PROCEEDINGS OF

OSPERT 2018

the 14th Annual Workshop on
Operating Systems Platforms for
Embedded Real-Time Applications

July 3rd, 2018 in Barcelona, Spain

in conjunction with

the 30th Euromicro Conference on Real-Time Systems
July 3–6, 2018, Barcelona, Spain

Editors:
Heechul YUN
Adam LACKORZYNSKI

Contents

Message from the Chairs

Welcome to OSPERT'18, the 14th annual workshop on Operating Systems Platforms for Embedded Real-Time Applications. We invite you to join us in participating in a workshop of lively discussions, exchanging ideas about systems issues related to real-time and embedded systems.

The workshop will open with a keynote by Kai Lampka. Dr. Lampka will discuss mastering security and resource sharing challenges of high-performance controllers in automotive applications. In the afternoon, we also have a second keynote by Dr. Michael Paulitsch, based on his experiences in aviation and chip industries. We are delighted that Dr. Lampka and Dr. Paulitsch volunteered to share their experience and perspective, as a healthy mix of academics and industry experts among its participants has always been one of OSPERT's key strengths.

The workshop received a total of twelve submissions. All papers were peer-reviewed and nine papers were finally accepted. Each paper received three individual reviews.

The papers will be presented in three sessions. The first session includes four papers on real-time operating systems. Best paper will be presented in the second session, while the third session will present four papers on shared memory hierarchy and GPU management.

OSPERT'18 would not have been possible without the support of many people. The first thanks are due to Francisco J. Cazorla and Gerhard Fohler and the ECRTS steering committee for entrusting us with organizing OSPERT'18, 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 twelve experts for volunteering their time and effort to provide useful feedback to the authors, and of course to all the authors for their contributions and hard work.

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,

Heechul Yun
University of Kansas
USA

Adam Lackorzynski
TU Dresden / Kernkonzept
Germany

Program Committee

Marcus Völp, *Université du Luxembourg*
Robert Kaiser, *RheinMain University of Applied Sciences*
Michael Engel, *Coburg University of Applied Sciences*
Michal Sojka, *Czech Technical University in Prague*
Gabriel Parmer, *George Washington University*
Olaf Spinczyk, *Technische Universität Dortmund*
Hyoseung Kim, *University of California Riverside*
Renato Mancuso, *Boston University*
Andrea Bastoni, *SYSGO AG*
Juri Lelli, *Redhat*
Daniel Lohmann, *Leibniz Universität Hannover*
Euiseong Seo, *Sungkyunkwan University*

Keynote Talks

Mastering Security and Resource Sharing with future High Performance Controllers: A perspective from the Automotive Industry

Dr. Kai Lampka
System Architect, Elektrobit Automotive GmbH

On safety and real-time in embedded operating systems using modern processor architectures in different safety-critical applications

Dr. Michael Paulitsch
Dependability Systems Architect (Principal Engineer), Intel

Deterministic Futexes Revisited

Alexander Zuepke, Robert Kaiser
RheinMain University of Applied Sciences, Wiesbaden, Germany
Email: first.last@hs-rm.de

***Abstract*—Fast User Space Mutexes (Futexes) in Linux are a lightweight way to implement thread synchronization objects like mutexes and condition variables. Futexes handle the uncontended case in user space and rely on the operating system only for suspension and wake-up on contention. However, the current futex implementation in Linux is unsuitable for hard real-time systems due to its unbounded worst case execution time (WCET).**

Based upon the ideas from our previous work presented at OSPERT in 2013 which addressed this problem, this paper presents an improved design for *Deterministic Futexes* which shows a logarithmic upper bound of the worst case execution time (WCET) and covers more futex use cases. The implementation targets microkernels or statically configured real-time operating systems.

I. Introduction

Support for Fast User Space Mutexes (Futexes) was introduced in Linux in 2002 [1] with the Native POSIX Thread Library (NPTL). Futexes allow to implement various POSIX-compliant high level synchronization objects such as mutexes, condition variables, semaphores, readers/writer locks, barriers, or one-time initializers with low overhead in the system's C library in user space. One major design goal of futexes was to reduce any system call overhead for these locking objects where possible, thus the implementation uses atomic modifications to handle uncontended locking and unlocking entirely in user space, while a generic system call-based mechanism is used to suspend and wake threads in the kernel on lock contention. Basically, a futex is a 32-bit integer variable in user space, representing a certain type of lock and its value is modified by a type-specific locking protocol [2].

Similar approaches where the kernel is entered only on contention are used by *Critical Sections* in Microsoft Windows [3] and *Benaphores* in BeOS [4].

We give a short introduction to futexes using a simple mutex implementation as example: in an integer variable, let bit 0 represent the locked state of the mutex, while bit 1 indicates contention. The unlocked mutex is represented by the value 0x0. A thread can lock and unlock the mutex by atomically changing the lock value from 0x0 to 0x1 and vice versa using a *Compare-and-Swap (CAS)* or *Load-Linked/Store-Conditional (LL/SC)* operation.

A lock operation on an already locked mutex atomically changes the value from 0x1 to 0x3 to indicate contention and then invokes a `FUTEX_WAIT` system call to suspend the calling thread until the lock becomes available again. Symmetrically, when the current lock-holder sees contention during an unlock operation, it atomically clears the locked bit in the futex value and calls the `FUTEX_WAKE` system call to wake a blocked thread which then acquires the lock by atomically setting bit 0 again. On contention, `FUTEX_WAIT` enqueues the thread on a *wait queue* which holds blocked threads referring to the same or a different user space futex. For wake-up, `FUTEX_WAKE` searches the wait queue and wakes up matching threads, if any.

The last important operation on futexes is `FUTEX_REQUEUE` to prevent *thundering herd effects* [5] when signalling condition variables: instead of waking up all threads and letting them compete to lock the associated mutex, this system call wakes only one thread and moves any remaining blocked threads from the wait queue associated to the condition variable to the mutex' one.

By design, futexes impose no restrictions on the number of user space variables used for futexes or on the number of threads blocked in a wait queue. This flexibility makes the concept very attractive and led to its recent adoption by other operating systems [6]–[8].

However, being designed for best case scenarios, the current futex *implementation* in Linux has drawbacks which make it unsuitable for hard real-time operating systems:

- *Hash table with shared wait queues:* Linux hashes the futex user space address and groups threads with the same hash value into a *shared wait queue*. This can lead to an unbounded worst case execution time (WCET) when, due to hash collisions, many unrelated threads are kept in the same hash bucket.
- *Linked lists:* Linux implements wait queues using *priority-sorted linked lists*, which show $\mathcal{O}(n)$ search time in shared wait queues and $\mathcal{O}(p)$ insertion time, for n threads and p priority levels.
- *Not preemptive:* When waking up or requeuing a large number of threads, the Linux implementation is not preemptive. Again, this can lead to an unbounded WCET.

In previous work [9], we presented a solution which tackles these problems by using a dedicated kernel-internal wait queue for each futex. To let the kernel look-up the wait queue, we placed the ID of the first waiting thread next to the futex value in user space. The solution then utilized $\mathcal{O}(1)$ insertion and deletion time of linked lists to bound the WCET. However, the solution in [9] supported only FIFO ordering in the wait queues, so it does not fulfill the POSIX requirement to wake up threads in priority order [10].

In this paper, we present an improved futex implementation with the following properties:

- dedicated wait queues for each futex,
- arbitrary ordering in the wait queues,

- bounded $\mathcal{O}(\log n)$ worst case execution time in the kernel for all futex operations targeting a single thread,
- preemptible implementation of futex operations which wake up or requeue all threads, and
- no dependency on dynamic memory allocation.

The rest of this paper is organized as follows: Section II describes all futex operations in detail and defines requirements for determinism and reliability. Section III presents our new approach. We discuss our new approach and compare it with the current Linux implementation and our previous approach in Section IV and we conclude in Section V.

II. Fast User Space Mutexes and Condition Variables

A. Terminology

Before we discuss the futex operations, we define the terminology used in the rest of this paper: a *process* is an instance of a computer program executing in an *address space*. A process comprises one or more *threads*. Threads can be independently scheduled on different processors at the same time. Different processes have their own distinct address space, but processes can share parts of their address spaces via *shared memory segments*. A shared memory segment is usually *mapped* at different virtual addresses in each address space. A *waiting* or *blocked* thread suspends execution until the thread is *woken up* or *unblocked* again.

B. Futex Operations in User Space

Here, we briefly present the user space parts of a futex-based mutex and condition variable implementation to help understanding the corresponding kernel parts. The mutex protocol extends the one shown in Section I and uses different kernel operations. Note that the presented user space implementation is simplified for ease of understanding. An actual user space implementation will usually be more complex, as the calls also have to handle asynchronous signals, thread cancellation, etc., but the interaction with the kernel side of the presented futex implementation remains the same. The presented futex API also deviates from the existing Linux API in that the handling of an arbitrary number of `count` threads is reduced to the two most common use cases, *one* or *all*. This helps to bound the WCET, as we will explain later.

Mutex: For a mutex, the futex value comprises two pieces of information: the thread ID (`TID`) of the current lock holder or 0 if the mutex is free, and a *waiters* bit if the mutex has contention. Also both user space and the kernel need to understand this mutex protocol.

`mutex_lock` first tries to lock a mutex by atomically changing the futex value from 0 to `TID`. If the mutex is already locked, `mutex_lock` atomically sets the waiters bit in the futex value to indicate contention, then calls `futex_lock` to suspend itself on the current futex value. The `futex_lock` operation in the kernel checks the futex value again and tries to either acquire the mutex for the caller if it is free, or, if not, atomically sets the waiters bit in the futex value and suspends the calling thread. On successful return from `futex_lock`, the calling thread is the new lock owner.

Conversely, `mutex_unlock` tries to unlock the mutex by atomically changing the futex value from `TID` to 0. If this fails (the waiters bit is set), `mutex_unlock` calls `futex_unlock` in the kernel. If no threads are waiting, `futex_unlock` sets the futex value to 0, or wakes up the next waiting thread and makes it the new lock owner by updating the `TID` in the futex value. `futex_unlock` sets the waiters bit as well if other threads are still waiting.

Condition Variable: For a condition variable, the futex value represents a counter that is incremented on each wake-up operation. The kernel does not need to know the exact protocol. When doing any operation on a condition variable, we assume the caller also has the associated mutex locked [10].

`cond_wait` reads the condition variable's counter value, unlocks the associated mutex, and then calls `futex_wait` to block with an optional timeout on the condition variable if the current counter value still matches the previously read value. Additionally, `cond_wait` provides the mutex object to later requeue to as well.

`cond_signal` and `cond_broadcast` increment the counter and call `futex_requeue` to requeue either one or all blocked threads from the condition variable's wait queue to the mutex' wait queue. In case the caller has not locked the mutex before, `futex_requeue` checks whether the associated mutex is unlocked, wakes up the first blocked thread and makes it the new lock owner instead of requeuing it. Remaining threads are requeued.

After wake-up, `cond_wait` needs to check the cause of the wake-up: if the thread was requeued, the condition variable must have been signalled, and the caller already owns the mutex. Otherwise, if the timeout expired or the counter's current value mismatched, the caller was not requeued to the mutex' futex and the function needs to lock the mutex again.

Note that the `cond_wait` operation exposes a race condition which may result in a *lost wake-up*. Lost wake-ups are normally prevented by the kernel comparing the futex value, but if – between the time `cond_wait` unlocks the mutex in user space and the time the kernel checks the futex value– exactly 2^{32} wake-up operations are performed, the futex value overflows to exactly the same value and the check would succeed. However, this problem is unlikely to appear in practice, unless the system overloads and low priority waiters do not progress anymore.

Corresponding futex operations in Linux with similar API and behavior are `FUTEX_LOCK_PI` and `FUTEX_UNLOCK_PI` for mutexes, and `FUTEX_WAIT_REQUEUE_PI` and `FUTEX_CMP_REQUEUE_PI` for condition variables [11]. The Linux implementation additionally supports a priority inheritance protocol which is not in the focus of this paper.

C. Futex Operations in the Kernel

We now describe the futex kernel operations. We consider a *wait queue* to be a set of blocked threads waiting on a futex. The kernel creates and destroys wait queues *on demand*. Note that in the following description, a wait queue is *specific to a*

single futex and is never shared between multiple futexes. The Linux implementation *differs* from this model insofar as the Linux kernel shares a single wait queue for multiple futexes, but the description still matches the Linux model if we ignore unrelated threads in a wait queue and assume that wait queues always exist, as the wait queues in Linux are created at boot time and remain persistent.

Wait Queue Look-up: As mentioned before, the kernel's `futex` operations must relate the user provided futex address to a wait queue by a *look-up function*. If the futex object is shared between processes, the kernel uses the physical address of the futex. For futexes local to the caller's address space, the kernel can use the virtual address for look-up instead. We define further requirements for the corresponding look-up function later. For now, we assume that the kernel maintains *sets* of wait queues and distinguishes local and shared futexes properly, e.g. address space-specific sets for local futexes, and a global set of wait queues for shared futexes.

`futex_lock(&futex, timeout)` handles locking for a mutex. The function first checks whether a wait queue for the futex exists in the set of wait queues. If not, it creates a new wait queue and adds it to the set. Then the kernel evaluates the futex user space value: if the mutex is unlocked, the kernel tries to atomically acquire it for the caller and returns if successful. If the mutex is locked, but the waiters bit is not set, the kernel atomically sets the waiters bit in the futex value. Finally, the kernel enqueues the thread into the wait queue and blocks it with the given timeout. When the timeout expires or the blocking is cancelled for other reasons, e.g. by a signal, the kernel removes the thread from its wait queue. Otherwise, the thread is already successfully dequeued from the wait queue. It is woken up, and becomes the new lock owner. In all error cases, the kernel also removes empty wait queues from the set and destroys them.

`futex_unlock(&futex)` first looks up the wait queue, and if one exists, it wakes up a waiting thread and makes it the new lock owner by updating the futex value in user space. If there are still blocked threads in the wait queue, the kernel additionally sets the waiters bit. Once a wait queue becomes empty after wake-up, the kernel removes and destroys it.

`futex_wait(&futex, compare, timeout, &futex2)` first checks whether a wait queue exists in the set of wait queues, otherwise it creates a new one and inserts it into the set. Then, before enqueuing the calling thread into the wait queue, the kernel checks if the futex user space value still matches the provided compare value, and returns an error if not. The rest of `futex_wait` follows `futex_lock`, but without any updates of the futex value in user space. `futex_wait` accepts an optional second futex which is the target mutex in a requeue operation. `futex_wait` also makes sure that all blocked threads refer to the same second futex (or `NULL`) to simplify the requeue operation.

`futex_wake(&futex, ONE|ALL)` first looks up the wait queue, and, if one exists, wakes up one or all threads. Again, empty wait queues are removed afterwards.

`futex_requeue(&futex, ONE|ALL)` works similarly to `futex_wake`: First, the kernel looks up the wait queue and operates on the given number of blocked threads. Then the kernel requeues threads to their associated mutex wait queue, which it has to look-up as well and possibly create. Eventually, the kernel also checks the mutex value, and if the mutex is currently unlocked, the kernel wakes up the first thread instead of requeuing it, and makes it the new lock owner with the waiters bit set accordingly. The threads are expected to have set a mutex to requeue to, otherwise the call fails.

Locking: All operations also require internal locks in the kernel: Usually, a whole set of wait queues is either protected by a specific lock, or a wait queue provides a specific lock itself (Linux). These internal locks are necessary for the futex protocols to serialize concurrent user space access and concurrent futex operations.

D. Requirements for Determinism

The presented futex operations in the kernel are quite complex. If they are to be used in a real-time system, they must be deterministic, i.e. have a WCET which is (i) *analyzable* and (ii) *bounded*. The main idea is to prevent sharing of wait queues and to use *dedicated* wait queues for each futex instead. This means we have to manage a *set of wait queues* (one for each futex), and each wait queue only contains a *set of blocked threads* specific to the futex. Here we define the requirements for such an implementation:

1) No dynamic memory allocations shall be used for creating wait queues. The problem is simply that dynamic memory allocations can fail at runtime. Also, having fewer dependencies on other components simplifies the WCET analysis.
2) For wake-up and requeuing operations to achieve real-time scheduling, POSIX requires that threads with the highest scheduling priority have to be woken up first. For threads with the same priority, FIFO ordering must be used. This means that wait queues shall be properly ordered.
3) All operations on a set of blocked threads in a specific wait queue i.e. $find$, $insert$, and $remove$ of threads, shall have at worst $\mathcal{O}(\log n)$ execution time, for n threads in the wait queue. This suggests to use *self-balancing binary search trees*, a data structure where the execution time of all operations stays within logarithmic bounds.
4) Similarly, all operations on the set of wait queues, e.g. insertion of a new wait queue into the set, shall have at worst $\mathcal{O}(\log m)$ execution time as well, for m wait queues in the set.
5) `futex_wake` and `futex_requeue` handle a potentially large number of threads in the `ALL` case, so their execution shall be preemptible after handling each thread.
6) `futex_wake/requeue` operations on all threads in a wait queue shall eventually terminate, i.e. threads are not allowed to sneak in into a currently processed wait queue again. This condition follows from the previous requirement that futex operations shall be preemptible.

7) The preemptible operations on all blocked threads in a wait queue shall not be observable by these threads if the threads follow the usage constraints properly. This condition also follows from requirement 5.
8) The implementation should support fine granular locking, i.e. locks on the set of wait queues and a particular wait queue are decoupled to reduce interference between operations on unrelated wait queues.

Note, requirements 3–6 have the same upper bound n, i.e. the overall number of threads in the system, when all threads block on either a different or the same futex. Also, requirements 3 and 4 are *not* required for determinism in the first place, as $\mathcal{O}(n)$ time is deterministic as well. But having an upper bound of $\mathcal{O}(n)$ execution time is only acceptable if n is both known and small. Thus the approach would not be applicable to systems with a very large number of threads.

Preemptible execution of the `ALL` operations is a good compromise with respect to the worst case time an operation holds internal locks, but it introduces its own problems, as requirements 6 and 7 state.

The last requirement helps to simplify WCET analysis, but this is not a hard requirement.

III. Implementation

In this section, we describe our implementation.

As described before, futexes in general require two different data structures in the kernel: (i) a wait queue handling all blocked threads waiting on the same futex, and (ii) a data structure to locate this wait queue, based on the futex user space address as look-up key. We explicitly need this *two-tier design* to isolate threads waiting on unrelated futexes and to support a preemptive implementation of the `ALL` operations.

For both data structures, self-balancing binary search trees (BST) are suitable, e.g. red-black trees or AVL trees. In our futex implementation, we chose to use AVL trees.

Like in Linux and our previous implementation [9], we keep all data related to futex management inside the *thread control block* (TCB) of the blocked threads to get rid of the dependency on dynamic memory management, thus fulfilling requirement 1.

A. Binary Search Trees

From the BST implementation, we require the standard operations $find$, max, $insert$, and $remove$, and additionally $root$ and $swap$. The $root$ operation locates the root node of the BST from any given node, thus requiring that nodes in the BST use three pointers: two for the left and right child nodes, and a third one to the parent node. The $swap$ operation allows to swap a node in the tree with another node outside the tree in $\mathcal{O}(1)$ time without altering the order in the tree. Lastly, the BST implementation requires a *key* to create an ordered tree. The key may not be unique, e.g. threads with the same priority are allowed to exist in the tree. If nodes with duplicate keys need to be inserted, we require FIFO ordering of the duplicate nodes.

B. Wait Queue Look-up in the Address Tree

To locate a wait queue from a futex address, we designate *one* of the blocked threads in a wait queue as *wait queue anchor*. The anchor thread has the root pointer to the wait queue. All wait queue anchors are enqueued in an *address tree*, which is rooted in an *address tree root*.

Key: For shared futexes, we use the *physical address* of the futex as key; and for per-process futexes, we use the *virtual address* as key. Also, both shared and per-process futexes are kept in distinct trees: shared futexes are kept in a global tree shared between all processes, while per-process futexes are kept in process-specific data, e.g. in the process descriptor.

We use the fact that futex variables in user space are 32-bit integers that are aligned on a 4-byte boundary. As the last two bits of a futex address are always zero, we use them to encode further information.

We define that a wait queue is *open* if threads can be added to it, i.e. new threads can block on a futex, and a wait queue is *closed* if new threads can not be added.

We decode the open/closed state of a wait queue in its key: An open wait queue has the lowest bit *set* in the key, for a closed wait queue the bit is *cleared*. By clearing the open bit, we can change a wait queue from open to closed state without altering the structure of the tree. Also, we do not allow open wait queues with duplicate keys, as each key relates to a unique futex in user space. However, multiple wait queues with the same closed key may exist, and they become FIFO ordered due to the ordering constraints in the BST when changing a wait queue from open to closed state. We later exploit this mechanism in `futex_wake` and `futex_requeue` to wake or requeue all threads in a preemptible fashion.

For closed wait queues, we also define a *drain ticket* attribute, a counter value which helps during `ALL` operations later. The drain ticket is a global 64-bit counter incremented each time a wait queue is closed. It should not overflow in practice.

The last specialty in the address tree is the following: if the thread used as wait queue anchor changes, we simply $swap$ the old anchor thread in the tree with a newly designated anchor thread without altering the structure of the tree and we copy the wait queue root pointer, the current drain ticket, and the current open/closed state in the key as well.

This design allows us to perform look-up, insertion, and removal of wait queues in $\mathcal{O}(\log n)$ time, while changing a wait queue from open to closed state and changing the wait queue anchor both need $\mathcal{O}(1)$ time. This fulfills requirement 4.

C. Wait Queue Management

As stated before, the wait queue anchor thread is an arbitrarily chosen blocked thread in the wait queue which holds the root pointer of the wait queue and the open/closed state of the wait queue encoded in the key. We refine this now and define that the thread being the *current root node* of the wait queue is to be used as anchor. If the root node changes due to insertion or removal in the wait queue tree, we swap the root nodes in the address tree as described above. Using the root node thread as its anchor is not mandatory, as any node in

the wait queue would do, but this simplifies the implementation when threads are woken up for other reasons, e.g. timeouts, as explained below.

When a thread blocks on a unique futex address, the kernel creates a new wait queue on demand in open state and inserts it into the address tree with this first thread as anchor. Note that this does not involve allocation of memory. Similarly, the wait queue is implicitly destroyed when the last thread (which again must be the anchor) is woken up. The kernel then removes the wait queue from the address tree.

The kernel inserts threads in priority order into an existing wait queue. Also, when waking or requeuing threads, we remove the highest priority thread first.

Removal of an arbitrary node, e.g. on a timeout, requires to find the associated wait queue root to rebalance the tree afterwards. We do not look-up the wait queue in the address tree in this case, as it might have been set to closed state and then up to two look-ups in the address tree would be required. Instead, we simply traverse the wait queue tree to the root node to locate the anchor and remove the thread. This is also necessary when a thread's scheduling priority changes while the thread is blocked. In this case, we remove the thread and re-insert it with its new priority.

If during insertion or removal the wait queue root changes due to the necessary rebalancing in the BST, we transfer the wait queue root pointer and the other current wait queue attributes to the new root and update the address tree accordingly.

This design allows to perform all internal operations on wait queues in at most $\mathcal{O}(\log n)$ time. With it, we are now able to implement `futex_lock`, `futex_unlock`, and `futex_wait` in $\mathcal{O}(\log n)$ time. Also, a `futex_wake` and `futex_requeue` operation targeting a single thread takes $\mathcal{O}(\log n)$ time. This fulfills requirements 2 and 3.

D. Preemptible Operation

We now discuss the preemptibility of `futex_wake` and `futex_requeue` if `ALL` threads need to be handled. In this case, both operations set the wait queue to closed state first, so it can no longer be found by enqueuing operations, then we draw a unique *drain ticket* and save the ticket in the anchor node.

Then the kernel wakes up or requeues one thread after another, but becomes preemptible after handling each thread. After preemption, the kernel is always able to find the wait queue again by looking for the now closed wait queue. If multiple closed wait queues with the same key are found, the drain ticket decides what to do. The FIFO ordering in the BST makes sure that nodes are found with increasing drain ticket numbers. If the drain ticket number of a node is less than the originally drawn ticket, the wait queue relates to an older, but still unfinished operation, and draining older wait queues on behalf of some other thread is fine. So the caller can safely perform its operations as long as the drain ticket number is less than or equal to the drawn drain ticket. The drain ticket is therefore necessary to prevent already handled threads to re-enter these wait queues.

Since at most n-1 threads can be blocked before a draining operation starts and a drain ticket is drawn, the upper limit of steps to complete a `futex_wake` or `futex_requeue` operation is therefore n. This fulfills requirements 5 and 6.

But is it acceptable in general to drain other thread's wait queues? We can answer this question if we look at the following usage constraint of condition variables: the caller of `cond_signal` and `cond_broadcast` shall have the support mutex locked as well, so none of the requeued threads will run before the caller unlocks the support mutex. Therefore, handling threads of a previous waiting round can only happen when `cond_signal` and `cond_broadcast` do not have the support mutex locked, and in this case, POSIX does not longer guarantee "predictable scheduling". This means the answer is yes, and we fulfill requirement 7.

A different use case is a POSIX barrier implementation where a given number of threads block until all threads have reached the barrier. An implementation of `barrier_wait` could then use `futex_wake` to wake all blocked threads. A preemptive `futex_wake` operation could get immediately preempted by a higher priority thread which is woken up as first thread and then the other threads are kept blocked until the original thread continues draining the wait queue. Note that this would not happen in a non-preemptible implementation. However, POSIX also notes that applications using barriers "may be subject to priority inversion" [10]. Alternatively, the barrier implementation can mitigate this issue by temporarily raising the caller's scheduling priority to a priority higher than the priorities of all blocked threads during wake-up.

E. Locking Architecture

The final point to be discussed is the locking architecture to fulfill requirement 8. In this case, we cannot easily provide a solution. We could, for example, implement a nested locking hierarchy where the kernel first locks the address tree, locates a wait queue, locks the wait queue, and then unlocks the address tree again. The strict order in which locks are taken is necessary to prevent deadlocks. But this design approach does not allow to remove an empty (and locked) wait queue from the address tree without holding the address tree lock. Doing this would require unlocking the wait queue first, then locking the address tree, and then finally locking the wait queue again. However, this kind of re-locking exposes races, as the re-locked wait queue may no longer be empty due to concurrent insertion on other processors. And this problem becomes even worse in our design as changes to a wait queue anchor require frequent updates in the address tree.

Still, we assume that a solution can be found, e.g. using a lock-free look-up mechanism in the address tree, but it is still questionable if such an approach would improve the WCET or would simplify the WCET analysis in the end.

For now, we decide to not implement a nested locking scheme as requested by requirement 8, but to use a shared lock for both the address tree and all wait queues. Note that we use dedicated locks for each per-process address tree and the shared global address tree.

Table I
COMPARISON OF FUTEX IMPLEMENTATIONS

	Our new approach	**Our old approach**	**Linux**
Futexes share wait queues	no	no	yes
Wait queue look-up	BST $\mathcal{O}(\log m)$	via `TID` $\mathcal{O}(1)$	hash table $\mathcal{O}(1)$
Wait queue implementation	priority-sorted BST	FIFO-ordered linked list	priority-sorted linked list
- find	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
- insertion	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(p)$
- removal	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Locking	global	global	per hash bucket
`futex_requeue`			
- one thread	yes	yes	yes
- arbitrary number of threads	no	no	yes
- all threads	yes	yes	yes
- preemptive implementation	yes	not needed	no
`futex_wake`			
- one thread	yes	yes	yes
- arbitrary number of threads	no	no	yes
- all threads	yes	not provided	yes
- preemptive implementation	yes	not needed	no
Priority ceiling protocol	yes	yes	yes
Priority inheritance protocol	no	no	yes
for n threads, m futexes, and p priority levels			

IV. DISCUSSION

In this section, we compare our *new approach* presented in this paper with our *old approach* in [9] and the current Linux implementation in kernel 4.16.

We briefly repeat the key points of our previous implementation in [9]:

- All futex-related data is kept in the TCB.
- Threads on a wait queue are kept in FIFO order.
- Wait queues use linked lists with FIFO ordering.
- For wait queue look-up, the kernel saves the `TID` of the first waiting thread next to the futex value in user space, and updates the `TID` value each time a wait queue changes.
- The *requeue all* operation appends the whole linked list of threads to requeue at the end of the mutex wait queue list in $\mathcal{O}(1)$ time.
- A *wake all* operation is not provided.
- All other operations handle insertion or removal in $\mathcal{O}(1)$ time as well.

Table I shows the differences between the implementations. The complexity of the Linux implementation clearly show that it was designed for the best case, e.g. when only a small number of threads block and collisions in the futex hash table are rare. And this is *usually* the case during normal operation of a system. However, if one considers certification or needs to determine deterministic upper bounds of the WCET, the possible corner cases in the Linux implementation lead to potentially unbounded execution time, e.g. a malicious application could exploit collisions in the hash.

Our old implementation in [9] already addressed these issues, but it does not support priority ordered wait queues which are required for POSIX scheduling. Also, the old implementation does not support POSIX barriers.

Our new implementation presented in this paper is superior in all these respects, however the overhead of a BST compared to linked lists seems quite heavy if the number of used futexes and blocked threads is low. This needs to be evaluated in future work.

Also, our presented locking approach is restricted to a single lock for all futexes, which is worse in the average case compared to the Linux implementation, as Linux uses a dedicated lock for each hash bucket.

Finally, our old and new implementations do not support all futex uses cases available in Linux, as we restrict our implementation to handle either just one or all threads, not an arbitrary number. Regarding other missing features: All discussed approaches can support the priority ceiling protocol defined by POSIX, which adjusts a thread's scheduling priority *before* locking a mutex [10]. But in addition, Linux also supports a priority inheritance protocol for mutexes. This would be possible for our presented design, but this is currently left to future work.

V. CONCLUSION AND OUTLOOK

We have shown an approach to improve the determinism of the kernel parts of a futex implementation by using a two-tier design using two nested self-balancing binary search trees, namely one tree to look up futex wait queues by their address, and a second tree to manage blocked threads in priority order. The shown design has a bounded WCET of $\mathcal{O}(\log n)$ time for all non-preemptible kernel operations with respect to the number of concurrently used futexes and/or blocked threads.

