system introduction at Daimler and serving as a consultant for several tier-1 suppliers regarding OSEK software architectures.

Starting in 2007, Robert became team leader of the AUTOSAR MCAL driver integration team and main contact to the microcontroller vendors. In 2011, he took over as the Product Manager responsible for the EB Safety Products and Operating Systems. Since 2014, he is part of the Solution Manager Team. The team defines and manages OEM-specific solutions using EB products and services. Robert is responsible for Daimler, JLR, and Functional Safety Solutions from Elektrobit.

Back to the Roots: Implementing the RTOS as a Specialized State Machine

Christian Dietrich, Martin Hoffmann, Daniel Lohmann
Department of Computer Science 4 - Distributed Systems and Operating Systems
Friedrich-Alexander University Erlangen-Nuremberg
{dietrich,hoffmann,lohmann}@cs.fau.de

***Abstract*— Real-time control systems, originally arisen from simple, state-machine–based discrete elements, nowadays comprise sophisticated and manifold software-based algorithms consolidated with different applications on single, yet powerful microcontrollers. Real-time operating systems were introduced to handle this complexity by providing APIs to describe the desired system behavior, however, at the cost of losing the clarity and explicitness of state-machine–based representations.**
This paper presents an approach to bring the RTOS back to the roots of a hardware-implementable finite state machine. The concept is based on a detailed static analysis of the application–kernel interaction to distill the real-time operating system behavior and find a FSM-based representation of the expected OS states and transitions. We apply our idea to a realistic control application based on an OSEK operating system, which results in a feasibly sized programmable logic array implementation. Having such a representation at hand might further leverage thorough system verification and validation based on existing and mature FSM analysis tools.

I. Introduction

Up to twenty-five years ago, embedded real-time control systems were typically designed by electrical engineers as *finite state machines* (FSMs) out of discrete elements. With the advent of cheap 4-bit and 8-bit microcontrollers, software has begun to take over the role of wiring discrete elements, but the paradigm of implementing control systems as FSMs remained. In comparison, the employment of a full-blown *real-time operating system* (RTOS) as underlying system software is a relatively young trend, triggered by the increasing complexity of control applications and the necessity of hardware consolidation. This is not always warmly welcomed by control-system engineers [18, 15], which is understandable, as the simple FSM paradigm has had some clear advantages: It is well understood (especially by certification authorities) and there is a large body of formal methods, heuristics, and tools available for optimization and validation, which leads to highly specialized, efficient implementations with low hardware requirements. On the other hand, employing an RTOS and its concepts (e.g., tasks, events, resources) can significantly ease the development of more complex control applications.

In this paper, we explore the possibility to get the best of both worlds: The idea is, to keep the RTOS interface for application development, but implement the *RTOS itself* (or more precisely: its concrete instance) as a FSM. Thereby, it becomes possible to use existing FSM-based analysis and validation tools (also) on the RTOS – or to push the RTOS completely "back into hardware" for perfect isolation.

This work was partly supported by the German Research Foundation (DFG) under grants no. LO 1719/1-3 (SPP 1500) and SCHR 603/9-1

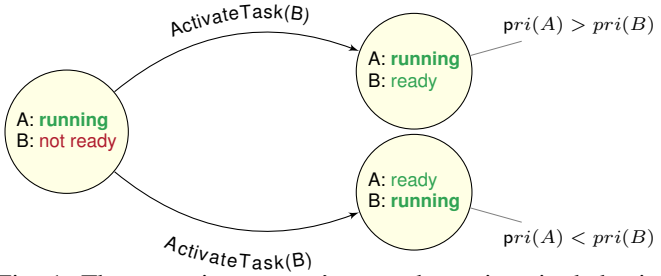


Fig. 1: The operating system's state determines its behavior. On system-call events, the OS changes this internal state.

A. Our Idea in a Nutshell

In theory, every computing system could be modelled as a FSM. This also holds for the RTOS: Every syscall, triggered synchronously by the application or asynchronously by an interrupt, can be considered as a transition on the OS-internal state (such as the ready list). The problem, however, is state explosion, caused by complex states and indeterminism in the control flow: Every syscall is a potential point of rescheduling at which, depending on the dynamic state of the ready list, some other task may be selected to continue execution.

The core idea of our approach is to reduce such indeterminism as far as possible at compile time: We exploit static knowledge about the RTOS configuration and *its semantics* in combination with a whole-system analysis across all control flows of the application to figure how the RTOS is actually used. Thereby, we derive a model on how the concrete application interacts with the kernel. We replace parts of the traditional OS implementation by an implementation of the derived model and (partially) specialize each syscall in the application at caller side to interact with the model.

The possible transitions on the kernel's state (such as the outcome of a scheduling decision) can thereby be greatly reduced at compile time, in many cases even to exactly one: If for instance, some task `A` triggers another task `B` for execution (`ActivateTask(B)`), this is a potential point of rescheduling. In a strictly priority-based system, however, the result can be reduced (by considering the scheduler semantics) to exactly two possible follow-up states: Depending on the relative priorities of `A` and `B`, either `A` is running and `B` is set ready (as shown in Figure 1) or vice versa. If we can further determine their priorities by static analysis, the effective result of this concrete syscall invocation can be reduced to exactly one follow-up state.

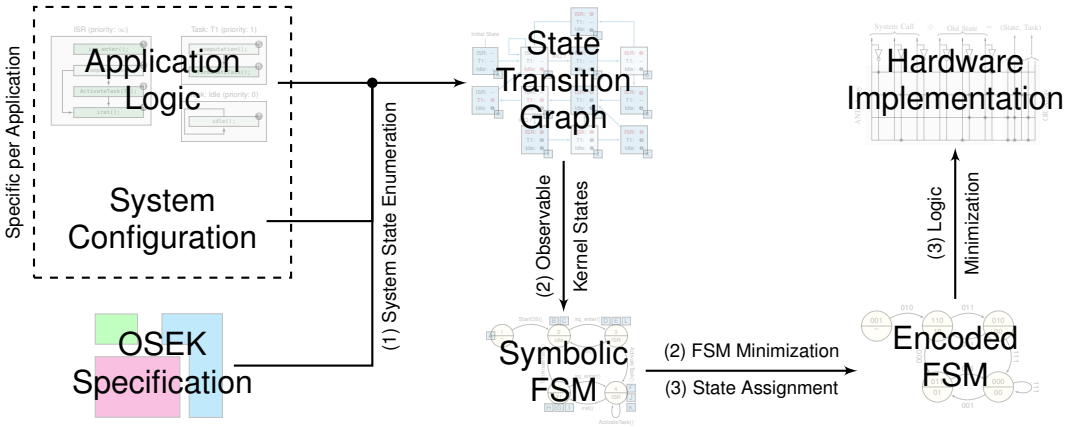


Fig. 2: Methodic Overview. From the general OSEK specification, and one concrete application, we generate a specialized OS implementation in several steps.

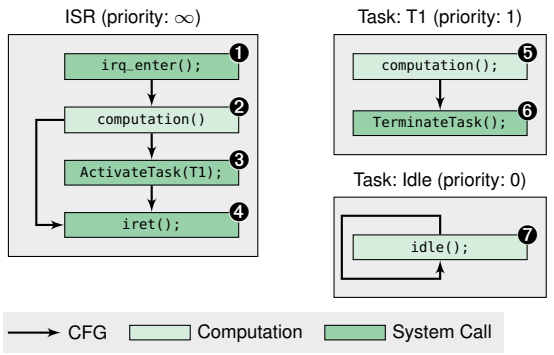


Fig. 3: Application Logic of a small (complete) OSEK System

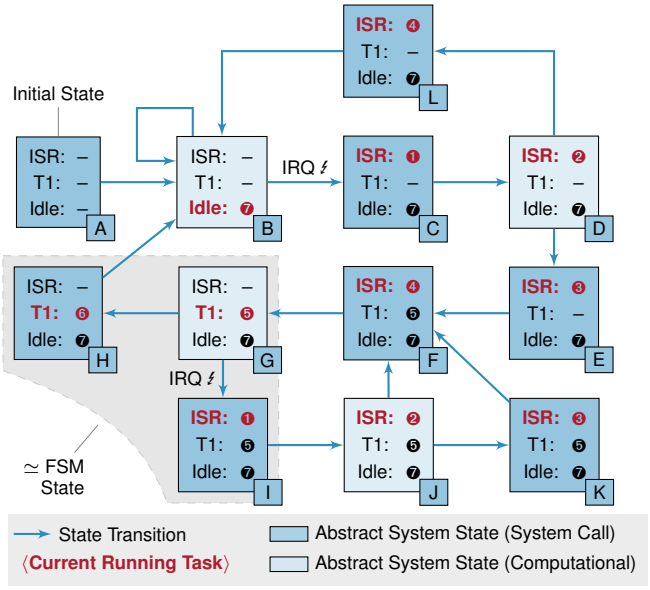


Fig. 4: State-Transition Graph for Figure 3. Each node is an abstract representation of the system at a one point in time.

Of course, in real-world systems, not all kernel interactions can be reduced that easily – especially interrupt-based alarms are a significant source of indeterminism. Nevertheless, our results show that the resulting state reduction makes it still feasible to generate the RTOS instance as a simple FSM.

B. Structure of the Paper

We apply our idea to the OSEK [13] / AUTOSAR [1] standards employed in the automotive industry. The RTOS included in these standards is an event-triggered, priority-driven, preemptive kernel. Its static configuration includes the number of tasks, their priority, the events they can wait for, and the resources they synchronize on using a static stack-based priority ceiling protocol. Without loss of generality, we choose OSEK as the running example throughout the paper.

In Figure 3, an example OSEK application is shown. It consists of one ISR, one normal task, and the idle loop. On an *interrupt request* (IRQ), the ISR may or may not activate the task. After the task finished its execution, it terminates and the OS executes the idle loop until the next IRQ occurs. Based on this example, the following Section II presents the static analysis and FSM construction. Finally, we provide first preliminary results on applying our concept to a realistic application scenario, and discuss further possible use cases.

II. Implementation

We divide our approach into three distinguishable parts: (1) The extraction of fine-grained interaction knowledge from the application. (2) The transformation to an executable model of the operating system. (3) The concrete implementation of the executable model. Figure 2 depicts the information flow of all three stages. With the *system-state enumeration* (SSE), we extract the interaction as a *state-transition graph* (STG) that enumerates all possible system states and their execution sequences. We identify all visible kernel states and construct a (minimized) FSM. As one possible implementation, we assign binary vectors for inputs, states, and outputs of the FSM and encode the minimized truth table as *programmable logic array* (PLA) simulation in software. In the following, we will investigate these steps in a greater detail.

A. System-State Enumeration

In the first step, we statically analyze the interaction of a given application with an abstract OSEK operating system. We already described this extraction step in previous work [5]. Therefore, we outline the *system-state enumeration* (SSE) mechanism only briefly and focus on the extracted fine-grained interaction knowledge, which is expressed as a STG.

The system-state enumeration combines three different sources of information in a forward simulation of the system: First, the *system semantics*, as defined by the OSEK specification [13]. Second, the *system configuration*, as declared in a domain-specific configuration language (OIL). And, third, the application logic, which is extracted from the control-flow graphs of the compiled application. The configuration already contains coarse-grained information about the system, like the set of tasks and their priorities. Together with the system semantics, we calculate fine-grained knowledge to predict the operating system's decisions in presence of the given application logic.

The SSE discriminates two block archetypes in the application: *computation* and *system-call blocks*. In computation blocks, the application does not issue system calls and therefore the OS state cannot be changed synchronously. Nevertheless, IRQs can *only* occur in computation blocks, and are modeled as asynchronous activation of ISR proxy tasks. The other block archetype contains system calls, which interact with the kernel synchronously and modify its state.

The central data structure for the SSE is the *abstract system state* (AbSS), which captures information about a system at a given point in time. For each task, an AbSS includes the ready flag, the current priority, and which block should be executed next in a task's context. Except the initial state, each AbSS has one task marked as the currently running task. In Figure 4, each node represents a simplified AbSS for the example system from Figure 3. For each task (interrupt-service routines and idle task included), the node contains the blocks to be executed next, while the currently active task is highlighted.

The SSE discovers all possible AbSSs for the given application, by repeated application of a `systemSemantic()` function on already discovered states until no new states appear. This transition function evaluates the block of the currently running task, calculates the block's influence on the current system state, and emits one or more follow-up states. For example, in Figure 4 only AbSS H executes block ❻ next. Since block ❻ contains a `TerminateTask()` system call, the transition function emits one follow-up state B with `T1` marked as *not-ready*. Furthermore, the transition function applies the OSEK scheduling rules and marks the idle task as *running*. All discovered AbSSs and their follow-up states are connected in the *state-transition graph* (STG).

interrupt-service routines (ISRs) are modeled with proxy tasks, which are assigned the highest possible priority and are executed under interrupt blockade. They are activated by the transition function within computation blocks. In Figure 4, the idle state B has two follow-up states: first, a self loop, since it is its own CFG successor. Secondly, the idle state can proceed to state C. This transition is the result of a virtual IRQ and the ISR entry block ❶ will be executed next.

The STG contains all possible state–state transitions for the given application. Depending on the application and its structure, it can become very large, but remains always finite. It is important to note, that each AbSS in the graph represents the system immediately *before* a block is executed. For a more detailed discussion on the SSE and mechanisms to ease the state explosion we refer to our previous work [5].

B. Kernel-Visible System States

As desired, the STG subsumes the application's control flow, as well as the kernel's scheduling decisions. We aim to implement only the OS' behavior. Therefore, we have to separate state transitions into application transitions and OS transitions. The application transitions are implemented by the application itself, in terms of branches, loops, and function calls. They are executed directly by the processor. Our specialized kernel should only implement the OS transitions, since only those are dictated by the OSEK specification.

As already said, each state represents the system right before a certain block is executed. Some states execute a computation block next, some a system-call block. Only the latter ones, *system-call states*, will ever be visible to an OS implementation. Therefore we partition all AbSSs in the STG into *regions of states* which are indistinguishable from the kernel's perspective. These regions are connected subgraphs within the STG; system-call states can only occur as leaf nodes in a region. In Figure 4, the states G, H, and I form such an region. This region cannot be extended to AbSS F, since F is a system-call state and must, therefore, be a leaf node in a different region.

These regions are constructed by repeated merging of initial minimal regions: Initially, each AbSS is located in its own region. For each state in a region, we merge the successor regions into the region, if the originating state is computational. Furthermore, we merge a predecessor region, if the preceeding state is a computational state. This process is repeated until no further changes happen.

With this construction, all states with a successor outside their region are system-call states. Since the OS state is only modified at the region's border, all inner states, which are computational, have the same task marked as running.

From these regions, we construct the initial *finite state machine* (FSM) for the kernel: Each region corresponds to a state in the FSM. An FSM transition from state A to state B is present, if a system-call state in region A can proceed to region B. The input event for this transition is the execution of the system-call block. Each FSM state exposes the currently running task as an output. It is noteworthy, that each system-call block results in a different FSM input signal, even if they invoke the same system service.

Figure 5 shows the resulting state machine with the AbSS regions drawn next to each FSM state. The constructed FSM matches the observation that an OS is a FSM with system calls as inputs and the currently scheduled task as output. In our construction, the resulting FSM is a Moore machine.

C. State-Machine Minimization

The resulting FSM already exhibits the required kernel transitions when triggered by external events and system calls.

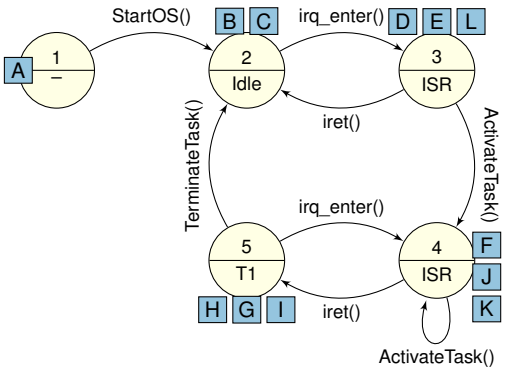


Fig. 5: Symbolic Finite State Machine with abstract system state next to each state.

Nevertheless, the number of states and transition edges is not minimal yet. Minimization of state machines is a well covered and long standing topic [12, 8]. Therefore, we will only investigate on the specifics for our operating-system FSM.

For the minimization of FSMs, states are grouped into *equivalence classes* (ECs), where each state within exposes the same observable behavior. From each equivalence class, a new state in the minimized FSM is generated, and transitions are added accordingly to the EC connections.

Our FSM is not an acceptor for a formal language. Furthermore, we are allowed to remove triggers from the system by wiping out system-call sites. We only have to ensure that the scheduling sequence remains the same. Therefore, we adapt the EC construction to fit these requirements.

First, we demand that each state in an EC results in the same *current running task*. Furthermore, the set of possible follow-up ECs must be equal for all states within an EC. The follow-up ECs of an state are those ECs which are reachable in the FSM when following the transitions. We used an adapted Moore algorithm [12] to find the most coarse EC partition of the FSM which fits both requirements.

In the minimized FSM, many transitions are self loops. If all transitions that are triggered by one system-call block are self loops, we wipe out the system-call site. The specific system-call signal never transfers the system into an observable different state; it is useless for our implementation. In the example (see Figure 5), the FSM is already minimal after its construction, but in general the size of FSM decreases significantly. With the FSM minimization, we have completed the construction of the executable model.

D. State Assignment and Logic Minimization

The last step is the implementation of the executable model and its linking to the application. The possibilities to implement the calculated FSM are endless. We chose to present an approach directed towards an OS implementation that fully resides in hardware. This would result in a specialized OSEK implemented as a processor extension.

However, while this is still a topic of further research, we currently provide a software simulated programmable logic array implementation of the generated FSM. While dispatching,

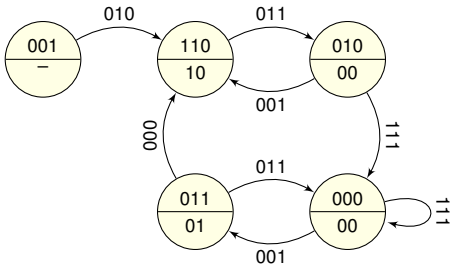


Fig. 6: Finite State Machine with Assigned Binary Vectors for Inputs, Outputs, and State Encodings

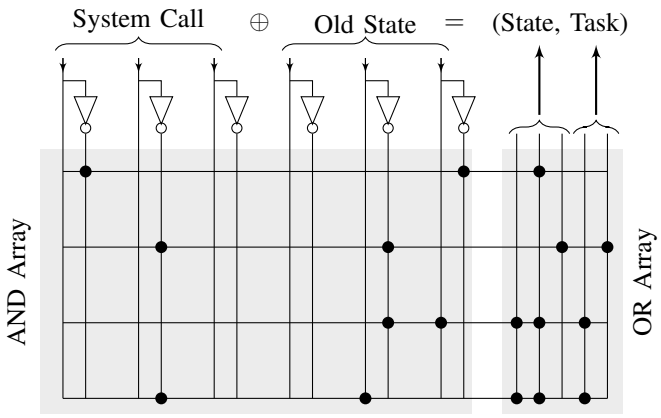


Fig. 7: Implementation as Programmable Logic Array

interrupt handling, and timer control is still implemented traditionally, the OS logic is already suited for a hardware implementation.

One main challenge of implementing a FSM in hardware is the selection of bit vectors for inputs, states, and output signals. This encoding largely influences the minimal required complexity of the hardware implementation. Luckily, many methods were already proposed to solve this problem for different hardware designs [17, 4, 16].

We decided to use the NOVA program [17] to choose the encoding of our FSMs. The driving factor of this decision was the availability of the NOVA source code. NOVA targets optimal encoding for two-level logic implementations. NOVA chooses input and state encoding for our FSM, while we choose the output encoding arbitrarily. The result of the assignment process is shown in Figure 6.

From the FSM and the encoding of inputs, states, and outputs, we generate a truth table with one line for each transition. Each line consists of the input word, the current state, the next state, and the desired output. To achieve an efficient implementation of this truth table in hardware, we use the ESPRESSO [2] heuristic logic minimizer.

From the minimization result, a PLA implementation can be derived in hardware. Figure 7 shows the final OS execution model for our running example. The resulting component takes the current system-call–block number and the saved system state as inputs. Each line in the AND array checks a certain

```
TerminateTask() called from T1
disable_interrupts();                                    ❹
OS_state, task = fsm_step(0b000, OS_state);
switch_to(task);
// Never returns, IRQ enable in next task
```

Fig. 8: `TerminateTask()` implementation called from T1.

TABLE I: Size of graphs, machines, and implementation after each step for the *I4Copter* task setup (11 tasks, 3 alarms, 1 ISR, 1 Resource).

Step		w/o Ann.	w/ Ann.
State-Transition Graph	[S(T)]	1,563,169 (2,098,236)	20,063 (23,876)
Symbolic FSM	[S(T)]	407,530 (942,597)	6,242 (10,055)
Minimized FSM	[S(T)]	2,938 (8,822)	667 (1,212)
Two-Level Logic	[AND Terms]	5,144	728
Software PLA Table	[Bytes]	35,798	4,566

bit pattern and emits a logic `1`, if the pattern matches. The OR array decides which outputs of the AND array will enable a bit in the output word. In our case, the output word consists of a new FSM state and the currently running task.

In our current implementation, we simulate this PLA in software by iterating over all lines in the ESPRESSO output. We use the task output word as an input for the dispatcher.

We replace every *system-call site* with a specialized code fragment that calls the FSM. Figure 8 exemplifies the implementation of the system-call block ❹. The `fsm_step()` function contains the PLA simulation, while the bitstring `000` identifies the call location exactly.

III. Preliminary Results

Currently, we do *not* produce hardware components from the execution model, but use a (slow) PLA software simulation. Therefore, we will only show some preliminary results for a realistic scenario to give an impression of the general feasibility.

We implemented the presented approach for the *d*OSEK [7] system generator[1]. As evaluation scenario, we use a realistic real-time workload. We revive a setup, already presented in previous work [6], resembling a real-world safety-critical embedded system in terms of a quadrotor helicopter control application. The scenario consists of 11 tasks, which are activated either periodically or sporadically by an interrupt. In total, 4 asynchronous events can trigger within computation blocks. Inter-task synchronization is done with OSEK resources and a watchdog task observes the remote control communication.

In the first column of Table I, the sizes of the system at different steps is given. While the STG has more than 1.5 million states and 2 million transitions, the (unminimized) FSM already reduces the size significantly. The minimization of the FSM removes 99.28 percent of the internal states. The state assignment and the logic minimization achieve a implementation of the execution model with 5,144 AND terms (rows in the PLA). In our software implementation, the minimized truth table occupies 35,798 bytes of read-only memory, while the implemented FSM requires 4 bytes of volatile memory for storing the current state.

In the second column of Table I, we show the results for the same system, but with additional annotations for the SSE analysis. We declared four task groups. Each group handles a different job in the system, which is released through an external signal (alarm or IRQ). The annotation forbids the retrigger of the signal while not all tasks of a group have finished their execution. This annotation is a qualitative statement that the deadline of the job is smaller than its period. This qualitative statement, which has to be supplied by the real-time developer, was already described in previous work [5].

With the annotation, the system has a 98.72 percent smaller STG, which, of course, was the intention of the annotation in the first place. Surprisingly, the state count of the minimized FSM shrinks only by 77.3 percent with annotations. This smaller decrease factor indicates an unnecessary edge redundancy in the STG without annotations.

IV. Discussion

In this paper, we derive an OS instance specifically tailored towards a given application. We used the OSEK API as a markup language to annotate the desired task orchestration and interaction. When we perceive the system configuration and placement of system calls as the abstract intentions of the real-time engineer, we can switch our focus from the traditional way of implementing the specification, to realizing only the developer's intended behavior. Encoding the minimized FSM in hardware is only one of many possible options. More importantly, this demonstrates the expressive power of the STG and the various FSMs as immediate representations of the system. Furthermore, pushing the OS logic fully into the hardware, we achieve perfect isolation. Not a single instruction would be needed for the OS execution. Only special opcodes would be reserved for giving inputs to the hard-coded FSM.

Apart from that, a FSM representation is not only useful for implementing the desired OS logic, but can also be used as watchdog for an off-the-shelf OSEK system. Fed with the same inputs, the actual OS must expose the same behavior. Combined with a WCET-based intrusion detection [19], an effective security scheme could be derived from static analysis of the system behavior.

Besides implementing the system behavior, the immediate representations make the actual kernel behavior accessible to other tools: The minimized FSM representation can be used to test whether the behavior of one real-time system is equivalent to or partially embedded in another system.

Our immediate representations may also assist the verification of tailored OS implementations: If we prove the equivalence of STG (or FSMs) to the OSEK standard for a certain application, and furthermore show the equivalence of the actual implementation to the STG, we get an OSEK implementation that is verified for a certain application; even in the presence of extensive system tailoring.

V. Related Work

The RTSC [14] that significantly inspired this work also uses the OSEK API as markup language to annotate the desired real-

[1]Code is released as free software at https://github.com/danceos/dosek

time behavior. It translates the system from an event-triggered to a table-driven, time-triggered system. Unlike our approach, their immediate representation is flow insensitive.

Chen and Aoki [3] use a formal model of OSEK and model checking techniques to automatically generate test cases for OSEK/OS. Their approach does not incorporate information about the configuration or the inner structure of a specific application, but emits whole applications as test-cases. Our application specific FSM could be used to generate application-specific event sequences to test the application, as well as the kernel.

In the sensor-network community, state machines are recognized as mean to compactly implement application and control logic. Kim and Hong [9] proposed state machines as well-suited paradigm for sensor nodes. Their *SenOS* kernel is an executor for transition tables, where each task comes with its own table. In contrast to our approach, the tables are not derived automatically.

Kothari, Millstein, and Govindan [10] proposed an automatic derivation of FSMs from TinyOS applications through symbolic execution. They derived "user-readable FSMs" in order to make the application logic more comprehensible to developers. As they state, their interrupt semantic is incomplete. Additionally, TinyOS has a simpler execution model than OSEK, since tasks have no wait states and only execute in a run-to-completion manner. Also, all their inferred FSMs do not exceed 16 states.

There are many projects implementing parts of the (or the whole) operating system in hardware. As one example, the ReconOS project [11] extends the multithreaded programming model across the hardware/software boundary. ReconOS provides a unified synchronization and communication API for hardware, which is executed on an FPGA, and software threads. Nevertheless, ReconOS is not tailored explicitly to fine-grained application logic, but mimics a generic, POSIX-like, interface.

VI. Conclusion

Many years of embedded real-time control engineering piled more and more abstraction layers on top of each other to ease the development process at the cost of complex software stacks and operating systems. In this paper, we presented an approach to descend these layers from an abstract RTOS-based control application back to the roots of an FSM-based PLA. Preliminary results already show the feasibility of our approach on the example of a realistic real-time application. Distilling the RTOS behavior not only allows to push it back into hardware, but might also leverage profound verification and validation of the system as a whole.

References

[1] AUTOSAR. *Specification of Operating System (Version 5.0.0)*. Tech. rep. Automotive Open System Architecture GbR, Nov. 2011.

[2] Robert King Brayton, Alberto L. Sangiovanni-Vincentelli, Curtis T. McMullen, and Gary D. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. Norwell, MA, USA: Kluwer Academic Publishers, 1984. ISBN: 0898381649.

[3] Jiang Chen and Toshiaki Aoki. "Conformance Testing for OSEK/VDX Operating System Using Model Checking". In: *18th Asia-Pacific Software Engineering Conference (APSEC 2011)*. (Ho Chi Minh). Los Alamitos, CA, USA: IEEE, Dec. 2011, pp. 274 –281. ISBN: 978-1-4577-2199-1. DOI: 10.1109/APSEC.2011.26.

[4] S. Devadas, Hi-Keung Ma, A.R. Newton, and A. Sangiovanni-Vincentelli. "MUSTANG: state assignment of finite state machines targeting multilevel logic implementations". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 7.12 (Dec. 1988), pp. 1290–1300. ISSN: 0278-0070. DOI: 10.1109/43.16807.

[5] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. "Cross-Kernel Control-Flow-Graph Analysis for Event-Driven Real-Time Systems". In: *2015 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES '15)*. (Portland, Oregon, USA). New York, NY, USA: ACM, June 2015. DOI: 10.1145/2670529.2754963.

[6] Martin Hoffmann, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Rüdiger Kapitza, Olaf Spinczyk, and Daniel Lohmann. "Effectiveness of Fault Detection Mechanisms in Static and Dynamic Operating System Designs". In: *17th IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '14)*. (Reno, Nevada, USA). IEEE, 2014, pp. 230–237. DOI: 10.1109/ISORC.2014.26.

[7] Martin Hoffmann, Florian Lukas, Christian Dietrich, and Daniel Lohmann. "dOSEK: The Design and Implementation of a Dependability-Oriented Static Embedded Kernel". In: *21st IEEE Int. Symp. on Real-Time and Embedded Technology and Applications (RTAS '15)*. Washington, DC, USA: IEEE, 2015.

[8] John Hopcroft. *An $n \log n$ algorithm for minimizing states in a finite automaton*. Tech. rep. Computer Science Department, University of California, 1971.

[9] Tae-Hyung Kim and Seongsoo Hong. "State Machine Based Operating System Architecture for Wireless Sensor Networks". In: *Parallel and Distributed Computing: Applications and Technologies*. Ed. by Kim-Meow Liew, Hong Shen, Simon See, Wentong Cai, Pingzhi Fan, and Susumu Horiguchi. Vol. 3320. LNCS. Springer Berlin Heidelberg, 2005, pp. 803–806. ISBN: 978-3-540-24013-6. DOI: 10.1007/978-3-540-30501-9_158.

[10] Nupur Kothari, Todd Millstein, and Ramesh Govindan. "Deriving State Machines from TinyOS Programs Using Symbolic Execution". In: *IPSN '08: 7th Int. Conf. on Information Processing in Sensor Networks*. Washington, DC, USA: IEEE, 2008, pp. 271–282. ISBN: 978-0-7695-3157-1. DOI: 10.1109/IPSN.2008.62.

[11] Enno Lübbers and Marco Platzner. "ReconOS: Multithreaded Programming for Reconfigurable Computers". In: *ACM Trans. on Embedded Computing Systems (TECS)* 9.1 (Oct. 2009), 8:1–8:33. ISSN: 1539-9087. DOI: 10.1145/1596532.1596540.

[12] Edward F. Moore. "Gedanken-experiments on sequential machines". In: *Automata studies*. Annals of mathematics studies, no. 34. Princeton University Press, Princeton, N. J., 1956, pp. 129–153.

[13] OSEK/VDX Group. *Operating System Specification 2.2.3*. Tech. rep. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2014-09-29. OSEK/VDX Group, Feb. 2005.

[14] Fabian Scheler and Wolfgang Schröder-Preikschat. "The RTSC: Leveraging the Migration from Event-Triggered to Time-Triggered Systems". In: *13th IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '10)*. (Carmona, Spain). Washington, DC, USA: IEEE, May 2010, pp. 34–41. ISBN: 978-0-7695-4037-5. DOI: 10.1109/ISORC.2010.11.

[15] Jim Turley. "Operating Systems on the Rise". In: *embedded.com* (June 2006). http://www.eetimes.com/author.asp?section_id=36&doc_id=1287524. URL: http://www.eetimes.com/author.asp?section_id=36\&doc_id=1287524.

[16] D. Varma and E.A. Trachtenberg. "A fast algorithm for the optimal state assignment of large finite state machines". In: *Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on*. Nov. 1988, pp. 152–155. DOI: 10.1109/ICCAD.1988.122483.

[17] T. Villa and A. Sangiovanni-Vincentelli. "NOVA: State Assignment of Finite State Machines for Optimal Two-level Logic Implementations". In: *26th ACM/IEEE Design Automation Conference*. (Las Vegas, Nevada, USA). DAC '89. New York, NY, USA: ACM, 1989, pp. 327–332. ISBN: 0-89791-310-8. DOI: 10.1145/74382.74437.

[18] Collin Walls. *The Perfect RTOS*. Keynote at embedded world '04, Nuremberg, Germany. 2004.

[19] Christopher Zimmer, Balasubramanya Bhat, Frank Mueller, and Sibin Mohan. "Time-based Intrusion Detection in Cyber-physical Systems". In: *1st ACM/IEEEInt. Conf. on Cyber-Physical Systems*. ICCPS '10. Stockholm, Sweden: ACM, 2010, pp. 109–118. ISBN: 978-1-4503-0066-7. DOI: 10.1145/1795194.1795210.