The presented approach is suitable to implement the standard POSIX thread synchronization mechanisms, like mutexes, condition variables, or barriers on top [2]. Also, the presented approach supports the POSIX priority ceiling protocol.

In future work, we would like to improve internal locking in the kernel implementation to reduce interference between unrelated processes. Finally, we would like to evaluate means to support *priority inheritance protocols*.

REFERENCES

[1] H. Franke, R. Russell, and M. Kirkwood, "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux," in *Proceedings of the Ottawa Linux Symposium*, 2002, pp. 479–495.

[2] U. Drepper, "Futexes Are Tricky," White Paper, Nov. 2011. [Online]. Available: https://www.akkadia.org/drepper/futex.pdf

[3] "Windows InitializeCriticalSection function." [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms683472(v=vs.85).aspx

[4] B. Schillings, "Be Engineering Insights: Benaphores," *Be Newsletters*, vol. 1, no. 26, May 1996.

[5] D. Hart and D. Guniguntalay, "Requeue-PI: Making Glibc Condvars PI-Aware," in *Eleventh Real-Time Linux Workshop*, 2009, pp. 215–227.

[6] "Windows WaitOnAddress function." [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/hh706898(v=vs.85).aspx

[7] "Fuchsia zx_futex_wait function." [Online]. Available: https://fuchsia.googlesource.com/zircon/+/master/docs/syscalls/futex_wait.md

[8] "OpenBSD futex manual page." [Online]. Available: https://man.openbsd.org/futex

[9] A. Zuepke, "Deterministic Fast User Space Synchronisation," in *OSPERT Workshop*, 2013.

[10] IEEE, "POSIX.1-2008 / IEEE Std 1003.1-2017 Real-Time API," 2017.

[11] Linux futex manual page. [Online]. Available: http://man7.org/linux/man-pages/man2/futex.2.html

Implementation and Evaluation of Multi-Mode Real-Time Tasks under Different Scheduling Algorithms

Anas Toma, Vincent Meyers and Jian-Jia Chen
Department of Computer Science
TU Dortmund University
Dortmund, Germany
firstname.lastname@tu-dortmund.de

***Abstract*—Tasks in the multi-mode real-time model have different execution modes according to an external input. Every mode represents a level of functionality where the tasks have different parameters. Such a model exists in automobiles where some of the tasks that control the engine should always adapt to its rotation speed. Many studies have evaluated the feasibility of such a model under different scheduling algorithms, however, only through simulation. This paper provides an empirical evaluation for the schedulability of the multi-mode real-time tasks under fixed- and dynamic-priority scheduling algorithms. Furthermore, an evaluation for the overhead of the scheduling algorithms is provided. The implementation and the evaluation were carried out in a real environment using Raspberry Pi hardware and FreeRTOS real-time operating system. A simulation for a crankshaft was performed to generate realistic tasks in addition to the synthetic ones. Unlike expected, the results show that the Rate-Monotonic algorithm outperforms the Earliest Deadline First algorithm in scheduling tasks with relatively shorter periods.**

I. Introduction

In modern automotive systems, *Electronic Control Units* (ECUs) are used to control and improve the functionalities, the performance and the safety of various components. These embedded systems are in continuous interaction with various parts of the automobile such as the doors, the wipers, the lights and most importantly the engine [14]. In order to guarantee a correct behavior, the embedded system should react within a specific amount of time, i.e. the deadline. The timing correctness in these systems is very important, because a delayed reaction can result in a faulty behavior and then affect the reliability and safety of the automobile.

The software of an automotive application can be modeled as a set of recurrent tasks with timing constraints, i.e. periodic real-time tasks. For instance, to control the engine of an automobile, an *angular* task may release jobs depending on the engines speed. Such a task is linked to the rotation of specific devices such as crankshaft, gears or wheels. It could be responsible for calculating the time at which the spark signal should be fired, adjusting the fuel flow, or minimizing fuel consumption and emissions [9]. The period of this task, i.e. the time between the release of two consecutive jobs, is inversely proportional to the speed of the crankshaft. With an increasing rotation speed, the time available for the task to execute all of its functions may not be long enough, which results in deadline misses. This could lead to catastrophic consequences in hard real-time systems [6].

TABLE I: An example of a multi-mode task with three different execution modes.

Rotation Speed (rpm)	Mode Type	Executed Functions
$[0, 3000]$	A	$f1$, $f2$ and $f3$
$(3000, 6000]$	B	$f1$ and $f2$
$(6000, 9000]$	C	$f1$

In order to meet the timing constraints and prevent a potential system failure, the job has to react before the next job is released. Therefore, the task might have to drop some of its functions, the non-critical ones, to meet its deadline. This can be achieved by using tasks with different execution modes, i.e. *multi-mode tasks*, to adapt to the changing environment [15]. In some cases, tasks may react differently according to an external input and thus switch into different modes accordingly. In our example of the automobile's engine, the input is the engine speed and the functionalities of the tasks are part of the fuel injection system. Every time the crankshaft finishes a rotation, the tasks have to execute their respective functions. If the engine speeds up, the tasks may need to use another algorithm or functions to achieve their goal and avoid deadline misses. In other cases, the engine may be more stable at higher rotation speeds, but requires additional functions to be executed at lower speeds to keep it stable. Consequently, these functions are not required to be executed at higher speeds, which can be exploited to reduce the execution time of the tasks [7]. Table I shows an example of a multi-mode task with 3 types of execution modes: A, B and C. The selection of the mode depends on the rotation speed, where the task executes different functions in each mode. The rotation speed of the engine is measured in *revolutions per minute* (rpm).

Such a task model was presented by Buttazzo et al. [7]. They also provide schedulability analysis under Earliest Deadline First (EDF) algorithm. Furthermore, another analysis under Rate Monotonic (RM) algorithm is provided in [9], in addition to simulation for the effectiveness of the proposed test. However, none of the studies above performed the evaluation of the system in a real environment. In this paper, we provide an empirical evaluation of multi-mode tasks under EDF and RM algorithms. The evaluation was performed on a real hard-

ware running a real-time operating system. The contribution of this paper can be summarized as follows:

- Modifying the FreeRTOS real-time operating system to consider the periodic and multi-mode real-time tasks. Furthermore, several cost functions were implemented for a comprehensive evaluation[1].
- Implementing the EDF and RM scheduling algorithms in FreeRTOS which can be used in further studies and researches[1].
- Empirical evaluation for the schedulability of the multi-mode tasks under EDF and RM algorithms in a real environment, i.e. FreeRTOS running on Raspberry Pi. Moreover, overhead evaluation of both algorithms is provided in this work.

II. Background and Literature Review

A. FreeRTOS

In this Subsection, we introduce the FreeRTOS and its main components that were modified in our implementation [4]. FreeRTOS is a real-time operating system kernel that supports about 35 microcontroller architectures. It is a widely used and relatively small application consisting of up to 6 C files [3]. FreeRTOS can be customized by modifying the configuration file *FreeRTOSConfig.h*, e.g. turning preemption on or off, setting the frequency of the system tick, etc. Tasks in FreeRTOS execute within their own context with no dependency on other tasks or the scheduler. Upon creation, each task is assigned a *Task Control Block* (TCB) which contains the stack pointer, two list items, the priority, and other task attributes. Tasks can have priorities from 0 (the lowest) to *configMAX_PRIORITIES*-1 (the highest), where *configMAX_PRIORITIES* is defined in *FreeRTOSConfig.h*. A task in FreeRTOS can be in one of the following four states:

- Running: The task is currently executing.
- Ready: The task is ready for execution but preempted by an equal or a higher priority task.
- Blocked: The task is waiting for an event. The task will be unblocked after the event happens or a predefined timeout.
- Suspended: The task is blocked but does not unblock after a timeout. Instead the task enters or exits the suspended state only using specific commands.

The following are the main functions and data structures in FreeRTOS which will be mentioned in the following sections:

- *xTaskCreate()*: Creates a task and add it to the ready list.
- *prvInitialiseTCBVariables()*: Initialize the fields of the TCB.
- *vTaskDelayUntil()*: Delays a task for a specific amount of time starting from a specified reference of time.
- *vTaskStartScheduler()*: Starts the FreeRTOS scheduler.
- *pxReadyTasksLists*: An array of doubly linked lists with size of *configMAX_PRIORITIES* that contains the ready tasks according to their priorities. Each array element and a corresponding list represents a level of priority.
- *uxTopReadyPriority*: A pointer to the task with the highest priority in the ready list.

[1]The implementation is available on https://github.com/multiModeFreeRTOS/multiMode

The scheduler in FreeRTOS is responsible for deciding which task executes at a specific time. It is triggered by every system tick interrupt and schedules the task with the highest static priority in the ready list for execution. It loops the ready list from the pointer *uxTopReadyPriority* to the lowest priority that has a non-empty list. If two tasks have the same priority, they share the CPU and switch the execution for every system tick.

B. Scheduling the Multi-Mode Tasks

Buttazzo et al. [7] provide analysis for the feasibility of multi-mode tasks under the EDF algorithm. Furthermore, a method is provided to determine the switching speed that keep the utilization of the tasks below a predefined threshold. On the contrary, Huang and Chen [9] present a feasibility test for such a task model under RM algorithm. Furthermore, they show the advantages of using the fixed-priority scheduling over the dynamic-priority scheduling. Both of the studies above evaluated their approaches by simulation.

III. Real-Time Multi-Mode Task Model

Multi-mode tasks are periodic tasks that can be executed in several modes [9]. Given a set $\mathcal{T}$ of n independent real time tasks. Each task i (for $i = 1, 2, \ldots, n$) has m_i execution modes, i.e., $\tau_i = \{\tau_i^1, \tau_i^2, \ldots, \tau_i^{m_i}\}$. In each mode τ_i^j, the task has different *worst-case execution time* (WCET) C_i^j, *period* T_i^j and *relative deadline* D_i^j. The task consists of an infinite sequence of identical instances, called jobs. T_i^j represents the time interval between the release of two consecutive jobs of the same task. Once a job is released, it should be executed within the deadline D_i^j. The mode of the task may change based on an external interrupt or any other event, which can be used to change the execution time of the tasks and then the total utilization accordingly. If the mode is changed during the runtime, it will take effect in the next period.

IV. Design and implementation

This section covers the implementation of the multi-mode task model and both scheduling algorithms in FreeRTOS (A ported version to Raspberry Pi [1]).

A. Multi-Mode Task Model

1) Periodic Real-Time Tasks: It is necessary to have a periodic task model in order to implement the multi-mode tasks. Therefore, the tasks in FreeRTOS were modified by expanding the *task control block* (TCB) structure with the typical fields used in periodic real-time systems [6]. In addition to the original TCB attributes in FreeRTOS, the following ones with *portTickType* data type were added:

- *uxPeriod*: Period.
- *uxWCET*: Worst-case execution time.
- *uxDeadline*: Relative deadline.
- *uxPreviousWakeTime*: The previous wake time of the task.

The absolute deadline of a task can then be calculated as $D = uxDeadline + uxPreviousWakeTime$. Those attributes were also added to the parameters of the *xTaskGenericCreate()*, *xTaskCreate()* and *prvInitialiseTCBVariables()* functions to be initialized upon task creation. To guarantee the

periodicity of the tasks, i.e. constant execution frequency, the *vTaskDelayUntil()* function is used to delay the task for the specified period of time T_i^j starting from the arrival time captured by the *xTaskGetTickCount()* function and stored in *uxPreviousWakeTime* variable.

2) Modes: Now, we have a periodic task model and it will be modified to have different execution modes. To achieve that, the TCB attributes described in Subsection IV-A1 should have many values corresponding to the modes of the task. Since the number of the modes are fixed and known upon system setup, an array data structure is used to store the several values of the same attribute. The TCB fields were modified as follows:

- `portTickType *uxPeriods;`
- `portTickType *uxWCETs;`
- `portTickType *uxDeadlines;`

Additional attributes were added to store the number of the modes and determine threshold values for each mode level as follows:

- **`unsigned int`** `uxNumOfModes`: The number of the modes.
- **`unsigned int`** `*uxModeBreaks`: The range for each mode.

uxModeBreaks contains the maximum value of each mode level. For example, the first mode (indexed by 0) will be chosen if the external input is between 0 and *uxModeBreaks*[0]. Similarly, the range of the second mode is (*uxModeBreaks*[0],*uxModeBreaks*[1]]. The parameters were also added to the corresponding functions as described in Subsection IV-A1.

To switch to the corresponding mode during the runtime, the function *vUpdateMode()* was implemented. It chooses the appropriate mode based on an external input and the defined mode ranges in the array *uxModeBreaks*. The value of the external input is stored in a global variable named *externalInput* with type *volatile unsigned int*. It is declared as volatile, because its value might change at any moment during the runtime. So, any application can change the mode easily by updating this variable according to an external input or any other event. The *externalInput* is initialized to 0, which means that the first mode is the default one. According to the definition of the multi-mode tasks in Section III, tasks do not change their mode once a mode change request is arrived, even if they are blocked. Any changes will be applied starting from the next release. Therefore, the mode is updated in our implementation right before the next wake-up time. This was done by calling *vUpdateMode()* at the start of the function *prvAddTaskToReadyQueue()*.

B. Rate-Monotonic Scheduler

According to the RM algorithm, the priorities of the tasks are assigned statically before the execution according to their periods, i.e., the tasks with a shorter period has a higher priority [12]. We reserve the priority level 1 in FreeRTOS and the corresponding ready list *pxReadyTasksLists*[1] for the tasks to be scheduled under RM algorithm. All of these tasks are assigned to priority 1 upon creation temporarily. Then, their priorities are assigned according to RM algorithm before the scheduler is started. A new function named *vAssignPriorities()* was implemented and is called in *vTaskStartScheduler*() function after the creation of the idle task to assign those priorities. Another attribute, *unsigned int *uxPriorities*, was added to the TCB to store the priorities of the same task for all the corresponding modes. Moreover, the following doubly linked list was created to sort the tasks according to their periods in all of their modes:

```
1 struct doublyLinkedListNode {
2    unsigned int value;
3    void *task;
4    int mode;
5    volatile struct doublyLinkedListNode *↩
         prev;
6    volatile struct doublyLinkedListNode *↩
         next;
7 };
```

Where *value* and *task* store the period of each mode and a pointer to the corresponding task's TCB respectively. The tasks in *pxReadyTasksLists[1]* are inserted into the doubly linked list and sorted according to their periods. Then, the priorities are assigned for each task for all the modes by filling the *uxPriorities* array. Finally, the tasks are moved to their corresponding ready lists according to the new assigned priorities.

C. Earliest Deadline First Algorithm

The EDF algorithm assigns the highest priority to the job with the earliest absolute deadline among of the ready jobs [13]. Before implementing the EDF algorithm, task creation functions were modified, so the tasks can be scheduled dynamically. The static priority parameter in the *xTaskCreate()* function is discarded by setting it always to 1. The FreeRTOS uses an array of linked lists to store the ready tasks according to their priorities. The array size can be defined by the variable *configMAX_PRIORITIES*. However, it is not suitable to use an array with a fixed size for dynamic priority assignment. Of course this array can still be used, but either it should be big enough for any eventual number of tasks, or its size should be always reallocated. To avoid such an overhead, we replaced the the ready list *pxReadyTasksLists* with a doubly linked list that has the same name to maintain all the ready tasks. We apply a binary heap on the ready tasks to find the one with the highest priority. Every time a task is added to the ready list by calling the *prvAddTaskToReadyQueue()* function, the absolute deadline is calculated, as shown in Subsection IV-A1, and the task with the earliest absolute deadline is scheduled for execution.

D. Additional Modifications

1) Shared Processor Behavior: In this subsection, we present the additional modifications to the system in order to improve the overall performance and make our EDF implementation work appropriately. In the FreeRTOS, the tasks share the processor equally if they have the same priority. The processor executes the tasks in a round-robin behavior, which results in a context switching for every system tick and then additional overhead. The actual cost of switching between two tasks is approximately $4\mu s$ per every context switch according to our measurements. Even if the ready list has just one task or only one task has the highest priority, the FreeRTOS performs

context switching on the same task for every system tick. This includes saving the state of the task and restoring it every system tick which results in a high overhead. We solved such a problem by performing context switching only if we have a new task with a higher priority or if the current task under execution is moved to the blocked state. Context switching is then only conducted when necessary. Tasks with the same priority are scheduled according to their insertion order in the ready list.

2) Performance and Evaluation metrics: For system evaluation, we implemented the following cost functions to measure the performance of the implemented schedulers [6]:

- System overhead: The time required to handle all mechanisms other than executing jobs such as scheduling decisions, context switching and system tick interrupts.
- Success ratio: The percentage of the schedulable task sets among the total number of the task sets.
- Average response time:

$$t_r = \frac{1}{n}\sum_{i=1}^{n}(f_i - a_i)$$

  where a_i and f_i are the arrival time and the finishing time of task execution respectively.
- Maximum lateness:

$$L_{max} = \max_i(f_i - d_i)$$

3) Configurations and Definitions: Several configuration parameters were added to the system which are required for the evaluation or visualization of the scheduling. The following definitions were added to the file *FreeRTOSConfig.h*:

- *configANALYSE_METRICS*: Trace the data of the tasks for the evaluation metrics (1: Enabled, 0: Disabled).
- *configANALYSE_OVERHEAD*: Measure the total time consumed by the tick interrupts (1: Enabled, 0: Disabled).
- *configPLOTTING_MODE*: Trace tasks at the context switches (1: Enabled, 0: Disabled).
- *configTICKS_TO_EVAL*: The period time in milliseconds for any of above modes to run.
- *configEVAL_THRESHOLD*: The time between evaluation rounds. It must be long enough for tasks to delete themselves.
- *configUSE_TASK_SETS*: Consider more than one task set for evaluation. (1: Multiple task sets, 0: Only one task set).
- *configSET_SIZE*: The number of the task sets used in the evaluation.
- *configNUMBER_OF_TASKS*: The number of the tasks per a task set.

Python scripts for the evaluation metrics, the overhead and the plotting were also implemented.

V. Experimental Evaluations

Two evaluation methods were conducted in our work. In the first one, we implemented a python script to generate tasks synthetically. In the second evaluation method, we generated task sets with timing characteristics similar to the tasks in a real-world automotive software system. The first and the second types of tasks are called *synthetic* and *realistic* task sets respectively.

Period	Share
1 ms	3 %
2 ms	2 %
5 ms	2 %
10 ms	25 %
20 ms	25 %
50 ms	3 %
100 ms	20 %
200 ms	1 %
1000 ms	4 %
angle-synchronous ms	15 %

TABLE II: Task distribution among periods

Mode	0	1	2	3	4	5
Min.	0	1001	2001	3001	4001	5001
Max	1000	2000	3000	4000	5000	6000
Period	30	15	10	7.5	6	5

TABLE III: 6 modes ranging from 0 to 6000 rpm with their periods in milliseconds.

A. Setup

The FreeRTOS was used as a real-time operating system to implement the multi-mode tasks and both scheduling algorithms on Raspberry Pi B+ board [1, 2]. The hardware board has ARM1176JZF-S 700 MHz processor and 512 MB of RAM. The UART interface of the Raspberry was used to generate an external interrupt. The corresponding interrupt service routine sets the global variable *externalInput* to the number of the mode determined by the evaluation script used in each respective evaluation method. The function *setupUARTInterrupt()* was implemented in the file *uart.c* located in the drivers directory in order to set up the UART interface.

Two types of task sets were generated: (1) synthetic and (2) realistic. For the synthetic tasks, a set of utilization values were generated in the range of 10% to 100% with a step size of 10 according to the *UUniFast* algorithm [5]. The approach in [8] was used to generate periods in the range of 1 to 100 ms with an exponential distribution. The WCET C_i^j of each task was calculated by $T_i * U_i$. The deadlines are implicit, i.e. equal to the period. A proportion p of those tasks were converted to multi-mode tasks with M modes. Note that the normal periodic tasks are multi-mode tasks with only one mode $M = 1$. The generated values above were assigned for the first mode of all the tasks. For the multi-mode tasks, the values for the remaining modes were scaled by the factor of 1.5, i.e., $C_i^{m+1} = 1.5 * C_i^m$, $T_i^{m+1} = 1.5 * T_i^m$. For each multi-mode task, one of the modes was then chosen to have the highest utilization while the WCETs of the other modes were reduced by multiplying them with random values between 0.75 and 1. According to the configurations above, 100 task sets were generated with 50% multi-mode tasks and cardinality of 10, i.e. the number of the tasks per a task set. The number of modes used in the evaluation are 5, 8 and 10. Each task set was assigned 10 seconds for execution and 5800 ms to delete itself.

Furthermore, realistic tasks that share the characteristics of

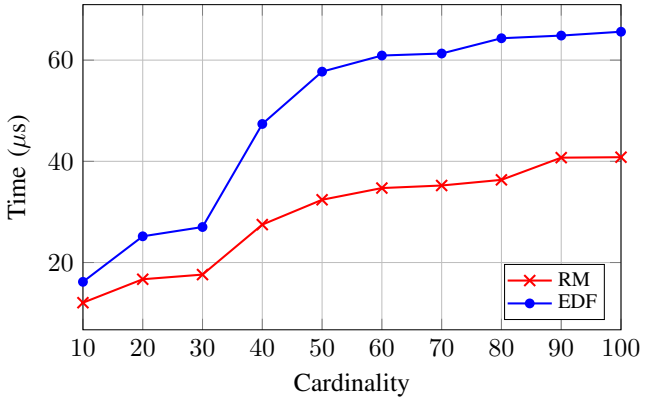
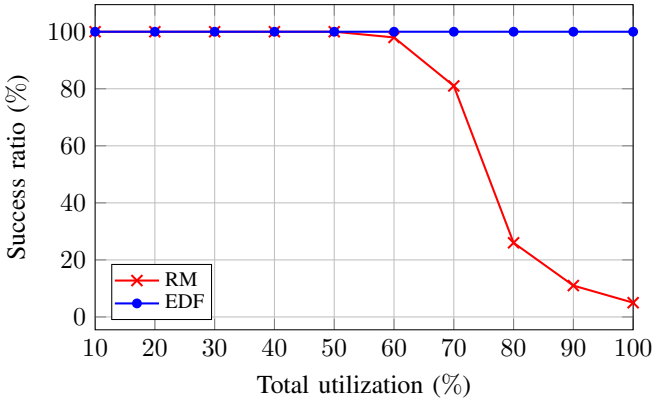
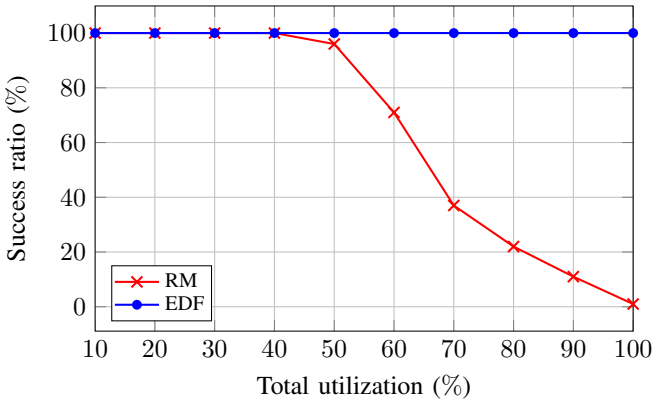


Fig. 1: The overhead of RM and EDF scheduling algorithms.



(a) M = 5



(b) M = 10

Fig. 2: Percentage of the schedulable task sets for 5 and 10 modes using the synthetic tasks.

an automotive software system were generated as presented by Kramer et al. [10]. These characteristics cover the distribution of the tasks among the periods, the typical number of the tasks, the average execution time of the tasks and factors for determining the best- and worst-case execution times. Table II shows the distribution of the tasks among the periods [10]. The angle-synchronous tasks, which take 15% of all the tasks, are converted to multi-mode tasks as their worst-case execution time needs to adapt to their reduced period. In our case, the maximum engine speed is 6000 rpm with 4 available cylinders. For the conversion to multi-mode tasks, the engine speed was divided into 6 intervals and the periods were calculated by the upper bound of each mode as shown in Table III. The WCET of the tasks was assigned to the lowest mode. For the remaining modes, it was calculated based on the utilization of the first mode, i.e. $C_i = T_i * U_1$ and $U_1 = \frac{C_1}{T_1}$. Moreover, we implemented a crankshaft simulation that starts at an angular speed of 1 rpm and increases by 1000 rpm over 500 ms, and sends a signal every time the piston reaches the maximum position. This happens every one full rotation of the crankshaft. Once the simulated crankshaft reaches its highest speed of 6000 rpm, it will slow back down to 1 rpm. The acceleration/deceleration is steady during the whole execution. 100 task sets were generated per each utilization level from 10% to 100% with a step size of 10.

B. Results

The success ratio of the tasks and the overhead of the algorithms used in this subsection are defined in Subsection IV-D2. Figure 1 provides an evaluation for the overhead of both scheduling algorithms. As expected, the EDF algorithm has a higher overhead than the RM algorithm due to the dynamic priority assignment, where the priority of the jobs may change during the runtime. The EDF algorithm should always keep tracking of the absolute deadlines of the jobs, whilst the priorities according to RM algorithm are fixed prior to the execution, and the algorithm should just pick the next task in the ready list. We also observe that the overhead of both algorithms increases as the cardinality (i.e. the number of the tasks per a task set) increases. The increase of cardinality results in a longer ready list, which explains the growth in the overhead.

Figures 2 and 3 show the impact of task utilization on the success ratio of the synthetic and realistic tasks respectively. They also compare between RM and EDF algorithms. What can be clearly seen in Figure 2 is that the EDF algorithm was able to find more feasible schedules than the RM algorithm. All the task sets with a utilization of up to 100% and up to 10 modes were feasibly scheduled under EDF. However, the RM algorithm could only achieve that for a utilization of up to 50% and 40%, and for a configuration of 5 and 10 modes respectively. After those levels of utilization, the success ratio of the RM algorithm decreases significantly. This is due to the fact that the EDF algorithm has a higher utilization bound than the RM algorithm.

If we now turn to the realistic tasks, we observe that the EDF algorithm performs worse than the RM algorithm, which is unexpected. It was able to schedule all the task sets with a total utilization of only 10%. It failed to schedule any task set with a total utilization of more than 50%. However, the RM algorithm could schedule all the task sets with a total utilization of up to 40% and was still able to find feasible schedules for some of the task sets with a total utilization of up to 60%. This behavior is due to the high overhead of the EDF algorithm and the distribution of the tasks among the periods in this data set. The realistic data set has more tasks with shorter periods than

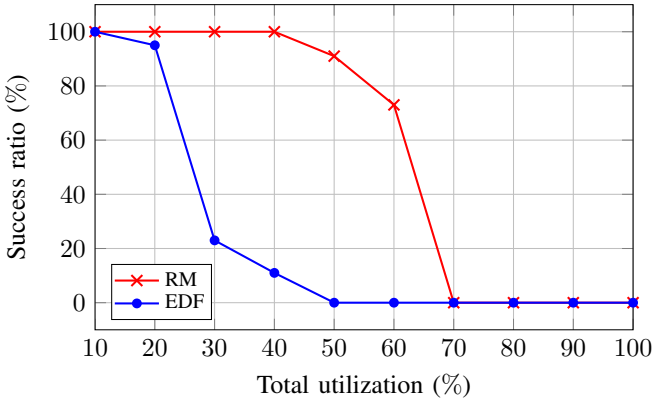


Fig. 3: Percentage of the schedulable task sets using the realistic tasks.

the synthetic one. For high workloads, the sum of the time required for the scheduling decision and the execution time of the job exceeds the relative deadline and then results in unschedulable task sets.

VI. Conclusion

In this paper, we evaluate the multi-mode tasks under the EDF and the RM scheduling algorithms in a real environment. To achieve that, the FreeRTOS real-time operating system was modified to implement this task model and both scheduling algorithms. Moreover, additional modifications were performed to provide configurable evaluation metrics. The experiments were performed on Raspberry Pi B+ board. Synthetic and realistic data sets were used in the evaluation. For the realistic data set, we generated angular tasks with periods tied to the rotation of a simulated crankshaft. The experiments confirmed that the EDF algorithm was in general able to find more feasible schedules than the RM algorithm for the synthetic task sets with high utilization values. However, it performed poorly when the realistic data set with relatively shorter periods was used, although a binary heap was used in the implementation to reduce the overhead of the scheduling decision. More feasible schedules were derived under the RM algorithm for this data set due to the low scheduling overhead.

VII. Future Work

Further work could usefully improve the implementation of the EDF algorithm by using a hardware accelerated binary heap to reduce the overhead caused by the dynamic scheduling [11]. However, such an implementation requires a special or additional hardware. Moreover, the system could be modified to handle task overruns.

VIII. Acknowledgements

This work is supported by the German Research Foundation (DFG) as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/) and as part of the Transregional Collaborative Research Centre Invasive Computing [SFB/TR 89].

References

[1] Freertos ported to raspberry pi. URL https://github.com/jameswalmsley/RaspberryPi-FreeRTOS.

[2] Raspberry Pi 1 Model B+. URL https://www.raspberrypi.org/documentation/hardware/.

[3] The FreeRTOS Kernel. URL http://www.freertos.org.

[4] An implementation of multi-mode real-time tasks, rate monotonic algorithm and earliest deadline first algorithm in FreeRTOS. URL https://github.com/multiModeFreeRTOS/multiMode.

[5] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, 2005.

[6] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 2004.

[7] G. C. Buttazzo, E. Bini, and D. Buttle. Rate-adaptive tasks: Model, analysis, and design issues. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

[8] R. I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Transactions on Computers*, 57(9):1261–1276, 2008.

[9] W.-H. Huang and J.-J. Chen. Techniques for schedulability analysis in mode change systems under fixed-priority scheduling. *2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 00:176–186, 2015. doi: doi.ieeecomputersociety.org/10.1109/RTCSA.2015.36.

[10] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmarks for free.

[11] N. C. Kumar, S. Vyas, R. K. Cytron, C. D. Gill, J. Zambreno, and P. H. Jones. Hardware-software architecture for priority queue management in real-time and embedded systems. *International Journal of Embedded Systems*, 6(4):319–334, 2014.