Partial Paging for Real-Time NoC Systems

Adrian McMenamin and Neil C. Audsley
Department of Computer Science, University of York, UK
email: [acm538,neil.audsley]@york.ac.uk

***Abstract*—In multiprocessor Network-on-Chip (NoC) architectures it is common that CPU local memory space is limited, with external memory accessed across the NoC infrastructure. Clearly it is imperative for real-time performance that local memory is used effectively, with code and data moved from external memory when required. One common approach is for the local memory to be comprised of two levels, ie. cache and memory. Software mechanisms are used to move code and data between local memory and external memory, eg. scratchpad mechanisms. In this paper we explore the issue of using paging to supplement this approach, ie. a hardware mechanism to automate movement of code and data between external memory and per-CPU local memory within the NoC. This has wide-ranging potential benefits in from efficiency and real-time performance, through application programmability (ie. potential support of logical address spaces). However, the limited amounts of local memory raise the problem of thrashing. Therefore, we examine the effect of limiting thrashing effects by only loading the parts of pages that are referenced (rather than the entire page). The approach is assessed against a real-time video application, considering different page replacement policies.**

I. Introduction

Both transistor scaling [1] and power density limitations [2] have motivated the move towards multiprocessor architectures. However, it is often not possible to provide the many CPUs within a chip large local memories. In multiprocessor Network-on-Chip (NoC) architectures it is common that CPU local memory space is limited, with external memory accessed across the NoC infrastructure - eg. Tilera [3], Intel SCC [4] and Epiphany [5].

The management of this hierarchical memory architecture efficiently so that real-time performance can be maintained is challenging. We note that this is a historic problem – CPUs speeds have generally increased faster than memory (and bus) speeds, forming a memory bottleneck as systems had to wait excessive times for new code and data to be loaded from slower layers in the memory hierarchy. If management of the memory hierarchy is not sufficient, then the overall architecture will spend more time moving code and data between local and external memory than actually computing – the phenomenon of "thrashing" [6].

The most efficient way of populating this local, faster, memory uses the optimal paging algorithm (OPT) – pages with the longest reuse distance are discarded [7]. OPT is "clairvoyant" as it relies on knowledge of future events. While occasionally this knowledge is available to programmers of embedded devices, a more general solution to the problem of thrashing was demonstrated by Denning's "working set" method, which, relying on the strong tendency of computer programs to show locality of reference in the short-term, stipulates that the most effective practical paging policy will be that which retains in memory those pages referenced in the past within a pre-defined time, called the working set window [8]. In fact, Denning's algorithm has proved to be difficult or impractical to implement, but most general computing devices and operating systems use an approximation, typically some form of "least recently used" (LRU) algorithm.

This paper explores the issue of using paging within NoC architectures. CPUs within the NoC typically have a cache and a small bank of SRAM. Large DRAM banks and permanent storage are available externally, accessed via the NoC mesh [3], [4]. Memory resources on the chip are limited — but time to access external memory is much higher than local memory (partly due to contention over the shared NoC mesh). As a consequence the problem of thrashing reappears. Therefore we examine the effect of limiting thrashing effects by only loading the parts of pages that are referenced (rather than the entire page). The approach is assessed against a real-time video application, considering different page replacement policies.

In section 2 we review relevant related work. In section 3 we model the performance of conventional paging systems. Sections 4 and 5 introduce a new approach where only part of a page is loaded. Section 6 offers a discussion and conclusions.

II. Related Work

The wide variety of parallel programming frameworks is perhaps a testimony to the essential difficulty of programming parallel systems. The problems, such as the limitation imposed by the need for at least some code to be serial - "Amdahl's Law" [9] - as well as the difficulties of maintaining coherence and efficiency across a large number of centres of execution are familiar. They are joined by the need to master a novel technology when considering NoC systems. As the authors of [10] state, it has been difficult to "make it easy to write programs that execute efficiently on highly parallel computing systems." Perhaps this is one reason why research has tended to concentrate on the use of NoCs as specialist accelerators [11]. This is also true of researchers' discussions of virtual memory use on NoCs. For instance, in [12] the authors discuss an efficient caching scheme to accelerate sorting.

Other researchers have examined how memory management for GPUs, which, while being "single instruction, multiple data" devices unlike the "multiple instruction, multiple data" devices we are considering, have much in common with NoCs. In [13] it is noted that OPT is not, in fact, optimal when the size of the working set of the data is much greater than the available local memory capacity. In [14] a method of improving cache performance by dynamically altering memory reuse distance is discussed.

Recent research into paging systems has concentrated on large memory systems. While, in [15], it was shown that smaller page sizes could reduce the fault count, more recent research, such as [16], has emphasised that, with large quantities of physical memory (relatively) cheaply available, minimising the cost of translation between virtual and physical addresses larger page sizes are better options to speed up computing in common use domains.

In [17] alternatives to traditional hardware designs to support virtual memory are explored and a model proposed that saves power and adds flexibility to operating system design.

III. Modelling the Performance of Paging Systems

Standard paging approaches move whole pages of code and data *en bloc* the memory hierarchy. This allows a logical address space to be presented to the application programmer – the familiar abstraction of a single and unified address space. However, this is not common within real-time systems (and largely unsupported on existing NoCs). The remainder of this section considers a standard real-time application and assesses its performance with respect to paging.

The x264 program from the PARSEC benchmark suite [18] was used. It was configured to run with a maximum of 16 threads (as we proposed to model a system with 16 cores) – note 18 threads in total were created, though simulations run no more than 16 at once.

Running the benchmark under a modified version of the Valgrind Lackey program [19], we could separate the memory references of each thread of execution and classify every such reference as one of the following:

- *instruction* – like a *load* sees a memory location is accessed but not modified;
- *store* – where a memory location is written to;
- *modify* – a location is first accessed and then written to in a single interaction.

Whilst every *instruction* has an initial impact similar to that of a *load* (in that the address of the instruction itself must be accessed), an instruction may also cause consequent *loads*, *stores* or *modifies*. Additionally, the point at which each new thread was released was marked.

The modified Lackey program produced an XML stream recording every memory access by every thread in time order. This is then used to model different models of on- and off-chip memory interaction and storage. The XML stream recorded the order in which memory addresses were accessed by each thread but contained no specific timing information and thus did not record any delays for thread synchronisation - so by its nature any processing of the XML could only be an approximation of how different paging policies would behave.

The modelled hardware system has 16 cores, each with 32KB of local memory (forming a 512KB pool of on-chip memory), this was loosely based on the Tilera example [11]. We assumed that all on-chip memory was immediately (i.e., in one "tick") available to all cores (i.e., we ignored both the issues of on-chip synchronisation and on-chip communication delays) and assumed that a standard cache line of external memory (128 bits, or 16 bytes) was available after a delay of 100 cycles/ticks. So, for instance, a 4KB page would take 25,600 ticks to load. The experiment does not model caching behaviour or the costs of writing-back modified pages as these aspects do affect the broad behaviour of the NoC model when using paged memory.

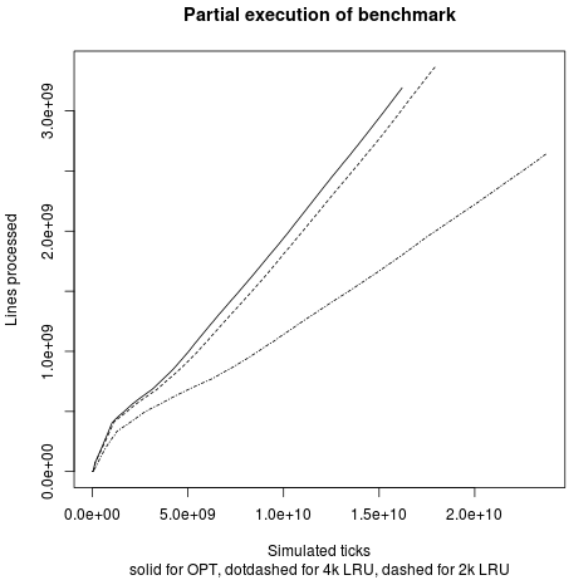


Figure 1. OPT and LRU compared

Our central finding was that FIFO, LRU (including LRU 2Q varieties) and even OPT replacement policies all showed the characteristics of thrashing as the system became memory I/O bound. Additional CPUs did not speed the system up, rather slowing each individual CPU as they were constrained by the small overall pool of memory[1].

Figure 1 shows the simulated performance of OPT and LRU for 4KB pages and also the performance of an LRU algorithm with 2KB page sizes[2]. The number of lines processed indicates progress in completing the benchmark, while the simulated ticks is an analogue for time. It will be seen that although using 2KB page sizes increases performance (despite resource restraints), all the lines, including that for OPT, display a common characteristic - that the rate of progress becomes constant. As Figure 2 shows, applying more CPUs to the task does not speed up its execution: the lines processed per simulated tick remaining constant even as more threads are being executed and more processors are being used. The graph shows that the simulated system is memory I/O bound: additional CPUs cannot squeeze any more computing power from the system as they simply fight each other for access to the limited memory pool.

[1]The model employed barrier synchronisation and if two threads both requested the same page both would gain access to it when it loaded on the earliest request. Threads simply blocked when waiting.

[2]To compensate for the additional size and cost of page tables that 2KB pages would require we allocated 30KB per core and increased the access time to 2 ticks for a present page.

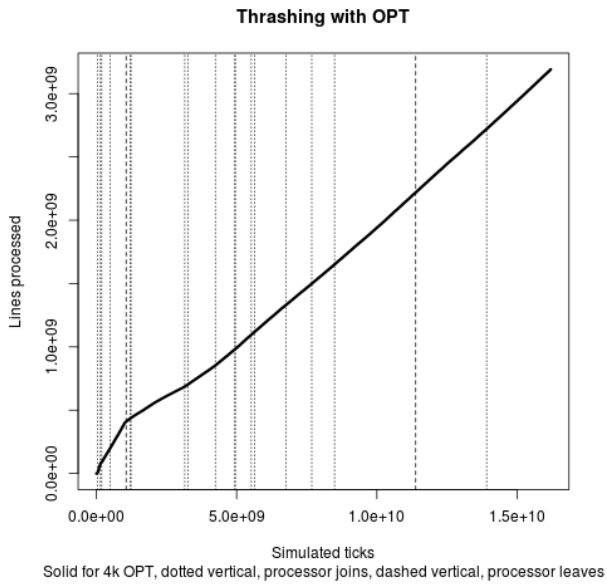


Figure 2. OPT algorithm: more processors do not speed execution

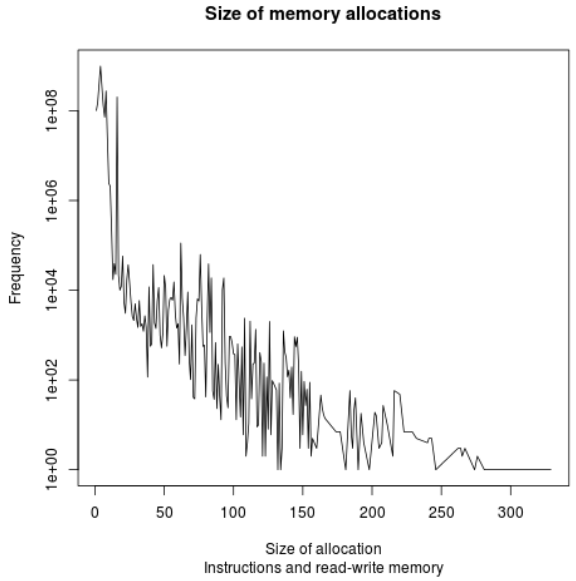


Figure 3. Logarithmic plot of the frequency of different sizes of contiguous memory allocations

IV. Partial Paging Approach

Figure 3 shows small (16 bytes or fewer) contiguous memory allocations were orders of magnitude more likely than larger allocations. Since pages were being pushed out quickly, we tested the proposition that a **partial paging allocation policy** – pages are populated one cache line (ie.16 bytes) at a time – could improve performance.

In this case we used 2KB pages and 30KB per core, with a cost of four ticks to access a present memory block and we tracked whether a given 16 byte block was present through a bitmap. The result, seen in Figure 4, was improved performance: as more threads are executed and additional CPUs used, the processing rate increases – mitigating thrashing.

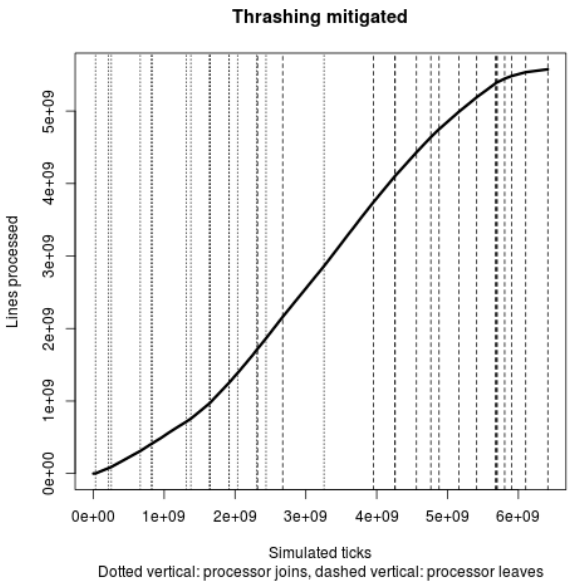


Figure 4. Partial paging: additional processors speed execution

A. Testing the Partial Paging Approach

The partial paging approach was tested using the OVPSim instruction accurate simulator [20] with MicroBlaze soft CPU [21] which delivers one instruction per cycle, enabling instruction count to be a good approximate to cycle counting.

1) Unmodified Microblaze: Each thread's XML output from the modified Valgrind Lackey was converted into MicroBlaze memory load and write instructions and was executed using simple page tables. In an unmodified MicroBlaze such code will continue to run (assuming no other problems) so long as a translation lookaside buffer (TLB) is able to translate the virtual address being accessed into a physical address. If address was not translatable by a TLB then an exception would be raised – ie. when the memory being accessed is not available "locally" (as though in the on-chip pool) and so must be copied from a "remote" address.

Three TLB entries were "pinned" (ie. made permanent and unchangeable), so ensuring the code providing basic VM services and the generated code, the page tables and the page frames would always have appropriate translations.

The system was configurable, eg. to have more page frames of physical memory than TLB entries. However within this paper we focus on the case where the number of page frames of physical memory was the same as the number of TLB entries (up to the maximum supported 64 TLB entries). In this case every TLB miss corresponds to a "hard fault" – ie. it requires a new page to be loaded into physical memory and, in all cases after the system has used all available physical memory, the eviction of a currently present page[3].

The demand paging FIFO page replacement system was tested to determine the fault count of 4KB and 1KB pages (the two smallest sizes supported on the MicroBlaze). As can

[3]The MicroBlaze has no timing device within OVPSim with so eviction policies followed a "first-in, first-out" (FIFO) policy as opposed to the more efficient CLOCK-type LRU approach

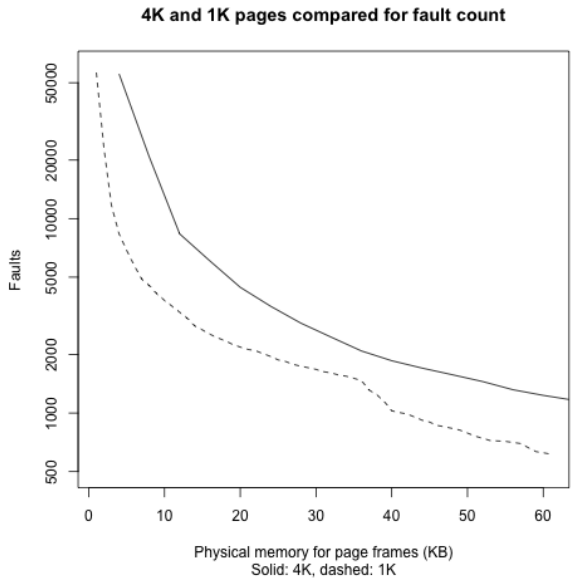


Figure 5. Fault count for traditional paging approach for different page sizes

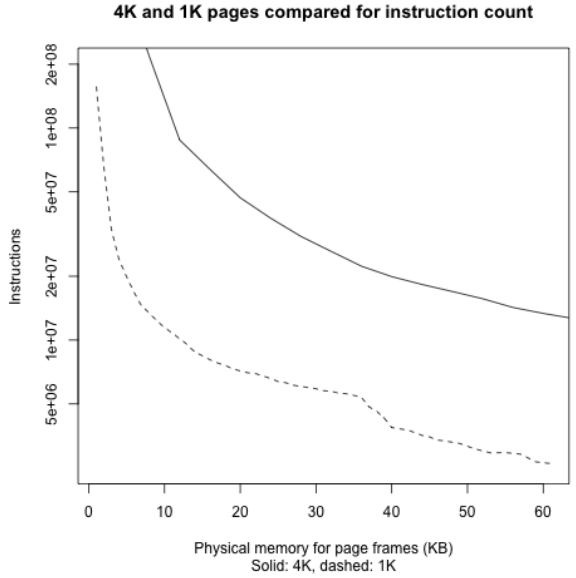


Figure 6. Instructions required to complete task

be seen in Figures 5 and 6, for a fixed amount of local memory, the 1KB pages delivered a lower fault count and required many fewer instructions to be executed to complete the task.

On each page fault that led to an eviction, as well as executing code to manage the page tables, the system was required to write back an evicted page as well as copy the incoming page into memory designated as holding a "local" page frame - no DMA functions were available on this simple model and so this was all carried out in assembly loops that copied memory from one address to another. As can be seen in Figure 6, this made the 1KB page model substantially more efficient than even the lower fault count along might suggest: there were fewer faults and each cost less to handle. At this point we made no allowance for the cost of transferring

TLBs	Instructions: traditional paging	Instructions: alternative paging
4	157,493,205	n/a
8	20,219,450	18,545,020
12	12,651,719	14,717,082
16	9,930,702	13,457,998
20	8,215,518	12,614,663
24	7,457,021	12,270,902
28	6,844,912	12,079,901
32	6,468,068	11,834,218
36	6,140,900	11,717,928
40	5,329,413	11,558,408
44	4,226,715	10,619,623
48	3,897,005	10,453,064
52	3,651,137	10,315,069
56	3,322,324	10,092,510
60	3,296,123	10,076,433
64	2,991,081	9,910,243

Table I
INSTRUCTION COUNTS FOR "TRADITIONAL" AND "ALTERNATIVE" 1KB PAGING SYSTEMS

memory from a "remote" to a "local" address, merely counting the number of instructions required to execute the copy.

2) Microblaze with Partial Paging: The OVPSim Microblaze code was modified to include partial paging – ie. pages loaded in 16 byte blocks. Now, while a TLB miss exception would be thrown in the normal way if an address translation was not available, each reference to an address mapped to "local" memory would raise an interrupt. The interrupt handler then would check a bitmap to see if the addressed 16 byte block has been loaded from remote memory to local memory. If it has no further action was taken and the interrupt handler returns, if it has not then a "small fault" is raised and the appropriate 16 byte line loaded, bitmap updated, and the interrupt handler returns. This means a substantial code block was executed on every memory reference, though the code executed when the fragment being accessed was present was significantly shorter than when it was missing. Hard faults still occur and in most cases (after the initial period when empty physical pages are being written to) require a page write-back (again, we did this for all pages) as well as a low cost bitmap reinitialisation. In such cases, only those 16 byte lines marked as present are written back. On a hard fault only the initially requested 16 byte block was loaded.

As Table I[4,5] shows, comparisons show higher instruction counts for all but the smallest amounts of available local memory. However instruction counts do not provide a full comparison between the two systems. Although partial paging generally executes more instructions to complete the task, it also loads smaller amounts of memory. Each fault on a 1KB traditional system requires a minimum of a 1KB page load - typically costing somewhere between 4800 cycles (if global memory is 75 cycles "away") and 8000 cycles (if global memory is 125 cycles per 16 byte cache line away).In contrast the alternative system only needs to load those lines it requires.

Partial paging shows superior performance when the timing

[4]For the traditional system three TLBs are pinned so, for instance 16 TLBs leaves 13KB for physical pages, for the alternative system four TLBs are pinned and 16 TLBs leaves 12KB for physical pages

[5]The bitmaps were pinned in memory, so losing a further TLB entry and so the alternative system needs a minimum of 5KB or 5 TLBs

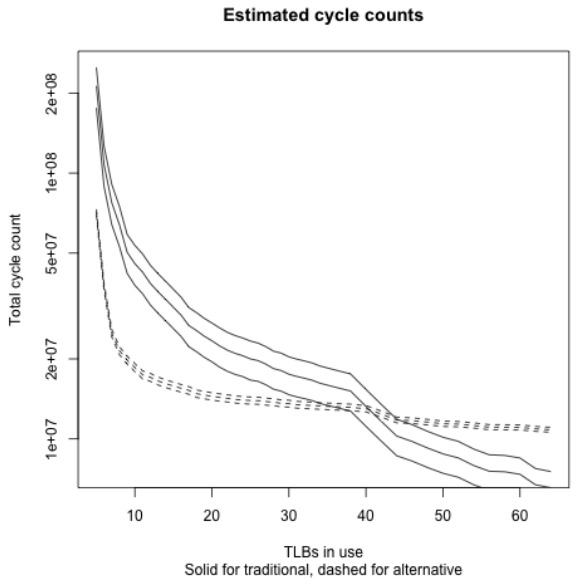


Figure 7. Estimated total cycles required by different paging algorithms: in each case the top line is for global memory 125 cycles away and the bottom 75 cycles away

TLBs in use	(Hard) Faults	Instructions
8	88,875	249,531,853
16	71,404	222,944,264
24	63,276	205,776,575
32	56,527	194,463,472
48	47,217	180,191,027
64	40,905	171,472,116

Table II
FAULT AND INSTRUCTION COUNT RESULTS FOR ILLUSTRATIVE LOW LOCALITY LOAD (TRADITIONAL PAGING)

is normalised. Figure 7 illustrates: the estimated total cycles required if global memory access cost is 75, 100 and 125 cycles per 16 byte cache line is compared for the two algorithms. Here partial paging requires fewer cycles (for this memory access pattern) when local memory is around 32KB or less. The flat performance profile of partial paging suggests this is dominated by the interrupt handler code rather than the number of faults (completing the task requires a set number of memory accesses and so the handler is run a set number of times regardless of the number of TLB entries in use). Improving the performance of this part of the process, such as making the checking of the bitmap a sub-cycle task in hardware could dramatically increase the advantage of the alternative approach.

TLBs in use	Hard and small faults	Instructions
8	113,150	108,389,860
16	112,668	108,556,316
24	112,134	109,586,781
32	111,594	110,708,748
48	110,261	114,716,046
64	108,769	118,125,334

Table III
FAULT AND INSTRUCTION COUNT RESULTS FOR ILLUSTRATIVE LOW LOCALITY LOAD (PARTIAL PAGING)

It should be further noted that, as we did not differentiate between page types[6], we did not account for the cost of writing back pages in this comparison, beyond the instructions required to be executed: such a count would certainly increase the advantage of partial paging. For instance, with 32 TLB entries, the average page has 144 bytes loaded on eviction and so only nine 16 byte blocks would need to be written back. The use of instruction count for comparison does account for the relative complexity of the two situations: in the case of the alternative approach the bitmap must be read to decide which blocks are to be written back.

We further tested the partial approach with a semi-randomised[7] selection of pages and, unsurprisingly, the partial paging approached showed a very strongly enhanced performance, as illustrated in Tables II and III.

V. POTENTIAL ADDITIONAL ADAPTATIONS

We were able to consider some additional adaptions to the partial paging algorithm.

A. Testing other loading sizes

Partial paging was tested with 32 byte and 64 byte loads. Such larger loads reduce the number of small faults and Table IV summarises the results. The marginal efficiency of the larger loads increases with the amount of TLB entries in use - for 8 TLB entries there are 2.9 more small faults with a 16 byte load size than for a 64 byte load size, while for 32 TLB entries the ratio is 3.1:1 and for 64 it is 3.2:1, but the gains are not dramatic and, given that the number of interrupts raised is the same regardless of the load size used then it is plain that, without hardware adaption, there is no benefit to using larger load sizes.

TLBs in use	Hard faults	Small: 16 bytes	Small: 32 bytes	Small: 64 bytes
8	8357	21122	12612	7375
12	4526	18858	10953	6249
16	3301	17209	9988	5702
20	2543	15822	9105	5203
24	2184	15144	8651	4936
28	1956	14688	8377	4763
32	1741	13893	7876	4472
36	1609	13400	7557	4272
40	1469	12866	7230	4079
44	1027	10623	5983	3367
48	919	10183	5733	3219
64	626	8513	4764	2649

Table IV
FAULT COUNTS FOR DIFFERENT LOAD SIZES COMPARED

B. Moving from FIFO to LRU

The presence of an interrupt on every memory access does allow experimentation with an LRU page replacement policy – noting additional costs of management of page lists etc.

[6]We could have assumed that no instruction pages were to be written back but for the sake of simplicity we treated all pages in the same way, so write-back code is executed for all pages

[7]Pages were selected from the same range of addresses and with approximately the same frequency and with allocation sizes modelled on the results shown in Figure 3, but with no stronger bound of locality.

We tested two forms of LRU: a partial policy where the page order was updated only on a hard or small fault, and a full LRU where the page list order was updated on every access. The results are summarised in Table V – both approaches significantly lower the total fault count compared to FIFO. For a 32 TLB system (ie. with 28KB of local memory), there are 9% fewer faults with the partial approach and 25% fewer with the full LRU policy. These would save 142,500 cycles and 388,100 cycles respectively in load time from global memory 100 cycles away. However, the cost of implementing the LRU policies in additional instructions greatly outweigh these, as shown in Table VI. The high cost of manipulating the ordered list decisively counts against the full LRU approach in particular.

TLBs	FIFO	Partial LRU	Full LRU
16	20510	18847	16492
32	15634	14209	11753
48	11102	10337	8355
64	9139	8762	7455

Table V
FAULT COUNTS- HARD AND SMALL COMBINED - FOR DIFFERENT PAGE REPLACEMENT ALGORITHMS

VI. CONCLUSIONS

Virtual memory has been part of the standard programming toolkit for around half a century. In recent years much research focus has been on how to improve the performance of machines with large amounts of memory, yet, at the same time, a problem from the dawn of virtual memory - thrashing - has also reappeared, especially in devices that might be otherwise expected to run highly parallel real time computing tasks, such as video processing, at speed. Our simulations suggest that such systems, if using virtual memory, could improve performance by both using smaller page sizes (and so travel in the opposite direction of systems processing "big data") and adopt a new sub-paging approach of loading in memory in cache line size blocks. However, our initial research also suggests that significant speed improvements will only come if we can match the bitmaps that record which parts of a page have already been populated to accessed addresses in hardware and thus sub-cycle.

We propose that such hardware adaptions would be possible: hardware memory management units (MMU) have long supported address translation and lookup on a sub-cycle basis. We have adopted a bitmap as an efficient method with which to map internal memory allocations in software, but it may be that other methods are more hardware efficient. In [22] a hardware bitmap-based memory allocator is discussed, while [23] discusses an MMU designed specifically for system-on-chip hardware.Further work includes investigation to see if a suitable hardware modification can be made (using an FPGA based software). This can then be used within an existing NoC architecture to evaluate the approach fully.

TLBs	FIFO	Partial LRU	Full LRU
16	13,457,998	15,448,631	31,501,330
32	11,834,218	14,951,934	43,505,073
48	10,453,064	13,852,267	55,552,516
64	9,910,243	13,728,863	68,390,135

Table VI
INSTRUCTIONS EXECUTED FOR EACH PAGE REPLACEMENT ALGORITHM

REFERENCES

[1] Ethan Mollick, "Establishing Moore's Law", *IEEE Ann. Hist. Comput.*, vol. 28, no. 3, pp. 62–75, 2006, 1158837.

[2] M. Bohr, "A 30 year retrospective on Dennard's MOSFET scaling paper", *Solid-State Circuits, IEEE*, vol. 12, no. 1, pp. 11–13, 2007.

[3] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect", in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, feb. 2008, pp. 88–598.

[4] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the Intel SCC many-core processor", in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, july 2011, pp. 525–532.

[5] Adapteva, "Epiphany Architecture Reference", http://adapteva.com/docs/epiphany_arch_ref.pdf, 2015.

[6] Peter J Denning, "Virtual memory", *ACM Computing Surveys (CSUR)*, vol. 2, no. 3, pp. 153–189, 1970.

[7] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer", *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, June 1966.

[8] P. J. Denning, "Working Sets Past and Present", *IEEE Trans. Softw. Eng.*, vol. 6, no. 1, pp. 64–84, January 1980.

[9] M.D. Hill and M.R. Marty, "Amdahl's Law in the Multicore Era", *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[10] R. Bodik B. C. Catanzaro J. J. Gebis P. Husbands K. Keutzer D. A. Patterson W. L. Plishker J. Shalf S. W. Williams K. Asanovic and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley", Tech. Rep. UCB/EECS- 2006-183, EECS Department, University of California, Berkley, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html, December 2006.

[11] D. Ungar and S. Adams, "Hosting an object heap on manycore hardware: an exploration", *SIGPLAN Not.*, vol. 44, no. 12, pp. 99–110, 2009.

[12] Alessandro Morari, Antonino Tumeo, Oreste Villa, Simone Secchi, and Mateo Valero, "Efficient sorting on the tilera manycore architecture", in *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*. IEEE, 2012, pp. 171–178.

[13] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt, "Cache-conscious wavefront scheduling", in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 72–83.

[14] X. Chen, L. Chang, C. I Rodrigues, J. Lv, Z. Wang, and W.i Hwu, "Adaptive cache management for energy-efficient gpu computing", in *Microarchitecture Annual IEEE/ACM Int. Symp. on*. IEEE, 2014, pp. 343–355.

[15] Donald J. Hatfield, "Experiments on page size, program access patterns, and virtual memory performance", *IBM Journal of research and development*, vol. 16, no. 1, pp. 58–66, 1972.

[16] A. Basu, J. Gandhi, J. Chang, M. Hill, and M. Swift, "Efficient virtual memory for big memory servers", in *ACM SIGARCH Computer Architecture News*. ACM, 2013, vol. 41, pp. 237–248.

[17] B. Jacob and T. Mudge, "Uniprocessor virtual memory without tlbs", *IEEE Trans. on Computers*, vol. 50, no. 5, pp. 482–499, 2001.

[18] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li, "The parsec benchmark suite: Characterization and architectural implications", Tech. Rep. TR-811-08, Princeton University, January 2008.

[19] Nicholas Nethercote and Julian Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation", in *ACM Sigplan notices*. ACM, 2007, vol. 42, pp. 89–100.

[20] OVPWorld.org, "Open virtual platforms (ovp) an introduction and overview".

[21] xilinix.com, "Microblaze soft processor core".

[22] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles, "Dynamic storage allocation: A survey and critical review", in *Memory Management*, pp. 1–116. Springer, 1995.

[23] Mohamed Shalan and Vincent J Mooney, "A dynamic memory management unit for embedded real-time system-on-a-chip", in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, 2000, vol. 17, pp. 180–186.

Transactional IPC in Fiasco.OC
Can we get the multicore case verified for free?

Till Smejkal, Adam Lackorzynski, Benjamin Engel and Marcus Völp

Operating Systems Group
Technische Universität Dresden, Germany
<name>.<surname>@tu-dresden.de

***Abstract*—Already announced in 2007 for Sun's Rock processor but later canceled, hardware transactional memory (HTM) finally found its way into general-purpose desktop and server systems and is soon to be expected for embedded and real-time systems. However, although current hardware implementations have their pitfalls, hindering an immediate adoption of HTM as a synchronization primitive for real-time operating-systems, we illustrate on the example of a transactional implementation of the L4/Fiasco.OC inter-process communication (IPC) how extended versions of HTM may revolutionize kernel design and, in particular, how they may reduce the verification costs of a multi-core kernel to little more than verifying a selectively preemptible uni-processor kernel. Removing L4/Fiasco.OC's half thousand lines-of-code cross-processor IPC path and making the local path transactional, we benefit from a principal performance boost for sending cross-core messages. However for the average case, we experience a 30 % overhead for local calls.**

I. Introduction

Cyber-physical systems such as autonomous cars, medical robots, and airplanes increasingly apply multi-core hardware and multi-core real-time operating systems (RTOS) to meet the performance demand of their applications. At the same time, these systems often operate with or in close proximity of humans, thus safety is a must and formal verification is most rigorous in assuring that a system is to be trusted. However, although fully verified single processor systems are at the verge (first microkernels have already been verified [1], [2]), multiprocessor verification remains a milestone to be taken.