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

[13] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[14] N. Navet and F. Simonot-Lion. *Automotive embedded systems handbook*. CRC press, 2008.

[15] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):243–264, 1989.

Jitter Reduction in Hard Real-Time Systems using Intra-task DVFS Techniques

Bo-Yu Tseng
School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology

Kiyofumi Tanaka
School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology

***Abstract*—In some real-time control applications, the predictability of task's timing behaviour is as important as energy consumption. That predictability includes the response time and short finish time jitter. This paper presents a jitter-aware Intra-task DVFS scheme for mitigating finish time jitter in hard real-time systems. This work exploits Dynamic Voltage and Frequency Scaling (DVFS) technique to proactively manipulate actual execution/response times of tasks. The strategy proposed in this paper mainly applies control and data flow analysis of task program to insert additional frequency scaling codes (instructions to change processors voltage and frequency). Moreover, it determines the appropriate frequency scaling factor. Through evaluation by multitasking simulation, it is shown that jitter can be reduced by up to 16.2%-19.4%.**

I. Introduction

In some real-time control applications, large jitter jeopardise the stability or processing accuracy of the systems [1].

To reduce jitter, **deadline assignment algorithm** by linear programming was proposed [7]. Deadline assignment alg. attempts to shorten relative deadlines of some periodic tasks whilst keeping the schedulability, by promoting priorities of certain tasks to reduces the number of preemptions. Variation in preemption duration makes contributes to jitter, accordingly the less the preemption, the less the jitter. However, this approach only takes preemptions into account. In fact, variation in execution time is another factor leading to jitter. Other works that exploit **Dynamic Voltage and Frequency Scaling (DVFS)** to handle jitter [9], [10]. DVFS has been widely utilised in energy efficiency, using slack time to scale down the operating frequency. In the meanwhile, DVFS enables the system to control the actual execution/response times of periodic tasks, thus it is applicable to reducing finish time jitter. Mochocki, et al. exploited only the suitable portion of slack time to scale down the operating frequency for some lower-priority tasks' instances instead of aggressively using all slack time for energy reduction [9]. Their work is based on **Inter-task** perspective, hence frequency scaling can be performed only at the start time of instances. Phatrapornnant and Pont proposed a similar jitter-aware DVFS algorithm called TTC-jDVS algorithm, which incorporates jitter reduction in an Inter-task DVFS scheme [10]. However, it reduces start time jitter only, ignoring finishing time jitter or the variation in execution time.

The objective of this paper is to reduce finishing time jitter under Rate-Monotonic scheduling (RM). We propose a **jitter-aware Intra-task DVFS scheme** to make task scheduling adapt to runtime variations due to both interference and execution time. The Intra-task DVFS approach [11]–[14] promises higher granularity of frequency scaling within one instance of task's execution. Thus, it relatively outperforms the Inter-task DVFS approach in terms of energy reduction. Apart from the effect of energy efficiency, it is expected that the Intra-task DVFS approach manipulate finishing time jitter. Our algorithm targets at reducing variation in both execution time and interference time.

This work is the first to control the finishing time jitter using Intra-task DVFS, to the best of the authors knowledge.

II. Preliminaries

A. The Causes of Jitter

It is useful to clarify the sources of jitter is necessary. Start time jitter directly depends on the task's priority, which further affects the variation in preemption/interference within the interval from release time to start time. On the other hand, finish time jitter is caused by variation in both preemption and execution time.

B. Jitter Margin

In real-time control systems (e.g., closed-loop control), a nearly constant computational delay of periodic tasks is essential due to the system requirement of stability/robustness. Hence the **jitter margin** covers time delay in a periodic control task was introduced to guarantee system stability [4]. Response time of a task instance consists of two parts: 1) constant delay $\boldsymbol{L}$ and 2) time-varying jitter. The jitter margin J_m is the maximum value of the time-varying part. The exact values of constant delay L and J_m for each periodic task can be calculated by response time analysis. Accordingly, L can be regarded as the best-case response time (BCRT) of the task whilst J_m is the worst-case response time of task (WCRT) minus L, as in the equations 1 and 2. In those equations, $hp(i)$ is the set of tasks with higher priority than task τ_i, P_j is the period of task τ_j, and $BCET_i$ and $WCET_i$ are its best-case and worst-case execution times respectively.

$$L_i = BCET_i + \sum_{j \in hp(i)} \left\lceil \frac{BCRT_i}{P_j} - 1 \right\rceil \cdot BCET_j \quad (1)$$

$$J_m^i = WCET_i + \sum_{j \in hp(i)} \left\lceil \frac{WCRT_i}{P_j} \right\rceil \cdot WCET_j - L_i \quad (2)$$

Although DVFS can control the actual response time, schedulability must be maintained. Hence we derive the model of jitter margin to perform safe DVFS operation in terms of schedulability of periodic tasks. We redefine the constant and varying delay parts as **constant response** and **variance response**, respectively.

The schedulability is checked for a task set $T = \{\tau_1, \tau_2, ..., \tau_n\}$, $\forall \tau_i \in T, BCRT_i \leq WCRT_i \leq D_i$, where D_i is the relative deadline of τ_i. The variance response of a periodic task differs from instance to instance due to the variations in execution time and total preemption time from higher-priority tasks. Consequently, this part leads to **possible range of finish time jitter**, and furthermore it can bound available frequency-scaling factor in terms of schedulability even if the system slows down the processing speed. The detail of utilising the jitter margin for DVFS operation are given in Section III-D.

C. *Runtime Profiling*

In any practical real-time operating system, there is one main data structure, Task Control Block (TCB_i), for each task τ_i [3]. TCB_i includes priority, computation time (WCET), period, and deadline. We add three additional control parameters into TCB_i to get required profiling information (recorded maximum response time $\boldsymbol{R_i^{max}}$, recorded minimum response time $\boldsymbol{R_i^{min}}$, and actual interference time $\boldsymbol{I^{actual}(i)}$). In addition, one global control parameter for the whole task set, global slack time $\boldsymbol{Slack_{global}}$. It represents the total difference between WCET and the actual execution time of the currently running task.

- **Recorded Maximum/Minimum Response Time**
  R_i^{max} and R_i^{min} are used to record the maximal and minimal response time among all past instances of task τ_i. Once τ_i finishes each instance execution, the system updates these two parameters if the response time of the current instance yields maximum or minimum response time, respectively.
- **Updating the Recorded Slack Time**
  Once a running task finishes its instance's execution, the system updates $Slack_{global}$. When the ready queue is not empty, $Slack_{global}$ is set to the difference between WCET and actual execution time unless the ready queue is empty then $Slack_{global}$ is reset to zero.
- **Updated Actual Interference Time**
  WCRT is the only offline information for schedulable guarantee. WCRT assumes that every higher-priority task runs up to its WCET when it preempts the target analysed task (lower-priority task). Accordingly, we can obtain the **worst-case interference time** encountered by τ_i, $I^{worst}(i)$. It corresponds to the difference between $WCET_i$ and $WCRT_i$. Obviously, there is a possibility that higher-priority tasks would have shorter execution time than the corresponding WCETs. This overestimation of response time degrades accuracy. To provide a more accurate interference time, we initialise $I^{worst}(i)$ to $I^{actual}(i)$ and then update $I^{actual}(i)$ by $I^{actual}(i) = I^{actual}(i) - Slack_{global}$ at start and resume time of the running task.

III. Jitter-aware Intra-task DVFS Scheme

In this section, **Jitter-aware Intra-task DVFS scheme** is presented, which is an extension of the existing Intra-task DVFS [12], [13]. Originally, the purposes are to reduce energy consumption of a single periodic task. On the other hand, our work aims to finish time jitter in multitasking environments. The response time of periodic tasks are controlled by changing the speed of the system according to both actual interference and execution times. The overall framework of the proposed approach is shown in Figure 1. It consists of four phases, e.g. 1) control and data flow analysis, 2) execution cycle estimation, 3) frequency-scaling point placement and 4) frequency-updated ratio calculation.

As shown in Figure 1, our scheme is mainly separated into off-line and run-time stages. In the off-line stages, source code (C codes) of given target tasks are analysed in order to obtain their control flow graphs (CFGs) and data flow information. Then each basic block of CFGs is examined by **execution trace mining** [13] to record the worst-case remaining execution cycles (processing cost). Finally, locations of frequency-scaling points are determined. The details are described in Section III-A to Section III-C.

In the run-time stage, the system mainly performs DVFS operation as a part of the task scheduling. The new operating frequency is decided by referring to the given frequency(and power) settings and scaling point lists. The details are described in Section III-D.

A. *Control and Data Flow Analysis*

In a task scheduling, response time of periodic tasks can be expressed as the combination of execution time and interference time. Although interference time encountered by task τ_i is the sum of execution times of higher-priority tasks. The actual execution time varies from one instance to another. Hence runtime variation from the interference time and execution time are the key factors which affect finish time jitter.

In this phase, runtime variation is estimated by static WCET analysis of tasks' source codes. The researches of static timing analysis share one of common idea, that is to break a task's source code into a control flow graph (CFG) with all execution paths. Figure 2a shows an example task source code, and its corresponding CFG in Figure 2b. Each basic block in CFG shows its calculated execution cycles[1].

[1]The execution cycles depend on the target micro architecture.

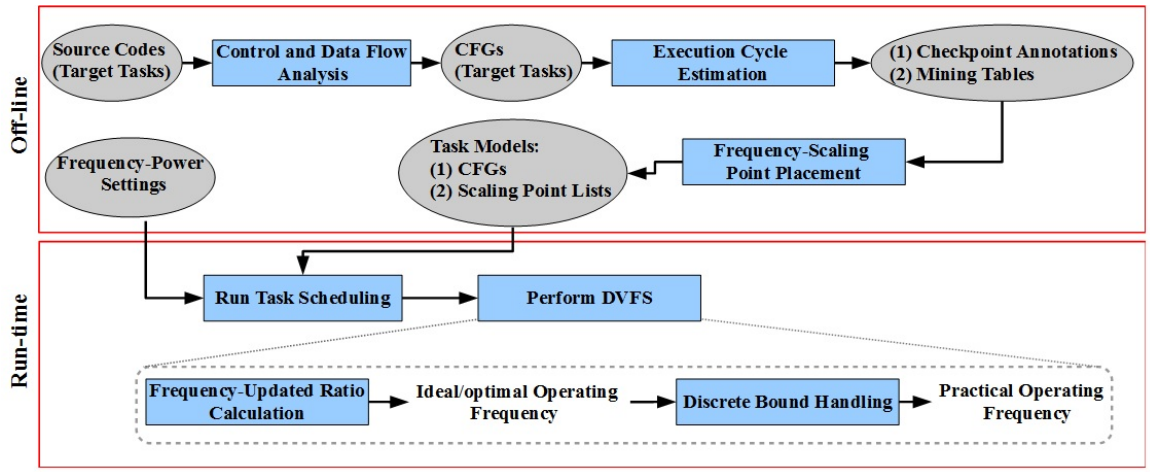
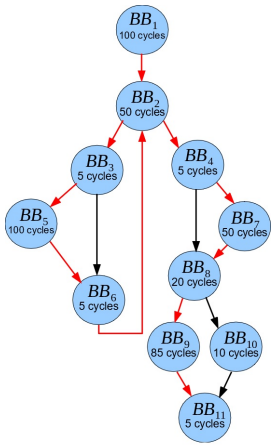


Fig. 1: **The framework of Jitter-aware Intra-task DVFS scheme**

```
1:  x = func1();
2:  func2();
3:  while(x != 0) {
4:    y = func3();
5:    if(y != 0)
6:      func4();
7:    z = func5();
8:    x-=1;
9:  }
10: if(z != 0)
11:   func6();
12: a = func7();
13: if(a == 1)
14:   func8();
15: else
16:   func9();
17: func10()
```

(a) Source code of target task



(b) CFG of target task

Fig. 2: **Control flow of target task's source code**

The control flow information is obtained by the techniques in [11], considering only the branch and loop structure. Additionally, data flow analysis is applied to detect loop dependency [14]. When a source code contains either **while-** or **for-loop**, the number of iterations is determined by particular variables. For instance, in Figure 2a, the number of iterations of while-loop depends on the variable **x**. Thus it is regarded as the induction variable of the while-loop. Since the value of induction variable can differ from one instance to another, it varies task's execution time. Thereby the Line 1 of Figure 2a is the point making system be possible to predict the execution time early.

B. Execution Cycle Estimation

This phase consists of two steps. In the first step, **B-type** and **P-type checkpoints are defined as the points where the execution flow changes**, this step prepares for deciding the locations of frequency-scaling described in Section III-C.

- **B-type Checkpoint**
  It deals with the execution flow change caused by branches. In Figure 2b, a B-type checkpoint is inserted right after basic block 4's execution. When the system finishes executing basic block 4's instructions, it will be known which one of the two following paths ($BB_7 \rightarrow BB_8 \rightarrow BB_9 \rightarrow BB_{11} \rightarrow$ end or $BB_{10} \rightarrow BB_{11} \rightarrow$ end).
- **P-type Checkpoint**
  It deals with loop and loop dependency. A checkpoint is inserted right after the basic block which contains loop dependency. For instance, in Figure 2a, the while-loop's dependency is at Line 1 and the corresponding instructions are inside the basic block 1. Thus, the system can predict the actual number of iterations of the while-loop in advance, that is, after the basic block 1's execution.

In the second step, the **remaining worst-case execution cycles (RWCECs)** from each checkpoint to the end of the task's execution is calculated. According to the **Execution Trace Mining [13]**, RWCECs of paths from each branch as well as their corresponding instruction addresses are recorded in a **mining table**. Our approach constructs two types of mining tables: 1) B-type mining table and 2) P-type mining table, as shown in Table I.

TABLE I: **Mining tables of Figure 2b's task CFG**

(a) B-type mining table

B-type Mining Table				
Address	$Successor_1$	$RWCEC_{successor_1}$	$Successor_2$	$RWCEC_{successor_2}$
#1(BB_4)	BB_7	160(*cycles*)	BB_8	110(*cycles*)
#2(BB_8)	BB_9	90(*cycles*)	BB_{10}	15(*cycles*)

(b) P-type mining table

P-type Mining Table				
Address	*Loop Entry*	*Loop Bound*	$WCEC_{iteration}$	$RWCEC_{outside\_loop}$
#1(BB_1)	BB_2	3(*iterations*)	160(*cycles*)	75(*cycles*)

B-type mining table records the locations of B-type checkpoints (**Address** column), the first basic block of each successive path and the corresponding RWCECs.

The system looks up the RWCEC when it reaches the checkpoint. On the other hand, in P-type mining table, every row deals with one specific loop in the CFG. It records the locations of P-type checkpoints, loop entries, loop bounds (the maximal possible number of iterations)[2], WCECs for one iteration, and RWCECs after the execution of the loop. In Figure 2b, there is one loop starting from basic block 2 (called loop entry) and this loop's dependency is basic block 1. Therefore, RWCECs from each P-type checkpoint can be calculated by the following function.

$$RWCEC = C(LoopEntry) + Iter_{actual} \cdot WCEC_{iteration} + RWCEC_{outside\_loop} \quad (3)$$

In this function, $C(LoopEntry)$ means the cost for executing the loop entry's basic block whilst $Iter_{actual}$ represents the actual number of iterations of the loop.

C. Frequency-Scaling Point Placement

In order to shorten the finish time jitter, variance of interference and execution time is handled. The main purpose of this phase is to determine when the system invokes DVFS operation to reduce those the variances. There are three execution points where performing frequency-scaling is possible: 1) task's start point, 2) B-type checkpoint, and 3) P-type checkpoint. The first one aims to reconfigure a default operating frequency due to updated actual interference time whilst the second and third cope with variance of execution time.

In Figure 3, τ_1 has WCET of 3 and period(=Deadline) of 5, and τ_2 has WCET of 2 and period(=Deadline) of 5, with τ_1 having higher priority than τ_2. This example shows a finish time jitter of one tick for τ_2 where response times of τ_1's first and second instances are not constant. This leads to different interference times on the τ_2's instances. It is obvious that the actual start time of lower-priority task is affected by higher-priority tasks. Here, frequency-scaling points are inserted at the start time of a lower-priority task. As a result, shorter response time for τ_2's first instance or longer response time for the second one is obtained, which can reduce the difference from all τ_2's instances.

On the other hand, as the aforementioned variety of execution paths in one task's CFG, it incurs variation in execution time. Therefore, frequency-scaling points are inserted at every location of branch and loop dependency. Such approach can equalise the response times of the running task even if it follows different execution paths.

The exact resume times of the running task which is preempted by higher-priority tasks is another factor affecting the final response time. A strategy for inserting frequency-scaling points right after resume time can further control response time. This enhancement is left for our future work.

[2]We assume that all loop bounds are given at the compile stage.

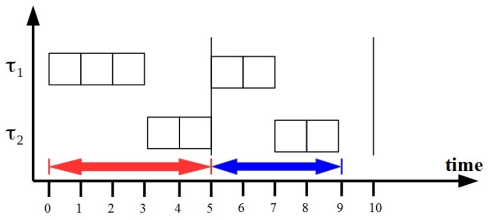


Fig. 3: **The finish time jitter caused by the variance of interference time**

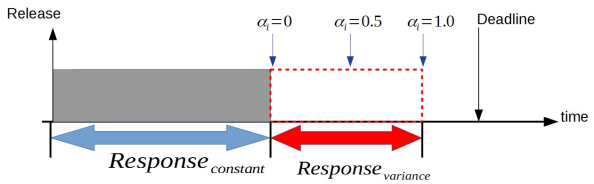


Fig. 4: **The target response time from the perspective of Jitter Margin**

D. Frequency-Updated Ratio Calculation

The next step is to calculate the frequency-updated ratio (frequency-scaling factor) that still makes the system meet its given timing constraint.

*1) **Assignment of Target Response Time**:* First, we give every jitter-sensitive task $\boldsymbol{\tau_i^{jitter}}$ (not tolerating large finish time jitter) an ideal guideline called **target response time** $\boldsymbol{R_i^{target}}$. Once the DVFS operation is invoked, the system starts calculating the frequency-updated ratio to make actual response time gets closer to the target response time. We propose two types of target response times from two different perspectives, user-specified and profile-based target response times.

- **User-Specified Target Response Time**
  According to the definition of **jitter margin** described in Section II-B, every jitter-sensitive task is given one target response time ratio $\boldsymbol{\alpha_i}$ ranging from 0 to 1 (or 0% - 100%) by user in advance. Hence the target response time is given by equation 4.

$$R_i^{target} = BCRT_i + \alpha_i \cdot (WCRT_i - BCRT_i) \quad (4)$$

  $\boldsymbol{\alpha_i}$ limits the jitter margin within low and upper bounds. An example of α_i is depicted in Figure 4.
- **Profile-Based Target Response Time**
  The system performs one procedure called dynamic assignment of target response time during runtime. It decides a target response time by referring to the profiling information as well as estimating the **currently expected response time** given by the following equation.

$$R_i^{expect} = time_i^{executed} + \frac{RWCEC_i}{f_{current}} + I^{actual}(i) \quad (5)$$

  In the above equation, $\boldsymbol{time_i^{executed}}$ is the total amount of time spent for executing τ_i. The obtained R_i^{expect} is compared with R_i^{min} and R_i^{max}. There are two cases for DVFS operation. The first case is that DVFS operation is not performed when $\boldsymbol{R_i^{min} \leq R_i^{expect} \leq R_i^{max}}$. In this case, the response time of the current instance will not increase

the finish time jitter even if the system keeps the current operating frequency $f_{current}$. Hence R_i^{target} does not need to be considered. Otherwise the target response time is assigned as follows.

$$R_i^{target} = \begin{cases} R_i^{min} & (R_i^{expect} < R_i^{min}) \\ R_i^{max} & (R_i^{expect} > R_i^{max}) \end{cases} \quad (6)$$

*2) **Ideal Operating Frequency**:* In order to get an ideal operating frequency at a frequency-scaling point, the system has to know the available time before R_i^{target} expires and the remaining worst-case execution cycles ($RWCEC_i$) which τ_i is supposed to spend from the current time. The ideal operating frequency is calculated by the following equation.

$$f_{idea} = \frac{RWCEC_i}{R_i^{target} - time_i^{executed} - I^{actual}(i)} \quad (7)$$

In this equation, $\boldsymbol{R_i^{target} - time_i^{executed} - I^{actual}(i)}$ represents the **available time** for task τ_i at the considered frequency-scaling point. The available time is substantially subject to the length of interference time $I^{actual}(i)$ from higher-priority tasks.

*3) **Discrete Bound Handling**:* The **ideal operating frequency** assumes that the system can use continuous frequencies from one to infinity. However it is impossible in practical processors which can operate only with a limited number of discrete operating frequencies(from maximal frequency f_{min} to minimal frequency f_{max}). Therefore, the obtained ideal operating frequency needs to be converted to one of those practical frequencies in the target processor model. We assume the set of practical frequencies $\boldsymbol{F^{discrete}} = \{\boldsymbol{f_1}, \boldsymbol{f_2}, ..., \boldsymbol{f_n}\}$ where f_1 and f_n are f_{min} and f_{max}, respectively. The frequency conversion is described as follows.

$$f_{new} = \begin{cases} f_{min} & (f_{ideal} \leq f_0) \\ f_{a+1} & (f_a < f_{ideal} \leq f_{a+1}) \\ f_{max} & (f_{ideal} \geq f_n) \end{cases} \quad (8)$$

If f_{ideal} is between f_a and f_{a+1}, f_{a+1} is chosen as the updated frequency f_{new} in order to avoid deadline misses.

IV. Evaluation

A. Experimental Setup

We built a CFG-based multitasking simulator for evaluating jitter reduction by the proposed approach. CFGs of target tasks are input with mining tables, processor model (DVFS settings), and lists of frequency-scaling points. The simulation is performed on a tasks' CFGs basis, where execution cycles of traversed basic blocks are counted.

We use five benchmark programs. Four of them are from [11], e.g., **bs.c, compress.c, matmul.c, and ludcmp.c**, and the other one is a simple case study's CFG (**cfg_1**) which we prepared. Each program is executed as as a periodic task in the simulation where rate monotonic (RM) scheduling is applied. The tool in [11] is used to obtain CFGs of the programs, execution cycles through each execution path and the worst-case execution path (WCEP). These five tasks' models are shown in Table II. In the table, the number of frequency-scaling points is obtained from the technique described in Section III-C.

TABLE II: **The features of target tasks**

Task	# Basic Block	# Scaling Point	WCEC (cycle)
bs	10	1	9750
compress	11	3	11950
matmul	23	6	1890395
ludcmp	46	13	27546
cfg_1	9	2	1810

We use the frequency settings of Texas Instruments Sitara AM335x processor in which the running clock frequency is set to 300, 600, 720, 800, or 1000 MHz [6]. To reflect runtime variation in executions of the target tasks, we built and used a **test pattern generator** which, for each task, randomly generates fifty execution paths (including loops with randomly chosen $Iter^j_{actual}$) to be traversed. When the simulator starts executing a task instance, it randomly picks one of the fifty generated paths. Two task sets which contain the five tasks are prepared as shown in Table III. WCET (ns) of each task is the total execution time calculated by $WCET = \frac{WCEC}{f_{max}}$. Each period (=deadline) is randomly obtained with exponential distribution and total system utilisation less than the RM's schedulability bound, $\boldsymbol{N \times (2^{1/N} - 1)}$ where N is the number of tasks.

TABLE III: **Two task sets**

(a) Task Set 1

Task	WCET (ns)	Period (ns)	Deadline (ns)	Priority
bs	9750	75582	75582	0
compress	11950	173189	173189	3
cfg_1	1810	164546	164546	2
matmul	1890395	9110699	9110699	4
ludcmp	27546	84239	84239	1

(b) Task Set 2

Task	WCET (ns)	Period (ns)	Deadline (ns)	Priority
bs	9750	162500	162500	3
compress	11950	35949	35949	0
cfg_1	1810	35951	35951	1
matmul	1890395	37807900	37807900	4
ludcmp	27546	121349	121349	2

TABLE IV: **The sets of jitter-sensitivity tasks**

Set	Jitter-sensitive Tasks for Task Set 1	*Set*	Jitter-sensitive Tasks for Task Set 2
1	(bs, comp.)	1	(comp.,cfg_1)
2	(bs,comp.,cfg_1)	2	(bs,comp.,cfg_1)
3	(bs,comp.,cfg_1,ludcmp)	3	(comp.,cfg_1)
4	(bs,cfg_1,ludcmp)	4	(bs,cfg_1)
5	(comp.,cfg_1)	5	(bs,comp.,cfg_1,ludcmp)

Furthermore, we prepare five sets of jitter-sensitivity tasks for each task set in Table IV. Finally the total

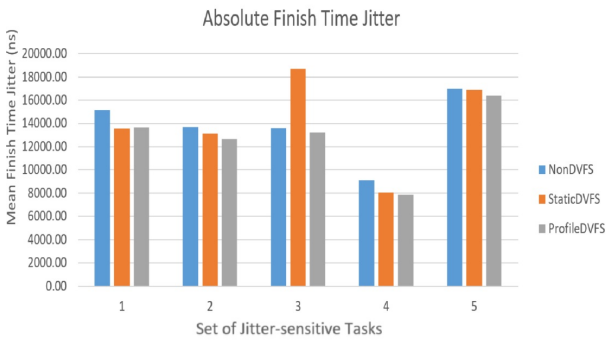


Fig. 5: **Finish time jitter of task set 1**

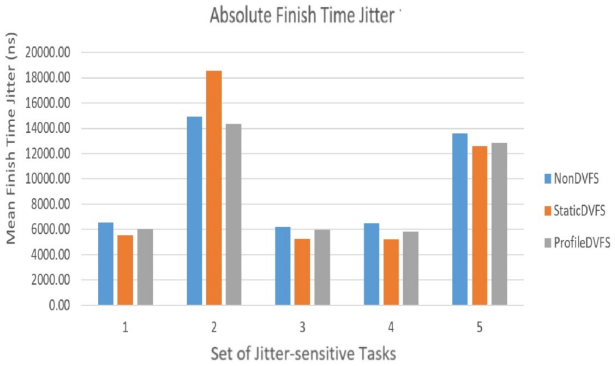


Fig. 6: **Finish time jitter of task set 2**

ten task sets including different combinations of jitter-sensitive tasks are evaluated.

In the experiments, The following three settings are compared: 1) a system without DVFS (with fixed f_{max}) (called NonDVFS, 2) a system with DVFS using the user-specified target response times for jitter-sensitive tasks (called StaticDVFS), and 3) a system with DVFS using the profile-based target response times for jitter-sensitive tasks (called ProfileDVFS).

Each target task set is simulated five times with different execution paths generated by the test pattern generator, and the average value of absolute finish time jitter of the jitter-sensitive tasks are used on in the comparison.

B. *Experimental Results*

Figure 5 and 6 show the results in terms of absolute finish time jitter for.Task Set 1 and 2, respectively. From Figure 5, it is clear that StaticDVFS and ProfileDVFS can reduce jitter by **16.8**% and **16.2**% at maximum compared to nonDVFS. Similarly, from Figure 6, StaticDVFS and ProfileDVFS reduce jitter by up to **19.4**% and **9.7**%, respectively.

V. Conclusion

This paper proposed jitter-aware Intra-task DVFS techniques for reducing jitter in hard real-time systems. We exploited DVFS technique to reduce runtime variation in both interference and execution time, with the cooperation of control and data flow analysis. To decide effective frequency-scaling factor at every DVFS operation, a jitter margin was defined to clarify the lower and upper bounds of possible finish time jitter, also four control parameters were prepared for profiling runtime situation manipulated by system. Through our simulation, it was shown that jitter can be reduced by 16.2% to 19.4%.

Currently, our ongoing work is trying to find a tradeoff between jitter and energy. Different power profiles are being mapped to the frequency settings used in this paper. Thorough assessment under various jitter and energy constraints are to be considered as our future extension. Together with the currently overlooked switching overhead, which could possibly limit the number of frequency-scaling points.

References

[1] Alireza Salami Abyaneh and Mehdi Kargahi. Energy-efficient scheduling for stability-guaranteed embedded control systems. In *Real-Time and Embedded Systems and Technologies (RTEST), 2015 CSI Symposium on*, pages 1–8. IEEE, 2015.

[2] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: an open toolbox for adaptive wcet analysis. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010.

[3] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[4] Anton Cervin, Bo Lincoln, Johan Eker, Karl-Erik Arzén, and Giorgio Buttazzo. The jitter margin and its application in the design of real-time control systems. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*, pages 1–9. Gothenburg, Sweden, 2004.

[5] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 8 of *International Workshop on Worst-Case Execution Time Analysis*, page 12, Dubrovnik, Croatia, Jun 2017.

[6] Texas Instruments. AM335x Power Consumption Summary. http://processors.wiki.ti.com/index.php/AM335x_Power_Consumption_Summary, 2016. [Online; accessed 19-July-2008].

[7] Taewoong Kim, Heonshik Shin, and Naehyuck Chang. Deadline assignment to reduce output jitter of real-time tasks. *IFAC Proceedings Volumes*, 33(30):51–56, 2000.

[8] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3):56–67, 2007.

[9] Bren Mochocki, Razvan Racu, and Rolf Ernst. Dynamic voltage scaling for the schedulability of jitter-constrained real-time embedded systems. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 446–449. IEEE Computer Society, 2005.

[10] Teera Phatrapornnant and Michael J Pont. Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling. *IEEE Transactions on Computers*, 55(2):113–124, 2006.

[11] D. Pinheiro, R. Goncalves, E. Valentin, H. d. Oliveira, and R. Barreto. Inserting dvfs code in hard real-time system tasks. In *2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 23–30, Nov 2017.

[12] Dongkun Shin and Jihong Kim. Optimizing intratask voltage scheduling using profile and data-flow information. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):369–385, 2007.

[13] Tomohiro Tatematsu, Hideki Takase, Gang Zeng, Hiroyuki Tomiyama, and Hiroaki Takada. Checkpoint extraction using execution traces for intra-task dvfs in embedded systems. In *Electronic Design, Test and Application (DELTA), 2011 Sixth IEEE International Symposium on*, pages 19–24. IEEE, 2011.

[14] Burt Walsh, Robert Van Engelen, Kyle Gallivan, Johnnie Birch, and Yixin Shou. Parametric intra-task dynamic voltage scheduling. In *Proceedings of the Workshop on Compilers and Operating Systems for Lower Power (COLP 2003)*, 2003.

Examining and Supporting Multi-Tasking in EV3OSEK

Nils Hölscher, Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen
Department of Informatics, TU Dortmund University, Germany
{nils.hoelscher, kuan-hsun.chen, georg.von-der-brueggen, jian-jia.chen}@tu-dortmund.de

***Abstract*—Lego Mindstorms Robots are a popular platform for graduate level researches and college education purposes. As a portation of nxtOSEK, an OSEK standard compatible real-time operation system, EV3OSEK inherits the advantages of nxtOSEK for experiments on EV3, the latest generation of Mindstorms robots. Unfortunately, the current version of EV3OSEK still has some serious errors. In this work we address task preemption, a common feature desired in every RTOS. We reveal the errors in the current version and propose corresponding solutions for EV3OSEK that fix the errors in the IRQ-Handler and the task dispatching properly, thus enabling real multi-tasking on EV3OSEK. Our verifications show that the current design flaws are solved. Along with this work, we suggest that researchers who performed experiments on nxtOSEK should carefully examine if the flaws presented in this paper affect their results.**

I. Introduction

Since 1998 Lego Inc. released a series of programmable robotics kits called Mindstorms [8], which have been extensively used in graduate level researches and college education. For the Lego Mindstorms robots of the NXT series, the OSEK standard [11] compatible real-time operating system (RTOS) nxtOSEK [4] has been widely adopted as an experimental platform [1, 3, 14]. However, EV3 as the latest generation of Mindstorms robots, released in 2013, is still not popularly used in the real-time community. One reason is that the only RTOSs for EV3 robots, namely EV3RT [9] and EV3OSEK [12], were release a few years after the EV3 robots, i.e., in 2016. In this paper we only focus on EV3OSEK, since it is the only RTOS for EV3 aiming at supporting the OSEK standard.

EV3OSEK is a porting of nxtOSEK to the EV3 platform, provided by a group at Westsächsische Hochschule Zwickau [5]. Hence, it is generally compatible to applications for nxtOSEK. Instead of using the limited sized display to capture the results, the output of EV3OSEK can be obtained directly via the EV3 Console [10] on a host machine. Moreover, unlike nxtOSEK that needs to flash the ROM on the brick, EV3OSEK can directly boot from a SD-Card.

During our experiments with EV3OSEK, we noticed that the task preemption mechanism did not function as expected. Gupta and Doshi [6] described similar problems after implementing nested task preemption in nxtOSEK and abandoned the project due to problems with the IRQ-Handler and dispatch routines. This motivated us to investigate if the problems were related. In course of this investigation, we discovered that EV3OSEK was unable to correctly restart preempted jobs but instead reexecuted them completely. A more detailed description of the preemption behaviour of EV3OSEK as well as of nxtOSEK can be found in Section III. We encourage researchers who performed experiments on nxtOSEK to carefully examine if the flaws presented in this paper affect their results.

To narrow down the source of the problem, we examined the ARM specifications, the hardware dependent IRQ-Handler, and the task dispatching routines. In this work, we provide the corresponding solutions to the errors in the current EV3OSEK, which are released on [7]. After solving these problems, EV3OSEK is now able to provide preemptive scheduling, and therefore multi-tasking, with all the advantages inherited from nxtOSEK.

Our Contributions: This paper presents the errors that exist in the current version of EV3OSEK when task preemption takes place and provides a solution to tackle these problems.

- We detail a flawed behaviour regarding task preemption in EV3OSEK in Section III, and explain the origin of these problems in Section IV.
- The corresponding solutions for the IRQ-Handler and the task dispatching routine are provided in Section V, hence enabling multi-tasking under EV3OSEK.
- We evaluated our solutions, the results are displayed in in Section VI, showing that the provided solutions solve the problems and allow fully preemptive fixed-priority scheduling, and therefore multi-tasking, in EV3OSEK.

II. System Model

A. Application Model

We consider the scheduling of n independent periodic real-time tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ in a uniprocessor system. Each task is defined by a tuple $\tau_i = (C_i, T_i)$ where T_i is an interarrival time constraint (or period) and C_i the tasks worst-case execution time. The deadlines is assumed to be implicit, i.e., if a task instance (job) is released at θ_a, it must be finished before $\theta_a + T_i \ \forall \tau_i$. We consider fully preemptive fixed-priority scheduling, i.e., each task τ_i is associated to a predefined priority $p(\tau_i)$[1], since the issues considered in this work only happens under a fully preemptive scheduling policy.

B. Lego Mindstorms EV3 and EV3OSEK

In this paper, we focus on the third generation of Lego Mindstorms robots (EV3), which are equipped with a uniprocessor ARM926EJ-S 300MHz and 64MB RAM on a TexasInstruments AM1808, running EV3OSEK with a C/C++

[1]Although EV3OSEK defines the lowest priority as 0, we use the more common notation that lower priority values indicate higher priorities.

compatible environment. EV3OSEK [12] is a real-time operating system which aims for compatibility to the OSEK standard [11]. It is a recent portation [5] of nxtOSEK [4], which is only available for the older LEGO Mindstorms NXT robots. EV3OSEK consists mainly of three parts:

1) Drivers for sensors and actors (leJOS)
2) API for development (ECRobot)
3) OSEK-OS for the EV3 robot

This work focuses on the OSEK-OS. To obtain the output from the EV3 robots with our host machines the EV3 Console [10] is used, which realizes an USB to UART bridge. It connects with one of the Lego sensor cables and a micro-USB cable. The suggested driver to access the device are provided by Texas Instruments [13].

C. Preemption in the OSEK Standard

Here we briefly review the specifications for task preemption defined by the OSEK standard [11]. The OSEK standard defines two different scheduling policies: non-preemptive scheduling and fully preemptive scheduling. In a non-preemptive scheduling policy, a job cannot be preempted once its execution has been started. In fully preemptive scheduling, any task is preempted at the point in time a higher priority task enters the system and that higher priority task is scheduled instead. The context of the preempted task is stored accordingly so that it can resume back later on.

III. Motivational Example

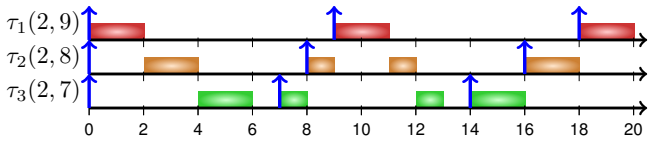
To demonstrate the flaws in the current EV3OSEK, Figure 1 provides and example that detail the EV3OSEK preemption behaviour. We consider three tasks: $\tau_1 = (2, 9)$, $\tau_2 = (2, 8)$, and $\tau_3 = (2, 7)$, indexed according to their priority, i.e., $p(\tau_1) > p(\tau_2) > p(\tau_3)$.

Figure 1a shows the expected behaviour. The second job of τ_3 released at time 7 is preempted by the second job of τ_2 released at time 8, which afterwards is preempted by the release of τ_1 at time 9, and both τ_2 and τ_3 have one unit of execution time left. After τ_1 finishes its execution, the remaining portions of τ_2 and τ_3 are executed. Note that in the original EV3OSEK also the problem occurs that not all tasks are activated at time 0, i.e., the first release of τ_1 was missing due to an index error. The array containing the tasks/alarms was read starting at index 1. In our code, we ensured a start at 0, hence the first job of τ_1 is released as well.

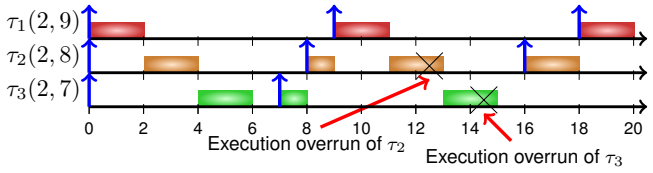
In contrast, Figure 1b shows the execution behaviour of EV3OSEK.[2] Both the second job of τ_2 and the second job of τ_3 are not resumed correctly but either resumed wrongly or completely restarted which leads to one additional unit of execution time for both jobs, called overrun in Figure 1b. Note that, due to the deadline miss at 14, the third release of τ_3 at 14 is skipped and the next job of τ_3 will be released at 21.

Since EV3OSEK is a portation from nxtOSEK, this behaviour could directly be inherited. However, the flawed behaviour in the original nxtOSEK was different and only

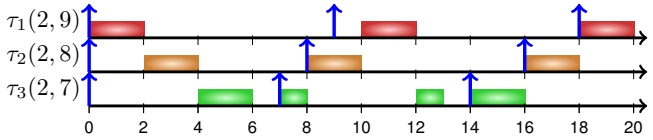
[2]The related source code is released on [2] as `NestPreemption`.



(a) **Expected behaviour:** τ_2 preempts τ_3 and is afterwards preempted by τ_1. The jobs of τ_2 and τ_3 are resumed where they were preempted.



(b) **Observed behaviour in EV3OSEK**: the jobs of τ_2 and τ_3 are restarted instead of resumed after a preemption.



(c) **Observed behaviour in nxtOSEK:** τ_3 is preempted by τ_2, but τ_2 cannot be preempted by τ_1.

Fig. 1: Expected behaviour compared to the actual behaviour of EV3OSEK and nxtOSEK.

effected nested task preemption as displayed in Figure 1c. Once τ_3 is preempted by τ_2 at time 8, the interrupt from the scheduler is deactivated and hence τ_1 cannot preempt τ_2 at time 9 although $p(\tau_1) > p(\tau_2)$. Only when τ_2 finishes at time 10, τ_1 is allocated to the processor. However, when Gupta and Doshi [6] tried to fix this problem, their efforts resulted in an identical behaviour as in Figure 1b due to the already existing problems with the IRQ-Handler and the task dispatching.

Overall, the current EV3OSEK does not match the expectation when resuming previously preempted tasks. Since the misbehavior is observed right after the preempting task finishes, e.g., τ_1, this motivated us to check the functions responsible for the IRQ-Handler and the task dispatching. It turned out that the IRQ handler, expended from TexasInstruments [13], has critical errors that could have lead to complete corruption of the program counter.

IV. Original Task Preemption in EV3OSEK

In this section, we first review the current design of the functions that are responsible for IRQ-Handler[3] and task dispatching in EV3OSEK[4]. Afterwards we point out the source of the aforementioned errors.

A. IRQ-Handler

To follow the OSEK standard, EV3OSEK has a hook routine named `user_1ms_isr_type2()`, which is invoked

[3]IRQ stands for Interrupt ReQuest from the underlying hardware.

[4]The reviewed files are downloaded from https://github.com/ev3osek/ev3osek/tree/master/OSEK_EV3. The latest update for exceptionhandler.S and cpu_support.S was on 18 Sep 2016.

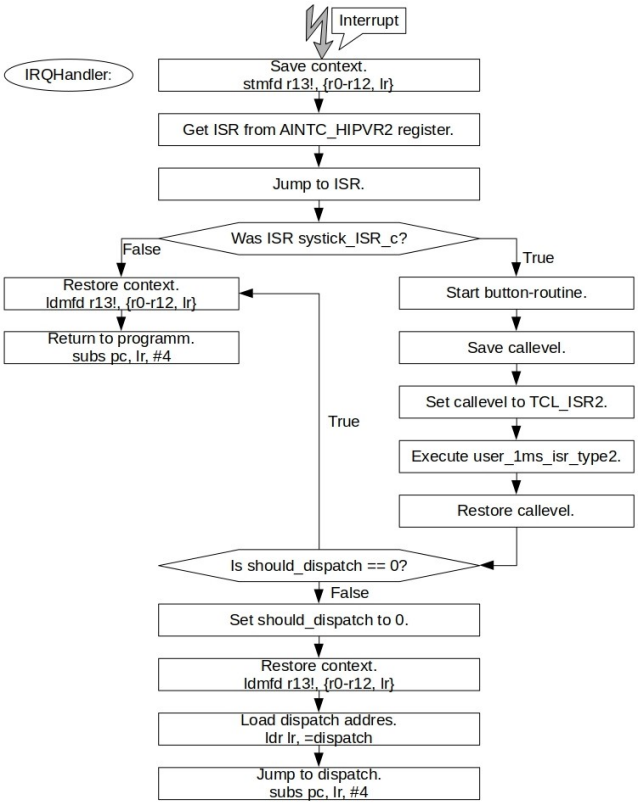


Fig. 2: Flowchart of the current IRQ-Handler in EV3OSEK.

from a periodic interrupt service routine in category 2 (ISR2) every 1 ms. This hook routine can be redefined by the programmer but it should always execute the system routine `SignalCounter()` to maintain the progress of EV3OSEK. However this design partially violates the OSEK standard.

Once an ISR occurs, the CPU loads the IRQ-Handler shown in Figure 2. It first saves the context of the interrupted task. Now it can handle the ISR without overriding registers of the interrupted task. The address of the ISR that called the interrupt is saved in the `AINTC_HIPVR2` register by the hardware interrupt handler. When the ISR has finished its execution, it returns back to the IRQ-Handler.

If the ISR was not `systick_ISR_c`, i.e., the function that handles the 1ms timer, the task context is restored and the IRQ-Handler returns to the interrupted task. But if the ISR was `systick_ISR_c`, the button-routine and `user_1ms_isr_type2()` are executed. In the hook function `user_1ms_isr_type2()`, `SignalCounter()` will set the Boolean `addr_should_dispatch` to TRUE if the current running task is not the highest priority task anymore.

In case that `should_dispatch` is false, the task context is restored and the IRQ-Handler returns to the interrupted task. In the other case, when `should_dispatch` is set to true, the task context is restored, i.e., all registers $r0$ to $r12$ and the lookup register. Afterwards the IRQ-Handler loads the dispatch routine address in the lookup register and loads it with an offset of -4.

Within the analyses, we noticed that there are five errors in the current implementation as shown in Listing 1:

1) The lookup register contains the return address of the preempted task and is always overwritten.
2) The lookup register has to be saved in the stack for the CPU User-/System-mode before jumping to the dispatch routine, since different CPU modes may have their own lookup registers.
3) The lookup register, which already contains the address of the dispatch routine, loads with an offset of -4. This is not necessary, since the address is loaded from the memory instead of the decoder.
4) The status register also has to be saved/restored, when interrupting a task, since it also contains information about the interrupted task.
5) `SignalCounter()` in ISR2 determines whether the task dispatching should take place or not. However, the OSEK standard defines that scheduling should be bound to ISR2 rather than `SignalCounter()`.

```
LDMFD    r13!, {r0-r12, lr}

LDR      lr, =dispatch
SUBS     pc, lr, #4
```

Listing 1: Assembler code fragment responsible for the five errors related to the IRQ-Handler.

B. Task Dispatching

Before introducing the current design of task dispatching in EV3OSEK, we list some notations used in the implementation:

- `runtsk`: Address of the running task ID.
- `schedtsk`: Address of the highest priority task.
- `tcxb_pc[]`: Array for the program counters of tasks.
- `tcxb_sp[]`: Array for the stack addresses of tasks.

For the simplicity of the presentation, we further use τ_{low} and τ_{high} in the rest of the section to describe the scenario that there is an executing task τ_{low} which is going to be preempted by a ready task τ_{high} with higher priority.

When τ_{high} is ready in EV3OSEK, the currently running task τ_{low} has to relinquish its right on the CPU. As shown in Figure 3, the scheduler in EV3OSEK has three main steps: *Dispatch*, *Preempt* and *Reload*, detailed as follows:

- **Dispatch:** To preempt a task, the IRQ-Handler calls the dispatch routine, which saves the context of the preempted task on the tasks stack, and stores the stack pointer in `tcxb_sp[runtsk]`. The address of `dispatch_r` is stored in `tcxb_pc[runtsk]`, allowing the task context to be restored when it is resumed.
- **Preempt:** After the dispatch step, the higher priority task is executed on the CPU. Once it finishes, it calls `TerminateTask()` to trigger the scheduler with `start_dispatch` to reload the lower priority task. In `start_dispatch`, at first `runtsk` is set to

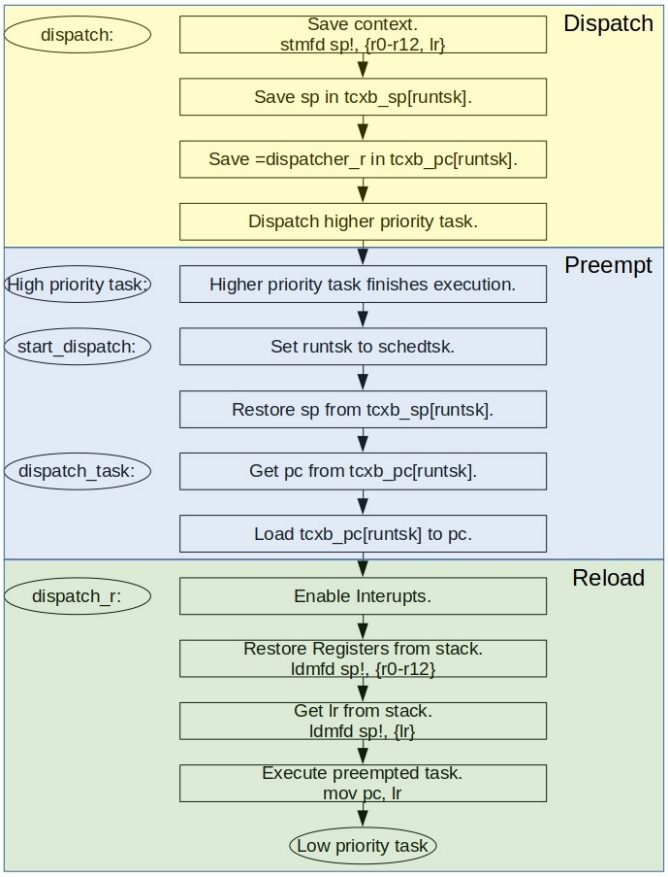


Fig. 3: Task dispatching and re-dispatching.

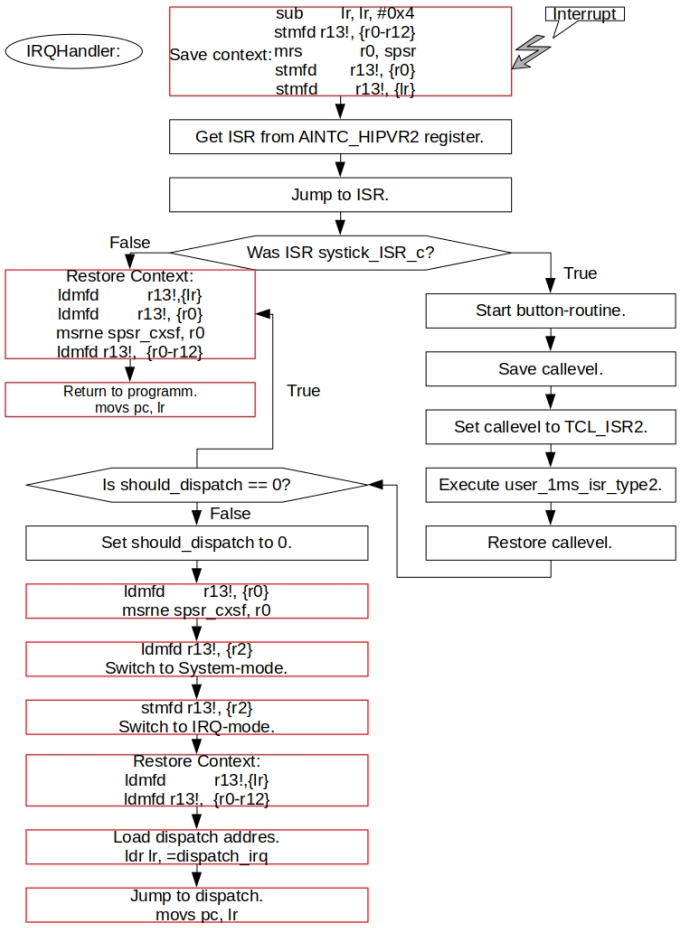


Fig. 4: Enhanced version of the IRQ-Handler.

`schedtsk`, so that the scheduler knows that the current running task is the currently highest priority task in the system. Afterwards, the stack pointer is restored back from `tcxb_sp[runtsk]` and `dispatch_task` is called.

- **Reload:** In `dispatch_task` the program counter of the preempted task is restored from `tcxb_pc[runtsk]`. Instead of loading the tasks program counter, the preempted task executes `dispatch_r` to restore its context from the stack and enable interrupts, which were disabled by `TerminateTask()`.

There are two errors in the current implementation:

1) In `dispatch_r`, the lookup register is loaded from the stack without ^ flag, and the status bits are not loaded as well. See Listing 2:

```
dispatcher_r:
        BL      IntMasterIRQEnable
        BL      IntMasterFIQEnable
        ldmfd   sp!, {r0-r12}
        ldmfd   sp!, {lr}
        MOV     pc, lr
```

Listing 2: The lookup register is loaded without ^ flag, the status bits are not loaded at all.

2) The status register has to be part of the save context routine in `dispatch` and of the restore context routine in `dispatch_r`.

V. Fixing Task Preemption in EV3OSEK

After discussing the flaws in the current EV3OSEK, we here present how we fix the task preemption accordingly. Please note that EV3OSEK's IRQ-Handler is not inherited from the portation of nxtOSEK and hence the nested task preemption problems in nxtOSEK are not inherited from the IRQ-Handler but the dispatch routines.

Based on the observations in Section IV, the proposed solutions can be summarized as follows:

- correcting the register operations in the IRQ-Handler,
- correcting the errors in `dispatch_r`,
- adding status register to context save/restore routines, and
- changing the trigger point of the task dispatching.

The flowcharts for the IRQ-Handler and the dispatching are shown in Figure 4 and Figure 5, respectively, where the red blocks are added or changed due to our solutions. In the rest of this section, we explain more details about our solutions.

Correcting the register operations in the IRQ-Handler: In the current EV3OSEK, the lookup register in the IRQ-Handler

contains the address of the preempted task and is overwritten. Moreover, the lookup register has to be saved in the User-/System-mode stack before jumping to dispatch, since IRQ- and User-/System-mode have their own lookup registers. Note that there are different execution modes in modern CPUs, where some modes have their own registers called banked registers which are not shared with other modes.

The errors can be solved by writing the lookup register in one of the registers r0-r12, switching to System-mode in the IRQ-Handler, pushing the register containing the lookup register on the System-mode stack, and switching back. This solution requires to remove the instruction that stores the lookup register on the system stack in the dispatch routine. As a result, the dispatch routine can no longer be called from User-/System-mode. To resolve this, the branch `dispatch_irq` is introduced right after the dispatch routine stores the lookup register, as this is already done in the IRQ-Handler. Now the IRQ-Handler calls `dispatch_irq` and it is still possible to call the dispatch routine from User-/System-mode.

Another error in the IRQ-Handler is that the lookup register contains the address of the dispatch routine, but it is loaded with an offset of -4. This can be easily fixed by removing the unnecessary offset from the branch instruction. The updated IRQ-Handler is displayed in Figure 4.

Correcting the errors in `dispatch_r`: As shown in Figure 5, the lookup register is loaded from the stack without the ˆ flag in `dispatch_r`, so that the status bits are not loaded as well. This can be easily resolved by adding the ˆ flag to the load instruction. By doing so, the program status is loaded into the status register correctly. The enhanced dispatching is detailed in the flowchart in Figure 5.

Save/Restore status register with context: In the IRQ-Handler and dispatch routines, the status register is not part of saving/restoring context. However the status register contains information about comparing instructions for the interrupted/dispatched task. By saving and restoring the status register together with the context of registers, the informations in $r0$ to $r12$ are not lost.

Changing the trigger point of the task dispatching: In the original implementation, `SignalCounter()` must be called by the hook routine `user_1ms_isr_type2()`, which is used to manage task scheduling. As defined in the OSEK standard, the task scheduling must be bound to ISR2. To fix this, we moved the code setting the flag `should_dispatch` to the function `SetDispatch()` and call it after `user_1ms_isr_type2()` has finished.

VI. Evaluation of the Proposed Solution

As illustrated in Section III, the current EV3OSEK is not able to provide task preemption correctly. With the enhancement mentioned in the previous section, task preemption, and hence multi-tasking, now should work properly. We present an additional example with three tasks to evaluate our proposed

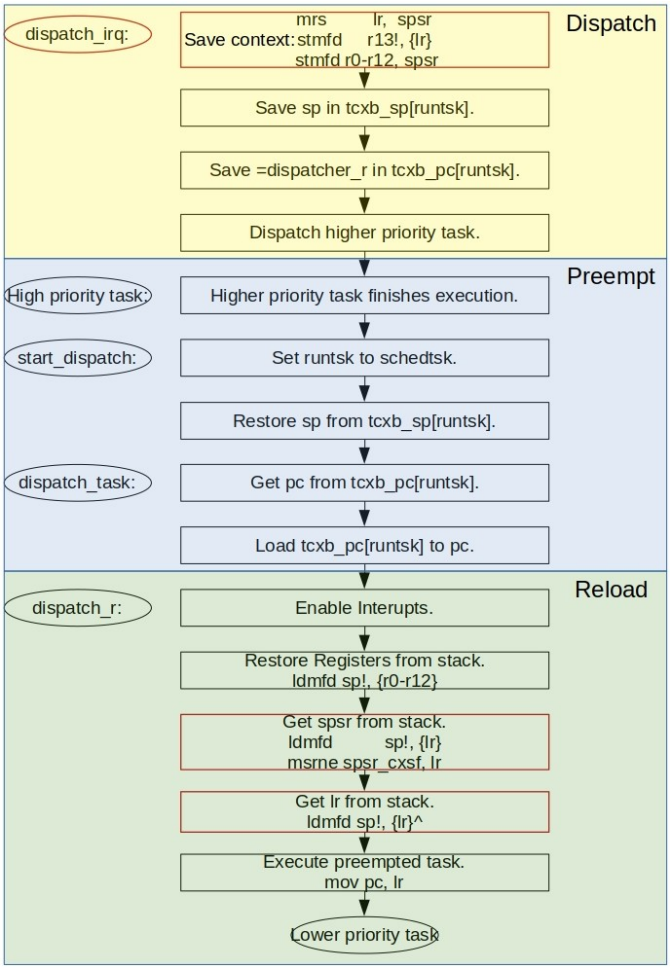


Fig. 5: Enhanced version of task dispatching.

solution in EV3OSEK[5].

In the following experiment, we considered a task set which is schedulable in a correct preemptive fixed-priority scheduling system while in the current EV3OSEK the unexpected additional workload due to task preemption leads to deadline misses. Once a job misses its deadline, the next job is only released after the current job is finished and hence the number of releases is reduced. Therefore, by checking if the number of jobs released in the current version of EV3OSEK and in our enhanced version of EV3OSEK is identical, we can determine whether our enhancement solved the discovered problem. The related source code can be found at [7].

Tasks τ_1, τ_2, and τ_3 all print out the following line right after it starts/finishes: "Task $\tau_i(l_1, l_2, l_3)$ starts/ends at t_{ms}". t_{ms} stands for the time point when a task starts or finishes its execution. τ_1, τ_2, and τ_3 all run roughly $2000\,\mathrm{ms}$ and priority's are $p(\tau_1) > p(\tau_2) > p(\tau_3)$. The tasks are released as follows:

- τ_1 releases at $0\,\mathrm{s}$ with a period $5\,\mathrm{s}$.
- τ_2 releases at $0\,\mathrm{s}$ with a period $8\,\mathrm{s}$.

[5]Please note that testing the nesting depth is not necessary. As the task stack for context-switch is managed in the OIL file, the management of the stack should be handled by the programmers.

- τ_3 releases at $0\,\mathrm{s}$ with a period $10\,\mathrm{s}$.

We verified that all the task preemptions behave as we expect over a certain amount of time, checking the resulting log file, and if the number of jobs for each task is exactly as we predict in advance. If there is no additional execution time after preemptions (like in the current EV3OSEK), there should be no unexpected interference affecting the job releases. We also intend to show that the program counter does not get corrupted any more, even after long run times, i.e., $10\,\mathrm{min}$.

We first derived an equation to predict the exact number of jobs l_i after a certain amount of time that is a multiple of 10 seconds. Since the least common multiple of three tasks' periods is 40 seconds, the so-called hyper-period, the following equation gives us the number of jobs from τ_i in a $10 \times t$ second long interval:

$$\begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix} = \begin{pmatrix} \frac{8}{4} \times t \\ \frac{5}{4} \times t \\ \frac{4}{4} \times t \end{pmatrix} = \begin{pmatrix} 2t \\ 1.25t \\ t \end{pmatrix} \tag{1}$$

The equation is detailed as follows:

- l_1 equals $2t$: τ_1 is released and finishes two times in $10\,\mathrm{s}$.
- l_2 is $1.25t$: τ_2 releases and finishes 5 times in a hyper-period of 40, every $10\,\mathrm{s}$ it has on average $1.25t$ releases.
- l_3 is t: τ_3 has one release every 10 seconds.

We can now predict l_1, l_2 and l_3 after an interval of $10\,\mathrm{min}$.

$$t = 600001(ms) \approx 60 \times 10sec \Rightarrow \begin{pmatrix} l_1(60) = 120 \\ l_2(60) = 75 \\ l_3(60) = 60 \end{pmatrix} \tag{2}$$

In the current version of EV3OSEK the example hangs after $7000\,\mathrm{ms}$, because the program counter is set to a random address. With our enhancement, the aforementioned problem does not exist anymore in the enhanced version of EV3OSEK. The output can be found at listing 3.

```
Task 1(0, 0, 0) start at 1.
Task 1(1, 0, 0) end at 2005.
Task 2(1, 0, 0) start at 2008.
Task 2(1, 1, 0) end at 4003.
Task 3(1, 1, 0) start at 4005.
Task 1(1, 1, 1) start at 5001.
Task 1(2, 1, 1) end at 6995.
...
Task 1(120, 75, 60) start at 600001.
```

Listing 3: Output generated with the evaluation example using the enhanced of EV3OSEK.