Verification of uniprocessor kernels typically proceeds by splitting the high level verification goal into smaller properties and invariants, which are then shown to hold for arbitrary sequences of non-preemptively executing pieces of kernel code. One, if not the challenge when comparing the verification of multiprocessor kernels with uniprocessor kernels is that non-preemptive execution no longer conveys atomicity at the granularity of non-preemptive execution, but at the granularity of individual processor instructions. Instead of having to consider arbitrary interleavings of large code pieces, one must therefore establish the desired results for all possible interleavings of machine instructions, which easily pushes verification complexity beyond manageable bounds. Of course, there are several tools to assist in this tasks, for example, concurrent separation logic [3] and the multitude of approaches that followed Owicki and Gries [4], [5] seminal work on assume-guarantee reasoning. However, despite these tools, one must still specify and verify the behavior of the kernel at a fine granular and machine-dependent level.

In this paper, we argue why we believe transactions can re-establish some of the simplicity one finds when verifying uniprocessor kernels by reintroducing atomicity at a coarse granularity and, most importantly, in a machine-independent way. Our goal is not to translate uniprocessor results to the multiprocessor case, which if possible at all requires careful argumentation. Instead, we propose to re-implement the kernel as sequences of large transactions to regain the atomicity of non-preemptive execution. We evaluate on the example of L4/Fiasco.OC's IPC path using the hardware transactional memory implementation found in Intel's Haswell processors to which degree this is possible and at which costs.

We present our transactional IPC path in Section III, compare its performance against mainline Fiasco in Section IV and illustrate in a semi-formal way in Section V how lifting atomicity from individual instructions to coarse grain transactions simplifies the multicore verification task to little more than what is required when verifying uniprocessor kernels.

II. Hardware Transactional Memory

In 1993, Herlihy and Moss [6] proposed transactional memory (TM) as a mechanism to assist developers in protecting shared data structure accesses in parallel systems. Unlike lock-protected data structures, which to scale require cumbersome to design and error prone fine-grain locking schemes, transactional memory performs modifications of data structures optimistically but is prepared to discard these modifications in case of conflicts. Especially for low-contended locks, TM avoids the locking overhead at the expense of guaranteed progress in situations where transactions abort.

To implement transactional operations in hardware, i.e., to ensure *atomicity* of updates in case the transaction completes and *isolation* in the sense that modifications remain invisible until the transaction commits, Herlihy and Moss proposed to exploit processor local caches as interim storage and cache coherence protocols (such as MESI) for conflict detection. External writes abort transactions if they are to any data loaded into the cache or accessed while executing transactionally; external reads abort a transaction if they are to cachelines that are cached in exclusive modified state (M) as a result of transactional writes. Further aborts may happen if transactional data exceeds the capacity of the cache or for other reasons that are specific to the concrete implementation of hardware transactional memory (HTM).

Many software implementations of transactional memory have been proposed over the years (see e.g. [7], [8]),

including hybrid hardware-software solutions [9]. However, their applicability is limited due to significant overheads as identified by Cascaval et al. [10]. The first full fledged hardware implementation as described by Herlihy and Moss has been announced in 2011 for IBM's BlueGene-Q servers [11], followed by Intel's Transactional Synchronization Extension (TSX) [12] for standard PC hardware in 2012.

TSX offers two distinct features: *Hardware Lock Elision* and *Restricted Transactional Memory*. Hardware lock elision [13] automatically replaces locks with transactions by replacing the acquisition of the lock with a transaction begin and the release with an attempt to commit the transactional state collected while executing the critical section. In contrast, restricted transactional memory (RTM) exposes the complete transaction interface to the programmer allowing her to start, commit and abort transactions through special processor instructions. The limitations of RTM are conflict detection only at the granularity of cachelines but no finer, the bounded amount of memory that can be accessed from within a transaction, and, as far as Intel's implementation is concerned, the lack of any progress guarantee with regard to which transaction will abort. In particular, RTM aborts transactions on interrupts, system calls and in many other situations, including the execution of some privileged instructions.

Our main focus in this paper is on safety, security and correctness but not on lifeness and guaranteed completion. Nevertheless, we will argue why transactions should be considered as a mechanism to simplify the kernel and why real-time systems require future implementations of hardware transactional memory to convey progress guarantees similar to those provided by IBM in BlueGene-Q.

III. Transactional Inter-Process Communication

With *TxLinux* Ramadan et al. [14] have already shown the value of hardware transactional memory for synchronizing access to kernel data structures. However, to leverage the full potential of HTM for both simplifying in-kernel locking and verifying multi- and manycore kernels, all system calls must execute transactionally, at least to the best degree possible.

To demonstrate the feasibility (and drawbacks) of almost fully transactional system calls, we use as an example an implementation of L4/Fiasco.OC's IPC path with Intel's RTM.

A. The L4/Fiasco.OC Microkernel

L4/Fiasco.OC is a 3rd-generation capability-based microkernel designed for use in both security and real-time critical scenarios. Following Liedtke's design principle [15], the L4 family microkernel provides only those functionality in the kernel, which cannot sensibly be implemented at application level. This is the functionality required to isolate user-level subsystems (capabilities and address spaces) and inter-process communication (IPC), which provides a safe and secure means for communicating between these subsystems.

IPC messages in L4 may contain both data and capabilities, which are required to invoke kernel-implemented objects (such as IPC gates to send messages to other threads). IPC is synchronous, that is the sender blocks until the receiver is ready to receive, which removes buffer allocation from the IPC path and allows the threads' user-level control block to be used as message buffer. Through IPC operations, threads may *send* or *receive* messages or they may *call* other threads, which is an atomic send and receive operation in the sense that when the callee receives the message, the caller is already ready to receive from this thread. IPC is transparent, that is IPC uniformly works in the same way irrespective of the core on which the receiver is executing. It may be on the same core, in which case we say IPC is local or on a different processor core than the sender, in which case IPC is cross processor.

Mainline L4/Fiasco.OC [16] comes with two tightly integrated IPC paths: a fast path for core-local communication and a cross-processor IPC path, designed to preserve the performance of local IPC as much as possible. In this paper, we explore how IPC and especially cross-processor IPC can be implemented with HTM mechanisms. Besides simplifying the cross-processor IPC path (when compared to existing non-TM approaches), we show that, with a few exceptions, transactions span the same parts of the code that executes non-preemptively in the core-local case. For these exceptions, we explain why they have to execute non-transactional and sketch how one can further reduce the amount of non-transactional code.

B. IPC with RTM

Ideally, from the viewpoint of verifying the kernel and to minimize transaction overhead, the entire IPC operation should be a single transaction. However, there are two general obstacles, which prevent us from turning IPC and, more generally, system calls into a single transaction each: (i) privileged instructions and device accesses abort transactions unconditionally; and (ii) transactional state may become too large to fit the L1 cache, which also leads to aborts. In the L4/Fiasco.OC IPC path, the transaction-aborting operations are the programming of timeouts, which involves setting the hardware-timer to the earliest pending timeout, and the reloading of the page-table base register, when IPC switches to a thread in another address space. In addition, on architectures such as ARM, the transfer of memory capabilities causes aborts when TLB entries have to be flushed as a result of upgrading page-table entries.

Fig. 1 shows a schematic of the L4/Fiasco.OC IPC path and the steps involved when the left-hand thread calls either one of the two right-hand side threads (in the same or in a different address space). Immediately after entering the kernel (e.g., with `sysenter` on x86-systems), execution may proceed transactionally (with `xbegin`) after setting the address of the abort handler (black dot #1 in Fig. 1). IPC proceeds by checking whether the receiver is waiting for the sender (i.e., it has already executed a receive operation) or whether the receiver is still involved in other operations (e.g., it may be running). In the first case, the sender and receiver rendezvous and the kernel starts the message transfer. After the transfer completes, which in case of a transfer of memory capabilities may require additional preemption points and hence transactions, the caller prepares its receive phase to ensure that it is ready to receive when the receiver replies. In case of capability transfers, the TLB shootdown can either be deferred to after the IPC operation or handled immediately after the capability transferring transaction commits. Switching to the receiver involves storing and reloading the register state and stack pointers of the IPC partners. These operations can be

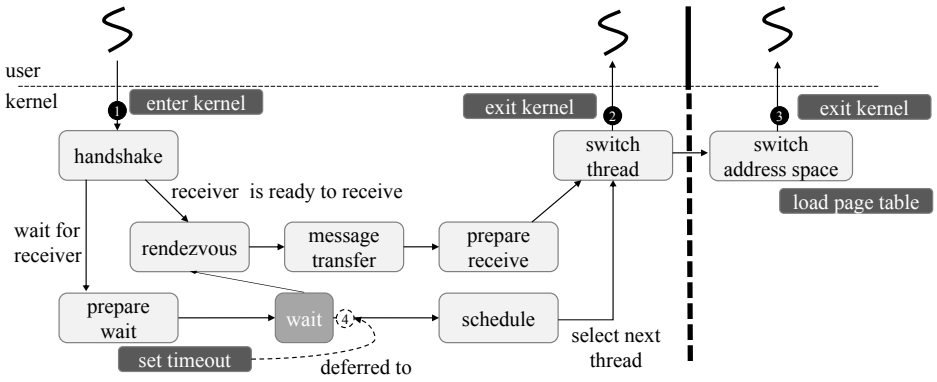


Fig. 1. Schematics of the L4/Fiasco.OC IPC path for an IPC call operation to a peer thread within the same (middle thread denoted by the wiggling line) or another (right thread) address space, that is intra vs. inter address space communication. Black (and dashed white) dots mark the begin and end of transactions. The path either directly proceeds to the receiver or stops at the preemption point *wait*. As part of waiting, the scheduler is invoked to select the next ready-to-run thread to which it then switches. Dark gray operations are privileged operations, which cannot be executed transactionally. They are deferred until after the end of the transaction.

executed transactionally without risk of abort. Therefore, when sending to a thread in the same address space, only one transaction is required, unless capability transfers needs to be preemptible. The transaction starts at the black dot $\#1$ and commits immediately before returning to user-level (e.g., with `sysexit`) at $\#2$.

If the receiver resides in a different address space, the page-table base register must be reloaded, which causes an unconditional abort when executed transactionally. Therefore, we defer the actual address space switch to the point in time after the transaction commits at $\#3$ and execute it immediately before returning to the user. The instructions that remain non-transactionally are the check whether an address space switch is pending and the `mov %1, cr3` instruction, which performs the switch of the page table and hence of the address space. For a verification, these arbitrary interleavings of these two instructions of the transactions and other non-transactional code must be considered. However, because only few operations must be deferred. We expect these interleaving to remain within manageable complexity.

So far we have only considered the case where the receiver is ready to receive from the sender and not involved in some other operation. If this is not the case, the sender blocks waiting for the receiver to execute the receive and message transfer[1].

L4/Fiasco.OC limits the time senders have to wait for receivers to participate in the IPC with timeouts. As we have already explained. Timeouts require programming the hardware timer, which is not possible from within a transaction. However, the actual programming of the timer can be deferred to the *wait* preemption point (white-dashed dot $\#4$) and with some additional restructuring of the implementation also to the point in time when the scheduler switches to the next thread to run. Notice, interrupts remain disabled and the timer will be programmed before any user-level code is executed on this core. The programming of the hardware timer is a second case where code must be executed non-transactionally and interleavings must be considered at the instruction level. To become ready to receive, a send timeout can be specified, which the kernel programs by writing to the hardware timer register. Like with the page-table load, we defer this programming of the hardware timer register to the point in time when the transaction is committed.

When enlarging the transaction in the prescribed way, we must of course validate that the transactional state stays small enough to fit in the transaction-storing cache (i.e., L1 in case of Intel Haswell). In addition, we have to ensure that transactions remain small enough to avoid frequent aborts due to conflicts. With the additional preemption point at $\#4$, no capacity aborts occurred and, as we shall see in Sect. IV, the probability of other IPC operations causing retries is little more than $10^{-7}\%$.

C. Manipulating Page-Table Entries Transactionally

One uncertainty that remained from the documentation [17] was whether page-table manipulations in Intel Haswell adhere to the transaction semantics, that is, whether page-table walks by one processor causes aborts of transactions that modify the walked page table. We therefore performed a small test, which transactionally updates the page table on one core while accessing the mapped memory on another core, to confirm that the page-table walker actually triggers aborts. As long as this implementation is maintained, only possibly required TLB shootdowns must remain outside the transaction. Otherwise, if the page-table walker bypasses the transaction mechanism and evaluates transactional state, large parts of the kernel's address space implementation would have to be moved out of the transaction because intermediate state would become visible that gets discarded if the transaction aborts.

IV. Evaluation

Similar to other research in the area of hardware transactional memory, there are two main aspects to consider when evaluating transactions in L4/Fiasco.OC: First, whether it is possible to reduce the complexity of the kernel code, and second, whether the performance of the kernel can be improved or not.

A. Reducing Kernel Complexity

With HTM, writing synchronized code is easier than with traditional locking mechanisms. This characteristic is

[1] For simplicity, Fig. 1 illustrates only the sender-driven part of the IPC path.

mainly related to two aspects: First, difficult problems such as deadlocks, priority inversion, and convoying do not exist with HTM because transactions do not block during their execution. Instead, they execute optimistically and roll back in case of conflict. Second, the programmer needs not to decide which portions of its code can run in parallel and hence which fine-grain lock to use where. Transaction detect automatically and at the granularity of cache lines, whether data accesses conflict. Hence, it is possible to use transactions also for larger critical sections because the conflict sets are determined dynamically. Still short transaction reduce the likelihood of conflict and the aborts they entail.

The IPC mechanism of the L4/Fiasco.OC microkernel already distinguishes core-local from cross-core communication in its locking mechanisms, by requiring that changes of critical process information is performed on the home core of the modified thread. Hence, if two threads perform an IPC operation while on the same core, no synchronization is needed. To protect locally unsynchronized critical IPC state from inconsistent modification during cross-core IPC, the cross-processor IPC path temporarily stops the partner's core to perform the modification there. This operation requires a comprehensive and time intensive synchronization via *inter processor interrupts* (IPI).

With the introduction of RTM in IPC, we removed the restriction that process information can only be changed on the process' home core. Instead, all modifications are executed transactionally. This way, the flow of executing local and cross-processor IPC have become identical and can be handled in one routine. The only remaining difference is in the way how the transitions from the sender context to the receiver context are realized. While the local IPC case requires only a scheduler activation, the cross-processor IPC case still requires an IPI to trigger scheduling on the remote core.

Hence, we were able to remove most of the complexity of cross-processor IPC path and replaced it with a simpler local IPC path. We expect to be able to make similar changes to other kernel routines and thereby further reduce the complexity of the kernel.

Unfortunately, since Intel®'s RTM extension does not provide any progress guarantee for the transactions, our implementation has a significant drawback. We always have to have a fallback mechanism to guarantee completion of all system calls in case of transaction aborts. Our current approach is to first retry the transaction for a couple of times and then to revert to the traditional cross-processor IPC path, which we could have removed otherwise to safe about 400 lines of code. In general, there is no need to abort all transactions, as demonstrated in IBM's HTM implementation [18], where later transactions cannot abort earlier transactions.

In situations where probabilistic completion and progress guarantees suffice, fall-back mechanisms are not required and the IPC operation could simply be aborted if it did not succeed within a limited number of retries. In Table I, we have collected a statistics to determine the number of retries required. During our performance benchmark (see Section IV-B), $8.64 \cdot 10^{-}5\%$ of the IPC operations failed to commit directly and only $1.05 \cdot 10^{-}7\%$ failed after a second attempt. We did not observe an IPC operation that did not complete after two retries.

TABLE I. STATISTICS ABOUT THE ABORT AND RETRY BEHAVIOR OF THE TRANSACTIONS USED IN THE L4/FIASCO.OC KERNEL

Total	Direct Commit	1 Retry	2 Retries	> 2	Fallback
10,446,981,951	10,446,972,918	9022	11	0	0

B. Performance

Yoo and Leis [19], [20] observed for their benchmarks a general performance advantage of using HTM, except in those that required no synchronization in the first place. To see whether, besides the above reduction in code complexity, IPC benefits from similar advantages when the number of parallel operations increases, we have performed the following experiments on an Intel Haswell i4770 running at 3.4 GHz. We expected significant improvements in the multiprocessor case and low overheads for local IPC.

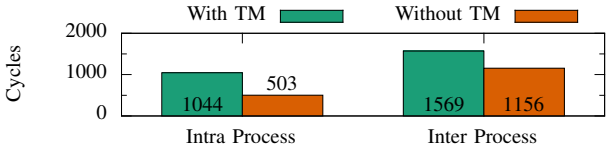


Fig. 2. Minimum number of processor cycles needed to perform a processor local IPC send-receive operation a) within one process (intra process) and b) between two processes (inter process) with the usage of TM and without it. (Deviation in the result is negligible.)

To determine the performance characteristics of local IPC, we measured the costs of transferring an empty message between two threads using an IPC send-receive operation. We compared intra process and inter process communication. As shown in Fig. 2, our implementation introduces a significant overhead of about 107 % for IPC between threads of the same process and of about 35 % for IPC between two processes. Our analysis indicates that the house keeping for the four transactions we need for one IPC send-receive operation introduces this performance decrease. Each transaction costs about 100 cycles.

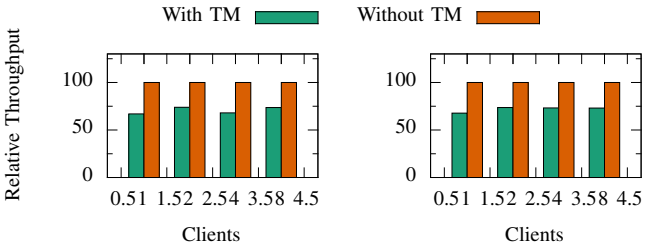


Fig. 3. Average number of full IPC round trips achieved in a second by 1, 2, 4, or 8 clients communicating with one server (left) or equally many servers (right) with the usage of TM relative to the same number without the usage of TM. Deviations in the results are negligible.

While IPC cycle counts reveal raw kernel performance, they generally reveal little insight on application performance. We have therefore also measured two benchmarks, which simulate client-server communication, a scenario common in microkernel-based systems. We measured the relative throughput in IPC send-receive operations between (a) an increasing number of client threads communicating with one server thread and (b) an increasing number of client threads communicating

with dedicated server threads. Fig. 3 shows that the transactional implementation introduces a performance degradation between 28 % and 35 % in all scenarios. This overhead is consistent with the results of the raw IPC performance benchmark. In total, the original local IPC implementation, which requires no further synchronization, performs significantly better than transactional IPC.

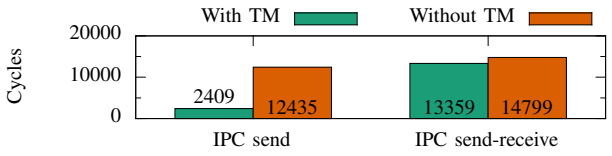


Fig. 4. Minimum number of processor cycles needed to perform a cross-processor IPC operation within one process with a) only sending and b) sending and receiving with the usage of Transactional Memory and without it. Deviations in the results are negligible.

For cross-processor IPC we tried to run a similar benchmark as described above. Unfortunately, this was not possible because this test triggered the RTM implementation bug in Haswell [21]. Our system failed silently. To still provide performance characteristics, we therefore measured the number of processor cycles required for a cross-processor IPC send operation as well as a cross-processor IPC send-receive operation. However, in contrast to the local IPC benchmark, every IPC operation had to wait for a constant time to avoid the above bug. Consequently, the measured values do not present the full potential of our system, but just an indication how future systems behave. As it can be seen in Fig. 4, our implementation performs better than the original code. Especially, the IPC send operation runs up to five times faster than its traditional counterpart. This large difference between the two implementations is mainly because we were able to remove the time expensive IPI from the critical path as we only need it to trigger the rescheduling on the remote core in a *fire-and-forget* fashion. The sender could proceed immediately. For the IPC send-receive operation, we have to wait for one IPI to trigger the scheduling of the receiver and for a second during the reply. As IPI costs dominate IPC send-receive costs, our transactional implementation performs as well but no better than the traditional path. For a saturated server, we expect these costs to be hidden because the server will then find the next request pending when it replies to the current one.

V. Simplifying the Multicore Verification Task

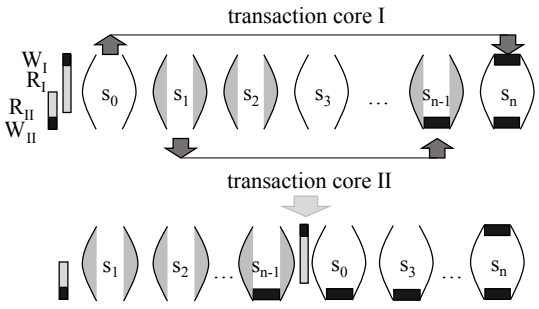


Fig. 5. Parallel interleaved execution of transactional operations exhibits the same visible states as a corresponding sequential execution.

Before we proceed with our argument why we believe that a consequent application of transactions will simplify the multiprocessor verification task to little more than what is required for a uniprocessor kernel, let us clarify our assumptions and goals. Our focus is on verifying multiprocessor kernels, not on translating uniprocessor verification results to a multiprocessor setting, which if at all possible requires additional arguments. We assume transactions to be correct and complete with regard to device side effects. That is, accesses to memory used by the kernel that origin from a device must adhere to the cache protocol and cause aborts if they conflict with transactional kernel state.

Our confidence is based on the following observation. If kernel code executes transactionally, interaction with this code is limited to points in time equivalent to the beginning of the transaction respectively to the time it commits. Any other interaction (by devices or remote cores) will cause an abort and unrolling of transactional state. By "equivalent times" we mean that all interacting write must happen before the transaction reads this data. Our argument, which we are currently transforming into a machine-checked proof, goes as follows. If the majority of the kernel executes transactionally, the trace positions, which characterize the execution of atomic machine instructions, can be rearranged to obtain a trace, which matches the execution behavior of a uniprocessor kernel with selectively preemptible system calls. Instead of considering all possible interleavings at the granularity of atomic machine instructions, it therefore suffices to consider only those interleavings where the instructions inside a transaction execute one after another and without other instructions interleaving. Fig. 5 shows this interleaving and the rearrangement into blocks. It suffices to consider only traces such as the one below, where transactions execute as blocks. Positions of transaction I (white) are combined to a single block and executed after the positions of transaction II (gray).

The rearrangement is possible because we know from the correctness of transactions that cached state becomes only visible if external writes went to a different set of physical addresses than transactional reads or writes. Let R_I, W_I, R_{II} and W_{II} denote these read and write sets for the two transactions. We conclude that for the interleaving of core I and core II, cached state of core II becomes visible only if $W_I \cap (R_{II} \cup W_{II}) = \emptyset$ and likewise for core I if $W_{II} \cap (R_I \cup W_I) = \emptyset$. But then we can shuffle the trace positions such that preserving the order of transactions, all positions of core II (who committed first in this trace) occur before those of core I (who committed last) follow. We realize that these traces are identical with regard to the visible memory updates. Notice in particular that the above address disjointness rules out that core I may depend on the state written by core II (black part in s_0, s_3, s_n) since otherwise core I's transactions would have aborted. But now the trace is identical to a sequential execution of the system calls in a non-preemptive manner while restricting the observation of state to the preemption points.

Notice, for deferred operations, we still require the machinery to verify kernel code at machine granularity. For these, we have to consider all possible interleavings of these instructions and of the transactions at their boundaries. The latter is because we require devices to abort transactions in case of conflict.

VI. Related Work

Ramadan et al. [14] were first to demonstrate the benefit of HTM for synchronizing operating-system code. However, unlike TxLinux, we take a more holistic approach trying to turn every system call into a sequence of transactions to benefit from the simplified interleaving in the verification task. In this regards, our work is more closely related to TxOS by Porter et al. [22] and their attempt to provide transactional kernel behavior for certain mechanisms such as I/O. Of course, there is a large body of work beyond the operating-system kernel. For example, Karnagel et al. [23] and Leis et al. [20] use RTM to improve the performance of in-memory database systems, Kleen [24] extends the GNU C pthreads library to use HTM for transactional synchronization. Ariel et al. [25] formally specify HTM for the purpose of verifying correctness of their implementation, a task which Gupta et al. [26] extend to HTM implementations with non-transactional writes as for example supported in AMD's ASF proposal [27]. To the best of our knowledge this is the first work to realize how a consequent application of HTM can simplify the verification task.

VII. Conclusion and Future Work

In this paper, we have shown how L4/Fiasco.OC's CPU-local IPC path can be converted into an almost completely transactional multiprocessor path. For scenarios where stochastic completion guarantees suffice, we observe a performance improvement in the cross processor case at the costs of significantly increasing uniprocessor costs by almost a factor of 2, and requiring retries in $1.05 * 10^{-7}$ % of all cases. In addition, we have shown how a consequent re-implementation using transactions may simplify the multiprocessor verification task by allowing similar reasoning for the transactions as in the uniprocessor case. Obvious directions for future work include a re-evaluation on newer-generation hardware, progress guarantees for HTM and the lifeness guarantees they entail, and an extension of the described approach to applications and user-level servers.

Acknowledgment

This work is in part funded by the German research council DFG through the cluster of excellence "Center for Advancing Electronics Dresden" cfaed and DFG-SPPEXA's project FFMK.

References

[1] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. ACM, 2009, pp. 207–220.

[2] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework," in *IEEE Security and Privacy*, Oakland, 2013.

[3] S. Brookes, "A semantics for concurrent separation logic," *Theor. Comput. Sci.*, vol. 375, no. 1-3, pp. 227–270, Apr. 2007.

[4] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs," *Acta Informatica*, vol. 6, pp. 319–340, 1976.

[5] ——, "Verifying properties of parallel programs: an axiomatic approach," *Communications of the ACM*, vol. 19, pp. 279–285, 1976.

[6] M. Herlihy and J. E. B. Moss, *Transactional memory: Architectural support for lock-free data structures*. ACM, 1993, vol. 21, no. 2.

[7] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[8] T. Harris and K. Fraser, "Language support for lightweight transactions," in *ACM SIGPLAN Notices*, vol. 38, no. 11. ACM, 2003, pp. 388–402.

[9] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson, "Architectural support for software transactional memory," in *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006.

[10] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: Why is it only a research toy?" *Queue*, vol. 6, no. 5, p. 40, 2008.

[11] R. Haring and B. Team, "The blue gene/q compute chip," in *The 23rd Symposium on High Performance Chips (Hot Chips)*, vol. 4, 2011, pp. 125–180.

[12] Intel®, "Intel® Architecture Instruction Set Extensions Programming Reference," https://software.intel.com/sites/default/files/m/9/2/3/41604, 2012.

[13] R. Rajwar and J. R. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 2001, pp. 294–305.

[14] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel, "MetaTM/TxLinux: transactional memory for an operating system," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 92–103, 2007.

[15] J. Liedtke, "On μ-kernel construction," in *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, Copper Mountain Resort, CO, Dec. 1995, pp. 237–250.

[16] "The Fiasco.OC Microkernel," http://os.inf.tu-dresden.de/fiasco/, 2014, [Online, accessed 27-Nov-2014].

[17] Intel®, "Intel® 64 and IA-32 Architectures Optimization Reference Manual," http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2014.

[18] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of Blue Gene/Q hardware support for transactional memories," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 127–136.

[19] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel® transactional synchronization extensions for high-performance computing," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 19.

[20] V. Leis, A. Kemper, and T. Neumann, "Exploiting hardware transactional memory in main-memory databases," in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*. IEEE, 2014, pp. 580–591.

[21] Intel®, "Haswell Specification Update," http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf, 2014.

[22] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel, "Operating system transactions," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009.

[23] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner, "Improving in-memory database index performance with intel transactional synchronization extensions," in *in Proc. 20th Int'l Symp. High-Performance Computer Architecture*, 2014.

[24] A. Kleen, "Lock elision in the gnu c library," http://lwn.net/Articles/534758/, 2013, [Online, accessed 29-Now-2014].

[25] A. Cohen, J. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck, "Verifying correctness of transactional memories," in *Formal Methods in Computer Aided Design, 2007. FMCAD '07*, Nov 2007, pp. 37–44.

[26] A. Cohen, A. Pnueli, and L. Zuck, "Mechanical verification of transactional memories with non-transactional memory accesses," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Gupta and S. Malik, Eds. Springer Berlin Heidelberg, 2008, vol. 5123.

[27] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier *et al.*, "Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 27–40.

A New Configurable and Parallel Embedded Real-time Micro-Kernel for Multi-core platforms

Antonio Paolillo, Olivier Desenfans, Vladimir Svoboda, Joël Goossens, Ben Rodriguez

PARTS Research Center, Université Libre de Bruxelles, Mangogem S.A. and HIPPEROS S.A.
Corresponding author: antonio.paolillo@ulb.ac.be

***Abstract*—One of the main on-going initiatives of the *PARTS Research Center* together with *HIPPEROS S.A.* is the creation of a new Real-Time Operating Systems family called *HIPPEROS*. This paper focuses on the design and the implementation of its new real-time multi-core micro-kernel. It aims to address the challenge of efficient management of computing resources for competing real-time workloads on modern MPSoC platforms while maintaining the level of assurance and reliability of existing production systems. The objective of this paper is to present an overview of its inner architecture.**

I. Introduction

For the past twenty years, real-time theory has widely explored the possibility to use multi-core and many-core platforms for embedded systems. However, while this topic seems to be very mature in the literature, the safety-critical software industry still relies on uni-core techniques for operating system implementations. The industry state-of-the-art regarding multi-core platforms is to fully separate the processing resources in time and space isolated partitions with very few possible communication channels between the different partitions and therefore to consider that each partition operates as an independent uni-core platform. Examples of de facto standards for these techniques are ARINC653 [1] and AUTOSAR [2].

While from a conservative point of view these are the most reliable and predictable solutions, these are not the best available options in the real-time research w.r.t. efficiency, resource utilisation and cost. Moreover, the rising demand of computational power per silicon area happening in every domain — including safety-critical systems — puts a pressure on the low-level, middleware and kernel developers to implement efficient policies for real-time process management.

In recent years, efforts in the research community have been made to adapt existing general-purpose kernels such as Linux in order to provide a real-time execution environment suitable for the evaluation of efficient multi-core real-time scheduling and resource allocation policies [3], [4]. The main advantage of this approach is to reuse the existing kernel code base which has been tested and validated by millions of users worldwide. However this approach cannot be directly exploited in production systems. Indeed Linux is not originally intended nor designed to support neither *hard* real-time constraints nor safety-critical applications. Moreover it is not conforming to the highest demanding certification standards of this industry such as DO-178-B (level A, B) or ISO26262. As a consequence, the latest real-time multi-core algorithms have not been tested in a strict and realistic hard real-time environment yet. As stated by Brandenburg in [5]:

> Ideally, [...], worst-case kernel overheads [...] should be determined analytically. However, for the foreseeable future, this will likely not be possible in complex kernels such as Linux. Instead, it would be beneficial to develop (or extend existing) μ-kernels of much simpler design with LITMUS^{RT}-like functionality.

This kind of kernel would have to be built from the ground up with hard real-time and multi-core constraints integrated as parts of its base design principles. This would allow for simpler, finer-grained measurements of the overheads introduced by different implementations of the various solutions the literature has to offer. Moreover the architecture of this kernel must scale with an increasing number of cores to allow execution on many-core platforms.

In order to address the challenge of providing efficient multi-core kernel implementations while still providing the same level of assurance and reliability of the existing industry quality standards, the *PARTS Research Center*, together with the company *MangoGem S.A.*, launched the *HIPPEROS* project in 2010. The development of the *HIPPEROS* kernel started in June 2013. *HIPPEROS* aims to provide a family of RTOS solutions, each adapted specifically to the different needs of the real-time system designer and including the implementation of the latest results of the research community. It stands for *HIgh Performance Parallel Embedded Real-time Operating Systems*.

II. System overview

We started the project by developing a new kernel from scratch running as a bare-metal system on ARM and x86 systems. The objective is to have a fully configurable kernel, running transparently on different architectures and platforms with an arbitrary number of cores, that will be the seed of the different RTOS solutions mentioned above. With such a flexible design it would be possible to deeply explore the practicability of real-time theory solutions. To reach this the kernel has the following design characteristics:

- for scalability reasons, it has a distributed asymmetric micro-kernel architecture, meaning that each core can execute a local part of the kernel (the lightweight and very local operations like simple system calls or process context switching), while a dedicated core executes the heavy parts of the kernel (complex system calls, scheduling decisions, shared resources handling, etc), allowing to execute several parts of the kernel *in parallel*; to the

best of our knowledge, this kernel design approach is very rare for real-time systems although it is already used in high performance computing and scalable non real-time kernels [6]–[8];
- it is configurable at build-time to efficiently suit the different needs of the system designer or application developer; e.g. the scheduling policy or the resource allocation protocol for real-time processes can be chosen at build-time; notice that only the chosen policies will be embedded in the production executable binary image of the kernel (mainly for code size reasons);
- to manage hard real-time workloads, it implements the popular process model used in the real-time scheduling research literature: the concept of periodic and sporadic tasks generating jobs to schedule with a finite time budget and deadline.

By combining the available configuration options, the *HIPPEROS* build system is able to generate a large variety of RTOS solutions, ranging from a low-overhead statically linked run-time executive implementing the simple rate monotonic scheduling policy [9] to a full fledged micro-kernel based operating system supporting several independent ELF applications with memory isolation between processes, inter-core message passing IPC and optimal scheduling policies.

This distributed and highly configurable kernel supporting real-time workloads aims to provides both a productive system to industry application designers and an experimental software platform to real-time researchers. The goal is to test, validate and run into production low-overhead energy efficient hard real-time systems running on modern embedded multi-core platforms with different instruction set architectures.

III. Process model

To derive straightforward implementations of state-of-the-art algorithms, we chose to faithfully interpret the task model w.r.t. real-time scheduling theory. We map the popular task model of real-time literature [9] to the internal HIPPEROS process abstraction. More specifically, we implemented *constrained deadline sporadic and periodic tasks*.

A set of tasks is statically registered to the kernel. Each task is configurable by providing the following information: an executable program and timing information (sporadic/periodic, offset, deadline, period and worst-case execution time). Time unit for these values is the *number of kernel ticks*, a configurable atomic time period. At kernel initialisation time, the process manager module registers one process for each of these tasks and configures it according to the task parameters.

The *scheduler API* is preemptive and priority-based: each time a process changes state, the scheduler module is called to decide if some process context switches must occur according to their priority. If a real-time process overruns its associated task's WCET or misses its deadline, a configurable policy is applied. It could be that the process is killed (the reason being the non-respect of its contract with the kernel), the event ignored or the priority of the process changed.

These simple mechanisms allow to easily implement and evaluate theoretical multi-core scheduling algorithms (like RUN [10], U-EDF [11] or power- and thermally-aware algorithms) and the associated resource allocation protocols. The model could be easily extended in the future to support mixed-criticality tasks: it would require vectorial timing information rather than scalars.

IV. Asymmetric kernel architecture

A recurring problem in kernel design for multi-core platforms is how to distribute the privileged work amongst the different processing resources. Usual implementations like Linux use a symmetric design, where each core goes through the same kernel code and protect data structures with fine-grained lock mechanisms. However, this approach can lead to *kernel serialisation*, meaning that each kernel thread is actually executed sequentially (each waiting for the completion of one other) and has been proven not to scale with an increasing number of cores [6], [12]. Furthermore, in [12], Cerqueira *et al* suggest an asymmetric distribution of the work, where one core has the responsibility to execute the scheduler and dispatches the processes to the other cores through message passing.

We adopted a similar solution in the *HIPPEROS* kernel design: a designated core called the *master core* is responsible for managing the global resources, keeping a coherent state of the system and calling the scheduler to decide which process has to be *preempted* or *dispatched*. We went further than [12] by implementing this design not only for the scheduler but also for system calls and process message passing mechanisms. It allows the kernel to be executed in *parallel*.

The principle is the following: each time a scheduling decision has to be made (e.g. a process changes state), the *master core* must be woken up. When the *master core* has to notify another core (called *slave core*) that it has to execute a context switch (process preempted or dispatched), the *master* sends a software-generated inter-processor interrupt (IPI) to the slave core to notify it of the changes. When a process executing on a slave core calls a system call that may impact scheduling, the *remote system call mechanism* is used. The slave part of the kernel serialises the system call arguments, triggers an IPI to the master and goes back to user mode to execute a busy loop waiting for the response of the master part of the kernel. Notice that this busy loop is process-specific, executed in user mode and can be interrupted by a context-switch request of the *master core*.

In opposition to the symmetric approach, this master/slave kernel architecture requires *almost* no locking mechanism as the system's global state must not be shared and is only visible by the *master core*.

To correctly implement the system calls and the context switches, some small data structures are shared between the *master* and each slave. These data structures are currently protected with mutexes, and wait-free data structures are considered to be integrated for the foreseeable future. Notice that as the contention on these data structures is limited by the process-to-kernel interactions, several slave cores require distinct mutexes. Therefore, the peak contention of the concurrency mechanisms is low. In the long term, our goal is to be able to predictably bound this contention. As the shared data structures between *master* and *slaves* are limited to what is necessary for system calls and context switches and the rest of the system state (e.g. scheduler data structures) is

maintained only by the *master*, we also expect to have limited performance-degrading cache-line bouncing.

The inter-process communication (IPC) scheme is built on top of this master-slave RPC mechanism. We support two different API for IPC: the *Copy buffer IPC* (CB-IPC), where the message is copied from the sender buffer to the receiver buffer and the *Zero copy IPC* (ZC-IPC), where a page is shared between the sender and the receiver (no copy is then performed when passing the message). When a process calls the *send* or *receive* system calls, the *master core* is warned through an IPI to update the process states accordingly. However, in case of CB-IPC, the message is copied locally by the slave to avoid overloading the master with memory operations.

We expect this approach to scale up to 8 cores of the embedded platform. More cores contacting the *master* would eventually overload it, resulting in a situation where some running processes have to wait for the execution of all the system calls of the processes executing on the other cores. For more cores (e.g. for execution of HIPPEROS on a many-core platform), we foresee the usage of techniques like clustering, where several independent micro-kernel instances would be executed in parallel, like the Helios Satellite Kernels [8]. Each parallel kernel would be responsible of a subset of the platform processing cores with independent scheduling, memory management and process message passing. Processes on different clusters that want to communicate would use a dedicated inter-kernel communication channel. This mechanism still needs to be implemented and evaluated.

V. Kernel configurability

As *HIPPEROS* targets deeply embedded systems, the majority of options is configured at build-time to suit the specific requirements of the embedded software. Policies and components must then be chosen at build-time.

One of the goals of the kernel is to be portable across a large variety of architectures and platforms. The kernel currently supports ARM and x86 architectures. There is a wide variety of hardware platforms implementing these architectures and these targets can have very different levels of complexity and features. For example, a Memory Management Unit (MMU) can be present or not on a given platform. Therefore, the kernel must be configurable to the point of presenting several memory models, according to the presence or absence of a MMU. It is necessary to provide a MMU-free memory model as some of the critical embedded platforms used in production today are still MMU- and cache-free.

The scheduling policy (Partitioned-RM, Global-EDF, etc.) in place is also a modular component that is chosen at kernel build-time. For energy efficiency reasons, the number of cores of the target platform actually used can be configured too: the user could decide to only use a subset of the resources available on the target platform. Moreover, the set of cores could be shared between several operating systems (several *HIPPEROS* instances as mentionned in section IV or other OSes). Therefore, decide which cores are used or not will allow *HIPPEROS* to be suited for mixed-criticality environment with space partitioning: the execution of several OSes with various levels of criticality on top of hypervisor software.

VI. Conclusion

In this paper, we introduced a new configurable kernel designed for embedded multi-core platforms. In opposition to the traditional research approaches, our kernel is written from scratch and explores new ways of distributing privileged work among the different cores of the platform by relying on its asymmetric architecture. System reliability is enforced by design using a micro-kernel architecture. By implementing scalable policies inside the kernel, it will be adapted to modern and future multi-/many-core platforms.

To enable straightforward implementation of existing real-time scheduling strategies, we faithfully implemented the literature task model: periodic and sporadic jobs with a limited execution budget and a deadline.

Thanks to the high level of configurability and modularity built in the kernel by design, we expect to provide a new benchmarking platform to the research community.

Future developments will involve the integration of the *HIPPEROS* RTOS in mixed-criticality environments where a RTOS running highly critical workloads can be executed in parallel with a general purpose OS like Linux to make an effective usage of the modern MPSoC platforms.

A free academic license of the product will be available for distribution. The RTOS is now being validated for various use cases within the ARTEMIS CRAFTERS project by the *PARTS Research Center* and *MangoGem S.A.*. This work is supported by the Innoviris grant RBC/12 EUART 2a. The kernel is used in industrial Proof of Concept projects by *HIPPEROS S.A.*, which further develops it into a full-blown certifiable RTOS. *HIPPEROS* is a registered trademark of *HIPPEROS S.A.*.

References

[1] *Avionics Application Software Standard Interface*, Airlines Electronic Engineering Committee, Aeronautical Radio INC, June 2013.

[2] *Guide to Multi-Core Systems*, AUTOSAR, March 2014.

[3] J. M. Calandrino, H. Leontyev, A. Block, U. Devi, and J. H. Anderson, "LitmusRT: A testbed for empirically comparing real-time multiprocessor schedulers," in *27th IEEE Int. Real-Time Systems Symposium*, 2006.

[4] D. Faggioli, M. Trimarchi, F. Checconi, M. Bertogna, and A. Mancina, "An implementation of the earliest deadline first algorithm in Linux," in *24th Annual ACM symposium on Applied Computing*, 2009.

[5] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina, 2011.

[6] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): The case for a scalable operating system for multicores." SOSP, 2009.

[7] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new os architecture for scalable multicore systems." SOSP, 2009.

[8] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, "Helios: Heterogeneous multiprocessing with satellite kernels." SOSP, 2009.

[9] J. W. S. Liu, *Real-Time Systems*. Prentice Hall, 2000.

[10] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *IEEE 32nd Real-Time Systems Symposium*, Nov. 2011.

[11] G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic, "U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks," in *ECRTS*, 2012.

[12] F. Cerqueira, M. Vanga, and B. Brandenburg, "Scaling global scheduling with message passing," in *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2014.

Adaptive Resource Sharing in Multicores

Kai Lampka Jonas Flodin Wang Yi
Department of Information Technology, Uppsala University

Adam Lackorzynski
Technische Universität Dresden

***Abstract*—This short paper presents an adaptive, operating system (OS) anchored budgeting mechanisms for controlling the access to a shared resource. Temporarily blocking accesses from a core reduces the waiting times of other applications executing in parallel on other cores. This helps to guarantee the assumed worst case execution time bounds at run-time. In addition to our previous work [1], the presented scheme considers shifting of unused access bandwidth among applications and takes advantage from a time-triggered scheduling policy for executing real-time applications at core-level.**

I. Introduction

a) Motivation: Sharing of hardware as found in COTS multicores brings in hidden dependencies when consolidating hard and soft real-time applications on a single processor. These dependencies can provoke timing faults that are difficult to foresee and can corrupt the functionality of the system.

The challenge inherent to the design of the run-time environment to support the timing correct execution of mixed critical workloads is three-fold.

Firstly, hard real-time tasks need to be isolated, such that their assumed upper bound on their execution time always holds. In addition, standard real-time analysis builds on task sets with known bounds on their execution times. The feasibility of a scheduling strategy, shown at design time, is guaranteed to hold at run-time if the upper bounds on the execution times (and activation frequencies) are not violated. As unaccounted waiting at a resource prolong execution times, it can become a threat to a systems timing correctness.

Secondly, resource sharing needs to be considerably dynamic, to avoid over-provisioning and thereby achieve good utilization of the used equipment.

Thirdly, the mechanism to coordinate the access to a shared resource must not be too complex to limit the computational overhead experienced at run-time.

b) Technical problem description: As an example to resource sharing, this short paper considers the sharing of the dynamic random access memory (DRAM).
When carrying out a worst-case response time analysis (WCRT) for quantifying the computation time consumption of an application, one has to assume that a memory access from a core can be delayed by all other memory accesses occurring while the respective access is waiting at the DRAM-controller. With n access requests from other cores, this yields a delay of $(n+1)$ times the worst case service time until a request is served. This assumption is conservative as it overapproximates the actual behaviour of the system at run-time. However, it is safe as long as less than n competing access requests occur. It is therefore of uttermost importance to ensure at run-time that the number of competing memory access requests is bounded by a pre-defined number and one does not experience unaccounted waiting times due to unaccounted memory requests.
In this short paper, we summarize our effort to do this efficiently and effectively and point out directions for improvements left to the future.

c) Related Work: For dealing with memory access contention effects in the setting of multicore architectures, several strategies have been proposed.
Time deterministic memory designs avoid interference by physically separating relevant parts of the memory hierarchy and exclusively assigning parts to cores. This ranges from the use of scratchpad memories [5] to the partitioning of main memory [3]. However, these techniques all rely on the layout of the memory hierarchy.
Another way to feature timing predictability is provision of isolation mechanisms as part of the run-time environment. At the level of OS, this can be done by controlling the virtual to physical address mappings [6] or by restricting access frequencies of the main memory for each core [7], [8].

d) Own Contribution: Advancing over the work of Pellizzoni et al. [7], [8], this short paper propose the following innovations when it comes to resource access budgeting schemes: (a) we enable lifting of budgets, namely once all real-time tasks are pre-maturely completed. (b) we also feature donation of budgets. But, donation is only allowed, if the donating real-time task has already terminated.
Both features can be considered safe, the safeness of budget lifting is demonstrated in [1]. The safeness of budget donation comes from the fact that we avoid premature shifting of resource accesses. This is important and this way we avoid starvation of real-time applications which could provoke timing faults.
In addition to our own work [1], this short paper presents a budgeting scheme which takes advantage of a time-triggered scheduling strategy of real-time applications at the levels of cores. This way, we not only lift unneeded budgets more often. We also hope to shift unused access budgets more often as this can take place every time all real-time applications of a time-frame have processed their workload.

II. System model

We consider a system deployed on a typical COTS multicore architecture. There are M CPU-cores, K of which are executing hard real-time software and $M-K$ cores are executing best-effort applications.

There are N sporadic hard real-time tasks $T = \{\tau_1, \tau_2, ..., \tau_N\}$, each defined by the quadruple $\tau_i = (C_i, P_i, D_i, H_i)$, with C_i as the WCET for the task when running alone on one hard real-time core, P_i as the minimum inter arrival time of the task, $D_i \leq P_i$ as the task's relative

This work is partly funded by DFG-SPPEXA's project FFMK.

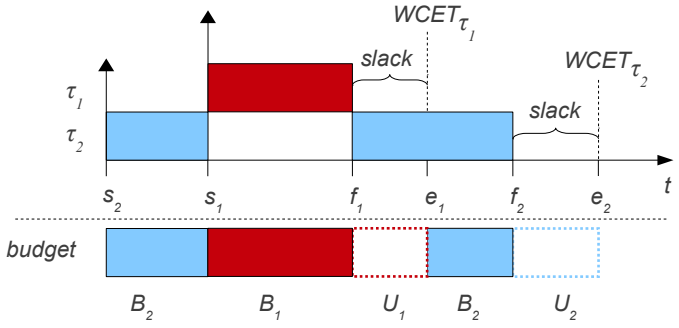


Fig. 1. Budgeting example with two tasks. Arrows pointing up denote job releases and dashed vertical lines denote the point in time when a job would have finished if it needed the entirety of its WCET.

deadline and with H_i as the largest number of memory access requests produced by τ_i during one task instance.

Each core has its own fixed priority scheduler and each task τ_i is mapped to one specific core out of the K hard real-time cores.
The other cores we collectively call soft real-time cores and they execute soft real-time or best-effort tasks, we do not make any assumptions about the soft real-time tasks. **It is these cores which we intend to control through the presented budgeting scheme and thereby ensure timing correctness of the hard real-time applications running in parallel.**
All cores share a single memory controller which acts as an arbiter for serving requests to DRAM.

III. Dynamic budgeting with lifting

The initial scheme of budget enforcement and lifting is presented in [1]. We briefly recall its working principle by means of an example.

Fig. 1 illustrates the execution of two tasks. The upper part depicts their interleaved execution on the hard real-time core. The lower part shows which budget is in effect on the soft real-time cores. The hard real-time core starts executing τ_2 and signals the soft-real time cores to use budget B_2 at time s_2. The hard real-time core continues executing τ_2 until time s_1, when it is preempted by the arrival of τ_1, which also triggers the soft real-time cores to switch budget to B_1. When τ_1 finishes early at f_1, the soft real-time cores are signaled to exchange the budget B_1 for U_1, which means that they have unlimited access to main memory until e_1. At the same time, the hard real-time core switches to executing τ_2. When U_1 expires at time e_1 the soft real-time cores fall back to use budget B_2 until τ_2 finishes at f_2. The budget B_2 is then switched for U_2 until it expires at e_2.

IV. Combining Ressource access budgeting and Time triggered application scheduling

A. Time-triggered execution of tasks

Scheduling of hard real-time tasks is organized according to a standard time-triggered scheme, e. g., as defined in [2].

A time-triggered schedule at core i is a sequence of K_i slots $s_{i,j}$, where $s^{\Delta}_{i,j}$ refers to the time length of each slot.

While executing a slot $s_{i,j}$, we need to guard that all the cores running soft real-time applications do not issue more than $B^{eff}(s_{i,j})$ accesses to the main memory in total.

Algorithm 1 Enforcing budgets on a soft core

Requires: timer T, active budget B,
set of active budgets $Budget$
Input: signal e mapping to a slot and action
procedure BSCHEDULER(signal e)
  $PREEMPTION = OFF$
  if `action`$(e) \in \{depleted, expired\}$ **then**
    `wait4Timer(T)`
    `goto line 28`
  end if
  `update`$(Budgets, B.b^{eff} -$ `readPMC`$(), B.t - T)$
  if `action`$(e) ==$ *activate* **then**
    `insert`$(Budgets,$ `slot`$(e))$
  else if `action`$(e) ==$ *deactivate* **then**
    `remove`$(Budgets,$ `slot`$(e)))$
  else if `action`$(e) ==$ *donated* **then**
    $C =$ `peek`$(Budgets,$ `slot`$(e))$
    `updateDonation`$(Budgets, B.d, C.t)$
  end if
  while $B =$ `peek`$(Budgets)) \neq \emptyset \wedge B.t \leq 0$ **do**
    `remove`$(Budgets, B)$
  end while
  if $B == \emptyset$ **then**
    `stopTimer`(T)
  else
    `setPMC`$(B.b^{eff})$
    `setTimer`$(T = B.t)$
  end if
  $PREEMPTION = ON$
end procedure

Below we detail on the algorithm to implement this basic functionality. For simplicity, we ignore the distribution of budgets and donations over multiple cores executing a soft real-time workload,. For the presented algorithms, the distribution could be arranged transparently, through a dedicated administering core.

B. Budget enforcement for soft real-time workloads

The required functionality for guarding the number of memory accesses such that timing correctness of the hard real-time tasks is ensured, is provided by Algorithm 1.

The implementation details of Algorithm 1 are as follows: we assume that there is a queue $Budgets$ of active budgets, with at most one active budget per hard real-time core.

Within the queue, the active budgets are ordered by increasing budget sizes. The following functions are used to access items of the queue: function `replace` and `remove`, which work as expected. Function `update`$(Budgets, a, b)$ decreases all budgets of the queue by value a and decreases their lifetimes by value b. This is needed once the decisive budget has reached its lifetime or is replaced by a newly activated budget. Function `peek` gives the head of the queue, i. e., the active budget with the smallest number of allowable cache misses. The functions does not remove the item from the queue.

The algorithm itself works as follows: upon depletion of the decisive budget or at the end of its lifetime the core suspend

execution for the remaining lifetime, which in case of the "*end of lifetime*" situation is 0 (line 5).

In case the decisive budget has reached the end of its life time or a new budget to be activated has arrived, we update all active budgets with respect to to the number of cache misses and the expired time occurred during the current budget has been made the decisive one.

In case of a premature deactivation the decisive budget, it is removed from the budget queue and the next active budget is fetched. This can either be the same, but updated budget, a new one, where budgets with invalid lifetime are discarded, or it is an empty budget (line 18-20).

In case of an empty budget all active budgets have been prematurely invalidated and the core has a non-restricted allowance to the main memory.

In case a valid budget is fetched from the queue, the LLC-register and the lifetime clock counter are set accordingly (line 25 and 26).

Budget donation executed by a hard real-time core is considered before actually fetching a budget from the queue. Function `updateDonation`$(Budgets, a, b)$ adds value a to each budget, here parameter $B.b^{eff}$ and does so only for those budgets which have a residual lifetime smaller than b.

V. Implementation

For evaluation, we use the L4Re microkernel system that provides the environment to run existing applications and operating systems through virtualization as well as native microkernel-based applications. The L4Re gives us the flexibility to use virtualization as well as specific native applications in a very controlled environment.

Scheduling in the L4Re microkernel applies scheduling contexts (SCs), a thread-specific data structure that contains all information required for scheduling [4]. A special features of the SC mechanism is that a thread, or vCPU, can have multiple SCs, allowing to give a thread/vCPU multiple different scheduling parameters. This is especially useful in virtualization contexts where the guest OS can use multiple SCs to express the requirements of its internal tasks to the microkernel. In our work we use the SC mechanism to implement budgets based on performance counters.

1) Hardware Performance Counters: Modern processors have a performance monitor counter (PMC) unit that allows to count hardware-related events in the CPU core, such as cache misses. The core can also generate interrupts when a counter reaches a predefined threshold. Using the PMC it is possible to count the number of last-level cache misses which is equivalent to the number of main memory fetches. If the number of memory fetches reaches a certain threshold, the microkernel may suspend the execution of soft real-time applications to avoid an overload of the main memory with memory access requests. The challenge is to use the PMC in such a way, that is dynamically resetting the PMC and adjusting the threshold, that the maximum amount of memory accesses can be placed on the DRAM without affecting real-time applications.

2) PMC Pecularities: All Intel Core-i CPUs have a minimal standard set of performance counters that includes the last-level-cache-miss counter. The first experiment we did was checking whether our test program indeed uses all of the memory bandwidth available. By running it on a different number of cores in parallel we expect the runtime of each program to increase with the number of cores. That is, on 4 cores each program shall run 4 times longer compared when running alone in the system. We observed this behavior.

However, when we added delays to the memory access loop in the test program, with the goal to not fully use up all the memory bandwidth, the respective last-level-cache-miss counter shows significantly less events although the same amount of memory was accessed. This is likely because of the hardware memory prefetcher where memory accesses are not counted, as they are no cache misses. We tried to disable the prefetcher via the `IA32_MISC_ENABLE` MSR [8], however, this yields to a general protection fault when writing the MSR on the used i7-4770 CPU. Using non-cached memory is no choice either because those accesses do not causes cache-relevant events, such as misses. Using other counters available on the specific CPUs showed either the same behavior (significantly different values for with and without delay loops), or did not count at all.

Intermediate result is that Intel-based x86 desktop CPU, such as the i7-4770, can not be used to implement memory access budgeting based on performance counters. We need to look at other CPU lines, such as Xeon CPUs, or older Intel CPUs, whether they are better suited, for example, because they allow to disable the prefetcher. Alternatively, looking at ARM Cortex-A CPUs shows a counter called `MEM_ACCESS` which sounds promising as well.

References

[1] Jonas Flodin, Kai Lampka, and Wang Yi. Dynamic budgeting for settling DRAM contention of co-running hard and soft real-time tasks. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems, SIES 2014, Pisa, Italy, June 18-20, 2014*, pages 151–159, 2014.

[2] Gerhard Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 152–161, 1995.

[3] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, M. Sullivan, Ikhwan Lee, and M. Erez. Balancing dram locality and parallelism in shared memory cmp systems. In *High Performance Computer Architecture (HPCA) 2012*, pages 1–12, Feb 2012.

[4] Adam Lackorzyński, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening hierarchical scheduling. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '12, pages 93–102, New York, NY, USA, 2012. ACM.

[5] I. Liu, J. Reineke, and E.A. Lee. A pret architecture supporting concurrent programs with composable timing properties. In *ASILOMAR 2010*, pages 2111–2115, Nov 2010.

[6] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 367–376, New York, NY, USA, 2012. ACM.

[7] Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 299–308, 2012.

[8] Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS 2013*, pages 55–64, 2013.

Implementing Adaptive Clustered Scheduling in LITMUS$^{\mathrm{RT}}$

Aaron Block
Department of Mathematics and Computer Science
Austin College
Sherman, TX
Email: ablock@austincollege.edu

William Kelley
BAE Systems
Fort Worth, TX

***Abstract*—In this paper, we develop an adaptive scheduling algorithm for changing the processor shares of tasks on real-time multiprocessor systems where tasks are assigned to *clusters* of processors. Additionally, we implement this adaptive algorithm as a LITMUS$^{\mathrm{RT}}$ plugin. Our focus is on adaptive systems that are deployed in environments in which tasks may *frequently* require *significant* share changes. Prior work on enabling real-time adaptivity on multiprocessors has focused primarily on systems where tasks are scheduled from a *global* priority queue. The algorithm proposed in this paper use *feedback* and *optimization* techniques to determine at runtime *which* adaptations are needed.**

I. Introduction

Real-time systems that are *adaptive* in nature have received considerable recent attention for both uniprocessor and multiprocessor environments [1], [4]–[6], [10], [11]. In prior work [5], we designed and implemented an adaptive multiprocessor scheduling algorithm (A-GEDF), in which all tasks are scheduled from a single *global* priority queue and can freely migrate between processors. As shown in [9], global systems have the advantage that they can fully utilize a multiprocessor system and still guarantee that deadlines will miss their deadlines by at most a bounded amount. However, Bastoni *et al.* [2] demonstrated that for soft-real time systems with *many* processors (*i.e.*, 12 or more cores), global scheduling algorithms are inferior to algorithms in which tasks can migrate between a *cluster* of processors that share a common cache. In this work, we designed and implemented (as a LITMUS$^{\mathrm{RT}}$ plugin) an alternative adaptive multiprocessor real-time scheduling algorithm (A-CEDF), which is a modification of A-GEDF that uses a clustered scheduling algorithm as its basis rather than a global scheduling algorithm. In this paper, we showed that adaptive behavior (which can improve the Quality of Service of a soft real-time systems) can be enabled on a clustered system without substantially increasing the scheduling cost.

II. Task Model

In this section, we describe our system model and the CEDF scheduling algorithm, upon which A-CEDF is based.

A. Sporadic Tasks

A *sporadic task* is defined by a *worst-case execution time* and *period*. The fraction of a processor required by a task is called the *weight* of the task, and is defined as the worst-case execution time divided by the period. The first job of a task may be invoked or *released* at any time at or after time zero. Successive job releases of task must be separated by at least the period of the task. The *deadline* of a job is period time units after the job is released. A job is said to be *active* if it has been released, but is not yet completed. In this work, we are concerned with *soft real-time systems* where it is acceptable for a job to miss a deadline as long as the amount of time that a job can miss a deadline is bounded (such a system is said to have *bounded tardiness*).

The *actual execution time of job* is the amount of time for which the job is actually scheduled. The *actual weight of a job* is the share of a processor that a job actually requires and is defined by the actual execution time of a job divided by the period of the task. We assume that actual execution time and actual weight for a job are unknown prior to the completion of the job since both values may differ between job releases.

The multiprocessor sporadic task scheduling algorithm that is the most relevant to this work is *clustered earliest deadline first.* (CEDF). Under CEDF, tasks are permanently assigned to "clusters" of processing cores that share a common cache. Jobs with work remaining are prioritized for scheduling on a cluster by their deadline. Jobs can be scheduled on any processor in their cluster, but cannot be scheduled outside of their cluster. As shown in [2] for soft real-time systems, CEDF-based scheduling tends to perform better than non-clustered approaches when clusters contain at least six cores.

B. Adaptable Sporadic Tasks

The *adaptable sporadic task system* [7] extends the notion of a sporadic task system in three major ways. First, *worst-case* execution times are not assumed. Second, each task has a set of *service levels*, which represent a different *Quality of Service* (QoS) levels of a task. Third, tasks have a *weight translation function*, which uses the actual weight and current service level of a task to estimate the actual weight of the task if it changed service levels.

Each service level of a task has three characteristics: a *QoS value*, a *period*, and a *code segment*. When a job is released, it is released at a given service level. That service level determines the code segment that the job will execute,

the deadline of the current job (via the period) and the earliest possible release time of the next job (again via the period).

The weight translation function of a task is an empirically-determined function that takes as an input the current active weight and service level of a task and provides an estimate of what the weight of task would be if it changed to a new service level. For example, if service level 2 for a task required twice as much computation as service level 1 and the current weight of a task was 0.25, then changing from service level 1 to 2 would change the weight of the task to 0.5. It is unnecessary for the weight translation function to be perfectly accurate, but the more accurate it is, the better an adaptive algorithm will be optimizing system QoS. It is important to note that tasks with lower QoS values must have lower estimated weights. Thus, an adaptive algorithm can trade QoS for schedulability.

III. A-CEDF

In this work, we introduce the *adaptive clustered earliest deadline first* (A-CEDF) scheduling algorithm. A-CEDF is a clustered-scheduled variant of the *adaptive global earliest deadline first* (A-GEDF) scheduling algorithm, which we proposed in prior work [7]. A-CEDF is designed to schedule adaptable sporadic task with the objective of maximizing the total QoS while maintaining bounded tardiness. A-CEDF consist of five primary components: **(1)** the *predictor*, which uses a *proportional-integral* (PI) feedback controller to estimate the weights of future jobs using the actual weights of previously completed jobs; **(2)** the *optimizer*, which given estimated job weights, attempts to determine an optimal set of functional service levels; **(3)** the *repartitioner*, which given the estimated job weights attempts to determine the optimal assignment of tasks to clusters; **(4)** several *reweighting rules*, which are used to change the functional service level of a task to match that chosen by the optimizer; and **(5)** the CEDF *algorithm*. At a high level, these components function as follows.

- *At each instant*, tasks are scheduled via CEDF.
- *At a job's completion*, the predictor is used to estimate the weight for the next job release.
- *After a developer-specified threshold based on task weight and time elapsed*, the optimization component uses the estimated weight to determine new service levels for each task. If the service level of a job changes, then the reweighing rules will enact it.
- *If the clusters are "imbalanced"*, then the repartitioner will correct this behavior by migrating tasks between clusters. If necessary, the optimization and reweighting rules may be run as part of this process.

The primary difference between A-CEDF and A-GEDF is that A-GEDF allow tasks to freely migrate between *all* processors. Thus, A-GEDF does not need or have a repartitioner component. That being said, A-CEDF and A-GEDF have similar predictors, optimizers, and reweighting rules.

IV. Implementation

To better understand A-CEDF, we implemented this algorithm in the LITMUS$^{\mathrm{RT}}$ version 2014.2 (**LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems), which is an extension of Linux (currently, version 3.10.41) that allows different multiprocessor scheduling algorithms to be linked as plug-in components [3], [8]. Our implementation of A-CEDF consists of both a user-space library and kernel support added to LITMUS$^{\mathrm{RT}}$. Our implementation of A-CEDF required 1,227 lines of code. Most of these changes were in modifying LITMUS$^{\mathrm{RT}}$'s default CEDF implementation. In prior work [7], we discussed how to modify LITMUS$^{\mathrm{RT}}$ to support adaptable sporadic tasks scheduled via *global* scheduling algorithms. In this work, we focus on the *additional* challenges that arise when implementing *clustered* real-time adaptive scheduling algorithms.

A. Challenge: Defining "Imbalanced"

As we mentioned in Sec. III, A-CEDF repartitions when the clusters are *imbalanced*. Informally, the clusters become imbalanced if one cluster is doing more or less work than another. However, it is not obvious how we should formally define "imbalanced." There are two metrics that we can use to measure the quality of a partitioning: **(1)** the total weight of all tasks assigned to a given cluster and **(2)** the total QoS of all tasks assigned to a given cluster. Under either metric, a system is "imbalanced" if the metric value (*i.e.*, total weight or QoS) of one cluster is higher than a *user-defined threshold* the metric value of another.

In this work, we repartition the system when there is an imbalance between the *QoS* of tasks assigned to different clusters. We chose to use a QoS-based metric because the objective of A-CEDF is to maximize the QoS without causing unbounded tardiness. Thus, the weight balance by itself is not useful if the system could run at a higher QoS after rebalancing. For example, consider the following scenario. Suppose that an external event occurs that increases the execution time for all tasks on a given cluster. The optimizer component of A-CEDF will reduce the service level (and hence the QoS) for every task on the cluster. If this reduction in QoS is larger than the user-defined threshold, then this will trigger the repartitioning to occur. It is possible that such an event would be unnoticed by a weight-based metric; particularly, if the total weight of all tasks was approximately the same before the external event and after the optimizer ran.