Hence, we conclude that our enhancement fixed the problems in EV3OSEK regarding task preemption which not only resulted in unexpected execution behaviour but also in system crashes.

VII. Conclusion

EV3OSEK as an OSEK inspired real-time operating system for the third generation of LEGO Mindstorms robots (EV3) has many benefits in graduate level researches and college education. In this work, we explain how we have fixed the IRQ handler and the task-dispatcher for the current version of EV3OSEK to achieve a generally expected task preemption feature. Consequently, the proposed solution fixes multi-tasking in EV3OSEK. The release source code of our enhancement can be found in [7].

Acknowledgments

This paper has been supported by DFG, as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/), subproject A1.

References

[1] M. Canale and S. C. Brunet. A Lego Mindstorms NXT experiment for Model Predictive Control education. In *2013 European Control Conference (ECC)*, pages 2549–2554, July 2013.

[2] K.-H. Chen. Motivational Examples for the flaws in EV3OSEK. https://github.com/kuanhsunchen/ev3osek/tree/master/example, 2017.

[3] K.-H. Chen, B. Bönninghoff, J.-J. Chen, and P. Marwedel. Compensate or ignore? meeting control robustness requirements through adaptive soft-error handling. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, LCTES 2016, pages 82–91, New York, NY, USA. ACM.

[4] T. Chikamasa. nxtOSEK. http://lejos-osek.sourceforge.net/, 2013.

[5] F. Grimm. Portierung des nxtOSEK-Frameworks auf die Lego EV3 Plattform, February 2016.

[6] S. Gupta and J. Doshi. Support for Nested Preemption in nxtOSEK. http://moss.csc.ncsu.edu/~mueller/rt/rt14/projects/p1/report4.pdf, 2014.

[7] N. Hölscher and K.-H. Chen. Enhanced ev3osek. https://github.com/kuanhsunchen/ev3osek, 2018.

[8] Lego Inc. Lego mindstorms. http://www.lego.com/en-us/mindstorms/.

[9] Y. Li, T. Ishikawa, Y. Matsubara, and H. Takada. A Platform for LEGO Mindstorms EV3 Based on an RTOS with MMU Support. In *Operating Systems Platforms for Embedded Real-Time Applications, OSPERT*, 2014.

[10] Mindsensors. Console Adapter for EV3. http://http://www.mindsensors.com/ev3-and-nxt/40-console-adapter-for-ev3, 2017.

[11] OSEK. OSEK/VDX Operating System Manual. https://www.irisa.fr/alf/downloads/puaut/TPNXT/images/os223.pdf, February 2005.

[12] A. Stuy. EV3OSEK. https://github.com/ev3osek/ev3osek, 2017.

[13] Texas Instruments Inc. AM1808/AM1810 ARM Microprocessor Technical Reference Manual. http://www.ti.com/product/AM1808/technicaldocuments, 2011.

[14] X. Weber, L. Cuvillon, and J. Gangloff. Active Vibration Canceling of a Cable-Driven Parallel Robot in Modal Space. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1599–1604, May 2015.

Levels of Specialization in Real-Time Operating Systems

Björn Fiedler, Gerion Entrup, Christian Dietrich, Daniel Lohmann
Leibniz Universität Hannover
{fiedler, entrup, dietrich, lohmann}@sra.uni-hannover.de

***Abstract*—System software, such as the RTOS, provides no business value on its own. Its utility and sole purpose is to serve an application by fulfilling the software's functional and nonfunctional requirements as efficiently as possible on the employed hardware. As a consequence, every RTOS today provides some means of (static) specialization and tailoring, which also has a long tradition in the general field of system software.**
However, the achievable depth of specialization, the resulting benefits, but also the complexity to reach them differ a lot among systems. In the paper, we provide and discuss a taxonomy for (increasing) levels of specialization as offered by (real-time) system software today and in the future. We argue that system software should be specialized as far as possible – which is always more than you think – but also discuss the obstacles that hinder specialization in practice. Our key point is that a deeper specialization can provide significant benefits, but requires full automation to be viable in practice.

I. Introduction

While the domain of real-time control systems is broad and diverse with respect to both, applications and hardware, each concrete system has typically to serve a very specific purpose. This demands specialization of the underlying system software, the real-time operating system (RTOS) in particular: An "ideal" system software fulfills exactly the application's needs, but no more [19]. Hence, most system software provides built-in static variability: It supports a broad range of application requirements and hardware platforms, but can be specialized at compile-time with respect to a specific use case. Historically, this has led to the notion of system software as *program families* [25], [14] as well as a myriad of papers from the systems community that demonstrate the efficiency gains by specializing kernel abstractions to the employed application, hardware, or both. Examples include [27], [6], [20], [26].

A. System Software Specialization

Specialization (of infrastructure software) for a particular application–hardware setting is a process that aims to improve on nonfunctional properties of the resulting system while leaving the application's specified functional properties intact. If the application employs an RTOS with a specified API and semantics (e.g., POSIX [2], OSEK/AUTOSAR [4], ARINC [3]), a specialized derivative of the RTOS does no longer fulfill this API and semantics in general, but only the *subset* used by *this concrete application* and hardware. If successful, this specialization leads to efficiency gains with respect to memory footprint, hardware utilization, jitter, worst-case latencies, robustness, security and so on; it increases the safety margins or makes it possible to cut per-unit-costs by switching to a cheaper hardware. For price-sensitive domains of mass production, such as automotive, this is of high importance [8].

Intuitively, any kind of specialization requires knowledge about the actual application: The more we know, the better we can specialize. In the domain of real-time systems (RTSs), we typically know *a lot* about our application and its execution semantics on the employed RTOS: To achieve real-time properties, all resources need to be bounded and are scheduled deterministically. Timing analysis depends on the exact specification of inputs and outputs, including their inter-arrival times and deadlines; schedulability analysis requires that all inter-task dependencies are known in advance – and so on.

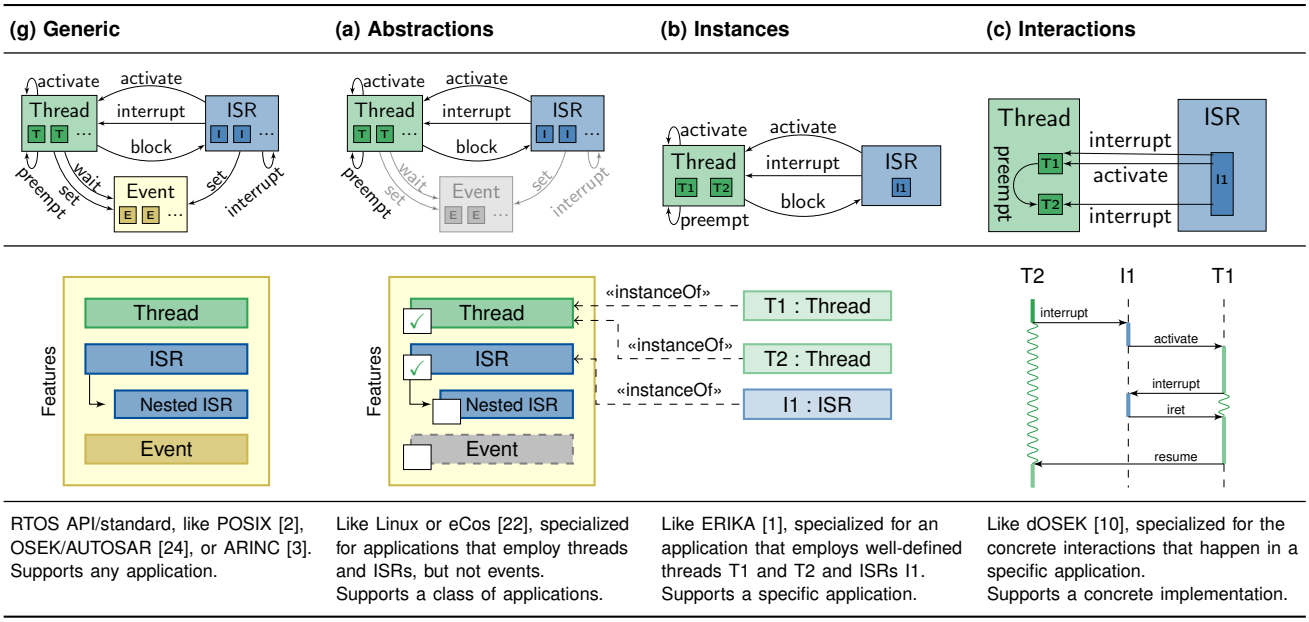
Even though all this knowledge should pave the road to a very rigorous subsetting of the RTOS functionality, this rarely happens in practice. Part of the problem is that the specialization of the RTOS typically has to be performed manually by the application developer or integrator, which adds significant complexity to the overall system development and maintenance process. We are convinced that automation is the key here, as most of the required knowledge could be extracted by tools from the application's code and design documents – the RTOS specialization should become an inherent part of the compilation process, like many other optimizations.

Another part of the problem is, however, that static specialization itself is only rarely understood. This holds in our observation for both, RTOS users and RTOS designers, both of which typically have been educated (and tend to be caught) in the mindset and APIs of general-purpose operating systems, such as Linux or Windows. So while every system software provides *some* means for static specialization and tailoring, the rigorosity at which this (a) could be possible in principle, (b) is possible in the actual RTOS provisioning, and (c) is employable by users in practice, differs a lot.

B. About This Paper

Our goal with this paper is to shed some light on the aspects and the fundamental levels of specialization that are provided by system software today and, maybe, in the future. We claim the following contributions: (1) We provide a classification for specialization capabilities on three increasing levels (Section II). (2) We discuss the challenges and benefits of system specialization by examples from the literature (Section III). (3) We show, on the example of a small experiment with FreeRTOS [5], the potential of different specialization levels, even for an RTOS API that is supposed to "look like POSIX" (Section IV).

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



RTOS API/standard, like POSIX [2], OSEK/AUTOSAR [24], or ARINC [3]. Supports any application.

Like Linux or eCos [22], specialized for applications that employ threads and ISRs, but not events. Supports a class of applications.

Like ERIKA [1], specialized for an application that employs well-defined threads T1 and T2 and ISRs I1. Supports a specific application.

Like dOSEK [10], specialized for the concrete interactions that happen in a specific application. Supports a concrete implementation.

Fig. 1: Levels of RTOS specialization. From left to right, each level further constraints how the application may use the kernel.

Many aspects about specialization we describe in this paper are based on our own experience with the design, development, and employment of highly configurable application-tailorable system software. We apologize for the (shameless) number of self-citations, but felt that leaving them off would not have contributed to the accessibility of the paper.

II. A Taxonomy of Specialization Levels

In this section, we give a taxonomy of system specialization and the different levels specialization can reach. In short, a generic RTOS (g) can be specialized by (a) removing complete abstractions (e.g., threads or a specific syscall), (b) make instances fixed (e.g., there are only threads T1 and T2), and (c) make interactions fixed (e.g., only T1 waits on event E1). We examine these terms at the example of RTSs, which we specify for the purpose for this paper as follows:

A (hard) real-time system *RTS* consumes time-labeled input events $\vec{I}$ and produces observable, time-labeled output events $\vec{O}$, while fulfilling strict timing constraints between both event streams. An implementation $RTS^{\substack{A\\RTOS\\HW}}$ of the abstract *RTS* consists of a concrete application *A* that runs, mediated by a concrete RTOS implementation *RTOS*, on a concrete hardware *HW*. We encapsulate the specification and timing requirements of the *RTS* in an equality operator $\overset{RTS}{=}$ that compares two outputs.

$$RTS(\vec{I}) = \vec{O} \quad \overset{RTS}{=} \quad RTS^{\substack{A\\RTOS\\HW}}(\vec{I})$$

Every *correct* implementation of *RTS* produces an output stream that is equal, under the *RTS* specification, to the outputs of the abstract/ideal *RTS*. Therefore, we derive: Every specialized implementation $RTS^{\substack{A\\RTOS\\HW}}{}'$ has to be a correct implementation of *RTS* and the observable outputs must not change with respect to the specification of the real-time system.

However, not every $RTS^{\substack{A\\RTOS\\HW}}$ is a specialized implementation. Specialization is the process of reducing flexibility from one or more system components of an already existing implementation. For real-time systems, it can take place in the application *A*, the *RTOS*, or/and the hardware *HW*. For the rest of the paper, we focus on the specialization of the RTOS, while application and hardware remain unchanged.

The specialized $RTOS'$ fulfills all requirements of the specific application that runs on top and works on the specified hardware. However, this $RTOS'$ does not necessarily provide the correct semantics to execute an alternative A' or correct instructions to execute on an alternative HW'. Therefore, RTOS specialization always depends on the application that uses the RTOS and the targeted hardware.

In the following we exemplify this by a simple RTOS that supports only three abstractions: Threads, interrupt service routines (ISRs) and Events. Figure 1 (g) shows the whole range of functions provided by our example RTOS as an *interaction graph*. Nodes are system abstractions that are provided by the RTOS standard; edges are interactions between them. The generic RTOS (i.e., the respective standard) provides the illusion that abstractions can be instantiated arbitrarily often and all instances (nodes within nodes) can interact according to their abstraction. For example, every ISR can activate every thread.

When we specialize our generic RTOS, we (a) remove abstractions, (b) make instances fixed, and (c) forbid concrete interactions. The shrunk interaction graph reflects the reduced flexibility of the specialized RTOS. We define three levels of specialization, which subsequently need more information about the actual interaction graph of the application and remove more flexibility. Every level is a true superset of the previous one.

Specialization of Abstractions: remove complete abstractions and types of interactions.

Specialization of Instances: number and identity of instances become fixed; dynamic instantiation is not possible.

Specialization of Interactions: interactions are constrained to concrete instances instead of (generic) abstractions.

The following sections describe the levels in detail and outline the information is needed to reach the respective level. If we specialize a RTOS implementation to a certain level, it only ensures that applications with the corresponding interaction graph are executed correctly. For all other applications, the result is undefined. The effects of the specialization levels (Figure 1 (a)-(c)) are examined using the following example application code:

```
BoundedBuffer bb;

ISR I1 { // priority: ∞
  data = readSerial();
  bb.put(data);
  activate(T1);
}

Thread T1 { // priority: 2
  while(data = bb.get())
    handleSerial(data);
}
Thread T2 { // priority: 1, autostart
  while (true)
    handleADC(readADC());
}
```

The nonpreemptable ISR reads serial data into a bounded buffer, which is handled by the higher-priority worker thread T1. The background thread T2 continuously reads analog data and handles the result. For compactness reasons, we ignored the lost wake-up problem between I1 and T1.

A. Specialization of Abstractions

Specialization on the level of abstractions is the most generic one that is commonly used to select the availability of RTOS features. The needed knowledge to conduct this specialization is confined to the list of used abstractions, which could be derived from code or explicitly listed in a configuration. This kind of specialization is possible in most operating systems. For instance, Linux, eCos and FreeRTOS provide support to be specialized on the level of abstractions. The example application employs only threads and ISRs, while events are not used at all. Therefore, the RTOS specialized on abstractions (Figure 1 (a)) avoids everything event related. Furthermore, we can safely forgo the nesting of ISRs and, therefore, remove the "interrupt" interaction between ISRs.

B. Specialization of Instances

One level deeper, specialization of instances means to specify the concrete instances of each abstraction and their properties. In addition, knowledge about these concrete instances is necessary. For threads, this could be their name, priority, stack size, periodicity and initial activation state. Some RTOS specifications, such as OSEK, already require this information in a configuration file. For others this information may be gathered out of the source code. An instance-level specialized RTOS looses the capability to create system objects at run time. All instances need to be specified statically at compile time.

In an OSEK implementation like ERIKA [1], the OSEK Implementation Language (OIL) file [23] describes all system objects of the application and their properties. For our example application this would be two threads, namely T1 and T2 and one ISR, namely ISR1. The priority of ISR1 is ∞ and T1 and T2 have the priorities 2 and 1. In Figure 1 (b), only the three concrete instances (T1, T2, I1) remain in the interaction graph, while the interactions are still attached to the abstractions.

C. Specialization of Interactions

The most extensive specialization takes place at the level of interactions. Here, we limit the concrete interactions between the system-object instances rather than abstractions. By limiting interactions, we can derive optimized kernel paths, like removing dead code branches (e.g., syscall parameter checking). In essence, we take the viewpoint of an optimizing whole-system compiler that knows the RTOS semantics and thereby could, for instance, derive scheduling decisions already at compile time. To optimize the RTOS on this level, we have to know of all concrete interactions of our applications. This can be done by static code analysis or examination of external-event timing constraints to derive possible invocation sequences.

For our application, we can derive that there is no inter-thread activation, no interrupt blockade, and only T1 can preempt T2. Furthermore, we know that I1 can only activate T1, while it potentially interrupts both threads. This results in Figure 1 (c) contain just these interactions.

D. Summary

In summary, by specialization of the RTOS kernel we remove flexibility from the kernel implementation by restricting the possible run-time interactions of the application already at compile time. This can take place on the (subsequently stricter) levels of *(a) Abstractions*, *(b) Instances*, and *(c) Interactions*, which, in turn, subsequently cut of more from the unneeded RTOS functionality.

III. Specialization: Benefits and Challenges

In our experience, the less-is-more philosophy (i.e., it is a good thing to *reduce* flexibility) tends to be counter-intuitive for many software engineers and in any case it is arguable. In the following, we discuss some benefits and challenges of specialization in general and with respect to the different levels.

A. Benefits

Memory footprint reduction is the most obvious benefit – and still the driving factor for industries of mass production, such as automotive [8]. It is not a coincidence that OSEK (and later AUTOSAR) were designed for specialization on the instance level from the very beginning. The compile-time instantiation of kernel objects and their management in preallocated arrays instead of linked lists facilitates significant RAM savings. In [17], the transformation of an RTS from the abstraction-level specialized eCos [22] to the instance-level specialized CiAO [21] reduced the RAM footprint by half.
But also abstraction-level specialization alone can pay off, if applied systematically: The specialization of a Linux system running typical appliances, such as a LAMP server or an embedded media player, can reduce its code size by more 90 percent compared to a standard kernel [30], [28].

Security and safety improvements are less obvious, but a corollary from memory footprint reduction: What is not there can neither break nor be attacked or exploited and does not need to be later maintained in this respect. For instance, specializing the mentioned LAMP server on the level of abstractions did not only reduce its code size, but also cut the number of relevant entries in the CVE database[1] by ten percent [30]. The instance-level specialization of the RTS in [17] also increased its robustness regarding bit flips by a factor of five.

[1] https://cve.mitre.org

Further significant improvements in this respect are possible if one specializes down to the level of interactions, for instance, by inserting control-flow assertions [11].

Better exploitation of hardware by a direct mapping of RTOS abstractions. Modern µ-Controllers are not only equipped with an increasing number of cores, but also large arrays of timers, interrupt nodes and so on. Nevertheless, most RTOS implementations still multiplex a single hardware timer and IRQ context. In Sloth [16], [15] the specialization on instance-level is the prerequisite to map system objects at compile-time directly to the available hardware resources, which results in minimal kernel footprints and excellent real-time properties.
If specialization of the hardware itself is also an option, a kernel specialized on interaction-level could even be placed directly into the processor pipeline [12].

Reduction of jitter and kernel latencies is a further benefit of memory footprint reduction and the better exploitation of hardware. Intuitively, removing code, state, and indirection in the control and data flows of the kernel also reduces noise caused by memory access and cache pressure and increases determinism. Shorter kernel paths and the direct mapping of kernel objects to hardware yield a direct benefit on interrupt lock times and event latency.

Analyzability and testability is both improved as well as impaired (see below). In principle, any reduction of possible kernel states and execution paths increases determinism and makes it easier to analyze, test, and validate the kernel against the RTS specification. The model that is required for instance-level specialization can directly be used for static conformance checking to find, for instance, locking protocol violations. If specializing on interaction level, the underlying interaction model [11] further paves the path to whole-system end-to-end response-time and energy-consumption analysis [13], [31].

B. Challenges

However, specialization does not come for free. It depends on a very deep understanding of your RTS on the systems level, as well as the ability and willingness to express its properties and demands towards the RTOS. In our experience, deep specialization remains a hopeless attempt if the configuration of the RTOS is mostly based on experience and manual labor of the RTS developer. Full (or at least nearly full) automation of specialization by tools is the key to success.

You have to know what you need and this is probably the major challenge. In practice the burden is on the developer – and this already hits its limits when specialization takes place on the level of abstractions: Recent versions of Linux (4.16), but also smaller RTOSs like eCos, provide an unbearable number of configuration options (more than 17000 in Linux, respectively 5400 in eCos). Hence, most developers have long ago stopped specializing more than necessary and employ, in the case of Linux, a one-size-fits-all standard distribution kernel instead.
To be viable in practice, the RTOS configuration has to be derived automatically: In fact, the 90 percent code savings in Linux mentioned above were only achievable by an automatic specialization approach that measures the required features on a standard distribution kernel in order to derive a tailored configuration [30], [28]. Schirmeier and colleagues suggested automatic detection of required eCos features (level of abstractions) by static analysis of the application source [29]. However, they also identified limits of their approach when the decision about an abstraction (e.g., the need for a costly priority inheritance protocol in the mutex abstraction) depends on information only available on the instance or interaction level (i.e., who accesses a particular mutex at run time).
Hence, for the developer automatic configuration becomes actually easier with instance- or interaction-level specialization. As she has to think about the employed system objects anyway, specifying the requirements on the instance level is closer to the application and more natural, while the configuration tool can automatically derive the necessity of, for example, a priority inheritance protocol in mutex objects. OSEK, which is specialized on instance level, automatically derives the priority of the resource objects specified in the OIL file [23], [24].
If interaction-level information is required, a manual provisioning would become completely intractable. However, in this case static analysis of the application source code is even more promising than on the feature level: Programming is the act of writing down desired interactions between instances, which are technically expressed by syscalls, and we can use static analysis to extract these interactions. For example, Bertran et al. [7] analyze all libraries and executables of a concrete Linux system and remove system calls that cannot be activated. Furthermore, with a complete and flow-sensitive analysis of the application's execution across the syscall boundary we could retrieve a complete interaction model [11]. This, however, has exponential overhead if indeterminism by external events needs to be considered. Hence, the analysis needs to be constrained by further information that is commonly not expressed in the source code, such as event-activation frequencies.

You have to be able to express what you need is therefore another challenge – and unfortunately in many cases the RTOS interface even hinders the expression of instance-level developer knowledge [18]: Most RTOSs adhere to (or at least mimic) a POSIX-style API with dynamic allocation and instantiation of a conceptually arbitrary number of system objects at run time. This mindset stems from interactive multi-user systems (UNIX), but has to be considered as a strong misconception in the world of real-time systems – for both sides, developers and users of an RTOS. The already mentioned reductions in the kernel's memory footprint when switching from the POSIX-like eCos to the OSEK-like CiAO in [17] are rooted in the kernel-internal overhead of implementing an interface that favors (unneeded) dynamic instantiation. So, if the RTOS employs such a "flexible" syscall interface, more additional information has to be provided by the developer to enable instance- and interaction-level specialization.

Testability and certifiability is in our opinion becoming the most significant obstacle towards systematic specialization of system software. With the advent of autonomous driving features, the industry is facing new challenges with respect to functional safety; ISO 26262 and ASIL D demand the employment of a certified RTOS. While in principle the certification of a less flexible system should make this easier (see discussion of the respective benefit in the previous section), existing certification procedures mostly follow a certify-once-and-never-touch-again philosophy that is fundamentally the opposite of application-specific specialization. The certification of an RTOS kernel is extremely expensive, so vendors shy away from the even higher costs of certifying a kernel generator. However, without a certified generator, each specialized kernel instance

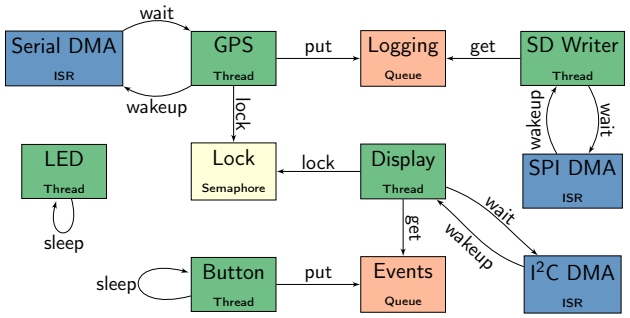


Fig. 2: Interaction Graph for GPSLogger

would have to be certified individually. In the extreme case (full interaction-level specialization) this would be necessary for every change of the application implementation. Hence, certified RTOSs, such as RTA-OS (ETAS), MICROSAR OS (Vector), or tresos Safety OS (EB) offer not more, but significantly less room for specialization.
So one has either to forgo the benefits of specialization or to swallow the bitter pill of certifying a complete kernel generator, which is highly unrealistic. A more promising direction could be to make a virtue out of necessity and extend the (automatic) specialization to the certification process as well: We do not need to validate the specialized kernel against the full RTOS specification, but only to those parts and interactions that are actually used on the concrete RTS. If the interaction model could be assumed as sound and complete, it can be employed with model checkers to automatically validate the generated kernel instance. [9]

C. Summary

Despite very high improvements regarding many nonfunctional properties, RTOS specialization is performed only half-hearted in practice, as explicit configuration puts to much burden on the developer. This is partly caused by unsuitable UNIX-inspired syscall APIs and misconceptions about "what the OS is and provides". Hence, deep specialization requires automation to remove the burden of having to understand and know the details from the developer. The analysis of the application's requirements interactions as well as the generation of a fitting RTOS instance has to be provided by tools.

Nevertheless, also with existing RTOSs implementations that offer a less-than-ideal API, significant savings are achievable. In the following, we exemplify this by re-analyzing an existing application running on FreeRTOS from the viewpoint of our taxonomy.

IV. An Experiment with FreeRTOS

Our example is the a freely available GPSLogger[2] application, which uses FreeRTOS [5] to orchestrate its threads. The system runs on a "STM32 Nucleo-F103RB" evaluation board that is equipped with a STM32F103 MCU. It is connected to a graphical display (I^2C), a GPS receiver (UART), a SD card (SPI), and two buttons (GPIO). The application consists of 5 threads, 3 ISRs, 2 blocking queues, and one binary semaphore. Due to a broken SD card library, we replaced the SD card operations with a `printf()`.

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

In Figure 2, we extracted the interaction graph for this application manually from the source code. For compactness reasons, we omitted some interactions from the figure (i.e. preempt). The inter-process communication is mainly done with blocking message queues. However, the GPS thread and the display thread bypass the kernel for the transferred data and use a shared memory region that is protected by a binary semaphore. For most IO operations, GPSLogger uses a pattern where one thread blocks passively until one DMA ISR signals the completion of a data transfer. However, for the button thread, GPSLogger uses active polling with a passive sleep. While the employment of full-blown queues is overkill to transmit small datagrams in 1:1 interactions, it is the primary abstraction offered by FreeRTOS.

a) Specialization of Abstractions: FreeRTOS provides abstraction-specialization capabilities by using conditional compilation and C preprocessor macros. However, there is no formal or semi-formal feature model, like it is provided by Linux KConfig or the eCos configuration tool, but the configuration is placed in a header file. As another specialization, unreachable functions are automatically removed by the linker, as the build system uses function- and data sections in combination with link-time garbage collection.

At this specialization level, the resulting binary uses $91{,}084$ bytes for code and $18{,}328$ bytes of mutable RAM. The kernel takes $60{,}426$ cycles of startup time, before the first task starts. Startup times were measured 100 times and the standard deviation always was below 35 cycles.

b) Specialization of Instances: For the instance level, we removed the dynamic system-object allocation in favor of statically allocating them in the data section. These system objects include the thread stacks, the thread control blocks, queues, and the ready list. FreeRTOS, since version 9.0.0, supports that the user provides a statically allocated memory to hold system objects and, thereby, gets rid of the special FreeRTOS heap. With static allocation, we use 112 more bytes of code, but save 856 bytes of RAM and $6{,}598$ cycles of startup time compared to baseline. The increase in code size stems from the additional parameter of the static object-initialization functions.

As a second step, we removed the dynamic initialization of stacks, thread-control blocks, and the scheduler. Instead, we initialized their values and pointers statically such that the data section already contains a prepared memory image to start FreeRTOS. Compared to baseline, the statically initialized GPSLogger saves 344 bytes of code and $7{,}327$ cycles of startup time. The RAM usage is equal to the variant only with static memory allocation.

c) Specialization of Interactions: After carefully examining the GPSLogger, we came to the conclusion that a interaction-level specialization that is restricted to the RTOS is not possible here. From the FreeRTOS API usage it is hard to tell why a specific API was used, since it hindered the expression of the developer's intention (see also Section III).

However, we extend the scope of the specialization to the application. From the interaction model (Figure 2), we know that the LED thread does not interact with any other thread as it only periodically blinks the LED. Furthermore, toggling a GPIO pin takes far less cycles than the thread-management

overhead. Therefore, we can safely inline the GPIO toggling into the timer ISR and remove the LED thread, including its stack and TCB. Compared to baseline, the system becomes 512 bytes of code and 1,616 bytes of RAM smaller. The startup time decreases by 9,397 cycles.

V. Conclusions and Future Work

In this paper, we described a taxonomy of specialization for real-time systems and define three levels of specialization that successively remove (unneeded) flexibility from the system. On the abstraction level, instance, and interaction level, we can remove abstractions, make instances fixed, and forbid concrete interactions. Furthermore, we discussed the benefits and challenges introduced by specialization. Although specialization yields significant improvements of nonfunctional properties, manual specialization has long outgrown engineers capabilities and is thus mostly applied on the coarse-grained abstraction level. Therefore, we argue that specialization on deeper levels requires automation to reach it's full potential.

To illustrate our taxonomy, we (manually) specialized an example application on the three specialization levels. Although the application was not designed with specialization in mind, we were able to extract the actually required interaction graph and, in consequence, were able to specialize the system to show improved nonfunctional properties. Therefore, we plan to integrate automated analysis and specialization into the build process and the compiler toolchain. If once automated, all levels of specialization can be generated and compared at compile time to choose the variant with the best nonfunctional properties for the specific use case.

References

[1] ERIKA Enterprise. http://erika.tuxfamily.org, visited 2014-09-29.