B. Challenge: Enacting a Repartioning

When the system determines that tasks should be repartitioned, the next question is when should that repartitioning be enacted. There are two primary approaches to this problem: **(1)** migrate all tasks to new clusters immediately or **(2)** gradually migrate tasks between clusters over time. In our implementation of A-CEDF, we migrate tasks gradually. Specifically, after a repartitioning event, we migrate each task when it finishes its active job. We chose this approach because, based on a simple extension to our work in [6], it is possible to show that frequently moving *incomplete* jobs between clusters can cause unbounded tardiness.

Additionally, it is worth noting that since repartitioning occurs because of QoS imbalances, the quicker the repartition is enacted, the better it is for the overal QoS for the system. Yet, quickly enacting a repartitioning is *not* crucial for the functioning of the system. Thus, while gradually migrating tasks between clusters will reduce the QoS of the system, we believe this tradeoff is worth the cost to preventing unbounded tardiness from occurring.

C. Challenge: Migrating a Task

In the typical implementation of CEDF, each cluster has its own spin lock for protecting the priority queue containing all active jobs. This prevents a race condition in which multiple cores on the same cluster attempt to change the priority queue at the same time. Moreover, under CEDF tasks never migrate between clusters. This is not the case in A-CEDF.

To enable A-CEDF to migrate a task from Cluster A to Cluster B, we need two layers of synchronization: **(1)** one layer to prevent any core on Cluster B from corrupting Cluster B's priority queue and **(2)** one layer to prevent any core on Cluster A *that is migrating a task to Cluster B* from corrupting Cluster B's priority queue. Moreover, Cluster A cannot simply acquire Cluster B's spin lock or a deadlock could occur (*e.g.*, if Cluster B attempted to migrate a task to Cluster A at approximately the same time that Cluster A is attempting to migrate a task to Cluster B). To enable task migration, we need a more sophisticated approach to synchronization. We do so by employing the following method:

- Each cluster has a unique ID number.
- Each cluster has a `prime` and `second` spinlock.
- When entering into *any* critical section, a core first acquires its cluster's `prime` lock, then its `second` lock.
- When a task that is flagged for migration from Cluster A to Cluster B, it executes the pseudo-code given in Fig. 1.

There are three keys to this synchronization technique. First, the `prime` lock on each cluster protects the priority queue from corruption by all cores in the same cluster. Second, the `second` lock provides a means to protect a cluster's priority queue from external corruption (*i.e.*, Cluster A must acquire Cluster B's `second` before migrating the task). Third, by releasing and reacquiring `second` locks in a globally established order (*i.e.*, the code in Fig. 1), we prevent the circular chain of dependencies that is a prerequisite for deadlock. Notice that this ordering heuristic is similar to the double-lock used by Linux for its native run queues.

D. Cost of Implementation

To measure the cost of an implemented A-CEDF, we ran a *simulated* virtual reality human tracking system (called Whisper [12]) on a Mac Pro with two 2.66 Ghz 6-core Intel Xeon processors (12 cores total). Each core has 512KB of L2 cache and each processor has 12 MB of fully shared L3 cache. Our clustered implementation of A-CEDF had two clusters, one for each processor. Our simulated human tracking system had 96 tasks each of which had both gradual and sudden changes in weight. We found that the introduction of adaptive techniques slightly increased the average scheduling cost compared to a non-adaptive variant. Specifically, A-CEDF took on average 5.8μs per scheduling decision while CEDF took on average 4.3μs per scheduling decision. The increased running time was primarily becasuse our implementation of the of the optimizer and repartitioner involves sorting a large number of tasks. It is possible to reduce the running time of A-CEDF by using a faster, but less accurate implementation of these two components. It is worth noting that neither the feedback predictor nor the double-locking mechanism appreciably increased the scheduling time.

```
Migrate task from Cluster A to B
1:  Release Cluster A's second lock
2:  if Cluster A's ID is less than Cluster B's ID then
3:      Acquire Cluster A's second lock
4:      Acquire Cluster B's second lock
5:  else
6:      Acquire Cluster B's second lock
7:      Acquire Cluster A's second lock
8:  fi
9:  Actually move task from cluster A to B
10: Release Cluster B's second lock
```

Fig. 1. Pseudo-code defining task migration

V. Conclusions and Future Work

In this work, we designed and implemented A-CEDF as a LITMUS$^{\mathrm{RT}}$ plugin. In the process of implementing A-CEDF, we came across multiple issues with implementing any type of adaptive clustered real-time scheduling algorithm. We also established that adaptive behavior can be enabled in clustered soft-real time systems with only a small additional scheduling cost. In future work, we plan to compare the performance of A-CEDF to A-GEDF at maximizing the QoS for a system.

References

[1] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *RTSS.* '02.

[2] A. Bastoni, B. Brandenburg, and J. Anderson. An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor Real-Time Schedulers. *RTSS*, '10.

[3] B. Brandenburg *Scheduling and Locking in Multiprocessor Real-Time Operating Systems.* PhD thesis, UNC, '11.

[4] A. Block, J. Anderson, and G. Bishop. Fine-grained task reweighting on multiprocessors. *Journal of Embed Comp*, '11.

[5] A. Block, *Adaptive Multiprocessor Real-Time Systems.* PhD thesis, UNC, '08,

[6] A. Block, J. Anderson, and U. Devi. Task reweighting under global scheduling on multiprocessors. *Real-Time Sys.*, '08.

[7] A. Block, B. Brandenburg, J. Anderson, and S. Quint. An Adaptive Framework for Multiprocessor Real-Time Systems. *ECRTS*, '08.

[8] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *RTSS*, '06.

[9] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Sys.*, '08

[10] N. Khalilzad, F. Kong, X. Liu, M. Behnam, and T. Nolte. A feedback Scheduling Framework for Component-Based Soft Real-Time Systems. *RTAS.*, '15

[11] C. Lu, J. Stankovic, S. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Sys.*, '02.

[12] N. Vallidis *Whisper: A Spread Spectrum Approach to Occlusion in Acoustic Tracking.* PhD thesis, UNC, '02.

Preliminary design and validation of a modular framework for predictable composition of medical imaging applications

M.M.H.P. van den Heuvel†, S.C. Crăcană†, H.L. Salunkhe†, J.J. Lukkien†, A. Lele† and D. Segers‡
†Eindhoven University of Technology, Eindhoven, The Netherlands — ‡Barco N.V., Kortrijk, Belgium

***Abstract*—In this work, we present a software framework which enables us to analyse the performance of medical imaging algorithms in isolation and to integrate these algorithms in a pipeline, thereby composing a medical application in a modular manner. In particular, we show how public-domain middleware can be configured in order to achieve predictable execution of a use-case application. On this use case we applied formal analysis and we validated the promised performance on a real platform.**

I. Introduction

Many safety-critical products are traditionally developed using hardware-software co-design. For example, the software of medical imaging devices is often run on dedicated hardware. However, these days custom-off-the-shelf (COTS) hardware has become an attractive alternative for the development of safety-critical devices, because the performance and programmability have significantly increased over the past decade. This trend is driven by innovations in the consumer electronics (CE) markets. Nevertheless, there are challenges that slow down the adoption of CE technology for medical devices. Firstly, the product design becomes more software oriented requiring companies to implement their existing imaging algorithms in software. Secondly, the medical application of such devices requires strict certification regarding their performance.

Just like in CE, medical imaging algorithms typically impose real-time constraints with highly transient variations in the rendering of their streams. For CE devices, however, allocating a static amount of processing resources to video applications is unsuitable [1], because it leads either to frame misses or to an over-provisioning of resources. To enable cost-effective video processing, many quality-of-service (QoS) strategies [2] have been developed. These strategies estimate the required processing resources by the processing pipeline dynamically and then allocate resources for image processing which may or may not be sufficient. In the latter case, a work-preserving approach is often taken in which the processing of the current frame is completed and a next frame is skipped [2]. However, for medical imaging applications, as considered in the current paper, the loss of video content and quality compromises are unacceptable.

In this paper, we analyze how a framework made from COTS hardware and COTS software fits the design process of medical imaging devices. The challenge with COTS hardware is that we miss a predictable execution architecture. Moreover, COTS software is not designed to give guarantees and often lacks real-time scheduling of the imaging algorithms that we use. We know however that in practice we may have good results.

This work was supported in part by the European Union's ARTEMIS Joint Undertaking for CRYSTAL under grant agreement No. 332830.

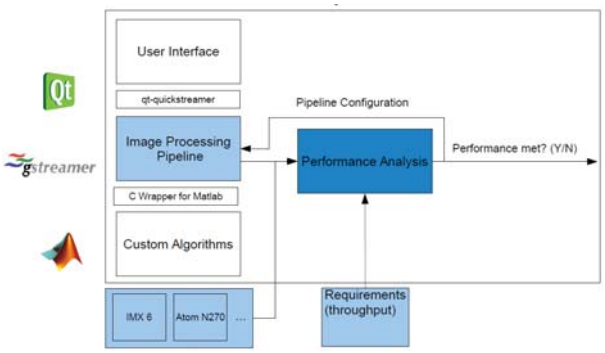


Fig. 1. Overview of tools and methods deployed in the engineering work flow in order to achieve predictable composition of medical video applications. For more details, we refer the interested reader to [3].

II. Modular software framework

In order to support modularity in the composition of a video application, we have decided to develop a flexible framework based on configurable public-domain middleware (see Figure 1), i.e., using Qt and gStreamer. The key idea behind this framework is that a video application can be decomposed into several imaging components (called plugins by gStreamer) with standard interfaces. These plugins can then be connected to each other, thereby creating a pipeline. Since Qt and gStreamer support different COTS hardware platforms, the combined framework allows for a reuse of imaging algorithms (wrapped in gStreamer's software plugins) in various setups and products.

The integration of Qt and gStreamer is work in progress. Firstly, our industrial partners are co-developing the Qt-quickstreamer plugin which extends the Qt Modeling Language (QML), so that QML can be used to compose an imaging pipeline from gStreamer plugins in an intuitive way. Secondly, Burks and Doe [4] investigated how custom imaging algorithms can be automatically imported from their development tools (Matlab Simulink) into a gStreamer plugin, i.e., an algorithm is wrapped into a plugin with a proper gStreamer interface. The integration of Qt and gStreamer is therefore expected to decouple the development of custom imaging algorithms and their integration.

Our aim is to integrate this modular software framework in the development flow of medical devices. We must therefore establish a predictable match between the execution model of gStreamer and the execution model being used at the stage of performance modeling. The remainder of the paper presents a case study in which prediction models are used to trade certain performance of an imaging application for its required processing resources during its real execution in our framework.

TABLE I
PREDICTED VERSUS EVALUATED RESOURCE USAGE FOR THE EXAMPLE PIPELINE, WITH OR WITHOUT A BACK-PRESSURED GSTREAMER IMPLEMENTATION.

Back-pressure	**Memory allocation** (number of frames per queue)	**Max. run-time memory usage** (number of frames per queue)	**Predicted throughput** (frames per second)	**Measured throughput** (frames per second)
yes	(2,1,1)	(1,1,1)	28	31.4
no	(2,2,2)	(2,2,2)	31	31.8

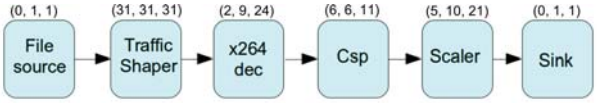


Fig. 2. An example video processing pipeline which we analysed using formal SDF analysis and which we implemented and validated in gStreamer. Each plugin in the pipeline has been benchmarked on a frame-by-frame basis, denoted by (BCET, ACET, WCET) in milliseconds.

III. USE CASE: FROM PERFORMANCE MODELS TO RESOURCE ALLOCATION AND VALIDATION

In this section, we model and implement an H264 client (see Figure 2). Since the software has to run on a medical device, we are interested in predicting, controlling and validating its execution time and memory usage. We therefore want to follow a standard design practice in which we control concurrency and memory usage to influence response times and throughput. Table I gives an overview [3] of the predicted performance and the real performance of such a controlled pipeline.

A. Experimental setup

We measure and validated the performance of our example pipeline on a X86-64 quad-core system. Each core offers two hardware threads. The example pipeline requires a number of software threads less than the number of hardware threads.

The threads are scheduled by Ubuntu 12.04 LTS (Linux kernel 3.11) and controlled by the gStreamer 0.10 and Qt 5.2 frameworks on top. The application running the pipeline is set to have the highest priority in the system and the threads get unique processor affinities (bound to separate cores). With this configuration we ensure that threads get executed as soon as possible, i.e., as mandated by our prediction models.

We fed the pipeline synthetic video sequences, generated using GStreamer's videotestsrc element (an open-source H264 encoder). They contain different patterns (white, checkers, noise and zone-plate). All sequences contain 1000 frames.

B. Constraining the data input stream

We compare two techniques to process all data in real time, i.e., without data loss and with finite sizes of queues. We therefore use a data source that reads compressed video content from a file. Some platforms (including gStreamer) support a synchronization mechanism, called *back pressure*, that suspends the data source when its output buffer is full and prevents data from getting overwritten. Alternatively, when the data source is uncontrollable, a traffic shaper can control the amount of data being pushed into the processing pipeline.

Synchronization may also be established over a network connection [5], so that the server stops sending packets when the client cannot handle more. This requires application-level streaming protocols on top of standard networking stacks, which need to be implemented and maintained. Alternatively, (without back-pressure support) the data source must constrain

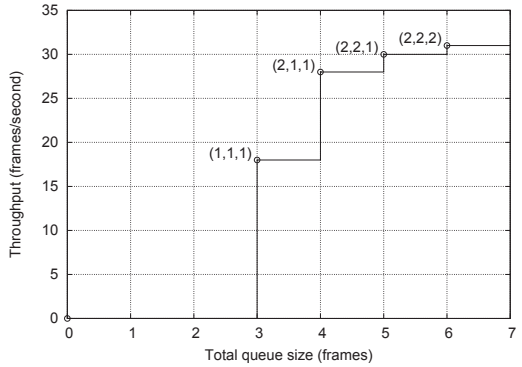


Fig. 3. Pareto optimal storage distributions of queues (AB, BC, CD) in a back-pressured example pipeline.

the amount of data being pushed into the pipeline. Some COTS network hardware is able to limit the data transmission rates by means of prioritization and buffering of specified real-time data [6]. We have implemented a traffic shaper in software as a gStreamer plugin in order to simulate streaming behaviour.

Our traffic shaper consumes and produces exactly one video frame periodically by inserting time delays between video frames. After the traffic shaper, we apply gStreamer's x264 plugin for decoding video frames, gStreamer's color-space conversion (Csp) and a synthetic spatial up scaler, which generates a random delay. These plugins all execute in a self-timed manner. Finally, the sink displays the processed video frames on the screen. For each of these gStreamer plugins, we have measured their execution times on a frame-by-frame basis for various video content; Figure 2 shows the best-case (BCET), average-case (ACET) and worst-case (WCET) execution times.

C. Concurrency control and allocation of processing resources

A gStreamer pipeline can be mapped onto several threads by explicitly placing queues between processing plugins. The plugins that are mapped upon the same thread execute in a static order, so that their execution times add up. A total of three queues, called *AB*, *BC* and *CD*, are placed after the traffic shaper, x264 decoder and Csp, respectively. With a certain positioning of queues, we can model the pipeline using the synchronous-dataflow (SDF) formalism.

An SDF graph allows us to compare the two algorithms by Stuijk et al. [7] and Salunkhe et al. [8] for computing the queue sizes and the corresponding throughput of the pipeline for setups with and without back pressure. The advantage of having a back-pressure mechanism is that waiting times of threads can be traded for throughput. Additional buffering at appropriate places in the pipeline may allow threads to work ahead and thereby increase the throughput. Figure 3 shows the Pareto optimal buffer allocations of our example pipeline obtained using the algorithm of Stuijk et al. [7].

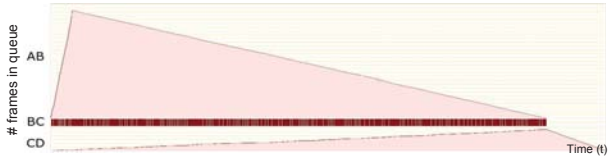


Fig. 4. Snapshot of unbounded memory usage of an unconstrained pipeline.

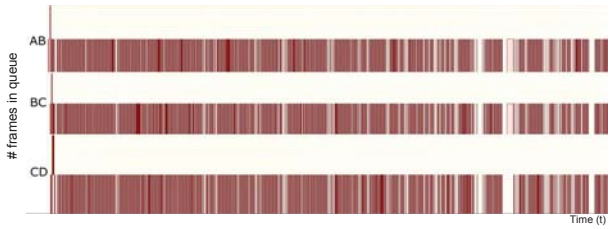


Fig. 5. Snapshot of controlled memory usage of a non-back-pressured pipeline.

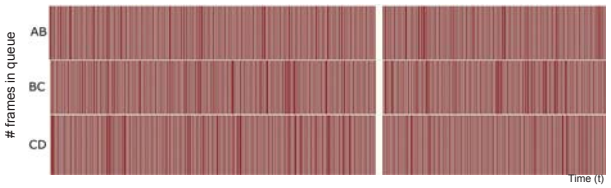


Fig. 6. Snapshot of controlled memory usage of a back-pressured pipeline.

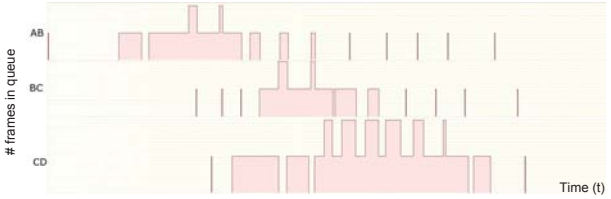


Fig. 7. Zoom in on the initialization phase of a non-back-pressured pipeline.

We recall that without back-pressure one must constrain the throughput at the input of the pipeline in order to bound the application's memory requirements. Moreover, plugins must execute in a self-timed manner, because delaying their execution may add buffer requirements to avoid data corruption. For such constrained applications, Salunkhe et al. [8] have proposed an algorithm to determine the Pareto point corresponding to the highest possible throughput. They use life-time analysis of data in the buffers based on the BCET and WCET of plugins in order to optimize the queue sizes[1]. In order to apply their algorithm, our traffic shaper limits the throughput at the input.

D. Performance validation

As shown in the methodology overview in Figure 1, the performance analysis is said to feed back configuration parameters to the application. The measured execution time parameters are the basis for a queue placement strategy, as tacitly applied in the previous subsection, and then allows us to mathematically predict trade offs in worst-case queue sizes and minimal throughput. We now validate the real-time memory usage and the real throughput of the pipeline (see Table I).

In gStreamer we log the number of buffered frames by instrumenting push and pop events of the queues in the pipeline; each buffer has the capacity of storing a video frame. Buffer access may or may not be be guarded by back pressure[2].

First, we look at a scenario of uncontrolled memory usage in which both our traffic shaper and gStreamer's back-pressure mechanism are disabled. In this scenario, the entire file is read from disk as fast as possible and stored into memory (see Figure 4). Since file readings have negligible WCETs compared to the later processing steps in the pipeline (see Figure 2), this experiment shows that, as can be expected, the memory storage requirements are proportional to the input size.

Secondly, we monitor the controlled memory usage for our pipeline (with and without back pressure). Figure 5 and Figure 6 show the number of frames $[0..2]$ stored in the queues. We confirmed that in both cases all frames in the file were actually being displayed at the output, i.e., both with and without back-pressure we report the absence of data loss. Table I reports the worst-case occupancies of the queues. Even without back pressure, the queues appear to be sized tightly and conservative.

As shown in Figure 5 and Figure 6, however, the queues in the pipeline are only occasionally fully occupied. Figure 5 also shows that initially the file reader works ahead one frame when back pressure is disabled. Figure 7 zooms in on the initial phase of the non-back-pressured pipeline. After an initialization phase, the execution pattern stabilizes and follows a repetitive order as dictated by our periodic traffic shaper.

[1]BCETs are irrelevant with back-pressure, because a delay of the earliest start time of plugins on new data can be enforced, which enables tighter life-time analysis based on just WCETs (see [8] for more details).

[2]The snapshots are created from the logged event traces with TimeDoctor [9].

IV. Conclusions

This paper presented a software framework for predictable composition of medical video applications. We configured middleware software in a way that the video pipeline is forced to execute closely in accordance with our formal application models. Formal (dataflow) analysis has been demonstrated on a case study in which we obtained optimized parallel executions of imaging algorithms by controlling execution delays and allocating memory appropriately. Since our initial experiments indicate that we can predict the performance of applications accurately, we consider our software framework a promising solution for the future design of medical streaming applications.

References

[1] D. Isović, G. Fohler, and L. Steffens, "Timing constraints of MPEG-2 decoding for high quality video: Misconceptions and realistic assumptions," in *Proc. ECRTS*, July 2003, pp. 73–82.

[2] C. C. Wüst, L. Steffens, W. F. Verhaegh, R. J. Bril, and C. Hentschel, "QoS control strategies for high-quality video processing," *Real-Time Syst.*, vol. 30(1-2), pp. 7–29, 2005.

[3] S. C. Crăcană, "Modular composition of imaging applications on commercial-off-the-shelf programmable hardware platforms," Master's thesis, Eindhoven University of Technology, Aug. 2014.

[4] S. D. Burks and J. M. Doe, "Gstreamer as a framework for image processing applications in image fusion," *Proc. SPIE*, vol. 8064, pp. 80 640M–80 640M–7, June 2011.

[5] G.-M. Muntean and L. Murphy, "Feedback-controlled traffic shaping for multimedia transmissions in a real-time client-server system," in *Springer, LNCS, ICN Networking*, 2001, vol. 2093, pp. 540–548.

[6] E. Wandeler, A. Maxiaguine, and L. Thiele, "On the use of greedy shapers in real-time embedded systems," *ACM TECS*, vol. 11(1), pp. 1–22, 2012.

[7] S. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *Proc. DAC*, 2006, pp. 899–904.

[8] H. Salunkhe, O. Moreira, and K. van Berkel, "Buffer allocation for real-time streaming on a multi-processor without back-pressure," in *Proc. ESTIMedia*, Oct. 2014.

[9] M. Rutten, "TimeDoctor Version 1.4.3," May 2013. [Online]. Available: http://sourceforge.net/projects/timedoctor/

Increasing the Predictability of Modern COTS Hardware through Cache-Aware OS-Design

Hendrik Borghorst
Embedded System Software
Computer Science 12, Technische Universität Dortmund
Email: hendrik.borghorst@udo.edu

Olaf Spinczyk
Embedded System Software
Computer Science 12, Technische Universität Dortmund
Email: olaf.spinczyk@udo.edu

***Abstract*—Real-time operating systems have been around for some time, but they are never designed for being used on modern multi-core processors with unpredictable timing behavior. An important source of unpredictability is the different timing between the processor and the DRAM-controller. Operating-system-based cache management is one possibility to reduce the timing variations of the processor by controlling the code and data which resides in the cache. The cache eliminates the timing differences between the memory and the processor.**

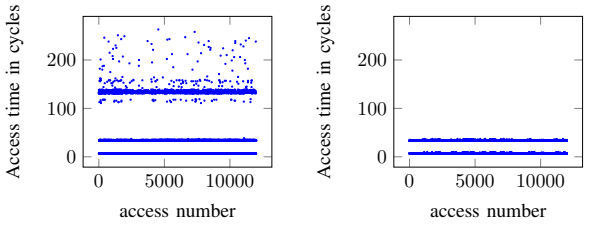
I. Motivation and related work

With increasing complexity of today's multi-core processors, their timing behavior gets more unpredictable, which leads to big fluctuations of the execution times for tasks and operating system functions like interrupt handling. This means that the overall response time of a system depends on the timing behavior of all the shared resources like the caches or buses [1]. This problem prohibits the use of such systems for time-critical applications like cyber-physical systems. Cyber-physical systems need to react on certain events within a predictable time bound. Therefore it is critical that the overall response time of the operating system is stable. Different timings of the main processor and the memory can be neutralized by the use of caches. But caches can introduce new problems like unwanted cache eviction which would also lead to unstable execution times.

Cache partitioning can be used to prevent cache eviction for multi-task or multi-core applications. Cache preloading can be used to prevent timing variations caused by simultaneous bus accesses from multiple participants.

R. Mancuso et al. proposed a cache management framework for applications running on the Linux operating system [2]. The approach, presented in their paper [2], loads specific application code and data to a partition of the shared cache and locks it afterwards. This approach shows an significant reduction of the application's execution time variation. Their method eliminates the timing variations caused by shared caches and random memory accesses. In contrast to this method the later presented approach works on the level of the operating system. The advantage of managing the cache within the operating system allows operating system functionality to be predictable as well.

J. Liedtke et al. worked on operating system controlled caches for single-core processors [3]. They used a technique called cache coloring to reduce the risk of cache eviction for multi-tasking applications. They could show that it is possible to reduce the variation of the execution time with the use of cache partitioning. Nonetheless their work is based on single-core systems and does not consider the properties of a multi-core system with shared resources such as the memory and buses.



(a) without preloading (b) with preloading

Fig. 1: Comparison of data access times (64kB range)

As a preparation to proof the later presented operating system concept, we created a prototype operating system which used a basic cache management to preload tasks on activation. To evaluate the concept of cache resource management, we ran four tasks on a dual-core *ARM Cortex-A9* processor. Each task was confined to a distinct memory area and accessed random memory addresses with and without preloading and locking of the shared L2 cache. The results of this test are shown in Figure 1 where single memory accesses are shown with their corresponding access time. The diagram in Figure 1a illustrates that there is a very high fluctuation of memory access times. For comparison Figure 1b demonstrates that the preloading of the data to access shows a significant reduction of the previously mentioned access time fluctuation. The benchmark was done for a memory area of 64 kB per task which is twice the size of the level 1 cache, so there are already level 1 cache misses which are represented by the upper one of the two distinct lines in the diagrams.

The execution times for the cache preloading itself were tested separately. It was measured that the execution times are proportional to the preloading size, if it is guaranteed that only one processor core is preloading at the same time. This knowledge is crucial to the whole idea to get the system predictable.

With this knowledge it is possible to create an operating system that takes control over the content inside the cache so that the execution times for operating system functions and interrupt handling become predictable. To achieve this we present an operating system model which is designed with the sources of unpredictability in mind.

II. Operating System Model

The idea of the new operating system is to sort out some problems that existing real-time operating systems present when they are executed on modern multi-core architectures that utilize some shared resources like caches and memory buses.

Modern multi-core processors often include shared caches that are structured as associative caches which features multiple cache ways to reduce the cache miss rate. Each cache way represents a part of the whole cache. The target architecture used for this paper features an shared second level cache with 16 cache ways with 64 kB capacity each.

To solve the issues of unpredictable caches and memory access latencies, the operating system and the applications have to fit inside the partitioned shared L2 cache. One solution to achieve this, is to divide the system into small pieces. We call each of these pieces *operating system component* (OSC). The implementation of the operating system is done in a highly modular way so that we can define very fine granular components. These components can than be grouped together into larger components to be optimal for the desired target platform. The optimal component size depends on the specific sizes of the cache structure of the hardware. For example a component needs to be smaller than the biggest shared cache and not to small which would effectively be the same like one random memory access. A good size would be a multiple of the cache way size.

One problem with existing embedded operating systems is that there is usually only one stack per core when using operating system functions. This makes it hard to predict where the local data is located when the processor jumps to operating system code. This could lead to cache eviction if operating system functionality is requested. To solve this, each OSC contains its own stack by what we enable the operating system to contain all code and data on the level of OSCs.

Another problem with existing solutions is that normal function calling allows no control over the data and control flow which could lead to cache eviction problems. To solve this the new operating system prohibits direct data passing between OSCs. Instead the system operates on a strictly event-based nature. These events are handled by the operating system so that it can control the contents of the cache.

Each OSC can define *input triggers* which will activate a specific OSC. Each input trigger needs a function which is called after the OSC is activated. To activate these input triggers, output events, that each OSC can define, are required. These events can be connected to the input triggers of other OSCs. The creation of the connections between events and triggers of OSCs is done during the time of compilation. For performance reasons this is a static linkage with hard coded function pointers. If an OSC wants to send an event it needs to do it by the use of a *system call*.

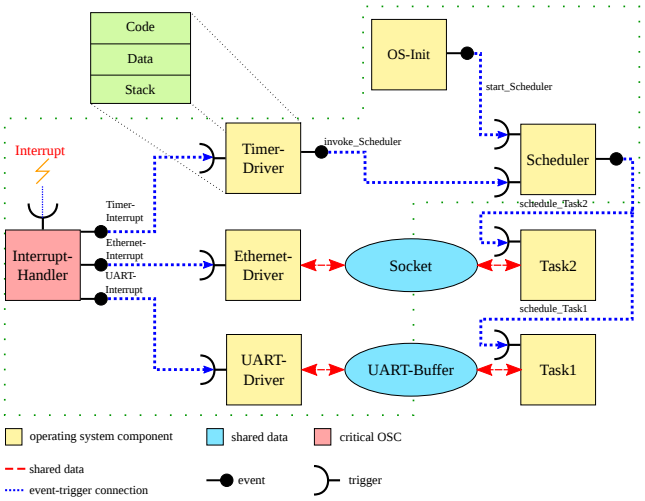


Fig. 2: Operating system model with critical/non-critical components

To solve the issue of uncontrollable data flow, the operating system specification allows shared data between two or more OSCs. Shared data must stay inside the cache until no OSC needs it anymore. These shared data objects need to be cache-aware by design so that the application developer needs to make the data structures efficient on constrained space. There are several approaches on cache-aware data structures and their optimizations. For example T. Chilimbi et.al. present a way to make pointer-based data structures cache-aware [4]. They introduce a method which can optimize different data structures, that are based on indirect data accesses, via a modified version of the dynamic memory management method *malloc*. In addition they present a way to specifically optimize tree-based data structures so that they reduce the number of cache-misses drastically. Those methods could be integrated within the operating system so that the application developer is presented with an API that takes care of the cache prefetching. It should be noted that the focus of this operating system is not on heavy data computation but on comparable small real-time task-sets with data structures that fit into the shared caches.

Another critical problem of the system is the interrupt handling because it is impossible to predict when interrupts arrive. Therefore it is critical that the whole minimal first stage of the interrupt handling is locked permanently to the cache. The first stage would then emit an event with the interrupt number. This event is handled like any other event. This ensures that the interrupt handling stays predictable by assuring that the unpredictable part always remains inside the cache. The preloading of the remaining interrupt handling is by definition predictable. With this model a periodic behavior is also possible to achieve by using a timer with a periodic configuration but the system is not limited to periodic configurations.

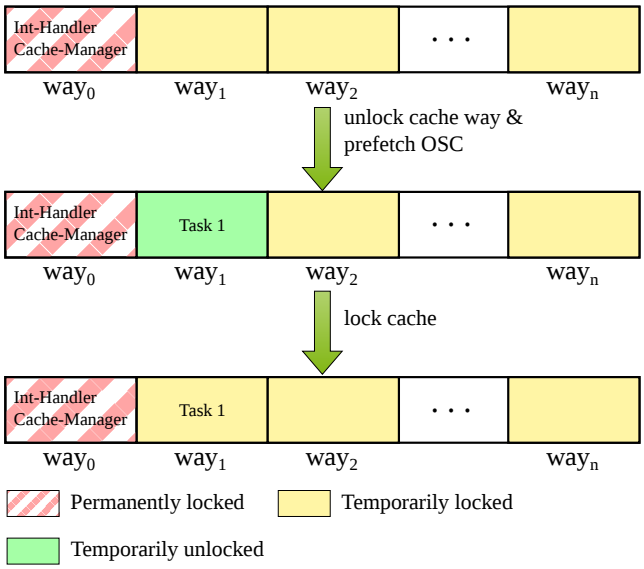


Fig. 3: Cache way states during an OSC-transition

A schematic representation of the presented operating system model is shown in Figure 2. The figure visualizes how different OSCs could be connected with each other. As highlighted in the figure, each OSCs consists of an separate code, data and stack segment. Events connect OSCs with each other as visualized by the punctuated lines. The *Interrupt-Handler* is marked in red because it is time critical and needs to stay locked permanently. The ellipsis in Figure 2 represents a shared data object. The figure shows an operating system which uses a timer component to emulate a time-based behavior. The operating system itself is not limited to time-based events and could react predictable to sporadic events as well because the critical part of the interrupt handler handles interrupts within guaranteed time bounds.

This operating system needs a special kind of scheduler because it does not schedule tasks directly but needs to schedule the execution of events. Events can be prioritized so that time critical events are handled before uncritical events. The scheduler needs to minimize the cache eviction and data flow from the main memory as well. As a result of this it needs to optimize which OSCs are active inside the cache and which can be swapped away.

Figure 3 shows the different states of the cache during the execution of the system. It represents an simplified version of a cache structured into n cache ways. Each cache way can be locked individually. Therefore it is possible to control the cache content manually by unlocking only one cache way at once which guarantees that the data is allocated to that specific way during prefetching. The uppermost row visualizes the state in which only the critical parts are locked and loaded inside the cache. This is the state in which the operating systems resides after successful initialization. The row in the middle of Figure 3 represents the cache state in which an OSC was prefetched, right before the needed cache way gets locked again. The cache management unlocks only the cache ways that are needed for the OSC to activate. This is not limited to only one cache way per OSC. It is also possible for OSCs to spread across multiple cache ways. In this case the cache ways would be unlocked and prefetched consecutively. After successful prefetching of the OSC the cache management locks all cache ways again to prevent cache eviction from happening which is the state of the bottom row in Figure 3.

III. Hardware Platform

For now the operating system needs special hardware features to control the cache. Cache locking is needed to prevent cache eviction when loading new OSCs. For the purposes of evaluation we used a Texas Instrument OMAP4460 ARM processor [5] that uses an external level 2 cache controller and is compatible to the ARM Cortex-A9 processor. This cache controller has sophisticated control features like cache lockdown by cache way and by core [6]. This means that it is possible to control in what cache way new cached data gets allocated. The processor was clocked at 921 MHz during the experiments.

The level 2 cache features 16 cache ways, each with a size of 64 kB. Thus a optimal size for the OSCs would be 64 kB or multiples of this value. For now the OSCs get aligned to this size during the linking process which makes it convenient to prefetch those components to specific cache ways.

IV. Ongoing and Future Work

The presented operating system is just a proof of concept for now. We evaluated that it is possible to take control over the contents of the shared cache with a basic cache control implementation that prefetched data and code to the cache and locked the cache afterwards. Another thing we measured is the required time to prefetch bulk data. Our results show that we can achieve a prefetch time which is linear to the prefetch size. It was measured that the prefetch time per byte is around 8 clock cycles if more than 128 bytes are prefetched in a bulk transfer. For the component size of 64 kB this bulk transfer require around 0.57 ms.

In the future we intend to focus our research on some specific topics regarding the operating system model. One part of this will be the scheduling of the event dispatching. There are several optimization criteria for the scheduling. For instance the minimization of cache evictions, to maximize the overall processor utilization and to keep the overall response time of the system minimal.

Furthermore we intend to analyze the timing behavior of the operating system. This includes analysis of the transition times, prefetch times and the OSC function execution times to guarantee that the execution time of the whole system will stay inside a time bound.

Also the operating system needs a good software development model. It is important that the implementation of the event-based system is not overly complicated. One possible solution for this could be the use of an aspect-orientated language like *AspectC++* [7].

Another topic to explore is how to extend the supported hardware base. One potential substitute for locking critical OSCs inside the cache could be a static ram which many new embedded processors include. It may also be possible to isolate one core of the system to interrupt handling. This would mean that the first stage interrupt handler should not be evicted from the level 1 cache if it is small enough. For systems lacking the support for cache locking the use of traditional software-based cache partitioning algorithms is necessary [8].

Finally the operating system needs evaluation under several circumstances. We expect that the manual management of the cache content will introduce some overhead on the computational performance of the system. Therefore an comparison with existing operating systems like RT-Linux [9] or RTEMS [10] is needed. The overall system response time also needs evaluation with various workloads.

V. Conclusion

This paper presents a possible solution for the unstable execution times of modern multi-core systems on the level of the operating system. This is done by manually controlling which data and program code resides in the cache. By this the operating system shifts the unpredictability of random DRAM-accesses to predictable bulk memory transfers. To realize this the operating system operates on a event-based nature and is structured as a set of OSCs, which can be loaded into the cache on-demand or permanently based on a cache scheduling strategy. At the moment the operating system only exists as a proof of concept but we intend to explore this concept further.

References

[1] D. Dasari, B. Akesson, V. Nelis, M. Awan, and S. Petters, "Identifying the sources of unpredictability in COTS-based multicore systems," in *2013 8th IEEE International Symposium on Industrial ES (SIES)*, June 2013, pp. 39–48.

[2] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, April 2013, pp. 45–54.

[3] J. Liedtke, H. Haertig, and M. Hohmuth, "OS-controlled cache predictability for real-time systems," in *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, ser. RTAS '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 213–.

[4] T. Chilimbi, M. Hill, and J. Larus, "Making pointer-based data structures cache conscious," *Computer*, vol. 33, no. 12, pp. 67–74, Dec 2000.

[5] "OMAP4460 ES1.x Technical Reference Manual," http://www.ti.com/lit/pdf/swpu235, accessed: 2015-02-20.

[6] "PL310 Cache Controller - Technical Reference Manual," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0246a/DDI0246A_l2cc_pl310_r0p0_trm.pdf, accessed: 2015-04-18.

[7] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An aspect-oriented extension to C++," in *Proceedings of the 40th International Conference on Technology of OO Languages and Systems (TOOLS Pacific '02)*, Sydney, Australia, Feb. 2002, pp. 53–60.

[8] F. Mueller, "Compiler support for software-based cache partitioning," *SIGPLAN Not.*, vol. 30, no. 11, pp. 125–133, Nov. 1995. [Online]. Available: http://doi.acm.org/10.1145/216633.216677

[9] "Real-Time Linux Wiki," https://rt.wiki.kernel.org/index.php/Main_Page, accessed: 2015-04-29.

[10] A. Colin and I. Puaut, "Worst-case execution time analysis of the RTEMS real-time operating system," in *Real-Time Systems, 13th Euromicro Conference on, 2001.*, 2001, pp. 191–198.

Evaluating the Isolation Effect of Cache Partitioning on COTS Multicore Platforms

Heechul Yun, Prathap Kumar Valsan
University of Kansas
{heechul.yun, prathap.kumarvalsan}@ku.edu

***Abstract*—Tasks running on a Commercial Off-The-Shelf (COTS) multicore processor can suffer significant execution time variations due to inter-core interference in accessing shared hardware resources such as shared last-level cache (LLC). Page-coloring is a well-known OS technique, which can partition the LLC space among the cores, to improve isolation.**

In this paper, we evaluate the effectiveness of page-coloring based cache partitioning on three COTS multicore platforms. On each platform, we use two carefully designed micro-benchmarks and perform a set of experiments, which generate very high interference at the shared LLC, with and without cache partitioning.

We made two interesting findings: (1) Without cache-partitioning, a task can suffer up to 103X slowdown due to interference at the shared LLC. (2) More surprisingly, we found that cache partitioning does not necessarily eliminate interference in accessing the LLC, even when the concerned task only accesses its dedicated cache partition (i.e., all memory accesses are cache hits); we observe up to 14X slowdown in such a configuration. We attribute this to contention in the Miss Status Holding Registers (MSHRs) of the LLC.

I. Introduction

Commercial Off-The-Shelf (COTS) multicore processors are increasingly being adopted in autonomous cars, unmanned aerial vehicles (UAV), and other critical cyber-physical systems (CPS). While these COTS multicore processors offer numerous benefits, they do not provide predictable timing—a highly desired property in many CPS applications.

In a COTS multicore system, the execution time of a task is determined not only by the task and the underlying hardware architecture, but also by co-runners on different cores due to interference in the shared hardware resources. One of the major source of interference is shared last-level cache (LLC). When more than two tasks execute in parallel on cores that share the LLC, tasks can evict each other's valuable cache-lines, which cause negative performance impacts. Cache-partitioning, which partitions the cache space among the cores, is a well-known solution to counter this problem [11], [15].

In this paper, we evaluate the effectiveness of cache partitioning in improving timing predictability on three modern COTS multicore platforms: one in-order (ARM Cortex-A7) and two out-of-order (ARM Cortex-A15 and Intel Nehalem) architecture based quad-core platforms. We use two carefully designed micro-benchmarks and perform a set of experiments to investigate the impacts of shared LLC to the application execution times—*with and without applying cache-partitioning*. In designing the experiments, we consider memory-level-parallelism (MLP) of modern COTS multicore architecture—non-blocking caches and DRAM bank parallelism—and intend to find worst-case scenarios where a task's execution time suffers the most slowdown due to cache interference.

From the experiments, we made several interesting findings. First, unlimited cache sharing can cause unacceptably high interference; we observe up to 103X slowdown (i.e., the task's execution time is increased by 103 times due to co-runners on different cores). Second, cache-partitioning is effective especially in the in-order architecture, as it almost completely eliminates cache-level interference. In out-of-order architectures, however, we observe significant interference even after cache partitioning is applied. Concretely, we observe up to 14X slowdown even when the task under consideration only accesses its dedicated cache partition (i.e., all memory accesses are cache hits). We attribute this to contention in the shared miss-status holding registers (MSHRs) [8] in the LLC (See Section V).

Our contributions are as follows: (1) experiment designs that help expose the degree of interference in the shared LLC; (2) detailed evaluation results on three COTS multicore platforms showing the performance impacts of the cache-level interference. To the best of our knowledge, this is the *first* paper that reports the worst-case performance impact of MSHR contention on COTS multicore platforms.

The rest of the paper is organized as follows. Section II describe necessary background on modern COTS multicore architecture. Section III describe the three COTS multicore platforms we used in this paper. Section IV experimentally analyze MLP of the hardware platforms. Section V investigate the impacts of cache (LLC) interference on the tested platforms. We conclude in Section VI.

II. Background

In this section, we provide necessary background on COTS multicore architecture and software based resource partitioning techniques.

A typical modern COTS multicore architecture is composed of multiple independent processing cores, multiple layers of private and shared caches, and a shared memory controller(s) and DRAM memories. To support high performance, processing cores in many embedded/mobile processors are adopting out-of-order designs in which each core can generate multiple outstanding memory requests [12], [4]. Even if the cores are based on in-order designs, in which one core can only generate one outstanding memory request at a time, they collectively can generate multiple requests to the shared memory subsystem. Therefore, the memory subsystem must be able to handle multiple parallel memory requests. The degree of parallelism supported by the shared memory subsystem—the caches and main memory—is called *Memory-Level Parallelism (MLP)* [5].

TABLE I: Evaluated COTS multicore platforms.

	Cortex-A7	Cortex-A15	Nehalem
Core	4cores@0.6GHz in-order	4cores@1.6GHz out-of-order	4cores@2.8GHz out-of-order
LLC	512KB, 8way	2MB, 16way	8MB, 16way
DRAM	2GB, 16banks	2GB, 16banks	4GB, 16banks

A. Non-blocking caches and MSHRs

At the cache-level, non-blocking caches are used to handle multiple simultaneous memory accesses. On a cache-miss, the cache controller allocates a MSHR (miss status holding register) to track the status of the ongoing request and the entry is cleared when the corresponding memory request is serviced from the lower-level memory hierarchy. For the last-level cache (LLC), each cache-miss request is sent to the main memory (DRAM). As such, the number of MSHRs in the LLC effectively determines the maximum number of outstanding memory requests directed to the DRAM controller. It is important to note that MSHRs are typically shared among the cores [7] and when there are no remaining MSHRs, further accesses to the cache—*both* hits and misses—are prevented until free MSHRs become available [1]. Because of this, even if the cache space is partitioned among cores using software cache partitioning mechanisms, in which each core is guaranteed to have its dedicated cache space, accessing the cache partition does not necessarily guarantee interference freedom as we will demonstrate in Section V.

B. DRAM and memory controllers

At the DRAM-level, a DRAM chip is divided into multiple *banks*, which can be accessed in parallel. As such, the number of banks determines the parallelism available on DRAM. To maximize the bank-level parallelism, DRAM controllers typically use an *interleaved mapping*, which maps consecutive physical addresses into different DRAM banks.

C. Cache and DRAM bank Partitioning

Cache partitioning has been studied extensively to provide better isolation and efficiency. Page coloring is a well-known software technique which partitions cache-sets among the cores [11], [15], [9], [16]. Also, there are a variety of hardware based partitioning mechanisms such as cache-way based partitioning [13], which is supported in some commercial processors [4]. More recently, several DRAM bank partitioning methods, mostly based on page-coloring, have been proposed to limit bank-level interference [17], [10], [14].

III. Evaluation Setup

In this paper, we use two COTS multicore platforms: an Intel Xeon W3553 (Nehalem) based desktop machine and an Odroid-XU+E single-board computer (SBC). The Odroid-XU+E board equips a Samsung Exynos 5410 processor which includes both four Cortex-A15 and four Cortex-A7 cores in a big-LITTLE [6] configuration. Thus, we use the Odroid-XU+E platform for both Cortex-A15 and Cortex-A7 experiments. Table I shows the basic characteristics the three platform configurations we used in our experiments. We run Linux 3.6.0 on the Intel Xeon platform and Linux 3.4.98 on the Odroid-XU+E platform; both kernels were patched with PALLOC [17] to be able to partition the shared LLC at runtime. When cache-partitioning is applied, the shared LLC is evenly partitioned among the four cores (i.e., each core gets 1/4 of the LLC space).

```
1  static int* list[MAX_MLP];
2  static int next[MAX_MLP];
3
4  long run(long iter, int mlp)
5  {
6      long cnt = 0;
7      for (long i = 0; i < iter; i++) {
8          switch (mlp) {
9          case MAX_MLP:
10             .
11             .
12         case 2:
13             next[1] = list[1][next[1]];
14             /* fall-through */
15         case 1:
16             next[0] = list[0][next[0]];
17         }
18         cnt += mlp;
19     }
20     return cnt;
21 }
```

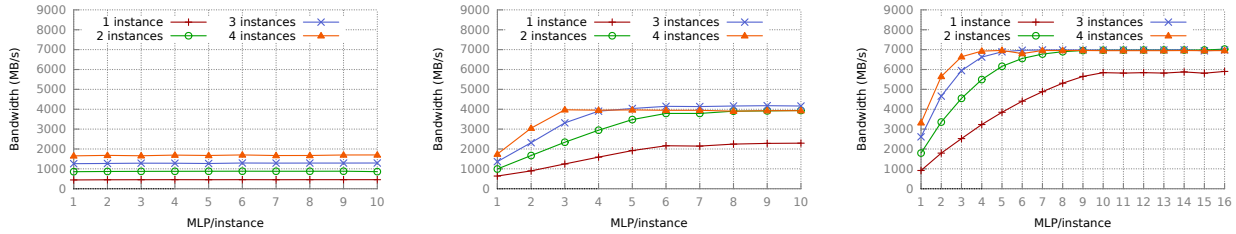
Fig. 1: MLP micro-benchmark. Adopted from [3].

IV. Understanding Memory-Level Parallelism

In this section, we identify memory-level parallelism (MLP) of the three multicore platforms using an experimental method described in [3].

In the following, we first briefly describe the method for better understanding. The method uses a pointer-chasing micro-benchmark shown in Figure 1. The benchmark traverses a number of linked-lists. Each linked-list is randomly shuffled over a memory chunk of twice the size of the LLC. Hence, accessing each entry is likely to cause a cache-miss. Due to data-dependency, only one cache-miss can be generated for each linked list. In an out-of-order core, multiple lists can be accessed at a time, as it can tolerate up to a certain number of outstanding cache-misses. Therefore, by controlling the number of lists (determined by *mlp* parameter in Figure 1) and measuring the performance of the benchmark, we can determine how many outstanding misses one core can generate at a time, which we call *local MLP*. We also varied the number of benchmark instances from one to four and measure the aggregate performance to investigate the parallelism of the entire shared memory hierarchy, which we call *global MLP*.

Figure 2 shows the results. Let us first focus on single instance results. For Cortex-A7, increasing the number of lists (X-axis) does not have any performance improvement. This is because Cortex-A7 is in-order architecture in which only one outstanding request can be made at a time. On the other hand, for Cortex-A15, the performance improves up to six lists and then saturates. This suggests that the Cortex-A15's local MLP is six. In case of Nehalem, performance improves up to ten concurrent lists, suggesting its local MLP is ten. As we increase the number of benchmark instances, the point of saturation become shorter in both Cortex-A15 and Nehalem. When four instances are used in



(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 2: Aggregate memory bandwidth as a function of MLP/benchmark.

TABLE II: Local and global MLP

	Cortex-A7	Cortex-A15	Nehalem
local MLP	1	6	10
global MLP	4	11	16

Cortex-A15, the aggregate performance saturates at three. This suggests that the global MLP of Cortex-A15 is close to 12; according to [2], the LLC can support up to 11 outstanding cache-misses (global MLP of 11). Note that the global MLP can be limited by either of the two factors: the size of MSHRs in the shared LLC or the number of DRAM banks. In the case of Cortex-A15, the limit is likely determined by the number of MSHRs of the LLC (11), because the number of banks is bigger than that (16). In the case of Nehalem, on the other hand, the performance saturates when the global MLP is about 16, which is likely determined by the number of banks, rather than the number of MSHRs; according to [7], the Nehalem architecture supports up to 32 outstanding cache-misses. Table II shows the identified local and global MLP of the the three platforms we tested.

V. Understanding Cache Interference

In this section, we investigate performance impacts of cache-level interference on COTS multicore platforms.

While most previous research on shared cache has focused on unwanted cache-line evictions that can be solved by cache partitioning, little attention has been paid to the problem of shared MSHRs in non-blocking caches, which also can cause interference. As we will see later in this section, cache partitioning does not necessary provide isolation even when the application's working-set fits entirely in a dedicated cache partition, due to contention in the shared MSHRs.

To find out worst-case interference, we use various combinations of two micro-benchmarks: *Latency* and *Bandwidth* [18]. Latency is a pointer chasing synthetic benchmark, which accesses a randomly shuffled single linked list. Due to data dependency, Latency can only generate one outstanding request at a time. Bandwidth is another synthetic benchmark, which sequentially reads or writes a big array; we henceforth refer *BwRead* as Bandwidth with read accesses and *BwWrite* as the one with write accesses. Unlike Latency, Bandwidth can generate multiple parallel memory requests on an out-of-order core as it has no data dependency.

Table III shows the workload combinations we used. Note that the texts with parentheses—(LLC) and (DRAM)—indicate working-set sizes of the respective benchmark. In case of (LLC), the working size is configured to be smaller than 1/4 of the shared LLC size, but bigger than the size of the last core-private cache. [1] As such, in case of (LLC), all memory accesses are LLC hits in both cache partitioned and non-partitioned cases. In case of (DRAM), the working-set size is the twice the size of the LLC so that all memory accesses result in LLC misses.

TABLE III: Workloads for cache-interference experiments.

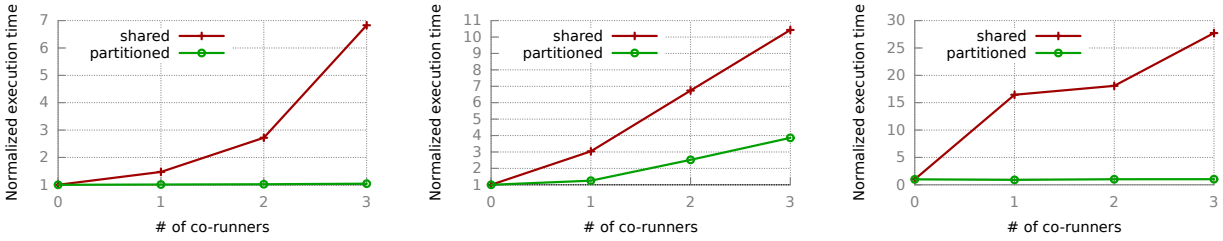
Experiment	Subject	Co-runner(s)
Exp. 1	Latency(LLC)	BwRead(DRAM)
Exp. 2	BwRead(LLC)	BwRead(DRAM)
Exp. 3	BwRead(LLC)	BwRead(LLC)
Exp. 4	Latency(LLC)	BwWrite(DRAM)
Exp. 5	BwRead(LLC)	BwWrite(DRAM)
Exp. 6	BwRead(LLC)	BwWrite(LLC)

In all experiments, we first run the subject task on Core0 and collect its solo execution time. We then co-schedule an increasing number of co-runners on the other cores (Core1-3) and measure the response times of the subject task. We repeat the experiment on the three test platforms with and without cache partitioning.

A. Exp. 1: Latency(LLC) vs. BwRead(DRAM)

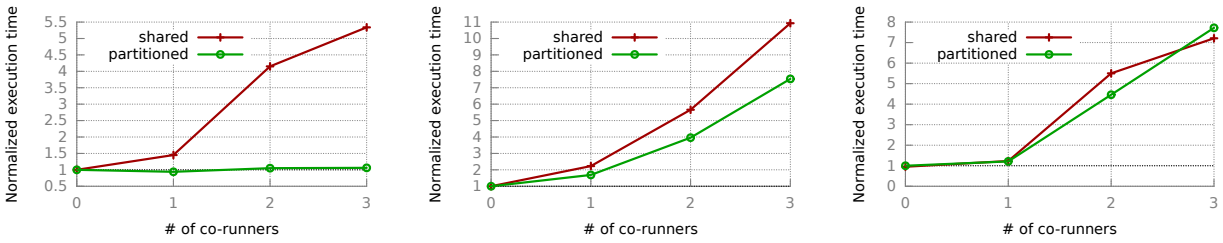
In the first experiment, we use the Latency benchmark as a subject and the BwRead benchmark as co-runners. Recall that BwRead has no data dependency and therefore can generate multiple outstanding memory requests on an out-of-order processing core (i.e., ARM Cortex-A15 and Intel Nehalem core). Figure 3 shows the results. When cache-partitioning is not applied, *shared*, the response times of the Latency benchmark are increased dramatically in all three platforms—up to 6.7X in Cortex-A7, 10.4X in Cortex-A15, and 27.7X in Nehalem. This is because cache-lines of the Latency benchmark are evicted by the co-running BwRead benchmark instances. If not the co-runners, those cache-lines would never have been evicted. On the other hand, applying cache-partitioning is shown to be effective in preventing such cache-line evictions hence providing performance isolation, especially in Cortex-A7 and Intel Nehalem platforms. In the Cortex-A15 platform, however, the response time is still increased by up to 3.9X even after partitioning the cache. This is an unexpectedly high degree of interference considering the fact that the

[1] The the last core-private cache is L1 for ARM Cortex-A7 and Cortex-A15 while it is L2 for Intel Nehalem.



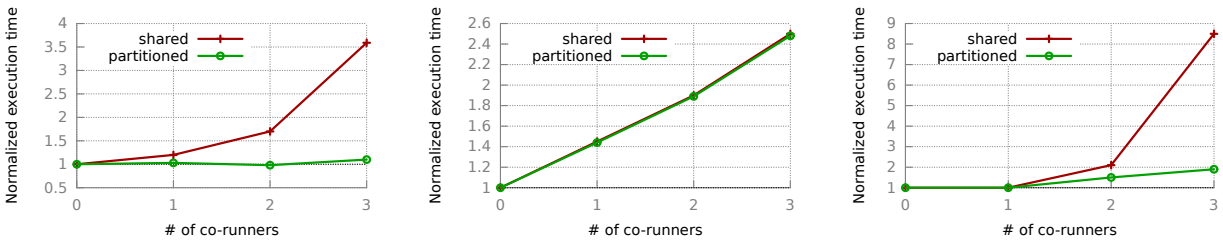
(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 3: [Exp.1] Slowdown of Latency(LLC) with BwRead(DRAM) co-runners.



(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 4: [Exp.2] Slowdown of BwRead(LLC) with BwRead(DRAM) co-runners.



(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 5: [Exp.3] Slowdown of BwRead(LLC) with BwRead(LLC) co-runners.

cache-lines of the subject benchmark, Latency, are not evicted by the co-runners as a result of cache partitioning. to the DRAM bandwidth contention [17]. The following experiment, however, shows it is not the case.

B. Exp. 2: BwRead(LLC) vs. BwRead(DRAM)

To further investigate this phenomenon, the next experiment uses the BwRead benchmark for both the subject task and the co-runners. Therefore, both the subject and co-runners now generate multiple outstanding memory requests to the shared memory subsystem in out-of-order architectures. Figure 4 shows the results. Note that while the behavior of Cortex-A7 is similar to the previous experiment, the behaviors of Cortex-A15 and Nehalem are considerably different. In the Nehalem platform, in particular, *the performance isolation benefit of cache partitioning is completely eliminated* as the subject benchmark suffers from the similar degree of slowdowns regardless of cache-partitioning. In other words, the results suggest that cache-partitioning does not necessary provide expected performance isolation benefits in out-of-order architectures. We initially suspected the cause of this phenomenon is likely the bandwidth competition at the shared cache, similar

C. Exp. 3: BwRead(LLC) vs. BwRead(LLC)

In this experiment, we again use the BwRead benchmark for both the subject and the co-runners but we reduced the working-set size of the co-runners to (LLC) so that they all can fit in the LLC. If the LLC bandwidth contention is the problem, this experiment would cause even more slowdowns to the subject benchmark as the co-runners now need more LLC bandwidth. Figure 5, however, does not support this hypothesis. On the contrary, the observed slowdowns in both Cortex-A15 and Nehalem are much less, compared to the previous experiment in which co-runners' memory accesses are cache misses and therefore use less cache bandwidth.

MSHR contention: To understand this phenomenon, we first need to understand how non-blocking caches processes cache accesses from the cores. As described in Section II, MSHRs are used to allow multiple outstanding cache-misses. If all MSHRs are in use, however, the cores can

no longer access the cache until a free MSHR becomes available. Because servicing memory requests from DRAM takes much longer than doing it from the LLC, cache-miss requests occupy MSHR entries longer. This causes a shortage of MSHRs, which will in turn stall additional memory requests even when they are cache hits.

D. Exp. 4,5,6: Impact of write accesses

In the next experiments, we further validate the problem of MSHR contention by using the BwWrite benchmark as co-runners. BwWrite updates a large array and therefore generates a line-fill (read) and a write-back (write) for each memory access. The additional write-back requests add more pressure in DRAM and therefore delay the processing of line-fill requests, which in turn further exacerbate the shortage of MSHRs. Figure 6, Figure 7, and Figure 8 show results. As expected, the subject tasks generally suffer even more slowdowns due to the additional write-back memory traffic.

E. Summary

Figure 9 show the maximum observed slowdowns in all experiments. When the LLC is partitioned, we observed up to 14.2X slowdown on Cortex-A15, 7.9X slowdown on Nehalem, and 2.1X slowdown on Cortex-A7. When the LLC is not partitioned, we observed up to 26.3X slowdown on Cortex-A15, 103.7X slowdown on Nehalem, and 6.8X slowdown on Cortex-A7.

In summary, while cache space competition (i.e., cache-line evictions) is certainly an important source of interference, eliminating the space competition through cache-partitioning does not necessary provide ideal isolation in COTS multicore platforms due to the characteristics of non-blocking caches. Through a series of experiments, we demonstrated that the MSHR competition can also cause significant interference, especially in out-of-order cores.

VI. Conclusion

Many prior works focus on cache partitioning to ensure predictable cache performance. In this paper, we showed that cache partitioning does not necessarily provide predictable cache performance in modern COTS multicore platforms that use non-blocking caches to exploit memory-level-parallelism (MLP). We quantified the degree of MLP on three COTS multicore platforms and performed a set of experiments that are specially designed to expose worst-case interference in accessing the shared LLC among the cores.

The results showed that while cache-partitioning help reduce interference, it can still suffer significant interference—up to an order of magnitude slowdown—even when the task under consideration accesses its own dedicated cache partition (i.e., all cache-hits). This is because there are other important shared resources, particularly MSHRs, which need to be managed in order to provide better isolation on COTS multicore platforms. We plan to address the issue as our future work.

References

[1] Memory system in gem5. http://www.gem5.org/docs/html/gem5MemorySystem.html.

[2] ARM. *Cortex-A15 Technical Reference Manual, Rev: r2p0*, 2011.

[3] D. Eklov, N. Nikolakis, D. Black-Schaffer, and E. Hagersten. Bandwidth bandit: quantitative characterization of memory contention. In *Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[4] Freescale. *e500mc Core Reference Manual*, 2012.

[5] A. Glew. MLP yes! ILP no. *ASPLOS Wild and Crazy Idea Session98*, 1998.

[6] P. Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 2011.

[7] Intel. *Intel ®64 and IA-32 Architectures Optimization Reference Manual*, April 2012.

[8] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *International Symposium on Computer Architecture (ISCA)*, pages 81–87. IEEE Computer Society Press, 1981.

[9] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High Performance Computer Architecture (HPCA)*. IEEE, 2008.

[10] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Parallel Architecture and Compilation Techniques (PACT)*, pages 367–376. ACM, 2012.

[11] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-Time Cache Management Framework for Multi-core Architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013.

[12] NVIDIA. *NVIDIA Tegra K1 Mobile Processor, Technical Reference Manual Rev-01p*, 2014.

[13] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 117–128. IEEE, 2002.

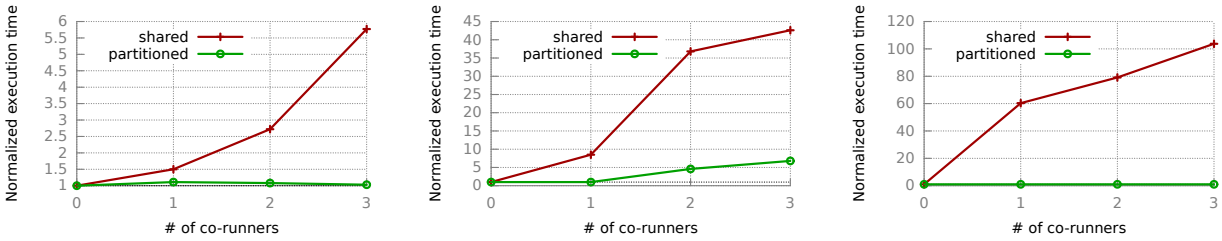
[14] N. Suzuki, H. Kim, D. d. Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *Computational Science and Engineering (CSE)*, pages 685–692. IEEE, 2013.

[15] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making Shared Caches More Predictable on Multicore Platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[16] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: a dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 381–392. ACM, 2014.

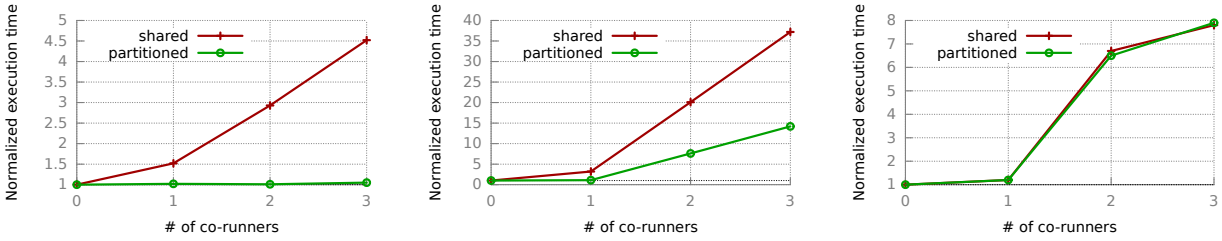
[17] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[18] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.



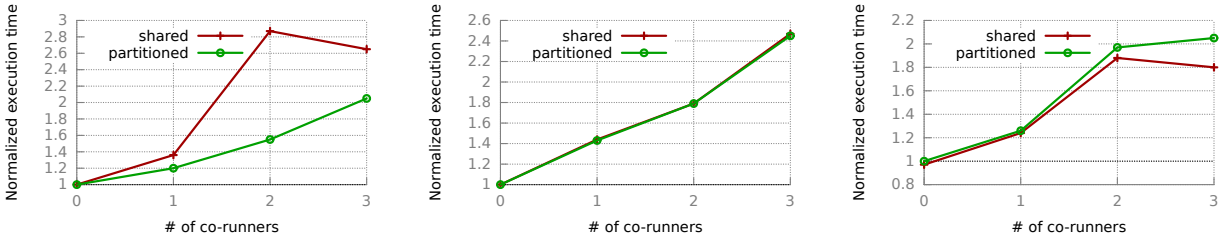
(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 6: [Exp.4] Slowdown of Latency(LLC) with BwWrite(DRAM) co-runners.