[2] Portable operating system interfaces (POSIX®) – part 1: System api – amendment 1: Realtime extension, 1998.

[3] AEEC. Avionics application software standard interface (ARINC specification 653-1), 2003.

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

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

[6] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *15th ACM Symp. on Operating Systems Principles (SOSP '95)*. ACM Press, 1995.

[7] Ramon Bertran, Marisa Gil, Javier Cabezas, Victor Jimenez, Lluis Vilanova, Enric Morancho, and Nacho Navarro. Building a global system view for optimization purposes. In *2nd Work. on the Interaction between Operating Systems and Computer Architecture (WIOSCA '06)*. IEEE Computer Society Press, 2006.

[8] Manfred Broy. Challenges in automotive software engineering. In *28th Intl. Conf. on Software Engineering (ICSE '06)*. ACM Press, 2006.

[9] Hans-Peter Deifel, Christian Dietrich, Merlin Göttlinger, Daniel Lohmann, Stefan Milius, and Lutz Schröder. Automatic verification of application-tailored OSEK kernels. In *17th Conf. on Formal Methods in Computer-Aided Design (FMCAD '17)*. ACM Press, 2017.

[10] 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)*. ACM Press, 2015.

[11] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. Global optimization of fixed-priority real-time systems by RTOS-aware control-flow analysis. *ACM TECS*, 16(2), 2017.

[12] Christian Dietrich and Daniel Lohmann. OSEK-V: Application-specific RTOS instantiation in hardware. In *2017 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES '17)*. ACM Press, 2017.

[13] Christian Dietrich, Peter Wägemann, Peter Ulbrich, and Daniel Lohmann. Syswcet: Whole-system response-time analysis for fixed-priority real-time systems. In *Real-Time and Embedded Technology and Applications (RTAS '17)*. IEEE Computer Society Press, 2017.

[14] Arie Nicolaas Habermann, Lawrence Flon, and Lee W. Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5), 1976.

[15] Wanja Hofer, Daniel Danner, Rainer Müller, Fabian Scheler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Sloth on Time: Efficient hardware-based scheduling for time-triggered RTOS. In *Real-Time Systems (RTSS '12)*. IEEE Computer Society Press, 2012.

[16] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. Sloth: Threads as interrupts. In *Real-Time Systems (RTSS '09)*. IEEE Computer Society Press, 2009.

[17] 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 *ISORC'14*. IEEE Computer Society Press, 2014.

[18] Tobias Klaus, Florian Franzmann, Tobias Engelhard, Fabian Scheler, and Wolfgang Schröder-Preikschat. Usable RTOS-APIs? In *10th Annual Work. on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '15)*, 2014.

[19] Butler W. Lampson. Hints for computer system design. In *9th ACM Symp. on Operating Systems Principles (SOSP '83)*. ACM Press, 1983.

[20] Jochen Liedtke. On μ-kernel construction. In *15th ACM Symp. on Operating Systems Principles (SOSP '95)*. ACM Press, 1995.

[21] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *2009 USENIX Annual Technical Conf.* USENIX Association, 2009.

[22] Anthony J. Massa. *Embedded Software Development with eCos*. New Riders, 2002.

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

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

[25] David Lorge Parnas. On the design and development of program families. *IEEE Trans. on Software Engineering*, SE-2(1), 1976.

[26] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, 2014.

[27] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1), 1988.

[28] Andreas Ruprecht, Bernhard Heinloth, and Daniel Lohmann. Automatic feature selection in large-scale system-software product lines. In *13th Intl. Conf. on Generative Programming and Component Engineering (GPCE '14)*. ACM Press, 2014.

[29] Horst Schirmeier, Matthias Bahne, Jochen Streicher, and Olaf Spinczyk. Towards eCos autoconfiguration by static application analysis. In *1st Intl. Work. on Automated Configuration and Tailoring of Applications (ACoTA '10)*, CEUR Work. Proceedings. CEUR-WS.org, 2010.

[30] Reinhard Tartler, Anil Kurmus, Bernard Heinloth, Valentin Rothberg, Andreas Ruprecht, Daniela Doreanu, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Automatic OS kernel TCB reduction by leveraging compile-time configurability. In *8th Work. on Hot Topics in System Dependability (HotDep '12)*. USENIX Association, 2012.

[31] Peter Wägemann, Christian Dietrich, Tobias Distler, Peter Ulbrich, and Wolfgang Schröder-Preikschat. Whole-system worst-case energy-consumption analysis for energy-constrained real-time systems. In *30th Euromicro Conf. on Real-Time Systems 2018*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. to appear.

Verification of OS-level Cache Management

Renato Mancuso*, Sagar Chaki+
*Boston University, USA, rmancuso@bu.edu
+ Mentor Graphics, sagar.chaki@acm.org

***Abstract*—Recently, the complexity of safety-critical cyber-physical systems has spiked due to an increasing demand for performance, impacting both software and hardware layers. The timing behavior of complex systems, however, is harder to analyze. Real-time hardware resource management aims at mitigating this problem, but the proposed solutions often involve OS-level modifications. In this sense, software verification is key to build trust and allow such techniques to be broadly adopted. This paper specifically focuses on CPU cache management, demonstrating that OS-level hardware management logic can be verified at the source code level in a modular way, i.e., without verifying the entire OS.**

I. Introduction

In the last decade, there has been an uptrend in the complexity of safety-critical real-time systems. Such a trend is the result of an ever increasing demand for performance, features and efficiency. Multi-core platforms and heterogeneous hardware largely represents the industry's answer to such an increase in computational demand. As the hardware grows in complexity to match the demand for performance, it becomes increasingly hard to fully understand or to *predict* its timing behavior.

Unfortunately, the loss of timing predictability makes real-time analysis significantly harder, with two unwanted consequences. First, the inability to produce *tight* upper-bounds on workload worst-case execution time (WCET) leads to overprovision and waste of hardware resources. Nonetheless, the decreasing cost of hardware components partially mitigates this problem. Second, safety-critical systems are required to undergo a rigorous certification process in order to be considered for large-scale deployment. Difficulty in determining the logical and temporal correctness of a system heavily impacts certification costs. These costs easily surpass the sheer cost of hardware components by several orders of magnitude.

A number of works [9], [20], [12] have proposed OS-level mechanisms to explicitly manage those hardware components that, if unregulated, represent major sources of unpredictability: i.e. shared CPU caches, DRAM memory, and I/O subsystem. Management techniques proposed in the literature have been shown to achieve substantial real-time benefits. Yet, many industries are reluctant to widely adopt such solutions due to a fundamental lack of *confidence* about the correctness of their implementation. The fear is justified considering that hardware management mechanisms often operate at high-privilege level, and thus their misbehavior can lead to substantial failures.

This work represents a first step toward the verification of system-level components that implement hardware management techniques for real-time purposes. In fact, in this work we demonstrate that it is possible to verify the logic of a kernel-level component at the source code level in a modular way; i.e. without verifying the entire OS that can be assumed verified or trusted. Specifically, this paper presents the verification approach for Colored Lockdown [11]: a real-time last-level cache management scheme implemented in the Linux kernel. Colored Lockdown is part of a larger framework of hardware resources management techniques for multi-core platforms that goes under the name of Single Core Equivalence framework (SCE) [12], [13].

The rest of the paper is structured as follows. In Section II we provide an overview of the related work. Section III provides the required background knowledge for this work. A high-level description of our verification approach is discussed in Section IV, while additional implementation details are provided in Section V. Next, a brief evaluation is reported in Section VI. Finally, concluding remarks and possible future extensions are discussed in Section VII.

II. Related Work

As increasingly higher level of assurance is required from safety-critical systems, there has been an uptrend in the popularity of verification methodologies. A consistent body of works has used the "verified by design" approach. In this context, the SPARK language and toolkit [3] provide extensive capabilities to reason about the correctness of applications at a source code level. In the SPARK environment, verification is performed with a combination of static analysis and deductive verification. Deductive verification on the other hand, has been widely used on industrial use-cases [7], [10], [4]. Similarly, the level of assurance provided by formal static analysis based on abstract interpretation often represents a good trade-off in terms of scalability [16], [6].

Automated assertion checking is often used as an alternative to deductive verification. With this approach, it is typically possible to confine the explored state space to a manageable subset that is fundamental for the considered properties/assertions. Among the different techniques for assertion checking, bounded verification is often used for source code debugging. A number of consolidated tools implement assertion checking, e.g. SLAM [2], TASS [19], and CBMC [5] used in this paper.

Recent works have explored the use of verification techniques to validate application-level software in the domain of control systems [8], aerospace and avionic software [21], and railways systems [15]. In seL4 [14], the design and verification of an entire OS is proposed. While closely related to [14], we take a fundamentally different approach: we consider certified systems where new kernel-level functionality can be introduced to improve/optimize performance and demonstrate how modular verification of OS-level code can be performed. Finally, many works perform verification of the interaction between kernel modules and OS routines [17], [1]. Conversely, we focus on the verification of kernel-level logic that interacts with (i) kernel sub-routines, (ii) virtual memory, and (iii) CPU cache space.

III. Background

The philosophy behind SCE is that performance in a multi-core system can be analyzed and certified using a modular approach with respect to the rest of the system. In order to attain this goal, four main components are used in SCE to mitigate inter-core interference arising from a correspondent number of major sources [18], [12], [22], [23], [11]. Apart from other components used to manage DRAM and I/O, Colored Lockdown [11] is used to perform deterministic allocation of real-time task data and instruction in last-level shared cache. When Colored Lockdown is used, the portion

+This work was done while this author was an employee of Carnegie Mellon University.

of task memory allocated in cache will exhibit 100% hit rate. In this paper, we specifically focus on verifying the OS-level logic of Colored Lockdown. In this section, we provide an overview of the design of Colored Lockdown and briefly describes its internal components.

On multi-core systems, the timing of an application running on core A can be affected by a logically unrelated application running on core B if they share cache space. This timing inter-dependence goes under the name of "inter-core (performance) interference". The goal of Colored Lockdown [11] is to use *cache locking* to address inter-core interference while providing a trade-off between efficiency and flexibility. Colored Lockdown involves two main stages: an offline profiling stage; and an online cache allocation stage.

Profiling: during the offline stage, each real-time task is analyzed using a memory profiler [11]. When the task runs in the profiling environment, memory accesses are traced and per-page access statistics are maintained. Next, (i) pages of the task's addressing space are ranked by access frequency; and (ii) a profile is produced identifying frequently accessed (hot) pages by their relative position in the addressing space. The final profile can be used online to drive the cache allocation phase. Given the produced profile, two mechanisms are used to provide deterministic guarantees and a fine-grained cache allocation granularity, described below.

Page Coloring: last-level caches in modern multi-core platforms are typically set-associative physically indexed caches. As such, multiple main-memory pages can be mapped to a given *set* of shared cache pages. Pages in the same set are said to have the same "color". Pages with the same color can be allocated across cache *ways*, so that as many pages as the number of ways can be simultaneously allocated in last level cache. Any application page can be re-colored transparently to the application by only manipulating physical memory and page-table translations. Colored Lockdown relies on this mechanism to reposition task memory pages within the available colors, in order to exploit the entire cache space.

Lockdown: real-time applications are dominated by periodic execution flows. This characteristic allows for an optimized use of last level cache by locking hot pages first. Relying on profile data, Colored Lockdown first colors frequently accessed memory pages to remap them on available cache ways; next, it exploits hardware cache locking support to guarantee that such pages (once prefetched) will persist in the assigned location (locked), effectively overriding the default cache replacement policy.

IV. Verification Approach

This section provides an overview of the approach followed to verify the main properties of Colored Lockdown. We first establish the boundaries of the performed verification; next, we discuss what memory model is being considered; and finally we describe what components of the hardware/OS are abstracted.

Verification Strategy: we perform source-level verification via bounded model checking of the main block of code that is responsible for the allocation of memory pages in last-level cache within the Colored Lockdown module. The considered code is compiled as a Linux kernel module and runs at the highest level of privilege in the target platform. Verifying its correctness is therefore of great value.

In order to perform cache allocation, the Colored Lockdown module tightly interacts with the rest of the Linux kernel in two main ways: (i) it uses data from many descriptors used in the kernel; (ii) it invokes memory manipulation/translation procedures provided by the Linux kernel. The code base of the entire Linux kernel is too large and complex to be formally verified. For this reason, we restrict the verification to that portion of the cache allocation logic that is directly related to Colored Lockdown.

In order to focus the verification on the important components, we abstract the behavior of any invoked kernel routine, as detailed in Section V. For instance, a routine used to allocate a new generic memory page is abstracted as a function that returns an unsigned integer. The return value is non-deterministic, and such that: (i) it is aligned to the memory page size; and (ii) it is within the range defined by the bit-width of the considered memory layout.

Similarly, only sub-fields of kernel data structures that are relevant to verification are initialized by the verification routines. A portion of the initialization procedure is parameter-dependent, so that different cache allocation scenarios can be analyzed.

Verification Boundaries and Assumptions: the hardware-level properties that are abstracted mostly concern the behavior of a typical cache controller that allows per-line cache locking. Hence, we make the following assumptions. First, we assume that the initial status of the cache is unknown. This reflects the status of a cold cache at the time of Colored Lockdown allocation. Second, we assume that the considered cache is physically indexed[1]. Third, we consider that the bits of the physical address that encode the index of a cache line correspond to the least significant bits following the cache offset bits. Hence, the structure of a physical address from the cache controller's perspective from most to least significant bit is: tag bits, index bits, offset bits. Since we consider cache controllers that support per-line locking, we assume that a special instruction is available to set a lock bit on a per-line basis. Once the lock bit has been set, the cache line cannot be evicted from cache. Finally, we assume that a cache look-up for a locked line will result in a cache hit.

We verify an implementation of Colored Lockdown as a Linux kernel module. The same logic, however, can be ported across different OS's, assuming that they provide kernel-level routines with similar semantics. In order to focus our attention on the target module, we assume that the descriptors belonging to the OS and used by Colored Lockdown have been correctly initialized (see Section V). Next, we assume that profile information about the process under consideration have been correctly passed from user-space to kernel-space. Finally, we assume that all the virtual memory pages of the process have a valid mapping in physical memory. The latter assumption is typically verified in RTOS's that do not perform demand-paging. Under Linux, this behavior can be achieved using the `mlockall` system call.

Memory Layout Specification: our verification is parametric with respect to the memory layout and cache controller configuration. Thus, it is possible to re-run the verification procedure on a specific memory/cache configuration and with a variable number of pages to be allocated, i.e. *profile pages*. The following five parameters suffice to fully define the considered memory subsystem as well as the address structure from the cache controller's perspective:

(1) P_s: Number of bits in a virtual address that encode the offset of a byte in a memory page, also known as *page shift*;
(2) B_w: Bit-width of a physical address in the considered platform, e.g. 32 for 32-bit architectures; 48 for 64-bit architectures[2].
(3) O: Number of bits in a physical address that encode the offset of a byte within a cache line;
(4) I: Number of bits in a physical address that encode the index in cache of a cache line;
(5) W: Associativity – i.e. number of ways of the cache.

[1]Last-level caches in multi-core platforms are typically physically tagged and indexed.

[2]Despite the bit-width of CPU registers is 64 bit, the memory subsystem typically works with 48 bit addresses. This results in 256 TB of addressable memory and a 4-level page tables layout is used.

Given the five parameters above, the rest of the parameters used to perform cache locking can be derived: size of a memory page; size of a single cache line; number of lines and pages (i.e., available colors) per way; number of cache sets; bit-width of the cache tag; and total size of the cache.

One more parameter controls the amount of memory that is allocated in cache for the process under consideration. This parameter defines a generic number of pages that is prefetched and locked in cache as a result of the coloring/locking logic. By default, all the pages are considered as process' heap pages. This however does not affect the generality of our approach since there is no difference in the way pages belonging to various regions are handled.

Verified Properties: a set of core properties of Colored Lockdown was successfully verified. The target of the verification is twofold: (i) that cache allocation is correctly performed when profiling data are correctly specified from user-space, and the amount of memory to be locked in cache is smaller than the cache size; and (ii) that the status of the system and cache is overall consistent. Note that using the current verification infrastructure, additional system/cache properties can be verified. The verified properties can be summarized as follows:

(1) If the number of pages to be allocated in cache is less or equal to the number of available cache space in pages, cache allocation for the considered process is entirely performed. Otherwise, no cache allocation is performed;
(2) If cache allocation is performed, then all the physical memory mapped to each virtual address within the range selected for allocation will be locked in cache;
(3) No more than the total number of locked pages are set as locked in cache at the end of the Colored Lockdown procedure;
(4) All the temporary kernel-level resources required by Colored Lockdown to execute are released at the end of the procedure.

Verification Challenges: we hereby summarize the challenges that had to be addressed to perform source-level verification of Colored Lockdown as a OS-level component. One of the first challenges we encountered in the attempt to verify a Linux kernel module was the large number of dependencies with the kernel source code that a module can exhibit. Three main type of dependencies exist: data type dependencies, procedural dependencies, and logic dependencies.

A Linux kernel module uses several types that are defined and exported by the kernel. Many of these types are complex C-language structures interconnected via pointers. Obviously, only a subset of the fields in such structures are required for focused verification. CBMC v. 5.2 [5], the source code verification tool we used, employs slicing to eliminate unused variables and reduce verification complexity. However, we found this to be inadequate for our target system. The first challenge was to manually prune the definitions of kernel-level structures to exclude all the irrelevant fields. In order to overcome this issue, we have incrementally transferred into the verification sandbox a number of kernel headers and systematically stripped them of unneeded data types and fields. For instance, one of the imported files was `sched.h` that in the Linux kernel defines constants and types relevant for process management. The file is about 2700 lines long in a typical Linux source tree. In the first pruning, we only maintained the process descriptor definition, reducing the file length to about 370 lines. Next, we identified the only two fields required for verification out of the 170+ fields included in a typical process descriptor.

The second type of dependency is procedural dependency. The code that needs to be verified uses at top level a set of routines defined in the kernel code. To reduce the state space and the amount of code logic to be verified, one challenge consists in abstracting the semantics of the invoked procedures (if possible) and making a reasonable assumption on their output. In Section V we describe as an example the abstraction performed on the kernel procedure `get_user_pages`.

Finally, many logic dependencies exist between the state of the kernel and the verified module. This problem sets our verification approach apart from verification of standalone components. In fact, the Colored Lockdown module expects the status of a number of kernel-level descriptors to be initialized and valid. Some of these descriptors are created at boot-time, while others are constantly updated upon system events. Hence, it would be unfeasible to verify the code responsible for their initialization. To tackle this challenge, we have first identified all the logic dependencies. Next, we have introduced an initialization routine that either explicitly sets each referenced variable to its expected value or assumes its value to be within the expected range. A closer look at the initialization procedure is provided in Section V.

Overall, CBMC revealed a good maturity in handling C source code. However, when verifying kernel-level code, we have encountered a few glitches that need to be carefully addressed to avoid false negatives in the verification process. Relatively simple workarounds have been found for all the encountered glitches. Such problems, however, can represent a serious overhead in the verification process when reasoning over a large base of system-level code.

The first problem we encountered regards the way `void` pointers are handled in CBMC. The standard C semantics enforces that the increment of a `void *` data type is performed at the granularity of a single byte. Consider the following code:

```
int void_test(void)
{
    void * ptr = (void *)(1 << 12);
    ptr += 0x100;
    return (ptr == (void *)0x00001100UL);
}
```

The code compiles without warnings/errors under a standard GCC compiler. The expected return of the `test` function is always 1 under standard C-pointer arithmetic. However, a CBMC verification instance that relies on this behavior will fail. Running CBMC 5.2 on the considered procedure, produces the following output:

```
Counterexample:

State 21 file ./cbmc_test.c line 18 function void_test thread 0
----------------------------------------------------
  ptr=NULL (00000000000000000000000000000000)

State 22 file ./cbmc_test.c line 18 function void_test thread 0
----------------------------------------------------
  ptr=NULL + 4096 (00000000000000000001000000000000)

State 23 file ./cbmc_test.c line 19 function void_test thread 0
----------------------------------------------------
  ptr=NULL + 3840 (00000000000000000000111100000000)

Violated property:
  file ./cbmc_test.c line 58 function main
  assertion return_value_void_test$1
  (_Bool)return_value_void_test$1

VERIFICATION FAILED
```

Clearly, State 23, which should reflect the pointer's status after the increment in the considered code extract, reports a wrong pointer value. This triggers a verification failure. A possible workaround consists in performing the pointer value increment after a conversion to unsigned long[3].

The second issue requires a longer explanation and due to space constraints we omit a detailed description. Briefly, CBMC seems to exhibit a glitch in the propagation of a variable value after it has been assigned using a bitwise operator. Consider the following snippet:

[3]The unsigned long type has typically the same width of a pointer.

```
int retval = 1;
for (...) {
    retval &= bool_function(...);
}
if (retval) ...
```

In this case, CBMC produced verification counterexamples that reported the execution of the `if` block even though the state value of the `retval` variable was 0 (false);

In our verification, we found that many subtle interactions in system-level code are hard to fully capture at a source level. Consider the following case. Colored Lockdown performs re-coloring of a process page. For this purpose, a new physical page is allocated and its content copied from the original page, appropriately modifying the page tables of the process under consideration. This behavior is "correct" as far as Colored Lockdown is concerned. However, if no action is taken to de-allocate the original page correctly, Colored Lockdown can indirectly trigger a fault somewhere else in the system as the original page descriptor remains in an inconsistent state. Similar interplay problems can occur when a module accesses a data structure without acquiring the required lock. In a typical multi-threaded application, this problem would be easy to detect since all the execution flows are known. The problem is however significantly harder to solve without knowing where in the kernel potential data races can arise.

Finally, a challenge that affects source-level verification at large, is the quick increase in complexity as the state-space expands. In our verification attempt, we were able to overcome the vast majority of challenges described in this section. In spite of this, verification settings with realistic parameters required significant computational resources. We provide additional insights on the feasibility and limits of our verification approach in Section V.

V. Verification Details

In this section, we provide additional details about the performed verification. First, we discuss how the cache hardware is modeled; next, we discuss the initialization of kernel structures and OS state. A detailed overview about how cache and memory layout are initialized is also provided. Finally, we detail the structure and verification statements used to verify the core properties of Colored Lockdown.

Cache Model: traditional source-level verification tools, including CBMC, do not provide primitives to model platform hardware behavior. For this reason, we use a supporting data structure to maintain the cache state and to perform assertions on its state. Colored Lockdown allows deterministic allocation of memory pages in cache. Thanks to coloring, the mapping set is explicitly controlled. Conversely, the decision about the allocation way is left to the cache controller. The key insight, however, is that when the replacement policy attempts to allocate a line with a certain set, and the line for that set is marked as locked in a given cache way, the way cannot be selected for eviction. Thus, as long as a number of lines less or equal to the cache associativity is locked, each locking request can be satisfied. It follows that the logical view of a cache is a 2D structure (sets vs. ways). One index (set index) is derived from the physical address being allocated; while the other index (way index) is non-deterministically determined by the replacement policy.

Following this structure, the cache status is defined as:

```
1  typedef struct {
2      void * addr;
3      char locked;
4  } cache_line_t;
5
6  typedef cache_line_t cache_set_t [CACHE_ASSOC];
7  typedef cache_set_t cache_t [CACHE_NSETS];
8  cache_t cache;
```

In the listing above, `CACHE_ASSOC` and `CACHE_NSETS` refer to the number of sets and to the number of ways (associativity), respectively. Note that there is no need to record the *value* of the cached data, as we are only concerned with hit/miss behavior. Hence, only cached *address* and *locked status* are being tracked.

The assumption we make about the initial state of the cache is that no line is currently locked. As such, we initialize the locked state on all the cache elements as 0, and assign a non-deterministic value to the address field.

Profile Structure and Initialization: as stated in Section IV, we assume that profiling information has been passed from user-space to kernel-space before the lockdown procedure is invoked. Hence, for verification purposes, we explicitly initialize the kernel structures that hold kernel-side profile data. In Colored Lockdown, profiling data is provided via the Linux CGROUP virtual file-system interface. For a task for which a profile has been loaded via the CGROUP interface, a custom structure, namely `struct task_profile` is associated with the task descriptor. The most relevant fields of the structure are: (i) number of memory regions with pages to be locked; (ii) list of descriptors for memory regions with data to be locked; (iii) total number of pages to be allocated in cache; (iv) list of descriptors for pages to be locked.

Since the state of the `struct task_profile` object is assumed to be valid, an initialization routine was added. The routine allocates enough data to contain the full list of memory regions and memory pages. These parameters are set at profile loading time, hence they are known at the time Colored Lockdown is invoked. In the context of this paper, they constitute parameters for the creation of a verification instance. A default scheme is used to associate memory pages to areas. This choice however does not compromise the generality of the verification, as there is no difference in the way pages in different areas are handled.

Within each memory region's descriptor, only the index that the considered region has in the list of kernel-maintained virtual memory areas (VMA) is initialized. The logic that resolves such a (relative) index into an absolute range of virtual memory addresses is part of the Colored Lockdown logic. Hence, it is part of the verification.

Task Descriptor Setup: when Colored Lockdown is invoked as a system call by a task, it heavily relies on information contained within the kernel-maintained task descriptor `struct task_struct` to perform cache allocation. Whenever any system call is invoked in the kernel, a globally visible expression, namely `current`, expands to a pointer to the `struct task_struct` object for the calling process. For verification purposes, the object pointed by `current` needs to be initialized. The following is an extract of the task descriptor setup routine:

```
1  int pages;
2  struct vm_area_struct * prev_vma;
3  struct vm_area_struct * cur_vma;
4  /* ... */
5  prev_vma->vm_start = 0x08048000UL;
6  current->mm->mmap = prev_vma;
7  pages = nd_int();
8  __CPROVER_assume(pages >= AREA_MINPAGES && pages <= AREA_MAXPAGES);
9  prev_vma->vm_end = prev_vma->vm_start + (pages << PAGE_SHIFT);
10 /* Link VMAs */
11 cur_vma->vm_start = prev_vma->vm_end;
12 prev_vma->vm_next = cur_vma;
13 /* Use cur_vma to setup next VMA */
```

The first area in the list of VMAs is typically the `text` (i.e. the executable code) section of a process. The start of the first area is taken as the default address at which code is logically placed in compiled executables (line 5). The address of the first VMA descriptor is recorded inside the `current` object (line 6). Next, a non-deterministic number of pages between the established boundaries is generated in lines 7–8, and the end of the first VMA is set accordingly (line 9). As VMAs are initialized, they are placed in an unidirectional linked list (lines 11–12).

Colored Lockdown Procedure: the Colored Lockdown module also performs a series of initialization routines as soon as it is loaded (once) into the kernel. The routines mostly initialize cache parameters and buffers required to perform page coloring. Due to space constraints, we omit the details about how initialization is performed inside the verification environment.

When Colored Lockdown core logic is invoked as a system call, the sequence of operations can be summarized as follows:

(1) Access to profile structure and validation of `current` object – to make sure Colored Lockdown is performed on the right task;
(2) Derivation of virtual addresses for each memory page in the profile to be allocated in cache;
(3) Resolution of virtual addresses into physical addresses and cache color calculation;
(4) Check of color availability in cache and assignment of first available color;
(5) If each page has been assigned a color, perform page re-coloring (as needed) and lockdown.

Hereby, we provide a few extracts of kernel logic that are relevant to understand the interaction with CBMC. The first point is trivially verified because we assume that profile data passing and Colored Lockdown invocation is performed correctly. The second step largely uses data in the `current` descriptor initialized as described in Section V. Next, in order to translate the virtual addresses of pages to be allocated, Colored Lockdown uses a kernel routine, namely `get_user_pages`. The `get_user_pages` routine represents an entry point for a number of page-wide kernel operations that can be selected via a `flag` parameter. When invoked with no flags, the function takes as input a range of (virtual) addresses and a task descriptor and returns an array of pointers to *page descriptors*. Each page descriptor corresponds to a page in the selected range. In Linux, the value of the pointer to a page descriptor is always a linear translation of the described page's physical address. Hence, knowing the pointer to the page descriptor for a page is equivalent to knowing its physical address. The `get_user_pages` logic is fairly complex, but since it is part of the kernel, it sits beyond our verification boundaries. As such, we have abstracted much of its functionality as follows.

```
1  long get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
       unsigned long start,unsigned long nr_pages, int write, int
       force, struct page **pages, int *locked)
2  {
3      struct page * page_ptr;
4      assert(nr_pages == 1);
5      assert(write == 0);
6      assert(force == 0);
7      assert(tsk == current);
8      assert(mm == tsk->mm);
9      page_ptr = __CPROVER_uninterpreted_void_ptr(tsk, mm, start);
10     __CPROVER_assume(page_ptr >= mem_map && page_ptr <  (mem_map +
           MAX_PAGES));
11     __CPROVER_assume(((unsigned long)page_ptr & ((1 << sizeof(
           struct page)) - 1)) == 0);
12     *pages = page_ptr;
13     return 1;
14 }
```

First, a set of asserts on the passed parameters is performed (lines 4–8), to verify the expected value of a number of parameters when `get_user_pages` is called within Colored Lockdown. An uninterpreted function is used (line 9) to construct a valid return value for the routine. In general, the returned value can be any pointer to a `page_struct` object (line 11) with a value between `mem_map`[4] and the end of that portion of kernel memory where page descriptors are stored (line 10). For any specified parameter value of `tsk`, `mm` and `start`, the same page pointer should be returned by successive invocations of `get_user_pages`. Hence the use of an uninterpreted function at line 9. The derivation of physical addresses from page descriptor pointers follows a similar logic.

In the following step the availability of colors is checked. The check is performed using an internal structure that remembers the color associated to each page to be allocated. The step is performed with minimal kernel interaction. When a "conflict" page is encountered, i.e. a page with an unavailable color, the module selects the closest available color. It also marks the internal descriptor for the page to reflect the change. At this stage, no recoloring is performed, hence no final changes are carried out.