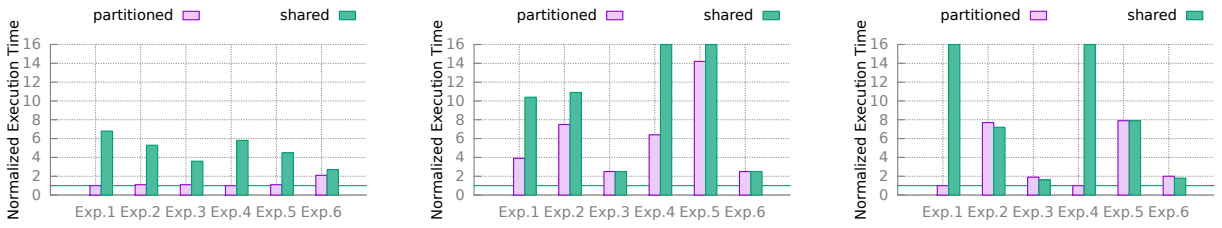
(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 7: [Exp.5] Slowdown of BwRead(LLC) with BwWrite(DRAM) co-runners.



(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 8: [Exp.6] Slowdown of BwRead(LLC) with BwWrite(LLC) co-runners.



(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 9: Maximum observed slowdowns in all experiments.

An experience report on the integration of ECU software using an HSF-enabled real-time kernel*

Martijn M.H.P. van den Heuvel, Erik J. Luit, Reinder J. Bril,
Johan J. Lukkien, Richard Verhoeven and Mike Holenderski

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven (TU/e),
Den Dolech 2, 5612 AZ Eindhoven, The Netherlands

***Abstract*—This paper gives an overview of the challenges we faced when integrating automotive software components on an embedded electronic control unit (ECU). The results include the design of a communication abstraction layer, management of scarce ECU resources and a demonstration of temporal isolation between components in an industrial case study.**

***Index Terms*—Automotive software; Virtualization; Real-time scheduling; Component-Based Software Engineering (CBSE).**

I. Introduction

Today's vehicles contain an ever increasing amount of software. These software functions consist of various components that replace mechanical controllers. The current market situation reinforces the challenges of integrating these software functions on a shared platform, because adding a new function into a vehicle often means purchasing pre-manufactured hardware and software with little information about the internal behavior [1].

The AUTOSAR consortium, however, recognized that a revolutionary performance increase of in-vehicle electronic systems comes from the composition and the integration of independently developed software functions. In AUTOSAR, functions are developed using components which are executed as tasks by an OSEK-certified operating system (OS). Some of these tasks may share memory-mapped input-and-output (I/O) devices, actuation devices (such as brakes) and software pieces [1] (such as object detection). The protocols that manage synchronization on these shared resources may further impact I/O delays experienced by the tasks of a component. Many components, especially those that implement control functionality, are sensitive to timing and fluctuations in actuation delays.

Hierarchical scheduling frameworks (HSFs) support promising techniques to control such timing delays and fluctuations. In order to support composition of components and temporal isolation between them, Nolte et al. [2] investigated the applicability of HSFs into AUTOSAR. The HSF is implemented using so-called *servers* as a layer between the AUTOSAR OS and the AUTOSAR Runtime Environment. The AUTOSAR standard allows for inclusion of proprietary technology, as long as the extensions can be abstracted to an AUTOSAR OS [2]. In this work we apply an HSF to real automotive software and we demonstrate its use in the field by means of video material.

*This work is supported by the Dutch High-Tech-Automotive-Systems innovation programme under the VERIFIED project (Grant number: HTASI10003).

The remainder of this paper is organized as follows. Section II gives a brief overview of the case study being explored in this paper. Section III then presents the software components that were developed for our use case. Section IV describes the deployment of those software modules on our ECU. Section V discusses some of the practical challenges we faced in the development and deployment of our ECU software. Finally, Section VI concludes this paper.

II. An automotive case study

In this work we integrated 3 software applications into a Jaguar XF (see Figure 1): an active suspension controller [3], a supervisory controller and a run-away process. We established timing predictable execution of these applications by means of an HSF, which allocates a server to each application.

The active suspension is part of a more comprehensive Integrated Vehicle Dynamics Controller (IVDC), which is meant to stabilize a vehicle in critical situations. The IVDC further improves the electronic stability program (ESP) of a car by adding suspension control to the integrated control [3].

A supervisory controller checks the correctness of the shared sensor and actuator data and handles faults when necessary. It is split up in a Central Supervisory Control (CSC) which coordinates central actions for the 4 wheels and a Local Supervisory Control (LSC) which controls a single suspension unit for one wheel. More precisely, the CSC implements logic to coordinate the suspension per axle and for the entire car.

The run-away process can be put in a mode where it consumes all processor cycles and it runs at the highest priority. It is used to demonstrate temporal isolation between the three applications, i.e., each application can consume only the resources allocated to its server and nothing more.

A. Logical view to hardware

We use various ECUs in the car which are connected to a fieldbus; some of these nodes are virtual ones. Each wheel is controlled locally. In our setup, one wheel is controlled by an ECU while the other wheels are controlled by a dSPACE [4] system (hence, the other ECUs are not deployed in real and their software runs on a central dSpace node). dSPACE provides a powerful hardware platform and tools for prototyping embedded applications. The CSC also executes on the dSPACE system.

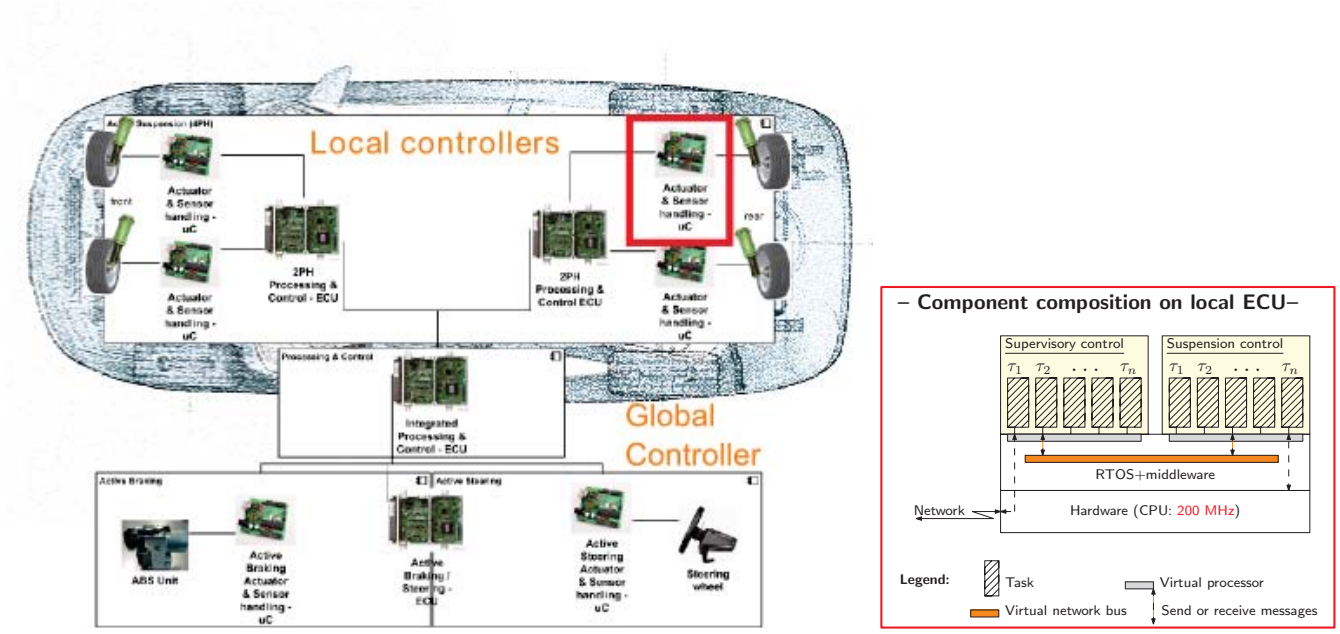


Figure 1. In this project we employed 4 ECUs at each corner of a test car for active suspension. A central dSpace coordinates the local ECUs. It therefore implements components for supervisory control and software-based integrated-vehicle-dynamics (IVDC) state estimation. We have integrated their local counterparts, i.e., 2 components which are (semi-)independently developed by various project partners, through an HSF with well-defined mechanisms for resource virtualization on a local ECU.

The ECU that we used is a Freescale EVB9S12XF512E evaluation board with a 16-bits, MC9S12XF512 processor and 32 kB on-chip RAM. The clock speed of the processor was set to 40MHz in order to accommodate the processing load. The board provides, among others, 16 Analog to Digital Converters (ADCs), several PWM outputs, a CAN controller and a FlexRay controller. The Freescale board is connected to an extension board which protects the processor hardware from electric overloads, it offers voltage division and it provides connectors to the processor board and to the environment.

B. This work

In this work, a dedicated ECU is deployed in order to control the suspension of one of the four wheels of a car. On this ECU, we implement and run three different applications:

- **Two control loops for active suspension:** these tasks run at 400 Hz and 100 Hz, respectively (i.e., tasks with periods of 2.5 ms and 10 ms). These loops execute a control model (developed using Matlab/Simulink) and they interact directly with the hardware.
- **The LSC process:** it receives commands from the CSC, sends commands to the control loops, receives data from the control loops and sends state information to the CSC.
- **Run Away Process (RAP):** on command it switches between a state in which it sends an *"I'm alive"* message each period and a state in which it tries to consume all CPU cycles.

Using our HSF extensions in MicroC/OS-II, temporal isolation is demonstrated between the three applications. Hence, the other applications are protected against the RAP. Moreover, we describe their mapping on a platform with scarce resources.

III. ECU Software

In this section we give an overview of the software modules that are integrated on our ECU. Figure 2 shows the

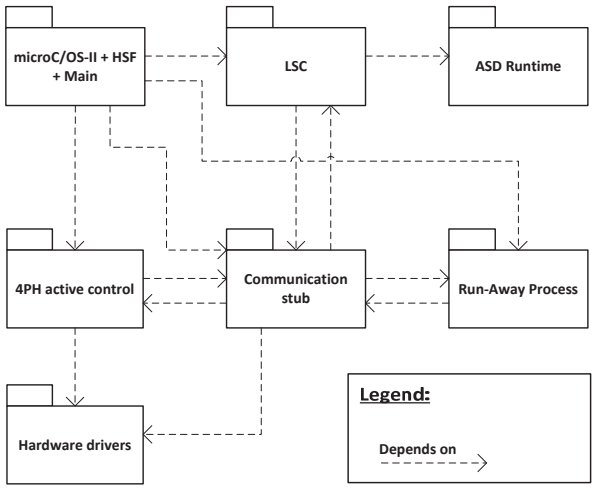


Figure 2. An overview of the software modules, including their dependencies, which we have integrated in our ECU.

dependencies between the different modules. Firstly, we briefly recapitulate MicroC/OS-II and its HSF. Secondly, we introduce the LSC and its run-time libraries. Thirdly, we introduce the 4-point-hydraulic (4PH) suspension control. Finally, we describe our communication stub.

The hardware drivers are not further described. These drivers were mostly delivered with MicroC/OS-II or by Freescale. Moreover, the most interesting part is described by our communication stub[1], which provides an abstraction layer for the underlying fieldbus drivers (either CAN or FlexRay).

Also the RAP is not discussed in further detail. The reason is that the RAP is a fairly simple process, i.e., an event-triggered infinite loop which is introduced for the purpose of demonstrating temporal isolation within an HSF.

[1]The dependencies of the communication stub to the application modules (LSC, RAP and 4PH active control) are just there to ease their definitions of message types; they can be avoided by means of singleton-like patterns.

A. MicroC/OS-II and its HSF

MicroC/OS-II is a microkernel which is maintained and supported by Micrium [5] and is applied in many application domains, e.g., automotive[2]. The kernel is open source and available for free for non-commercial purposes. The MicroC/OS-II kernel features preemptive multitasking for up to 256 tasks, and its size is configurable at compile time, e.g. services like mailboxes and semaphores can be disabled.

This section recapitulates our proprietary HSF module for MicroC/OS-II [6, 7]. Extending MicroC/OS-II with basic HSF support requires a realization of the following concepts:

1) *Server scheduling:* Similar to the MicroC/OS-II task scheduling approach, we introduce a ready queue for servers indicating whether or not a server has capacity left. When the scheduler is called, it activates the *ready* server with the highest priority. The fixed-priority scheduler of MicroC/OS-II then selects the highest-priority ready task from the group of tasks corresponding to the running server. The implementation of periodic servers turned out to be very similar to implementing periodic tasks [6].
2) *Task scheduling:* After masking the task groups of all servers except the tasks of the active one, the MicroC/OS-II fixed-priority scheduler subsequently determines the highest priority ready task; this code is unmodified.
3) *Idle Server:* We reserve the lowest task priority levels for an idle server, which contains MicroC/OS-II's idle task at the lowest local priority. This server cannot deplete its budget, so that the idle server can always be switched in whenever no other server is eligible to execute.

A major effort in the HSF's realization translates into a hierarchical representation of timed events. In a system we therefore employ four timer queues to control tasks and servers. In case of single level scheduling, we have just a single system queue that represents the timer events associated with the arrival of tasks. In an HSF, we use this existing system queue for the scheduling of servers. The timers in this queue represent budget-replenishment events corresponding to the start of a new period. In addition there is a local queue for each server which keeps track of the timers needed to manage the tasks inside a server such as the arrival of periodic tasks. At any time at most one server can be running on the processor; all other servers are inactive. When a server is suspended, its local queue is deactivated. In this configuration the hardware timer drives two timer queues, i.e., the local queue of the active (running) server and a system queue.

When the running server is preempted, its local queue is deactivated and the queue belonging to the newly scheduled server is activated. In order to ensure correct execution, the time that passed since the previous deactivation needs to be accounted for upon activation. To keep track of this time we introduce a third queue: the *stopwatch queue*. Upon deactivation of a server, a timer is added to this queue. Whenever a server is activated, its local queue is synchronized with the stopwatch, i.e., all timers in its local queue which would have expired if the server was running are handled. As a result, all local timers with a smaller value than the stopwatch timer are popped from the local queue and the corresponding stopwatch event is subsequently deleted from the stopwatch queue. The time spent to synchronize the local queue of the newly activated server with global time is accounted to this server and subtracted from its budget.

Finally, a fourth queue represents timers that expire relative to the server budget. These events trigger the depletion of (a fraction of) the server's budget. We call these *virtual timers* as their notion of time is limited to the server budget. Rather than putting these in the system queue we have a separate queue for them, since otherwise we would need to insert them into the system queue upon activation and remove them again upon deactivation. In this new configuration, at every tick interrupt at most four queues are updated: a system queue, an active server queue, a stopwatch queue, and an active server virtual queue. The last queue does not need to get synchronized when a server is resumed, because a deactivated server does not consume its budget.

We refer the interested reader to [6] for a detailed performance evaluation of MicroC/OS-II and our HSF.

B. Local supervisory control and its ASD runtime

The Local Supervisory Control (LSC) consists of code generated from formally verified state charts. These state charts are programmed using the *ASD:Suite* [12]. Although ASD's underlying model-checking techniques can guarantee absence of faults in the state-chart models, absence of faults is not automatically guaranteed in the modeled program unless code generation techniques are applied.

For this purpose, amongst other approaches, Broadfoot and Broadfoot [8] proposed to bridge the gap between formal methods and the informal world of software engineering by combining the sequence-based specification method (SBS) [9] and the process algebra Communicating Sequential Processes (CSP) [10]. Broadfoot and Hopcroft [8, 11] extended this work by developing automated translations between SBSs, CSP and executable code, such that the operational semantics are preserved. This led to the invention of Analytical Software Design (ASD) and together with the commercial product ASD:Suite [12], developed and owned by Verum, enables its full integration into industrial practices.

Using ASD, we describe the provided interface of the LSC component, which consists of the following methods:

- *comm_ok*;
- *controls_enabled*;
- *reset_system*;
- *reset_errors*.

These methods can be called by other components in the system, i.e, in our case, the communication stub.

The behaviour behind the interface of the LSC component is then captured by a state chart, as shown in Figure 3. It has the following states: uninitialized, passive and active. The state changes of the LSC are triggered by the received commands

[2]Unfortunately, the suppliers of MicroC/OS-II have discontinued the support for an OSEK-compatibility layer.

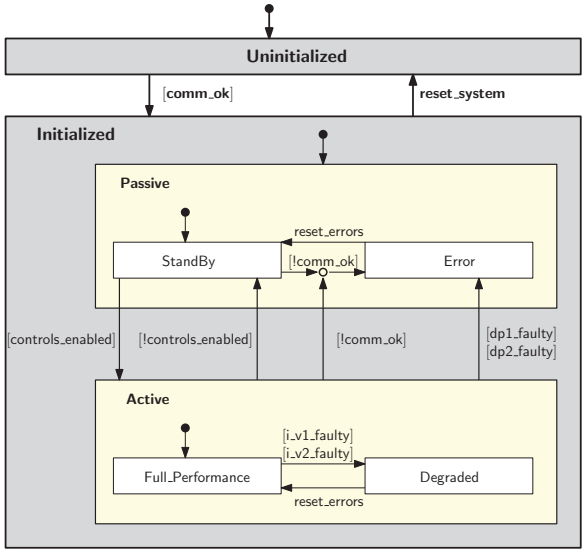


Figure 3. A state-chart representation of the LSC.

from incoming network messages. From the uninitialized state, a transition is made to the initialized/standby state when the LSC receives the *comm_ok* message from the communication stub. This message is sent as soon as the first message is received from the CSC. When the *controls_enabled* signal is received, the active/full performance state is entered.

When faults are detected, the LSC goes into either the degraded state or into the passive state. The degraded state is entered if the measured sensor data (i.e, the pressure and current) deviate from their expected values. These errors are reported to the CSC and the LSC can return to the full performance state when the *reset_errors* message is received.

In the passive states the local 4PH suspension control acts independently of the central control. This happens, e.g., when the communication between the dSpace box and our ECU fails. The communication is considered to be correct (see Section V-B) as long as maximally two messages from the CSC are missed, either because these did not arrive at the ECU or because the ECU could not process these in time. If the communication fails, i.e., when more than two messages are not received, then the passive state is entered. When messages are arriving again, the communication stub sends the *comm_ok* message again and the full performance state is re-entered. Otherwise, the LSC stays in the passive state.

Finally, the above design is formally verified by ASD. By modeling its environment, e.g., the interface of the communication stub which may use the LSC's provided interface, concurrency issues of tasks interacting with the LSC can be avoided. Subsequently, MISRA C compliant source code [13] has been generated which implements the model.

C. 4PH active suspension

The local 4PH suspension control of our ECU controls the suspension unit at one wheel of the car. The suspension unit for one wheel consists of a conventional suspension extended with a hydraulic system. The hydraulic system consists of a fluid-filled cylinder with a piston that divides the cylinder into two parts. The pressure on both sides of the piston can be varied by two electrically operated valves, so that the piston and the rod attached to it can move in both directions. The valves are actuated via Pulse Width Modulation (PWM), so the effective voltage applied is determined by the ratio of the duty cycle and the period of the PWM. Hydraulic pressure is measured on both sides of the valves. Also the actual current of the valves and the voltage of the power supply are measured.

The code generated from this active-suspension application for our ECU consists of 2 control loops: one controls the pressure of the valves at 100Hz and the other controls the current at 400Hz. The central dSpace box runs the software for the other 3 wheels of the car and it runs the CSC which implements logic to coordinate the suspension per axle and for the entire car. The entire control application has been modelled and tested using Matlab-Simulink. For details on the vehicle dynamics, we refer the interested reader to [3].

D. Communication stub

The communication stub optimizes concurrent use of the network bus and abstracts its underlying technology. In this section, we describe how we connected our ECU to a CAN bus; Section V-C shows how the CAN connection can be replaced by a FlexRay connection.

The communication optimization focuses on minimizing the number of messages to be transmitted from dSpace to the ECU and vice versa. Messages that are to be sent at the same time are therefore piggybackked into one packet. The abstraction takes care of a uniform message format and it hides variations in latency and jitter involved with communication. This is needed, because the data structures, that define the messages being communicated over the CAN network, are compiled differently by the dSpace and the Freescale compilers. The communication stub therefore encodes and decodes CAN messages.

Moreover, without any additional means, the clocks at the dSpace box and our ECU will not be synchronized, which may lead to jitter. Although the central controllers at the dSpace and the local controllers at the ECU may roughly run at the same speed, they will not be as tightly synchronized as they would be in case both run on the same dSpace box. This may have two consequences for a local controller:

1) When it runs ahead, it may expect an absent message;
2) When it runs behind, it may receive multiple messages.

Both problems are resolved by assuming that the local controller has a state-message semantics. That is, the last value that has been sent is returned and there is no synchronization between sender and receiver.

This way of communication may lead to conflicts with the LSC, because the LSC expects messages upon each event that requires a change of its internal state. We have therefore implemented a translation layer in the communication stub in order to support event messages (see Section V-A).

IV. Application mapping

In this section, we firstly describe the mapping of applications to tasks and servers. Secondly, we describe the mapping of applications to messages on the fieldbus. Finally, we discuss the mapping of applications to memory.

A. Servers and tasks

As suggested by Figure 2, the application settings of MicroC/OS-II and the integrated ECU software are together defined in a main file. This file includes declarations of tasks and servers, their priorities and the stack size of the start tasks, i.e., the task that creates the other tasks and that starts the real-time clock. The real-time clock operates at 4000Hz, which restricts the monitoring of the resource consumption to 10% of the execution of the most frequent control loop.

In total we define 3 servers, i.e., given in descending priority order: for the RAP, the active suspension and the LSC process. The RAP and the LSC are (arbitrarily) allocated 10% processor bandwidth each period of 10 ms. Based on our experiments, the processor budget of the server corresponding to the active suspension control is set to 80% of the processor bandwidth with a period of 2.5 ms.

The local 4PH suspension control consists of two control loops (for current control and for pressure control) which are running on the same server. For each of the control loops, a task is created and their priorities are assigned in a rate-monotonic manner. In order to reduce the number of context switches between these tasks, their execution is forced in a strictly alternating manner (using a release offset and semaphore protection), so that 1 execution of the 100Hz control loop is followed by 4 executions of the 400Hz control loop. Moreover, the offsets of the tasks are chosen such that the high-frequent task cannot be preempted due to the server's budget depletion.

B. Fieldbus communication

In our setup, the applications on the ECU report their status to the CSC. The fieldbus (by default CAN) is therefore used by three different applications, i.e., from the ECU's sides:

- **Active-suspension control:** every 2.5 ms, it reports the current and voltage set points of the valves.
- **LSC:** every 10 ms, it reports state and error information;
- **RAP:** sends an *"I'm alive"* message every 10 ms.

The messages from the LSC and the RAP are piggybacked on the control messages, because these have the highest frequency.

In return, the CSC on the dSpace box replies to our ECU every 10 ms. The messages received by our ECU contain:

1) set points and estimated valve flows for the control loops;
2) state-change commands for the LSC;
3) state-change commands for the RAP.

C. Memory management

A major challenge encountered was that the control application (generated from Simulink) did not fit into the non-paged memory of the processor, i.e., the application requires more than the 8KB directly accessible RAM. Additional RAM can be used by means of the so-called *banked memory model* which enables memory paging. By loading a page into the page window and making sure no other page is loaded into this window, 4KB additional RAM can be directly addressed.

The support for memory paging required us to change the functions involved in context switching, including the interrupt service routines (ISRs), because the stack needs to store the PPAGE register and a 24-bit function pointer (only a 16-bit pointer is stored in the non-paged case). Paging has only been implemented for code, not for data as this would have required additional effort which was unnecessary to solve our memory problems. For performance reasons, compiler directives (pragmas) were applied to ISRs in order to link them into non-banked memory.

V. Discussion: reliable communication

A. Joint event-triggered and time-triggered message handling

In our design, two types of message semantics have been integrated [14]: event-message and state-message semantics. For event-message semantics, a message is associated with an event that is processed upon receiving the message. Also, synchronization is needed between sender and receiver. For state-message semantics, the last value that has been sent is returned which represents the last known state of the sender. Since we cannot assume intermediate synchronization between the dSpace box and our ECU, we implemented a translation from event-message semantics to state-message semantics.

The 4PH suspension control loops are implemented using Matlab/Simulink. Matlab/Simulink implements time-triggered activations of control tasks and it polls for input data, corresponding to state-message semantics. The local 4PH suspension control will therefore automatically read the data of the latest received message, i.e., following the state-message semantics.

However, the LSC assumes event-message semantics, because the supervisory control is assumed to be activated upon a (relevant) state change in its environment. This requires a conversion from state-message semantics to event-message semantics in the communication stub. For this purpose, our communication stub provides dedicated send and receive primitives, which we briefly describe below.

1) Sending messages: When a *send* primitive is called from the communication stub interface, this will simply cause an update of local data within the communication stub corresponding to the send request. However, no message is being submitted at this time. Only periodically, messages are being packed and submitted to the CAN bus.

When multiple state changes happen for the local read data of supervisory control, it gives rise to multiple events to reflect those state changes. This may lead to overload situations, as discussed in the next subsection.

2) Receiving messages: When a *read* primitive is called from the communication stub, all messages (if any) will be retrieved from the message queue of the CAN driver, in the order of arrival. Only the latest received message will be taken into account for handling, because a state-message semantics is assumed. This is possible because only the local state of the LSC needs to be updated. This message may cause a state-change in the LSC, where only the last state matters.

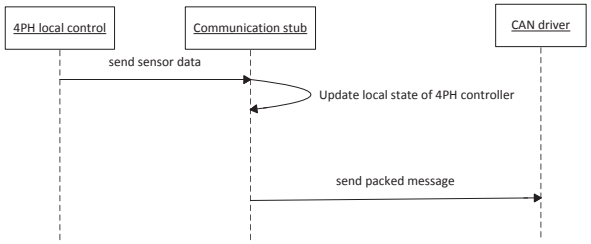


Figure 4. Interaction diagram for sending a CAN message.

Note that only a state change with respect to locally stored data (for example, by the LSC) is translated into an "*event*". In this way we effectively transformed state-message semantics into event-message semantics.

B. Handling communication errors and overloads

Within our system, we cannot assume that message communication is reliable. When a task attempts to send a message to an uninitialized node in the network, a CAN error interrupt is generated. If this interrupt is not handled properly, this causes a crash of the control software. An ISR is therefore developed to handle this interrupt, i.e., it resets the CAN bus with a call to *CANStart*. The execution time of this ISR is considerable and it exceeds the execution time of the control loops. In practice this is not problematic, because uninitialized nodes typically occur only once when the applications are bootstrapped.

Furthermore, once the communication has been initialized, our ECU may be unable to keep up with the produced messages of the central control running on the dSpace box. In consumer-producer situations, a consumer (e.g., a LSC) may not be able to keep up with the producer (e.g., the CSC delivering commands over the network). A common technique to prevent buffer overloads is to selectively delete incoming messages. In this way, unacceptable latencies between the reception of the remaining commands and their handling can be avoided.

Deleting events in a state-message semantics is only possible, if and only if the new state of the receiver depends on just the latest event (rather than all intermediate states). Given that state changes of the LSC do not appear often, we experienced that in our proof of concept pruning of messages can be ignored.

C. Replacement of CAN by FlexRay

FlexRay has been introduced in order to increase the available network bandwidth compared to CAN. The FlexRay technology defines a communication cycle which is divided into static and dynamic segments. The static segment enables time-triggered communication; the dynamic segment allows each node to transmit its messages in the remaining bandwidth using event-driven communications (like with CAN). In this work, we merely used the static segment. Freescale provides a library for this, which contains a set of functions and protocol-specific interrupt handlers to interact with the FlexRay controller.

The payload size of FlexRay slots is configured to be 16 bytes. This allows the resolution of the messages to be increased compared to CAN. Another advantage of this payload size is that encoding of the messages into the slots can be done efficiently. Consequently, all messages can be encoded and decoded by using the union of their data (i.e., fast piggybacking). The data structures of the FlexRay messages are defined as a union of a set of fields and a byte array. Such a union provides the possibility to approach the memory location at which the structure is stored as one of the fields or as a byte in the array. This makes the earlier described functions to encode and decode messages obsolete.

However, since FlexRay messages are larger than CAN messages, FlexRay communication requires more data memory compared to CAN. This reinforces the challenges related to efficient memory management of the applications running on our ECU (as discussed in Section IV-C).

VI. Conclusions and Future work

In this paper, we revisited HSFs for facilitating timing predictable integration of automotive software components. Previously, we have published both the theoretical [15] and the practical impact [6, 7] of resource virtualization on the timeliness of synthetic components. In this paper we presented our experiences with employing our HSF in an ECU with real automotive software. A Jaguar XF carries our HSF with active-suspension software, which we captured on video. Future cars are expected to rely even more on timing predictable composition, not just for further vehicle dynamics but also for car-to-car control (like collision avoidance). Here real-time systems and the internet-of-things may join their forces.

References

[1] M. Di Natale and A. Sangiovanni-Vincentelli, "Moving from federated to integrated architectures in automotive: The role of standards, methods and tools," *Proc. of the IEEE*, vol. 98, no. 4, pp. 603–620, April 2010.

[2] T. Nolte, I. Shin, M. Behnam, and M. Sjodin, "A synchronization protocol for temporal isolation of software components in vehicular systems," *IEEE Trans. on Ind. Inf. (TII)*, vol. 5, no. 4, pp. 375–387, Nov. 2009.

[3] B. Bonsen, R. Mansvelders, and E. Vermeer, "Integrated vehicle dynamics control using state dependent riccati equations," in *AVEC*, Aug. 2010.

[4] dSPACE GmbH, "Automotive Solutions – Systems and Applications," 2015. [Online]. Available: https://www.dspace.com/

[5] Micrium, "RTOS and tools," 2011. [Online]. Available: http://micrium.com/

[6] M. Holenderski, R. J. Bril, and J. J. Lukkien, "An efficient hierarchical scheduling framework for the automotive domain," in *Real-Time Systems, Architecture, Scheduling, and Application*. InTech, 2012, pp. 67–94.

[7] M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Transparent synchronization protocols for compositional real-time systems," *IEEE Trans. on Industrial Informatics*, vol. 8, no. 2, pp. 322–336, May 2012.

[8] G. H. Broadfoot and P. J. Broadfoot, "Academia and industry meet: Some experiences of formal methods in practice," in *APSEC*, 2003, pp. 49–59.

[9] S. J. Prowell and J. H. Poore, "Foundations of sequence-based software specification," *IEEE Trans. on Software Engineering (TSE)*, vol. 29, no. 5, pp. 417–429, 2003.

[10] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, Int. Ser. in Computing Science, 1985.

[11] P. J. Hopcroft and G. H. Broadfoot, "Combining the box structure development method and CSP for software development," *ENTCS*, vol. 128, no. 6, pp. 127–144, May 2005.

[12] "Verum® - Tools for building mathematically verified software," 2009. [Online]. Available: www.verum.com

[13] "MISRA - The Motor Industry Software Reliability Association," 2004-2009. [Online]. Available: http://www.misra-c2.com/

[14] S. Poledna, "Optimizing interprocess communication for embedded real-time systems," in *RTSS*, Dec. 1996, pp. 311–320.

[15] M. M. H. P. van den Heuvel, "Composition and synchronization of real-time components upon one processor," Ph.D. dissertation, TU/e, The Netherlands, June 2013, ISBN 978-94-6108-443-9.

Evolving Scheduling Strategies for Multi-Processor Real-Time Systems

Frank Feinbube, Max Plauth, Christian Kieschnick and Andreas Polze
Operating Systems and Middleware
Hasso Plattner Institute
University of Potsdam, Germany
Email: {frank.feinbube, max.plauth, christian.kieschnick, andreas.polze}@hpi.de

***Abstract*—In recent years the multi-core era started to affect embedded systems, changing some of the rules: While on a single processor, Earliest Deadline First has been proven to be the best algorithm to guarantee the correct execution of priorized tasks, Dhall et al. have shown that this approach is not feasible for multi-processor systems anymore. A variety of new scheduling algorithms has been introduced, competing to be the answer to the challenges multi-processor real-time scheduling is imposing. In this paper, we study the solution space of prioritization-based task scheduling algorithms using genetic programming and state-of-the-art accelerator technologies. We demonstrate that this approach is indeed feasible to generate a wide variety of capable scheduling algorithms with pre-selected characteristics, the best of which outperform many existing approaches. For a static predefined set of tasks, overfitting even allows us to produce optimal algorithms.**

I. Introduction

Following the trends in the personal computing sector, many embedded systems are nowadays equipped with multiple processing units. These resources are used for non-critical tasks like entertainment systems and critical ones, where wrong timing is considered a failure. In real-time systems, the latter are traditionally studied using the preemptive task model. A task T arrives at time A in the system and is supposed to finish its execution by its deadline AD. Furthermore, with these critical tasks, it is usually assumed, that the worst case execution time C is known upfront. Tasks can either be occurring only once or periodically, where AD is also considered to be the time interval, after which the task arrives again. If a given set of tasks includes only periodic tasks, it is called a *periodic* task set; otherwise it is called *sporadic*.

A task scheduling algorithm is used to schedule these tasks onto p processors so that no task misses its deadline. This is usually realized by assigning priorities to the tasks. If the task set is known upfront, *static* scheduling algorithms can be used, assigning fixed priorities to the tasks. This is very efficient since the scheduling algorithm only needs to be executed once. If the task set is not known upfront and new tasks arrive during system runtime, *dynamic* scheduling algorithms need to be utilized. They reevaluate the priorities of all known tasks and are usually executed when new tasks arrive or at predefined time intervals during runtime.

Born at a time when resources for embedded systems were very restricted, traditional scheduling algorithms are rather simplistic, usually assigning priorities based on a single attribute: the deadline. As discussed in Section II, more sophisticated algorithms are required in multi-core scenarios. Ideally, an algorithm should be optimal, which means that it is capable of finding a feasible schedule whenever there exists one. While it has been proven that an optimal algorithm for multi-core scenarios cannot exist, a number of algorithms have been proposed that can schedule certain classes of task sets. In Section III we describe our approach to the problem. By applying genetic programming and state-of-the-art accelerator technologies, we were able to evaluate a vast variety of prioritization-based scheduling algorithms. As shown in Section IV our implementation can be used to find close-to-optimal algorithms tailored to task sets with specific characteristics.

II. Related Work

For *single processor* scenarios, optimal algorithms have been around for a long time [1]: Rate Monotonic Scheduling (RMS) [2] is an optimal static scheduling algorithm for periodic task sets. RMS prioritizes inverse proportionally to period lengths. Earliest Deadline First (EDF) [2] is an optimal dynamic scheduling algorithm for sporadic task sets. Each time a new task arrives, EDF prioritizes based on the deadlines of all tasks. Least Laxity First (LLF) [3] is also an optimal dynamic scheduling algorithm. The priority of each task is based on the difference of its remaining execution time and the time until its deadline is violated. Since this difference constantly changes during runtime, LLF shows strong *oscillation effects* as shown in Figure 1 leading to a huge amount of task switches. In practice, task switching in embedded systems comes with a performance overhead. Thus, there are variations of LLF such as Modified Least Laxity First (MLLF) [4] that try to reduce the oscillation effect.

	A	AD	C
T_1	0	10	5
T_2	0	10	5

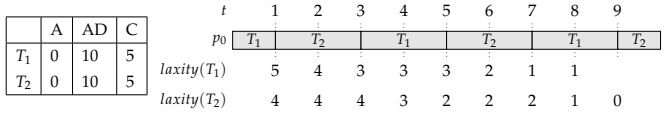


Fig. 1. Scheduling algorithms like Least Laxity First [3] show oscillating behavior where the priority is altered at each quantum.

In *multi-processor* scenarios, things get a little bit more complicated: Besides oscillation effects, task schedulers also have to cope with *Dhall's effect* and *pure global task sets*.

Dhall's effect is demonstrated in Figure 2. It describes the scenario where there are task sets which produce a very

low overall system utilization, but still miss a deadline when scheduled with traditional algorithms. A number of "hot fixes" to EDF and RMS were introduced that have been proven to circumvent the problem: e.g. EDF First Fit/Best Fit [5], Earliest Deadline Until Zero Laxity (EDZL) [6], and UMax algorithms [7], [8]. Although Dhall's effect is prevented, these scheduling algorithms only allow for low system utilizations: e.g. 35.425% for sporadic and 37.482% for periodic task sets [7], [8]. Since this is significantly lower than the 50% utilization, that is considered the actual limit [9], new approaches were evaluated. Lundberg has proven that by assigning task priorities based on the slack ($AD - C$) instead of the deadline, the acceptable utilization for sporadic task sets can be increased to 38.197% [10].

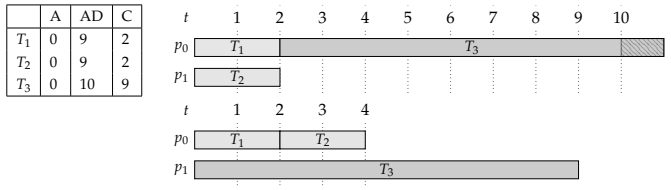


Fig. 2. This two-processor scenario with three tasks demonstrates Dhall's effect [2]. Although it is possible to schedule all tasks according to their deadline (bottom schedule), Earliest Deadline First (EDF) fails to do so (schedule on top).

A popular approach to multi-processor real-time scheduling is to statically allocate tasks to processors so that a task will never be migrated to another one. The alternative to this *partitioned* approach, is the *global* approach where each processor can execute each task and tasks will be migrated accordingly. Migrating tasks results in additional overhead, but it is the only way to handle *pure global task sets* as depicted in Figure 3.

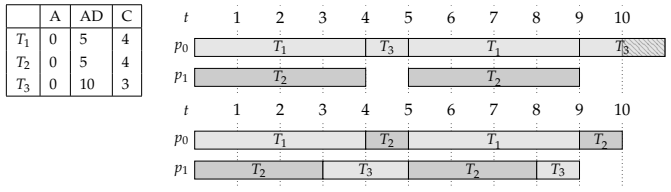


Fig. 3. This two-processor scenario with three tasks demonstrates Levin's pure global task sets [11]. Although it is possible to schedule all tasks according to their deadline (bottom schedule), it is impossible to do so by pinning tasks to a single processor (schedule on top).

There are algorithms that avoid *Dhall's effect* and are capable of scheduling pure global task sets while allowing utilizations of up to almost 100%. Proportionate Fair Scheduling [11], [12] and Dp-fair use a fluid scheduling model with fair task progress, which requires a reprioritization of all tasks at predefined time intervals. Largest Local Remaining Execution First (LLREF) [13] follows a similar model, but reprioritizes based on the laxity and execution time of the active tasks, instead of static time intervals. When it comes to theoretical maximal system utilization, these scheduling algorithms perform exceptionally well. However, depending on the frequency of the reprioritizations, they show oscillation effects and introduce significant scheduling overhead. This overhead is comprised of the execution time of the more complex scheduling algorithm itself, the overhead for switching the active tasks and the overhead for task migration between processors. Another restriction is that the aforementioned reprioritizing algorithms are only suitable for periodic task sets. Hong et al. [14] formulated the hypothesis that there is no optimal priority-driven algorithm for sporadic task sets. This hypothesis has been proven by Fisher [15].

A. Research Gap

In this work, we contribute to the field of real-time multi-processor scheduling by presenting an approach to:

- Identify novel algorithms by exploring the solution space for real-time scheduling algorithms.
- Create algorithms complying with desired characteristics such as the number of task migrations and maximal system utilization.

As a means to implement these goals, we use genetic programming to evolve real-time scheduling algorithms with pre-selected characteristics. Using genetic programming for the creation of our algorithms allows us to cover a wide variety of scheduling alternatives, thereby helping us to identify the attributes and functions that are most successful to reduce overheads while allowing for a solid system utilization. While being able to create optimal algorithms for many of the task sets we used in our evaluation, we were unable to identify an algorithm that is optimal for the general case. However, these findings harmonize with the proof of Fisher [15], which states that no optimal algorithm can exist for the general case. Running such a compute-intense simulation to identify suitable algorithms was only possible due to the performance of modern processor and state-of-the-art accelerator technologies.

We are not the first to apply genetic algorithms to the research area of scheduling algorithms for multi-processor systems. Hou et al. [16] and Greenwood et al. [17] used genetic algorithms and evolutionary strategies to generate heuristics for predefined task graphs. While demonstrating the feasibility of the approach, both studies focussed exclusively on task sets that are known upfront and created heuristics that, while useful in for multi-processor systems in general, did not consider real-time requirements.

Furthermore, there are existing studies that simulate scheduling algorithms to evaluate their qualitative and quantitative characteristics [18]–[20]. These approaches are sophisticated to gain insight into capabilities of a single selected scheduling algorithm, while our approach allows sift through a vast amount of scheduling algorithms to identify the interesting candidates for further examination.

To the best knowledge of the authors, we are the first to apply genetic programming for an exploration of the real-time scheduling algorithm solution space for arbitrary task sets.

III. Approach

Mathematical modeling of the task scheduling domain and proving the qualities of particular scheduling algorithms becomes increasingly complicated the more complex the scheduling algorithms are. Thus, the next best thing would be a simulation of all possible scheduling algorithms starting with a very

limited set of terms and functions and iteratively considering more, when the current complexity is exhaustively studied. Such an approach has to handle humungous state explosions with every additional variable and function. Evolutionary processes and genetic algorithms have proven to be ideal for these kinds of scenarios, since they confine unpromising states while iteratively exploring the more promising ones. [21]–[23] This section discusses the application of genetic algorithms to identify promising scheduling algorithms.

A. Architecture

The general architecture of our approach is depicted in Figure 4. We start by loading the three kinds of task sets, that we use as the workload for our simulation. The task sets are described in detail in Section III-B. Furthermore, we generate a number of initial prioritization schemes. Prioritization schemes form the core of our scheduling algorithms. They encapsulate everything that is needed to assign priorities to task sets. The generic task scheduler shown in Figure 5 will use these schemes to prioritize the tasks and then simply schedule them based on their priorities.

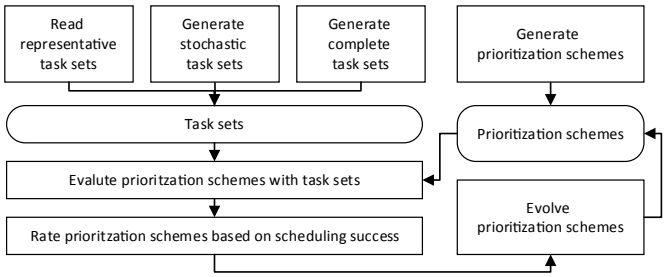


Fig. 4. Architecture: the evolutionary process iteratively refines the scheduling strategies using a variety of task sets.

```
1  for(runtime = 0;
2    runtime < simulationEnd && !missedDeadline(tasks);
3    ++runtime)
4  {
5    activeTasks = filterActive(tasks);
6
7    // this is exchanged with each prioritization scheme
8    prioritizationScheme->prioritizeTasks(activeTasks);
9
10   orderDescendantByPriority(activeTasks);
11   tasksToSchedule = selectFirst(activeTasks, processors);
12
13   simulateDiscreteStep(tasksToSchedule);
14 }
```

Fig. 5. The generic scheduler is the core of our implementation. In our implementation schedulers only differ in the way they assign priorities to tasks at any given point in time throughout the execution. This scheduling strategy is determined by the prioritization scheme.

The evolutionary process is conducted iteratively with the following consecutive steps: evaluation, rating, evolving. In the evaluation step, each prioritization scheme is used to schedule each of the task sets. It is monitored how many tasks switches and task migrations were required and how many of the task sets failed to be scheduled successfully, e.g. a deadline was missed. In the rating step, this information is used to assign a fitness value to each of the prioritization schemes. Based on the fitness value the well-known mechanisms of selection, mutation and crossover are applied to create the next generation of prioritization schemes. This process is repeated until candidates with fitness values that are sufficient to comply with our requirements have been found, e.g. prioritization schemes capable of scheduling all task sets successfully, or a predefined maximal runtime is exceeded.

By adapting the fitness rating accordingly, this architecture allows us to easily ensure that the scheduling algorithms comply with our requirements when balancing task migrations and maximal supported utilization.

B. Task Sets

The quality of the resulting prioritization schemes depends primarily on the task sets that are used for the fitness rating of the evolutionary process. We distinguish between three categories of task sets: *representative* task sets, *stochastic* task sets and *complete* task sets.

Representative task sets are a selection of tasks sets from the literature that is used to evaluate the capability of a prioritization scheme to handle the 'hard' cases. For single processor scenarios, we have task sets that can barely be scheduled by Rate Monotonic Scheduling (RMS), cases that RMS fails to schedule, but Earliest Deadline First (EDF) can schedule. In the multi-processor scenarios, we we extend these conventional task sets so that the workload increases according to the number of processors. Furthermore, we add task sets that show different effects discussed in Section II. Our set of representative task sets includes both periodic and sporadic task sets. Most of these task sets could be scheduled with a partitioning strategy, e.g. without task migration. Consequently, we added pure global task sets as described by Levin et al. [11] to complete our mix of representative task sets. An overview of aforementioned task sets and the ability of selected scheduling algorithms to find a feasible schedule is presented in Table I.

While representative task sets are well suited to remove prioritization schemes that fail to handle the problematic cases, *stochastic* task sets allow us to assess the overall scheduling performance by mitigating undesireable overfitting effects. To accomplish this, we generate a number of task sets with a pseudo-random generator based on a stochastic distribution.

Complete task sets are created by generating every possible combination of task distributions for a given number of processors and number of scheduling time slices (quanta). Since both representative task sets and stochastic task sets are included in complete task sets, they deliver the best quality for the evaluation. The drawback is, though, that the amount of task sets that have to be generated grows exponentially and renders computation unfeasible for all but very small amounts of processors and quanta. In our experiments, we studied complete tasks sets for up to 8 processors and quanta of up to 6 intervals, resulting in about 10^8 task sets.

C. Evolution

We represent each prioritization scheme as an abstract syntax tree (AST) that can be executed for a task to produce a priority. Figure 6 shows an example. The evolutionary process

TABLE I. CHARACTERISTICS OF OUR SET OF REPRESENTATIVE TASK SETS. THE RIGHT PART OF THE TABLE SHOWS WHICH PROCESSOR CONFIGURATIONS CANNOT BE SCHEDULED BY EXISTING SCHEDULING ALGORITHMS. 1, 2, 4, 8, 16 ARE THE NUMBERS OF PROCESSORS USED. CONFIGURATIONS MARKED WITH A * CAN ONLY PARTLY BE SCHEDULED. PLEASE NOTE THAT LEVIN'S PURE GLOBAL TASK SETS [11] CAN NEITHER BE SCHEDULED BY APPROACHES THAT APPLY A SIMPLE PARTITIONING, NOR BY APPROACHES THAT ARE SENSITIVE TO UTILIZATION.

	periodic	partitionable	Laxity-based	global EDF	EDF-US	EDZL
RMS3	✓	✓		2*		
RMS4	✓	✓		2* 4 8 16		4* 8* 16*
WikiEDF	✓	✓				
Partitioned	✓	✓	2* 4* 8* 16*	4* 8* 16*	2* 4* 8* 16*	4* 8* 16*
Dhall		✓		2 4 8 16	1*	
SlackDhall		✓	4* 8* 16*		1* 2* 4* 8* 16*	4* 8* 16*
Detail		✓		2		
Split		✓				
Interwoven		✓	2 4 8 16	2 4 8 16	1 2 4 8 16	2 4 8 16
Levin [11]	✓		2 4 8 16	2 4 8 16	2 4 8 16	2 4 8 16

of selection, mutation and crossover was realized according to the literature. [21]–[23] The initial population is generated purely randomly, with a restricted AST depth of up to 5.

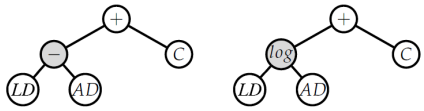


Fig. 6. Prioritization schemes are represented as abstract syntax trees. Mutations and crossovers are realized by varying and exchanging nodes. In this example $(LD - AD) + C$ mutates to become $log(LD, AD) + C$.

TABLE II. ATOMIC AND SIMPLE DERIVED TERMINALS, BASED ON THE CURRENT TASK AND THE SYSTEM.

x	random floating point values from -10.0 to 10.0
0, 1	constant values 0 and 1
m	number of processors
A	arrival time
RD	relative deadline (relative to arrival time)
C	capacity = worst case execution time
PT	amount of C that has already been executed
P	current task priority (starting with 0)
T	current point in time
AD	absolute deadline = A + RD
ST	slack = RD - C
L	remaining surplus time = (AD - T) - (C - PT)
U	utilization created by task = C / RD
LD	remaining execution time = C - PT
RU	remaining utilization = LD / (AD - T)

The nodes in the AST are the terminals listed in Table II. We distinguish between three types of basic terminals: numbers, system terminals, task specific terminals. System terminals comprise of the processor count and the time. Task specific terminals are deadline, worst case execution time and so forth. In addition to these, we provide a selection of derived terminals. These are not essential, since they would be generated by the evolutionary process anyway, but since they are the core of many of the popular scheduling algorithms like EDF [2] and LLF [3] we provided them, as well. Furthermore, the introduction of derived terminals improved the performance of the evolutionary process significantly. Please note that the resulting prioritization schemes do not consider the other tasks in the system, thereby guaranteeing a linear execution time of the represented scheduling algorithm.

The set of functions supported by our AST are: *addition, subtraction, multiplication, protected division, protected logarithm, exponentiation, check for equality, check for inequality, selecting the minimum, and selecting the maximum*. Checking for equality and inequality will produce either 1 for success or 0, allowing a combination with the other functions: AD * (L == 0).

The fitness of a prioritization scheme is rated according to multiple objectives [23]. A prioritization scheme is considered better than a similar one, if it can either schedule more task sets successfully or needs significantly less migrations for the scheduling. The impact of the objectives on the fitness functions can be configured by weights. For the selection process, we experimented with different population sizes. We observed that a tournament based selection process with 8 participants and a population size of 100 produced the best results.

In our experiments, we experienced overfitting effects [22], where the identified candidates were capable of scheduling all the task sets we trained them with. This is useful, if you want to use the approach, to find the perfect schedule for a specific task set. In the study of the solution space for scheduling algorithms, it is a hindrance, though, because overfitted prioritization schemes perform worse in the general case. To control overfitting, we created two distinct sets of task sets – the first to evolve the schemes and the second for the final evaluation. Furthermore, we applied randomizations and weighted function length negatively, since long functions tend to overfit more, than shorter ones.

D. Implementation and Performance Tuning

For the practical evaluation, we implemented the conceptual architecture presented in Figure 4. Fortunately, the repetitive steps of generation, evaluation and selection are suited for a parallel implementation. Our initial measurements indicated that the evaluation step is the predominant workload causing 99.99% of the overall execution time. As a consequence, all optimization efforts were directed at improving the efficiency of the evaluation step.

The time required for the evaluation process was greatly reduced using several optimization techniques: Using a stack-based representation of terms resulted in a decreased number of memory allocation operations compared to a tree-based data structure. At the same time, the stack-based structure

managed to increase the degree of data locality. Targeting the goal of data locality as well, an additional blocking method was applied to increased the amount of cache hits. Finally, we evaluated several strategies to vectorize our implementation. However, in contrast to the other optimizations, none of the vectorization strategies resulted in any significant performance improvements.

In addition to an x86_64 CPU-based implementation, we also created prototypes targeting Intel's Many Integrated Core (MIC) architecture exclusively as well as a hybrid version. The hybrid implementation applies an asymmetric load distribution scheme between the CPU and the MIC in order to maximize the execution speed.

The Xeon Phi accelerators based on the MIC architecture consist of 57-61 cores that are based on a modified P54C design. Unlike GPU compute devices, all cores of a MIC accelerator can act independently of each other. This property makes the MIC architecture a promising target for the parallel evaluation of diverse prioritization functions. Since the MIC architecture supports x86_64 instructions, the optimization we conducted improved the performance for both architectures.

IV. Evaluation

A. Qualitative evaluation

As described in Section III-C, we designed our implementation to assign fitness ratings based on weighted objectives. Figure 7 shows the impact of weighting migrations with 10%.

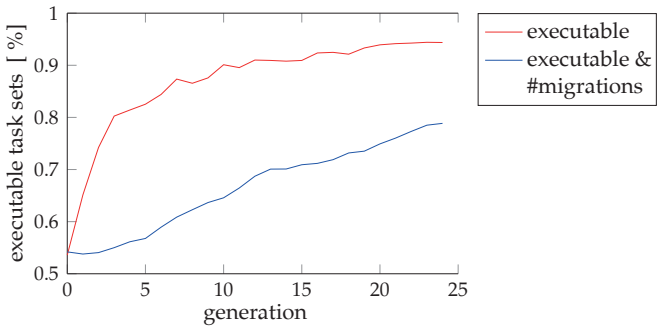


Fig. 7. Fitness ratings that are based on the number of executable task sets exclusively show a faster evolutionary progress, but introduce a considerable amount of task migrations.

A selection of the resulting prioritization functions is listed in Table III. In this example, *L/RU* was capable of scheduling all task sets, but required a substantial amount of task migrations. As another interesting candidate, *AD* reduced the number of migrations by a factor of 35.9, but failed with over 25% of the task sets. These examples show that even simple functions can handle the training task sets very successfully. Our second set of task sets proved to be greater challenge. We conducted elaborate simulation runs each with up to 200 generations. The most successful ones were capable of scheduling 83% of the task sets successfully. Some of them, such as *1 / L*, were capable of executing pure global task sets, but failed with others.

TABLE III. The quality of exemplary prioritization functions based on capability of scheduling task sets and the number of required task migrations.

function	# executable task sets		migrations / task set
L/RU	75	100 %	862
L	71	94.67 %	819
AD	56	74.67 %	24
$AD - 1.0$	56	74.67 %	24

Figure 8 and Figure 9 show which terminals and functions are most dominant. The terminals that are used by the state-of-the-art scheduling algorithms such as laxity *L*, remaining execution time *LD*, deadline *AD* are successful at surviving the selection process. Surprisingly, the processor count, that could be a mechanism to distinguish single-processor from multi-processor systems is only scarcely used for prioritization. The most prominent functions are basic arithmetic functions such as addition and multiplication as well as selecting the minimum and maximum. Functions allowing terminals to have strong influence on the results such as exponentiation and logarithm are only used rarely.

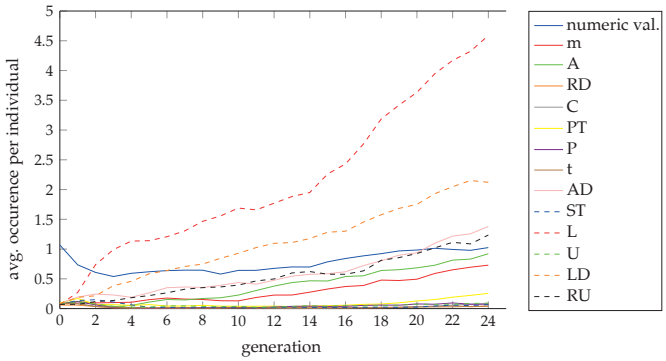


Fig. 8. Terminals with dynamic properties such as Laxity *L*, remaining execution time *LD* and remaining utilization *RU* were especially successful in the evolutionary process.

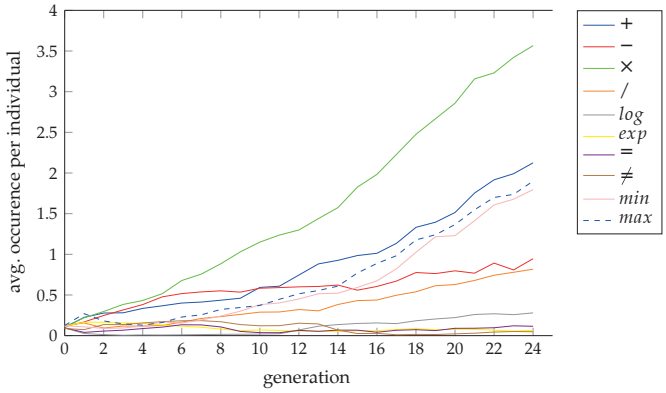


Fig. 9. In our evolutionary process arithmetic operations as well as minimum and maximum operations were predominant.

In the majority of our experiments, we found a vast amount of candidate prioritization schemes with interesting properties. However, a generic optimal solution was not found, concurring with the literature [11], [14], [15].

B. Performance evaluation

Our optimized implementation was able to retrieve valid prioritization functions for multiprocessor systems ranging from 1 up to 400 processors in a feasible amount of time. Benchmarks were performed in a test environment equipped with two Xeon E5620 processors, each containing 4 cores clocked at 2.40 GHz, and 24 GB of main memory. Furthermore, a Xeon Phi 5110P accelerator was employed, providing 8 GB of dedicated memory and 60 cores clocked at 1.053 GHz.

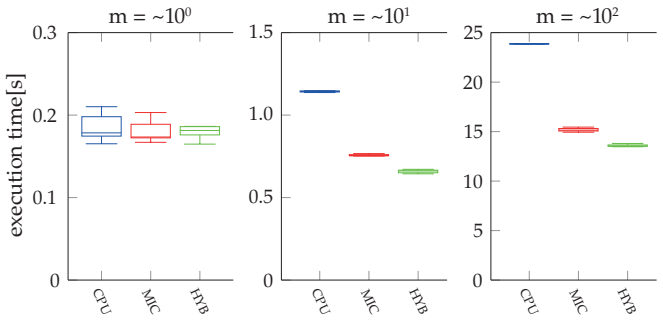


Fig. 10. Across all problem sizes for m, the *MIC* always outperforms the *CPU*. However, the hybrid approach *HYP* always provides an additional performance improvement on top of the *MIC* performance.

The measurements illustrated in Figure 10 demonstrate that even though evolutionary approaches require huge amounts of compute resources, modern CPUs empower us to accomplish the task in acceptable time. The first generation of MIC-based hardware accelerators allowed us to push the limit a little further by achieved speedup factors of 2 for $m = \sim 10^2$.

V. Conclusion

In this work we have studied the feasibility of genetic programming and the evolutionary process to explore the solution space of priority-based scheduling algorithms. We found that this approach is indeed helpful to identify the terminals and functions that are most dominant in promising prioritization schemes. Furthermore, we demonstrated that it is possible to weight desired characteristics like task migration and find optimal schedulers for static task sets by exploiting overfitting.

None of the scheduling algorithms that we generated, not even the most promising ones were capable to schedule all our task sets successfully. These findings harmonize with Fisher's proof [15] that no optimal priority-driven scheduling algorithm exists for arbitrary task sets.

Acknowledgement

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866.

Disclaimer

This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] Burns, A. and Wellings, A. *Real-Time Systems and Programming Languages*. Addison Wesley, 2001.

[2] Dhall, S. K. and Liu, C. L. *On a Real-Time Scheduling Problem*. Operations Research, 1978, Vol. 26, pp. 127-140.

[3] J. Y.-T. Leung, *A new algorithm for scheduling periodic, real-time tasks*. Algorithmica, vol. 4, no. 1-4, pp. 209219, 1989.

[4] S.-H. Oh and S.-M. Yang, *A modified least-laxity-first scheduling algorithm for realtime tasks*. Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on, pp. 3136, IEEE, 1998.

[5] J. M. López, M. García, J. L. Diaz, and D. F. Garcia, *Worst-case utilization bound for edf scheduling on real-time multiprocessor systems*. Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on, pp. 2533, IEEE, 2000.

[6] C. Seongje, L. Suk-Kyoon, and L. Kwei-Jay, *Efficient real-time scheduling algorithms for multiprocessor systems*. IEICE Transactions on Communications, vol. 85, no. 12, pp. 28592867, 2002.

[7] L. Lundberg, *Analyzing fixed-priority global multiprocessor scheduling*. Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE, pp. 145153, IEEE, 2002.

[8] L. Lundberg and H. Lennerstad, *Guaranteeing response times for aperiodic tasks in global multiprocessor scheduling*. Real-Time Systems, vol. 35, no. 2, pp. 135151, 2007.

[9] A. Srinivasan and S. Baruah, *Deadline-based scheduling of periodic task systems on multiprocessors*. Information Processing Letters, vol. 84, no. 2, pp. 9398, 2002.

[10] L. Lundberg, *Slack-based multiprocessor scheduling of aperiodic real-time tasks*. Real-Time Systems, vol. 47, no. 6, pp. 618638, 2011.

[11] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, *Dp-fair: A simple model for understanding optimal multiprocessor scheduling*, Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on, pp. 313, IEEE, 2010.

[12] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, *Proportionate progress: A notion of fairness in resource allocation*. Algorithmica, vol. 15, no. 6, pp. 600625, 1996.

[13] H. Cho, B. Ravindran, and E. D. Jensen, *An optimal real-time scheduling algorithm for multiprocessors*. Real-Time Systems Symposium, 2006. RTSS06. 27th IEEE International, pp. 101110, IEEE, 2006.

[14] K. S. Hong and J. Y.-T. Leung. *On-Line Scheduling of RealTime Tasks*. IEEE Transactions on Computers, 41:1326-1331, 1992.

[15] N. W. Fisher, *The multiprocessor real-time scheduling of general task systems*. University of North Carolina at Chapel Hill, 2007.

[16] E. S. Hou, N. Ansari, and H. Ren, *A genetic algorithm for multiprocessor scheduling* Parallel and Distributed Systems, IEEE Transactions on, vol. 5, no. 2, pp. 113120, 1994.

[17] G. W. Greenwood, A. Gupta, and K. McSweeney, *Scheduling tasks in multiprocessor systems using evolutionary strategies* Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on, pp. 345349, IEEE, 1994.

[18] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, *Cheddar: a flexible real time scheduling framework*, ACM SIGAda Ada Letters. vol. 24, pp. 1-8, ACM, 2004.

[19] F. Golatowski, J. Hildebrandt, J. Blumenthal, and D. Timmermann, *Framework for validation, test and analysis of real-time scheduling algorithms and scheduler implementations*, 13th IEEE International Workshop on Rapid Systems Prototyping, pp. 146-152. IEEE. 2002.

[20] G.A. Lloyd, *Comparing schedulability of global, partitioned and clustered multiprocessor platforms using empirical analysis*, 2010.

[21] T. Bäck and H.-P. Schwefel. *An overview of evolutionary algorithms for parameter optimization* Evolutionary computation, vol. 1, no. 1, pp. 123, 1993.

[22] J. R. Koza *Genetic programming as a means for programming computers by natural selection* Statistics and Computing, vol. 4, no. 2, pp. 87112, 1994.

[23] R. Poli, W. B. Langdon, and N. F. McPhee, *A field guide to genetic programming* Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008.

Notes

OSPERT 2015 Program

	Tuesday, July 7th 2015
8:00 – 9:00	Registration
9:00 – 10:30	Keynote talk: *Software Architectures for Advanced Driver Assistance Systems (ADAS)* *Robert Leibinger*
10:30 – 11:00	Coffee Break
11:00 – 12:30	Session 1: RTOS Design Principles Back to the Roots: Implementing the RTOS as a Specialized State Machine *Christian Dietrich, Martin Hoffmann, Daniel Lohmann* Partial Paging for Real-Time NoC Systems *Adrian McMenamin, Neil Audsley* Transactional IPC in Fiasco.OC - Can we get the multicore case verified for free? *Till Smejkal, Adam Lackorzynski, Benjamin Engel, Marcus Völp*
12:30 – 13:30	Lunch
13:30 – 15:00	Session 2: Short Papers A New Configurable and Parallel Embedded Real-time Micro-Kernel for Multi-core platforms *Antonio Paolillo, Olivier Desenfans, Vladimir Svoboda, Joël Goossens, Ben Rodriguez* Adaptive Resource Sharing in Multicores *Kai Lampka, Jonas Flodin, Yi Wang, Adam Lackorzynski* Implementing Adaptive Clustered Scheduling in LITMUS^{RT} *Aaron Block, William Kelley* Preliminary design and validation of a modular framework for predictable composition of medical imaging applications *Martijn M.H.P. van den Heuvel, Sorin C. Crăcană, Hrishikesh L. Salunkhe, Johan J. Lukkien, Alok Lele, Dominique Segers* Increasing the Predictability of Modern COTS Hardware through Cache-Aware OS-Design *Hendrik Borghorst, Olaf Spinczyk*
15:00 – 15:30	Coffee Break
15:30 – 17:00	Session 3: Isolation, Integration, and Scheduling Evaluating the Isolation Effect of Cache Partitioning on COTS Multicore Platforms *Heechul Yun, Prathap Kumar Valsan* An experience report on the integration of ECU software using an HSF-enabled real-time kernel *Martijn M.H.P. van den Heuvel, Erik J. Luit, Reinder J. Bril, Johan J. Lukkien, Richard Verhoeven, Mike Holenderski* Evolving Scheduling Strategies for Multi-Processor Real-Time Systems *Frank Feinbube, Max Plauth, Christian Kieschnick, Andreas Polze*
17:00 – 17:30	Discussion and Closing Remarks

	Wednesday, July 8th – Friday, July 10th 2015
	ECRTS main conference.

Max
Planck
Institute
for
Software Systems