If the procedure has determined that there exist enough available space to perform cache allocation, the following actions are performed. First, the module performs re-coloring of all the conflict pages. Second, it executes a cache lockdown operation on each line of each profile page. In the considered architecture, the lockdown is performed using a dedicated assembly instruction, namely `DCBTLS`[5]. In order to perform verification, however, we also update the status of the structure used to model the cache. More in detail, we invoke the `lock_line` procedure on each address corresponding to every line in a page being allocated. The `lock_line` procedure is reported below.

```
1  void lock_line(void * addr)
2  {
3    unsigned int index = get_index(addr);
4    unsigned int way = nd_int();
5    __CPROVER_assume(way >= 0 && way < CACHE_ASSOC);
6    __CPROVER_assume(!cache[index][way].locked);
7    cache[index][way].addr = addr;
8    cache[index][way].locked = 1;
9  }
```

The procedure is invoked on physical addresses, hence it is easy to calculate the cache index of the line, i.e. the cache set where the line will map (line 3). Since no specific cache replacement policy is assumed, the way selected for the allocation is generated as a non-deterministic integer (`nd_int()`, line 4) between 0 and the number of available ways (line 5). The ways where a line has been previously locked in the same set are excluded (line 6), as per assumed hardware behavior. Finally, with selected set/way, line locking is carried out as in lines 7 and 8.

To complete the verification, after Colored Lockdown is invoked, we check that: (i) every physical address (at the granularity of single cache lines) in pages to be allocated, as per the profile, can be found in our cache structure; and that (ii) no more locked lines than what specified in the profile is marked as locked.

VI. Evaluation

In this section, we provide a brief evaluation of the time required to perform verification using the proposed approach. The evaluation has been performed under two memory/cache layout scenarios using CBMC version 5.2 on a workstation machine featuring a 28-core Intel Xeon E5-2658 CPU running at 2.10 GHz with 32 GB of RAM. Unfortunately, CBMC only uses only one core and it is not possible to parallelize the verification effort due to the large amount of memory required to acquire each sample.

In the first scenario, we consider a 32-bit system ($B_w = 32$) with the following memory layout: memory pages of size 256 bytes ($P_s = 8$); a cache line size of 64 byte ($O = 6$); and a way size of 512 bytes ($I = 3$), so that each cache way can entirely hold 2 memory pages. We study the length of the verification for an increasing number of profile pages and cache associativity. Moreover, we set the timeout for the verification to 2 hours. The results for this setup are reported in Figure 1.

[4]In a Linux kernel, this symbol represents the beginning of the array of page descriptors.

[5]This instruction is common to PowerPC-based platforms, such as Freescale MCPxxx and QorIQ P40xx platforms.

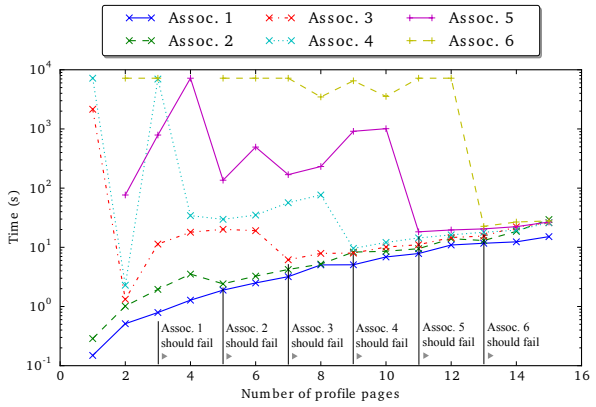


Fig. 1. Verification runtime for scenario: $P_s = 8, B_w = 32, O = 6, I = 3$ and associativity $W \in [1, 7]$.

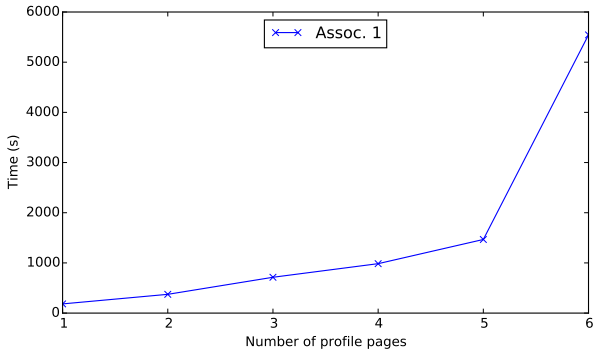


Fig. 2. Verification runtime for scenario with $P_s = 12, B_w = 32, O = 6, I = 10$ and $W = 1$.

In the figure, we use logarithmic scale to visualize in a compact way the runtime of the considered scenarios. As can be seen, the verification runtime can require from few milliseconds to entire hours, depending on the complexity of the system. For a low number of pages and higher associativity, we consistently observe peaks in execution time. We believe that these peaks originate from the increased flexibility of in-cache placement, which negatively impacts the size of the state space. In general, as the number of pages is incremented with a fixed associativity, the increment in runtime follows a regular trend and is exponential in time. Intuitively, this arises from the exponential increase in state space size to be explored by CBMC. It can also be noted that the verification time sharply decreases in those instances of verification that are not supposed to succeed. These cases, highlighted in the figure, correspond to those setup where the cache space is insufficient to carry out allocation, and where verification fails as it should. In this cases, CBMC stops after encountering a verification counter-example, hence it does not perform a complete exploration of the state space. Unfortunately, cases beyond associativity 6 consistently timeout in our evaluation.

In a second scenario, we evaluate the verification time for a more complex memory/cache layout by fixing the associativity to 1 and varying the number of pages. We consider a 32-bit system with 4 KB memory pages ($P_s = 12$), 64 byte cache line size ($O = 6$), and a way size of 64 KB ($I = 10$). In this layout, a single cache way can contain up to 16 memory pages. The results are depicted in Figure 2.

As shown in the figure, a sharp increment in runtime is observed at 6 profile pages. Although not included in the graph, any verification attempt for pages beyond that boundary runs longer than the selected 2 hours timeout threshold. Nonetheless, even with the current approach, verification is feasible on a general-purpose machine for a limited number of profile pages.

VII. Conclusions and Future Work

In this work, we focused our attention on verification of kernel-level cache management logic. We have demonstrated that it is possible to perform verification by reasoning directly on the system-level C code of the target module. Key properties for advanced kernel-level features were verified in a modular way with respect to the rest of the OS logic. In our approach, we relied on bounded model checking via CBMC. The work opens many possibilities for improvement. As a part of our future work, we will investigate how to include elements of deductive verification to allow verification of more complex scenarios. Additionally, we will attempt verification of complementary real-time hardware kernel logic with the goal of establishing an industry-ready, verified real-time resource management framework.

References

[1] T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: Static driver verification with under 4% false alarms. In *Formal Methods in Computer-Aided Design (FMCAD)*, Oct 2010.

[2] T. Ball and S. K. Rajamani. The slam toolkit. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, 2001.

[3] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[4] S. Boldo and T. Nguyen. Hardware-independent proofs of numerical programs. In *NASA Formal Methods Symposium*, 2010.

[5] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, Lecture Notes in Computer Science. Springer-Verlag, March–April 2004.

[6] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7), July 2008.

[7] S. Duprat, P. Gaufillet, V. M. Lamiel, and F. Passarello. Formal verification of SAM state machine implementation. In *Embedded Real Time Software and Syst. (ERTSS)*, May 2012.

[8] R. A. B. e Silva, N. N. Arai, L. A. Burgareli, J. M. P. de Oliveira, and J. S. Pinto. Formal verification with frama-c: A case study in the space software domain. *IEEE Transactions on Reliability*, 65(3):1163–1179, Sept 2016.

[9] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson. Rtos support for multicore mixed-criticality systems. In *IEEE Real Time and Embedded Technology and Applications Symposium*, April 2012.

[10] N. Kosmatov and J. Signoles. Frama-c, a collaborative framework for c code verification: Tutorial synopsis. In *International Conference on Runtime Verification*, Cham, 2016. Springer International Publishing.

[11] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society, April 2013.

[12] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun. WCET(m) estimation in multi-core systems using single core equivalence. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 174–183, July 2015.

[13] R. Mancuso, R. Pellizzoni, N. Tokcan, and M. Caccamo. WCET derivation under single core equivalence with explicit memory budget assignment. In *Euromicro Conference on Real-Time Systems (ECRTS), Dubrovnik, Croatia*, 2017.

[14] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. sel4: From general purpose to a proof of information flow enforcement. In *Security and Privacy (SP), 2013 IEEE Symposium on*, May 2013.

[15] V. Prevosto, J. Burghardt, J. Gerlach, K. Hartig, H. Pohl, and K. Voellinger. Formal specification and automated verification of railway software with frama-c. In *IEEE International Conference on Industrial Informatics (INDIN)*, July 2013.

[16] A. Puccetti. Static analysis of the xen kernel using frama-c. *Journal of Universal Computer Science*, 16, 2010.

[17] V. V. Rubanov and E. A. Shatokhin. Runtime verification of linux kernel modules based on call interception. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 180–189, March 2011.

[18] L. Sha, M. Caccamo, R. Mancuso, J. E. Kim, M. K. Yoon, R. Pellizzoni, H. Yun, R. B. Kegley, D. R. Perlman, G. Arundale, and R. Bradford. Real-time computing on multicore processors. *Computer*, 49(9):69–77, Sept 2016.

[19] S. F. Siegel and T. K. Zirkel. TASS: The toolkit for accurate scientific software. *Mathematics in Comp. Science*, 5(4), 2011.

[20] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. MERASA: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010.

[21] V. Wiels, R. Delmas, D. Doose, P.-L. Garoche, J. Cazin, and G. Durrieu. Formal verification of critical aerospace software. *AerospaceLab Journal*, May 2012.

[22] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014.

[23] 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 IEEE 19th*, pages 55–64, April 2013.

The case for limited-preemptive scheduling in GPUs for real-time systems

Roy Spliet, Robert Mullins
Department of Computer Science and Technology, University of Cambridge

***Abstract*—Many emerging cyber-physical systems, such as autonomous vehicles, have both extreme computation and hard latency requirements. GPUs are being touted as the ideal platform for such applications due to their highly parallel organisation. Unfortunately, while offering the necessary performance, GPUs are currently designed to maximise throughput and fail to offer the necessary hard real-time (HRT) guarantees.**

In this work we discuss three additions to GPUs that enable them to better meet real-time constraints. Firstly, we provide a quantitative argument for exposing the non-preemptive GPU scheduler to software. We show that current GPUs perform hardware context switches for non-preemptive scheduling in 20-26.5μs on average, while swapping out 60-270KiB of state. Although high, these overheads do not forbid non-preemptive HRT scheduling of real-time task sets. Secondly, we argue that limited-preemption support can deliver large benefits in schedulability with very minor impact on the context switching overhead. Finally, we demonstrate the need for a more predictable DRAM request arbiter to reduce interference caused by processes running on the GPU in parallel.

I. Introduction

An important class of cyber-physical systems now demand both significant compute and hard real-time (HRT) support. A prime example is the autonomous vehicle, where low-latency engine control systems are combined with time-critical AI classification and decision making procedures. These classification problems are solved using massively parallel algorithms such as neural networks [17]. From both a cost and performance-per-watt perspective it is attractive to offload these problems to massively parallel accelerators like GPUs. NVIDIA's introduction of Drive PX computers for assisted- and autonomous driving [2] are evidence of the shift of GPUs towards the domain of safety-critical HRT systems.

GPUs are designed with some real-time principles in mind, for example to resolve contention for the DRAM bus such that it never leads to a flickering image. Unfortunately, these real-time provisions are not applicable to the emerging cyber-physical use-cases. Instead, there is a strong desire to bound execution- and response time of GPU compute workloads.

A prerequisite to bound the worst-case response time of HRT tasks and to determine schedulability is a thorough understanding of the task scheduling policy. NVIDIA GPUs allow FIFO scheduling of kernels plus limited (sparsely documented) support for prioritising some kernels over others within the same hardware context [6]. Additionally, hardware supports non-preemptive context-switching between processes as a means to provide security mechanisms like per-task virtual memory spaces. Unfortunately, the criteria used for scheduling processes are unknown and not under the control of system developers. To achieve HRT scheduling on GPUs, systems must instead introduce a software abstraction layer [11], [14], [15], [16], [22]. These systems add overhead and force all tasks in a single hardware context, sacrificing inter-task protection mechanisms. Furthermore, their non-preemptive scheduling still imposes large worst-case blocking times on task sets, reducing HRT schedulability.

In this paper we present a case for three changes in GPU architecture. Firstly, we argue that exposing control over the current non-preemptive GPU context switching mechanisms to systems developers can facilitate low-overhead HRT task scheduling while providing desirable security mechanisms. From measurements we observe an average context switching time on NVIDIA GPUs of $20-26.5\mu$s. Using these numbers, we demonstrate the schedulability properties of random task sets under overhead-aware non-preemptive earliest-deadline first (npEDF) scheduling. Secondly, we motivate from an HRT perspective the proposal of Tanasic et al. [24] to perform context switches on the boundary of a work-group (SM draining) rather than the compute kernel. Measured by the schedulability of randomly generated task sets under limited-preemptive EDF scheduling, we show that this solution provides a good trade-off between blocking time and context switching overhead. By contrast, we show that fully preemptive scheduling on GPUs will perform similar or worse than non-preemptive scheduling as a result of high expected context switch overheads when exposed to task sets of the same parameters. Finally, we show the need for a predictable and analysable DRAM subsystem to provide optimistic bounds on the latency of GPU compute workloads. Using our measurement set-up, we expose interference between display scan-out and context switching by showing that increasing the bandwidth demand of scan-out increases the worst-case context switch time from $3.7\times$ average to more than $5.5\times$.

II. Background and related work

A. GPU nomenclature

Developers implement their data-parallel algorithms in one or more *compute kernels*, following the *Single Program, Multiple Data streams* (SPMD) programming model. A kernel typically describes the transformations on a single data element in the data stream. Hardware will spawn one thread or *work-item* for every data element.

Following OpenCL nomenclature, work-items are grouped into *work-groups*. On NVIDIA hardware a work-group consist

of multiple 32-thread groups called *warps* (AMD: *wavefronts*). Each warp will typically be executed in a SIMD fashion.

To execute SPMD programs, NVIDIA hardware implements the *Single Instruction, Multiple Threads* (SIMT) execution model [18] following a hierarchical structure. At the bottom level, a *Streaming Multiprocessor* (SM) contains many computational *cores* on which work is dispatched by *warp schedulers*. A warp scheduler issues one or two SIMD instructions per clock cycle at a warp granularity, temporally interleaving the instructions of multiple warps to minimise hardware stalls. A large register file ensures that the warp scheduler can interleave instructions of warps from the same hardware context with zero overhead. Further up the hierarchy, one or more SMs are contained within a *Graphics Processor Cluster* (GPC). A GPU contains one or more GPCs.

B. Non-preemptive context switching on NVIDIA GPUs

On current NVIDIA hardware a context switch is performed by dedicated custom "Falcon" microcontrollers [28], [29]: one at the top-level called *FECS* (Front-End Context Switch) and one per GPC called *GPCCS* (GPC Context Switch). Each microcontroller is connected to a set of FIFO buffers, used to coalesce register read/write actions to memory to improve DRAM efficiency. At the top-level, a hardware scheduling unit triggers context switches by notifying FECS.

When FECS receives a context switch request, it configures all execution engines (SMs, rasterisers etc.) to pause after finishing the currently running compute kernel. Once engines are paused, it notifies each GPCCS to swap state. FECS and GPCCS microcontrollers proceed by writing the MMIO address of every register that must be saved to their FIFOs. After all FIFOs are drained and their register values stored, the reverse process is initiated to restore registers of the next context. Finally the GPCCSs signal completion, after which FECS resumes execution of all engines.

Tanasic et al. [24] explore implementations for full- and limited-preemptive context switching on NVIDIA GPUs. They evaluate their approach using an in-house simulator by measuring average context switching times for several benchmarks. We extend this work by presenting a baseline for context switching under non-preemptive scheduling (henceforth "non-preemptive context switching") on commodity hardware and evaluating preemption models from an HRT point of view.

C. Real-time considerations for GPUs

We consider three key differences between popular GPU architectures and the CPU: the SIMT execution model, the lack of direct I/O access to external devices from GPU compute cores, and the absence of shared memory resources between different compute kernels.

SIMT execution allows GPUs to achieve high resource utilisation by executing the many work-items of a compute kernel on all available GPCs in parallel. We limit ourselves to the base case of temporal multitasking, in which case GPUs are best analysed as a *uniprocessor* where each compute kernel represents a task in the system. Limited support exists for spatial multitasking of kernels within a context [6], but in the absence of scheduler implementation details we consider this a throughput optimisation without analysable worst-case response time benefits.

NVIDIA GPUs will never encounter context switches due to self-suspending jobs. In traditional systems we can categorise self-suspensions in three classes: jobs waiting to be granted access to a shared resource, jobs blocked on I/O and jobs explicitly yielding their core. Alglave et al. [5] show that sharing resources between different jobs on a GPU is deemed infeasible by the weak memory consistency model found on current GPUs. I/O blocking is impossible because the GPU is a slave device without direct access to external devices. Finally, NVIDIA GPUs do not support a yield instruction. As a consequence of not encountering self-suspension we can bound the number of context switches in a system.

D. System model

In this work we consider the *periodic task model* [19] with *implicit deadlines*. For limited-preemptive execution, this model defines a set of tasks τ of size n where each task τ_i is described by a three-tuple (c_i, p_i, q_i). During execution, each task releases a series of *jobs* $J_{i,k}$. The *period* p_i describes the time between two successive job releases from the same task. In an implicit deadline system, a job's absolute deadline equals its launch time plus p_i. The *cost* c_i is the worst-case execution time (WCET) of a job. The final parameter q_i describes the maximum preemption delay or "non-preemptive blocking period". The utilisation of a task $U_i = c_i/p_i$ and the utilisation of a task set $U_\tau = \sum_{1 \leq i \leq n} U_i$.

We limit our experiments to EDF scheduling [19]. Although not implemented by commodity GPUs, EDF's optimality among both preemptive- and non-preemptive non-idling uniprocessor schedulers [12] removes a factor of uncertainty from the cause of a task set's non-schedulability. This results in a more accurate demonstration of the influence of the context switch times in our experiments.

Two concepts underlie EDF schedulability analysis. Firstly, the *critical instant* is the instant for which a task's response time is maximised [19]. For preemptive EDF this instant corresponds with the *synchronous arrival sequence*, releasing the first job of each task at time $t = 0$ and each subsequent job $J_{i,k}$ at time $t = k * p_i$. Secondly, Baruah et al. [7] define the concept of *demand bound* as the sum of the cost of all *jobs* in the critical instant whose absolute deadline is on or before t. We define $h(\tau_i, t)$ as the function returning this bound for task τ_i.

Building on this work, Baruah [8] proved that under EDF scheduling, limited preemptive (implicit deadline) task sets are not schedulable iff:

$$\exists t : 0 \leq t : \sum_{i=1}^{n} h(\tau_i, t) > t$$

or there is a $\tau_j, 1 \leq j \leq n$, and

$$\exists t : 0 \leq t < p_j : q_j + \sum_{i=1, i \neq j}^{n} h(\tau_i, t) > t$$

NVIDIA GeForce	SM #	GPC MHz	DRAM GiB/s	State KiB	Measured time (μs) Min	Avg	Max	Avg. BW util GiB/s	%
GT 710	1	953	14.4	63.9	9.2	21.5	80.1	2.83	19.6%
GT 640	2	901	28.5	68.2	13.6	26.5	43.7	2.45	8.6%
GTX 650	2	1058	80.0	68.2	12.7	23.2	36.0	2.71	3.4%
GTX 780	12	992	288.4	268.6	9.7	20.0	28.6	13.76	4.8%

TABLE I
MEASURED CONTEXT SIZE AND SWITCHING OVERHEAD

For schedulability analysis of non-preemptive tasks, we can define $\forall i \in [1,n], q_i = c_i - 1$, resulting in the original npEDF schedulability conditions ([12], [13]).

To date, the best algorithm to bound the set of relevant values for t is Zhang et al's QPA [30]. Short's [23] *lpQPA-LL* extends QPA with schedulability analysis of implicit-deadline periodic task sets under limited-preemptive EDF.

Under EDF scheduling, the number of context switches is upper bound by two per job [10]. The rationale is that a reactive implementation of this policy only takes decisions on two types of events: job release and job completion. For non-preemptive EDF, scheduling decisions caused by job releases are postponed until after completion of the current job. This tightens the upper bound to one context switch per job.

III. CONTEXT SWITCHING OVERHEAD

To make substantiated claims about the effectiveness of preemption models for GPUs, in this section we present the results of measuring context size and switching time on NVIDIA GPUs. By manipulating measurement conditions, we also demonstrate the effect of performance interference on worst-case context switch times, motivating further research in predictable DRAM subsystems for GPUs.

A. Measurement set-up

In this experiment we measure the size and switching time of non-preemptive contexts on several NVIDIA Kepler generation (2012-2014) graphics cards. Measurement is performed by a modified context switching firmware. The nature of our changes mandate the use of the open source "nouveau" driver for NVIDIA graphics cards [1] rather than the official driver. Source code and acquired data is available at https://github.com/RSpliet/RTGPU-Preempt.

We modify the FECS firmware to report context switching time in an available scratch register. This time spans from the moment all GPCs are paused to the moment they resume. We measure the context size and switching times using an instrumentation tool built using the envytools suite.

The timer used for this measurement has a granularity of 32ns. Our firmware modifications increase the runtime of a context switch by two register read operations. Based on 1,064,960 samples we determine that these operations skew our measurement by 160-224ns, averaging at 176ns.

GPUs are connected to a monitor operating at 1600x1200@60Hz. To trigger context switches, we run two generic workloads in separate contexts (XFCE on Xorg, windowed OpenArena @1024x768). The choice of workload should have minimal effect on the measured overheads, as all SMs are paused during the measured interval. We use our instrumentation tool to obtain 20 million samples per GPU.

B. Results

The fifth column in Table I lists the size of the state that needs to be stored to memory on a non-preemptive context switch. This state, significantly larger than that of a modern CPU, includes OpenGL/CUDA/OpenCL configuration, hardware settings, a pointer to the top-level page-table, and many other undocumented pieces of information. The contents of the register- and local-memory file are not included.

Such large state results in observed context switch times in the order of tens of microseconds. Our measured average context switch time (column 7) corresponds with NVIDIA's claim [26] of $\sim$25μs for the Fermi-generation of graphics cards (2010-2012). Such overhead clearly needs to be accounted for when performing schedulability analysis.

Experiments with lower GPC clocks, leaving all other clocks (including the DRAM interface) unaltered, reveals that average context switch time increases. This suggests that the process of context switching is not solely memory bound. However, the observed worst-case context switch times on the low-end GeForce GT710 are slightly lower ($<$5%) when the GPC clock is reduced by 15%. This worst case overhead reduction rules out the theory that context switch is compute bound in the worst case. Instead, data indicates that higher worst-case context switch times correlate with lower DRAM bandwidth. We will present further evidence of context switching being memory bound in the worst case in Section III-C.

Figure 1 shows a logarithmic histogram of samples for the GeForce GT710, displaying the extent to which our maximum sample introduces pessimism to schedulability analysis. We observe that the vast majority of the samples lie around the average of 21.5μs, whereas merely $\sim$0.3% of the samples lie in the tail of the measurement. The observed maximum is $\sim$3.7$\times$ average.

In the next section we demonstrate how interference affects the samples in this tail. In the light of these results we discuss the limitation of empirical measurements.

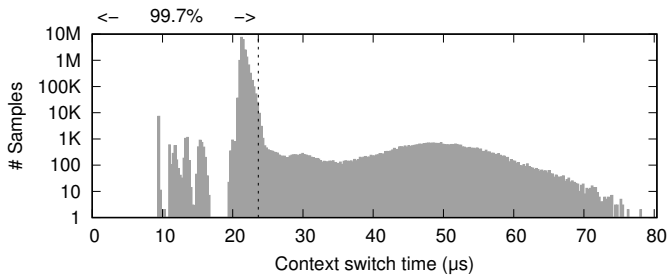


Fig. 1. Histogram of context switching overhead on NVIDIA GeForce GT710

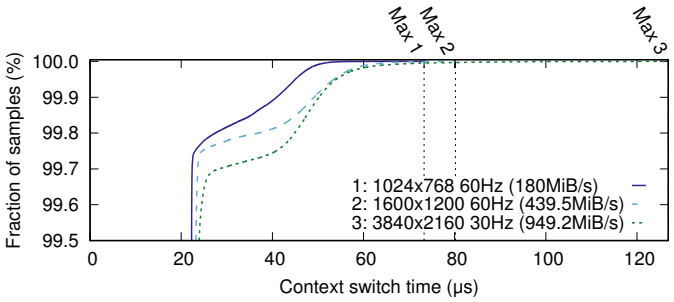


Fig. 2. Cumulative histogram of context switching overhead on GeForce GT710

C. Interference effects

To demonstrate interference within a GPU we repeat the experiment from Section III-B with different display resolutions. Figure 2 shows a cumulative histogram displaying the top 0.5% samples of context switch times of this experiment. From this graph we observe that increasing the required bandwidth for scan-out has a strong negative effect on the observed worst-case context switching overhead.

This interference is caused by sharing the DRAM subsystem between multiple workloads. If we consider a DRAM hierarchy, we find one or more channels on the top level. Each channel has a data bus to its RAM chips. If two memory operations transfer data from/to the same channel, these requests need to be serialised by an arbiter. This arbiter implements a prioritisation policy that makes a trade-off between performance and latency. If this policy is predictable it could be possible to determine a worst-case latency on individual memory requests, but unfortunately the prioritisation policy of GPU memory controllers is unknown.

Scan-out is merely one example of a GPU subsystem that requires access to DRAM in parallel with context switching. Other examples include DMA transfers and video decoding. Indeed, our observations on interference give reason to believe that e.g. the proposal of Verner et al. [25] to overlap DMA transfers with execution is likely to decrease response time predictability unless measures are taken to account for DRAM interference. Without analysable architectures and models, it is impossible to use quantitative measurements like these to distinguish between the worst-case execution time of a workload and its worst-case response time. This results in pessimistic GPU timing analysis.

In the next section we show how measured and extrapolated context switch times affect schedulability. We use these results to motivate further research in GPU preemption models.

IV. Schedulability analysis

To illustrate the effects of context switching overheads on schedulability, we performed a schedulability analysis, comparing non-preemptive, limited-preemptive and full-preemptive EDF. Next we explain how these scheduling policies map to microarchitectural solutions.

A. Models

Based on measured context switch overheads for non-preemptive execution on NVIDIA GeForce GT640, similar in specifications to the embedded Tegra K1 SoC, we extrapolate parameters for EDF and lpEDF. Resulting estimates are summarised in Table II.

For these estimates we make two simplifying assumptions. Firstly, we disregard cache-related preemption delays as they depend too much on the application and GPU microarchitecture to allow substantial claims. Secondly, divergence between warp schedulers will cause some SMs to wait idle for the last to finish. This idle time negatively affects the WCET of jobs. However, without knowledge of the scheduling policy implemented within the warp-schedulers, we cannot determine a bound on the divergence of warps in flight. This prevents us from modelling this effect in our analysis.

Scheduler policy	State (KiB) Ctx	Reg	Local	Total	Time (μs) Avg	Max	Preempt /job [10]
EDF	68.2	512	96	676.2	263	434	$\times 2$
lpEDF	68.2	0	0	68.2	27	44	$\times 2$
npEDF	68.2	0	0	68.2	27	44	$\times 1$

TABLE II
Parameters for schedulability analysis

Non-preemptive scheduling is currently implemented on NVIDIA GPUs. For non-preemptive EDF analysis we inflate c_i with the measured cost of one context switch.

Limited-preemptive scheduling applies to *SM draining* [24], a hardware solution that allows compute kernels to be preempted on the boundary of a work-group. At these boundaries the register and local memory contents do not need to be preserved, hence the state size and context switch time is estimated equal to that of the non-preemptive case. We account for this by inflating each task's cost by $2\times$ the measured context switching overhead.

For (full-)preemptive scheduling we must account for the larger context required to preserve register and local memory contents. To estimate the context switch time, we assume a linear correlation with the context size. Despite evidence that the DRAM subsystem provides more efficiency for bigger transfers on average [24], we cannot make optimistic assumptions for the worst-case without further research. For preemptive scheduling we inflate each task's cost with $2\times$ the projected context switching overhead.

B. Measurement set-up

For each utilisation $U \in (0.2, 0.21..1.0)$, we generated 100,000 implicit-deadline periodic task sets. Tasks have a period between 1,000 and 15,000 (μs), modelling kernels across the range of costs observed by Tanasic et al. [24]. Utilisation is randomly assigned to each task with a uniform distribution using the UUniFast algorithm [9].

Schedulability tests are performed using Brandenburg et al's schedcat, modified to support lpQPA-LL. [23] For limited preemption, we set $q_i = c_i / rand(5, 500)$, corresponding with $5 - 500$ work-groups per SM.

C. Schedulability

Figure 3 shows the result of this schedulability experiment when generating task sets of two tasks. We draw two conclusions from this graph. Firstly, the large overheads we measured do not prevent HRT schedulability. However, the large projected overhead greatly reduces the value of full-preemptive scheduling. Assuming worst-case context switch times, we find a minimal benefit for task sets with $U_\tau \leqslant 0.71$. For full-preemptive scheduling to become feasible in a real-time GPU, the overhead must ideally be bound to a value close to the average projection.

Secondly, we see that a limited-preemptive scheduler can benefit from the combination of paying the context switching overhead of non-preemptive scheduling and achieving response times close to preemptive scheduling. In practice this

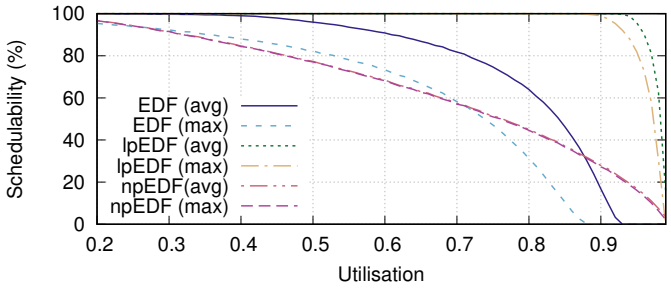


Fig. 3. Schedulability: 2 tasks, $c_i \in [1000, 15000]\mu$s

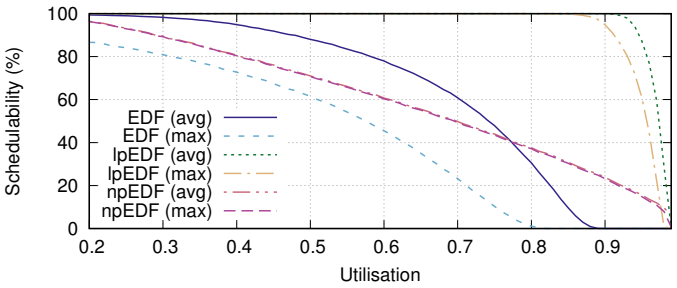


Fig. 4. Schedulability: 3 tasks, $c_i \in [1000, 15000]\mu$s

means that for the chosen parameters, we can schedule 99% of all task sets with $U_\tau \leqslant 0.89$ even when the worst case context switching time is assumed.

To show the influence of the task set size on schedulability, Figure 4 shows the results of this experiment when generating task sets of size 3. For preemptive-scheduling, the maximum projected overhead now outweighs the theoretical benefits of preemption completely. However, under limited-preemption EDF we would continue to be able to schedule many high-utilisation task sets.

D. Maximum blocking exploration

To demonstrate the impact of the maximum blocking parameter q, we perform an lpEDF schedulability analysis on random task sets where for a work-groups/SM ratio $w \in [2, 30]$, $q_i = c_i/w$. Figures 5 and 6 show the results of this analysis with w on the x-axis. On the y-axis we find the maximum utilisation for which $> 90\%$ (Figure 5) and $> 99\%$ (Figure 6) of the task sets are schedulable. We generated task sets with 3 tasks, for each task $c_i \in [1000, 15000]\mu$s

We see that even for two work-groups/SM, 99% of all task sets with $U_\tau < 0.29$ are schedulable. In Figure 4 we observe that this outperforms non-preemptive scheduling. Furthermore, for jobs containing 9 or more work-groups/SM, the figures demonstrate that the deciding factor for schedulability is not the preemption delay but rather the context switching overhead. For reference, 9 work-groups/SM corresponds to 122,880 work-items (e.g. a 351×351 image or matrix) on the largest Kepler generation GPU, the NVIDIA GeForce GTX780 TI. Such data sets are realistic for AI and computer vision workloads, supporting our claim that lpEDF kernel scheduling will result in increased GPU utilisation under HRT constraints.

V. Discussion and Future Work

A. DRAM interference

In Section III-C we explore the interference between context switches and display scan-out to show how contention for the DRAM subsystem reduces response time predictability. However, interference does not solely occur between these two tasks. Concurrent DMA- or video decoding activity further diminishes the worst-case latency of individual requests.

There are two known ways to mitigate this interference. Firstly, DRAM partitioning (e.g. bank privatisation [21]) could be applied to isolate subsystems. Although this has far-reaching consequences to the freedom a system has to allocate memory to each workload, it could serve as a way to reduce the worst-case interference on commodity hardware.

Secondly, designing a real-time DRAM request arbiter that prioritises requests based on their time of arrival and/or criticality level could make this interference predictable and analysable. Such arbiters have been studied for traditional multi-core architectures connected to DDR2 and DDR3 memory (e.g. [4], [20]), and prove effective at bounding the response time of individual request. Unfortunately, as improvements in DRAM latencies continue to stagnate and data buses are becoming wider, the bandwidth utilisation of memory controllers with such arbiters gets successively worse with each DRAM generation [27]. Future research should explore the design space of bound-latency high-throughput DRAM subsystems for GPUs under the constraints of present-day DRAM.

B. Task scheduling

In Section IV we describe how the limited-preemption model is a good fit for GPUs, assuming it reduces the maximum blocking time at a cost similar to that of non-preemptive context switching. One reason why this assumption could be too optimistic is that it disregards the context of subsystems that are irrelevant for most compute workloads, e.g. the rasteriser. The rasteriser keeps track of a lot of state during execution and does not appear to work on the granularity of work-groups. This raises questions on how the state of such fixed-function components should be treated in the preemptive execution models: Is it desirable to perform a context switch on these components in lock-step with the

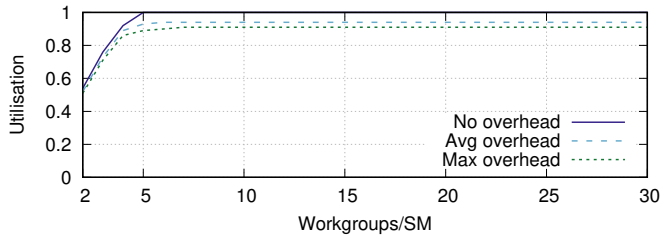


Fig. 5. Impact of work-groups/SM on 90% schedulability

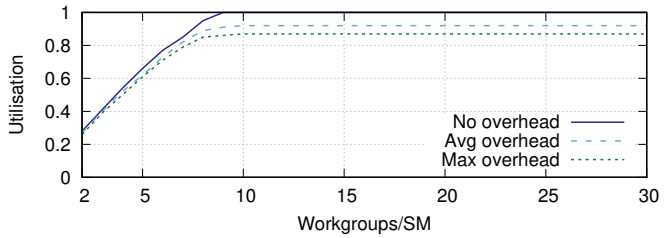


Fig. 6. Impact of work-groups/SM on 99% schedulability

compute subsystem? Can we find points in time at which these components have less state? Do we need to take the state of these components into account for (non-rendering) real-time compute workloads, or is it possible to take this out of the equation? Would this require the design of new, compute-oriented architectures?

Another avenue for research is the concept of GPU partitioning or "spatial multitasking" [3]. A partitioned non-preemptive GPU could permit a cost-based grouping of tasks, providing lower-latency guarantees to shorter tasks. Research could determine the architectural overhead of GPU partitioning, the implications to the context size, the schedulability implications for HRT workloads and the implications of DRAM-related interference on response-time analysis.

VI. Conclusion

In this work we have motivated the need for research in three areas of GPU design for real-time applications. Firstly, we show that it is possible to use existing non-preemptive EDF schedulability analysis to prove schedulability of task sets under the parameters we expect for massively parallel applications in the HRT domain running on contemporary GPUs. Prerequisite is that GPUs provide control over their non-preemptive task scheduler to software. We show that the measured average context switching overhead of 20-26.5μs has only a limited influence on schedulability. Secondly, we motivate research in limited-preemptive scheduling following the "SM draining" approach [24] to reduce the maximum blocking time of tasks while retaining the security benefits of task isolation. We show that this can result in significantly higher schedulability of task sets. Finally, we show that interference effects caused by contention for the shared DRAM subsystem has a negative effect on observed worst-case execution times of individual tasks. We suggest that further research should be conducted towards bound-latency DRAM request arbiters that enable more optimistic worst-case response times with minimal sacrifices to throughput.

Acknowledgements

We thank Andy Ritger (NVIDIA) and Joonas Lahtinen (Intel OTC) for their technical discussion, and Timothy Jones (University of Cambridge, Dept. CST) for his feedback.

References

[1] Nouveau: Accelerated Open Source driver for NVIDIA cards. URL: https://nouveau.freedesktop.org/wiki/.

[2] NVIDIA Tegra X1 - NVIDIA's New Mobile Superchip, 2015.

[3] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for GPGPU spatial multitasking. In *IEEE Int. Symp. on High-Performance Comp. Arch.*, pages 1–12, Feb 2012.

[4] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A Predictable SDRAM Memory Controller. In *Proc. of the 5th IEEE/ACM Int. Conf. on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '07, pages 251–256. ACM, 2007.

[5] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU Concurrency: Weak Behaviours and Programming Assumptions. *SIGPLAN Not.*, 50(4):577–591, March 2015.

[6] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *Proc. 38th Real-Time Systems Symp.*, pages 104–115, Dec 2017.

[7] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proc. 11th Real-Time Systems Symp.*, pages 182–190, Dec 1990.

[8] Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *17th Euromicro Conf. on Real-Time Systems*, pages 137–144, July 2005.

[9] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[10] A. Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Trans. on Software Engineering*, 21(5):475–480, May 1995.

[11] G.A. Elliott, B.C. Ward, and J.H. Anderson. GPUSync: A Framework for Real-Time GPU Management. In *Proc. 34th Real-Time Systems Symp.*, pages 33–44, Dec 2013.

[12] L. George, P. Muhlethaler, and N. Rivierre. Optimality and non-preemptive real-time scheduling revisited. Research Report RR-2516, INRIA, 1995. Projet REFLECS. URL: https://hal.inria.fr/inria-00074162.

[13] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *Proc. 12th Real-Time Systems Symp*, pages 129–139, Dec 1991.

[14] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A Responsive GPGPU Execution Model for Runtime Engines. In *Proc. 32nd Real-Time Systems Symp.*, pages 57–66, Nov 2011.

[15] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX ATC11*, page 17, 2011.

[16] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class GPU resource management in the operating system. In *USENIX ATC12*, volume 12, 2012.

[17] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada. An Open Approach to Autonomous Vehicles. *Micro, IEEE*, 35(6):60–68, Nov 2015.

[18] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March 2008.

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

[20] M. Paolieri, E. Quiñones, and F. J. Cazorla. Timing Effects of DDR Memory Systems in Hard Real-time Multicore Architectures: Issues and Solutions. *ACM Trans. Embed. Comput. Syst.*, 12(1s):64:1–26, Mar 2013.

[21] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Proc. of the 9th IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis*, pages 99–108, Oct 2011.

[22] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proc. of the 23rd ACM Symp. on Operating Systems Principles*, SOSP '11, pages 233–248. ACM, 2011.

[23] M. Short. Improved schedulability analysis of implicit deadline tasks under limited preemption EDF scheduling. In *ETFA2011*, pages 1–8, Sept 2011.

[24] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *ACM/IEEE 41st Int. Symp. on Computer Architecture (ISCA)*, pages 193–204, 2014.

[25] U. Verner, A. Schuster, and M. Silberstein. Processing Data Streams with Hard Real-time Constraints on Heterogeneous Systems. In *Proc. of the Int. Conf. on Supercomputing*, pages 120–129, 2011.

[26] C.M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU Architecture. *Micro, IEEE*, 31(2):50 –59, March-April 2011.

[27] Z. P. Wu, Y. Krish, and R. Pellizzoni. Worst Case Analysis of DRAM Latency in Multi-requestor Systems. In *Proc. 34th Real-Time Systems Symp.*, pages 372–383, Dec 2013.

[28] J. Xie. NVIDIA RISC-V Evaluation Story. 4th RISC-V Workshop, 2016. URL: https://www.youtube.com/watch?v=gg1lISJfJI0.

[29] Y.Fujii, T. Azumi, N. Nishio, and S. Kato. Exploring Microcontrollers in GPUs. In *Proc. 4th Asia-Pacific Workshop on Systems*, pages 2:1–2:6. ACM, 2013.

[30] F. Zhang and A. Burns. Schedulability Analysis for Real-Time Systems with EDF Scheduling. *IEEE Trans. on Computers*, 58(9):1250–1258, Sept 2009.

Scaling Up: The Validation of Empirically Derived Scheduling Rules on NVIDIA GPUs*

Joshua Bakita, Nathan Otterness, James H. Anderson, and F. Donelson Smith
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract— **Embedded systems augmented with graphics processing units (GPUs) are seeing increased use in safety-critical real-time systems such as autonomous vehicles. The current black-box and proprietary nature of these GPUs has made it difficult to determine their behavior in worst-case scenarios, threatening the safety of autonomous systems. In this work, we introduce a new automated validation framework to analyze GPU execution traces and determine if behavioral assumptions inferred from black-box experiments consistently match behavior of real-world devices. We find that the behaviors observed in prior work are consistent on a small scale, but the rules do not stretch to significantly older GPUs and struggle with complex GPU workloads.**

I. Introduction

Recent advancements in artificial intelligence and embedded computing have started to bring the revolution of self-driving vehicles closer to reality, but a multitude of unanswered questions still stand in their path to mass adoption. One key open question is *how good is good enough?* Recent fatalities [14, 18] have shown that the current standard of "good enough" falls short in more than one commercial system. To eliminate a subjective definition of "good enough", this paper envisions autonomous vehicle hardware eventually requiring *certification* for manufacturer, customer, and regulatory acceptance.

Unfortunately, the hardware increasingly used in vehicles and labs today to meet size, weight, and power (SWaP) requirements utilizes proprietary architectures with vague or underspecified public documentation. This presents a quandary to groups attempting to certify such systems. This issue is exemplified in systems based on NVIDIA's Parker system-on-a-chip (SoC). This SoC powers NVIDIA's TX2 development board as well as the NVIDIA's DRIVE PX AutoChauffeur and AutoCruise boards marketed towards autonomous vehicles [9]. Known users of this platform include Tesla's Autopilot 2.0 system [7]. Due to the TX2's public availability, recent work [1, 20] has focused on that board as a representative for NVIDIA's other, more tightly held, Parker SoC-based boards.

These embedded platforms contain the majority of their raw computing power in their graphical processing units (GPUs), so it becomes essential to thoroughly understand how they behave when multiple general purpose GPU (GPGPU) workloads share a single GPU. Danger lies in justifying the use of these components in a self-driving vehicle without reasoning from fundamental behaviors [13]. Making simulation-based statistical assertions about the overall observed lack of failures of a self-driving vehicle system cannot carry over to the real world. Systems must either be statically, provably safe or allowed to drive for millions of representative miles without demonstrating any error [13]. The infeasibility of the latter option leaves us with the former, and returns us to the importance of understanding GPU behavior in these systems.

Our prior work attempted to solve this problem by forming rules of behavior for *CUDA*, a common GPGPU programming API for NVIDIA GPUs. Unfortunately, safety-critical systems could not yet rely on our rules. We formulated the rules from empirical observation, but rigorous applicability of the rules remained untested. Our rules also suffered from fragility and limited scope. As new GPU architectures appear almost every other year and new processors based on those architectures appear every few months, the possibility of rigorously testing all these devices by hand becomes vanishingly small. A field dominated by rapid and regular change needs some automated method to validate past results on new or more complicated devices.

To emphasize this point, consider Fig. 1, which displays a GPU execution trace used in recent work [11]. Shaded rectangles represent GPU executions over time. The trace appears well-ordered and reasonably easy to step through by hand, given familiarity with prior work. Now take Fig. 2. This presents a trace from the same benchmark, but on a GPU with more compute capacity. A trained eye will pick out the subtly different rules in effect here, but even this simple modification tests the limits of empirical observation. Real-world executions can be far more complex. Fig. 3 provides an extreme example. It displays a trace from the execution of a randomly generated four-thread workload on a mainstream GPU. So many different interactions take place that even our graphing software struggles to cope.

No human can hand-validate what behavioral rules apply in traces like this. But comprehensive testing of our proposed rules requires the validation of these sorts of traces. To accomplish that, this paper introduces an automated rule-validation framework which provides a path to scalable, rigorous validation.

II. Background

Our solution builds on elements of NVIDIA GPUs, concepts from CUDA, and properties of a number of representative test platforms.

*Work supported by NSF grants CNS 1409175, CPS 1446631, CNS 1563845, and CNS 1717589, ARO grant W911NF-17-1-0294, and funding from General Motors.

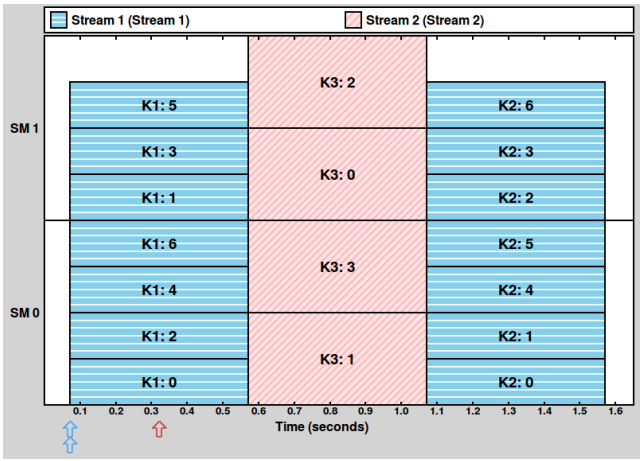


Fig. 1. EE queue benchmark trace on NVIDIA TX2

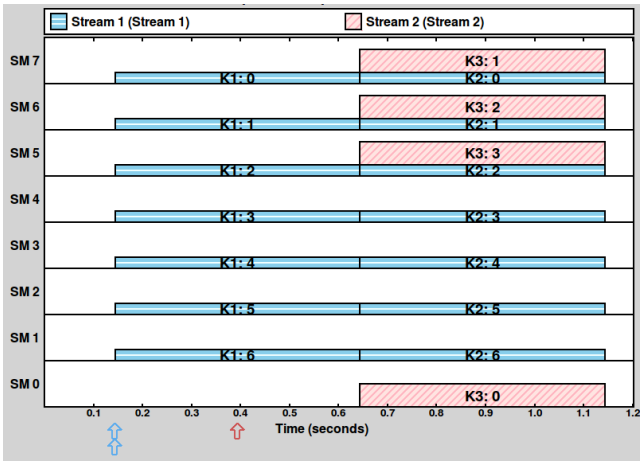


Fig. 2. EE queue benchmark trace on NVIDIA K5000

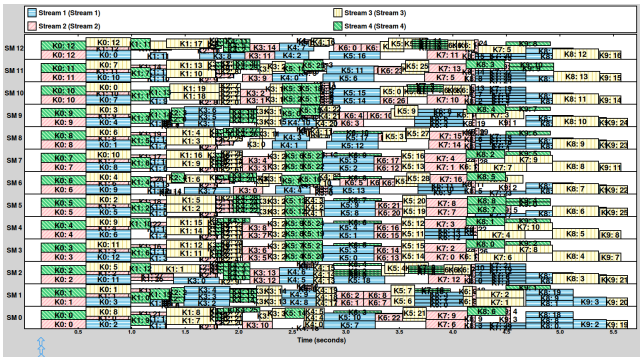


Fig. 3. Benchmark trace of random work on NVIDIA GTX 970

A. GPU and CUDA Fundamentals

A GPU is a highly parallel co-processor. Traditionally used for fast 3D scene rasterisation, recent years have seen GPUS used increasingly for general-purpose computing. For the NVIDIA GPUs considered in this paper, CUDA is the most popular API for creating and executing non-graphical computations on the GPU. Built as a C/C++ extension, valid CUDA code looks very similar to embedded C code. See Algorithm 1 for a loose example of vector addition in a CUDA program.

Some key GPU and CUDA terms used throughout this paper:

1) *CUDA Thread Block:* A group of GPU threads executing the same set of user-defined instructions in lockstep. This is the lowest-level GPU scheduling unit considered in this paper.
2) *CUDA Kernel:* A combination of instruction code and CUDA thread block specifications. Dispatched asynchronously by a user-space process.
3) *CUDA Stream:* A first-in-first-out (FIFO) work queue into which processes on the CPU can dispatch kernels.
4) *SM (Streaming Multiprocessor):* A subdivision of an NVIDIA GPU. Single thread blocks cannot be split across multiple SMs [10].
5) *EE (Execution Engine) Queue:* A special internal queue of kernels that our past work has defined to exist between CUDA stream queues and the actual GPU. Fig. 6 illustrates its location in the execution flow.

B. Test Platforms

To demonstrate the scalability and broad applicability of the framework presented in this paper, we test four generations of NVIDIA's graphics processors. The following list notes the specifications and release dates of the representatives from each generation:

Kepler The Quadro K5000 discrete GPU (Nov. 2012) with 8 SMs.

Maxwell The GeForce GTX 970 discrete GPU (Sep. 2014) with 13 SMs.

Pascal The GeForce GTX 1070 discrete GPU (June 2016) with 15 SMs and the Jetson TX2 (March 2017) with 2 SMs.

Volta The Titan V discrete GPU (Dec. 2017) with 80 SMs.

C. Related Work

Due to a lack of public documentation on how concurrent execution of GPU workloads behave, much past work in this area focuses on efficiently providing an exclusive locking mechanism for the GPU [4, 5, 6, 15, 16, 17, 19]. This effectively prevents any interference after lock acquisition (as processes only ever own an independent portion of the GPU) but may lead to powerful GPUs remaining underutilized.

Moving in a different direction, other work focuses on increasing utilization at the cost of predictability. For example, Zhong et. al.'s scheduling approach showed increased

Algorithm 1 Vector Addition Pseudocode from [20, p. 7].

```
1: kernel VECADD(A ptr to int, B: ptr to int, C: ptr to int)
   ▷ Calculate index based on built-in thread and block information
2:     i := blockDim.x * blockIdx.x + threadIdx.x
3:     C[i] := A[i] + B[i]
4: end kernel

5: procedure MAIN
   ▷ (i) Allocate GPU memory for arrays A, B, and C
6:     cudaMalloc(d_A)
7:     ...
   ▷ (ii) Copy data from CPU to GPU memory for arrays A and B
8:     cudaMemcpy(d_A, h_A)
9:     ...
   ▷ (iii) Launch the kernel
10:    vecAdd<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C)
   ▷ (iv) Copy results from GPU to CPU array C
11:    cudaMemcpy(h_C, d_C)
   ▷ (v) Free GPU memory for arrays A, B, and C
12:    cudaFree(d_A)
13:    ...
```

utilization, but adds overhead and may not fully account for potentially destructive interference between multiple concurrent GPU operations. [21] Several other works [21, 3, 5, 8] attempt to expose more degrees of scheduling freedom in CUDA by using software to approximate preemptive hardware.

Our recent work addresses an orthogonal problem. It attempts to address worst-case execution times (WCETs) in any GPU context by better understanding and leveraging existing undocumented GPU hardware behaviors to enable real-time systems. Our group has postulated scheduling rules for the Jetson TX1 [11, 12] and Jetson TX2 [1] development boards while also discovering some more generally applicable behaviors and pitfalls in CUDA [20]. This paper expands on that work.

III. Experimental Approach

Beyond rules, our past work also provides a GPU benchmarking framework that allows for sets of CUDA kernels to be run in a reproducible manner. The framework provides logs of these executions marking events such as a kernel dispatch or thread block end with high-precision timestamps. Our prior work visualized these traces for empirical analysis, but the traces also provide all the information necessary to enable an automated validation framework.

A. Validator Design

We use a state machine to validate if these logged behaviors adhere to what we expect from our rules. We prefer this approach over full-scale simulation due to its simplicity and applicability to our proposed rules. Some past work (namely GPGPU-Sim [2]) has applied a custom GPU simulator to confirm behaviors, but we find the inherent complexity of a full simulation too burdensome. Even the most recent simulators fall generations behind today's GPUs. Our approach instead takes a subset of the events recorded from actual executions and uses each event to trigger state transitions.

B. Sourcing Traces

The first step in our rule-validation approach is to parse a selection of the information logged by the benchmarking framework. From the original trace, we obtain a series of events that can trigger state transitions in our validator, and sort the events by timestamp. The current version of the validation tool focuses on the following events:

- Kernel launch start
- Kernel launch end
- Kernel end[1]
- Thread block start
- Thread block end

During parsing, we extract and attach contextual data about each event. For kernel events, that includes a list of child thread blocks and the ID of the associated CUDA stream. For thread block events, context includes the number of threads in the block, the parent kernel, and the SM used.

C. Building the State Machine

After preparing the series of events, the actual state machine can proceed. Our constructed state machine appears as a flow chart in Fig. 4, and builds off the scheduling rules postulated in our recent work [1]. This machine only validates a core, always-applicable subset of the full set of our published rules (6 of 16 rules). These chosen rules and labels have been reproduced from [1] in the following list:

G1 "A copy operation or kernel is enqueued on the [CUDA] stream queue for its stream when the associated CUDA API function (memory transfer or kernel launch) is invoked."

G2 "A kernel is enqueued on the EE queue when it reaches the head of its [CUDA] stream queue."

G3 "A kernel at the head of the EE queue is dequeued from that queue once it becomes fully dispatched."

G4 "A kernel is dequeued from its [CUDA] stream queue once all of its blocks complete execution."

X1 "Only blocks of the kernel at the head of the EE queue are eligible to be assigned."

R2 "A block of the kernel at the head of the EE queue is eligible to be assigned only if there are sufficient thread resources available on some SM."

Validation proceeds by processing execution events in chronological order. At each event, state updates and a validity check can occur on the path between states. Fig. 4 represents events in all-caps, states as vertical rectangles, updates with horizontal rectangles, and checks as diamonds. Validation succeeds or fails dependent on entry into the red, terminal failure state. Our Python implementation of this state machine and the event stream parser is open-source software available online[2].

[1]Pseudo-event; sometimes it is undesirable for a benchmark to perform a `cudaStreamSyncronize` to retrieve the actual kernel execution end time. In those cases, the parser uses the end time of the last thread block in the kernel as a substitute.

[2]See `https://github.com/JoshuaJB/cuda_scheduling_validator_mirror`

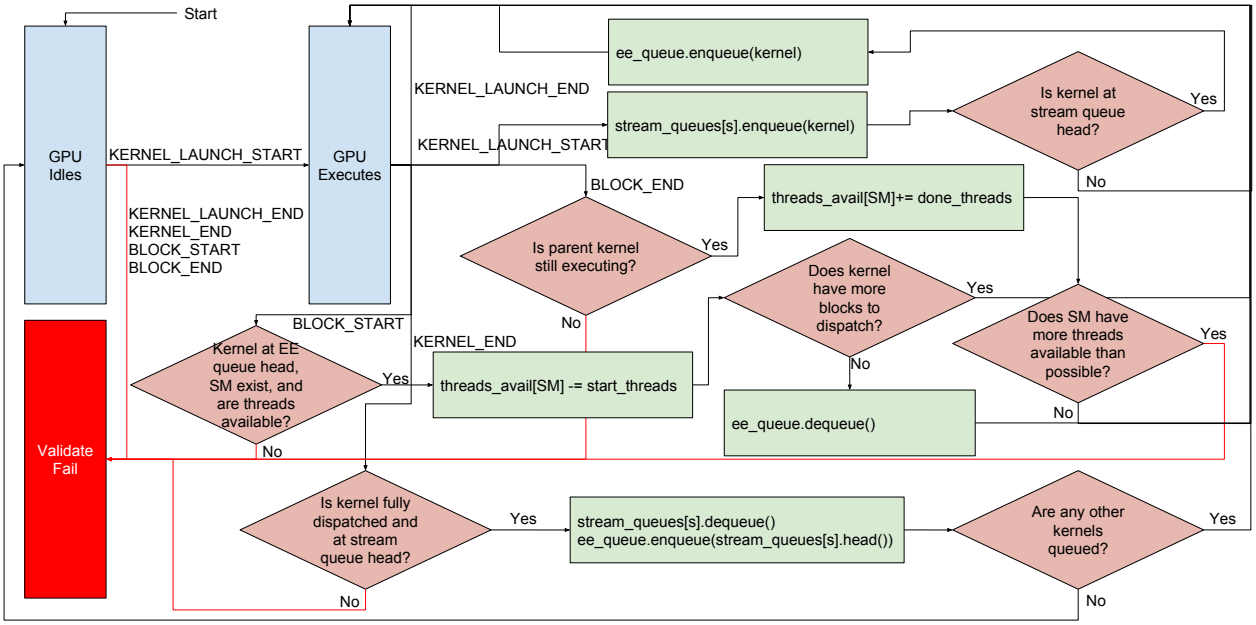


Fig. 4. State machine for rules G1-G4, X1, and R2 (all-caps annotations denote what events trigger each transition)

IV. Evaluation

We evaluated our framework by assuring that it properly categorized known good traces and known bad traces.

A. Validating Base Rules

We demonstrated that known correct traces do not falsely enter the validation failure state by applying the convenient fact that our selected scheduling rules mirror those first discussed in one of our papers from last year [11]. In that paper, we provided clear benchmark configurations to demonstrate each scheduling rule in action. For this paper, we ran those configurations again and confirmed that our automated validation of each of their traces succeeded. That verified that no single proper rule behavior would be flagged as incorrect by our framework

We demonstrated that rule violations produce validation failures in practice by testing handcrafted invalid traces. Some example modifications to previously correct traces involved swapping the execution order of two thread blocks, adding additional threads to a block, or creating timing abnormalities.[3] By testing all of the possible ways that each individual rule could fail, we assured that all real failures or combinations of multiple failures will be caught by the framework. Importantly, these invalid traces all originate from static modifications to previously generated valid traces - we never expect the GPU to directly generate an invalid trace.

B. Results on Maxwell, Pascal, and Volta

After using this approach to confirm that our validator behaves as expected for the specific platform analyzed by hand in our past work (the TX2), we expanded tests to cover all of the platforms detailed in Sec. II-B. We found our scheduling rules to be broadly applicable to GPUs running NVIDIA's Maxwell, Pascal, and Volta architectures. This encompasses all major NVIDIA GPUs released since late 2014. However, we ran into unexpected results when attempting to validate large, randomly generated workloads such as the one demonstrated in Fig. 3.

For example, on our GTX 970, only about 13% of 2,000 randomly generated 40-kernel tests passed validation. Upon further inspection, we found that the framework was correct; there appeared to be subtle violations of rule X1 (that only the head of the EE queue should be eligible for dispatching) recorded in the benchmark logs. However, the extent of this incorrect ordering never appeared to be more than a few microseconds. We currently hypothesize that these "violations" merely result from minor inaccuracies in our methods for recording timestamps. CUDA does not provide thread-block-level start or stop timestamps, so our benchmarking framework instead obtains these by reading a global GPU time register immediately on start and before end inside each thread block. We believe that momentary stalls or propagation delays may cause these reads to sometimes not perfectly correspond to actual block start and end times. Preliminary investigation into the traces that failed validation have found support for this hypothesis, but we hope to further analyze and clarify this behavior in future work.

C. Results on Kepler

While we found the rules seem to apply to the three-most-recent architectures considered, the older Kepler architecture behaved rather differently. The framework revealed that a rule violation occurred during validation of the trace from the benchmark designed to demonstrate rule G2 in action.

[3]To examine the exact violations added, the `tests/bad` directory in our online code repository contains all of these tests.

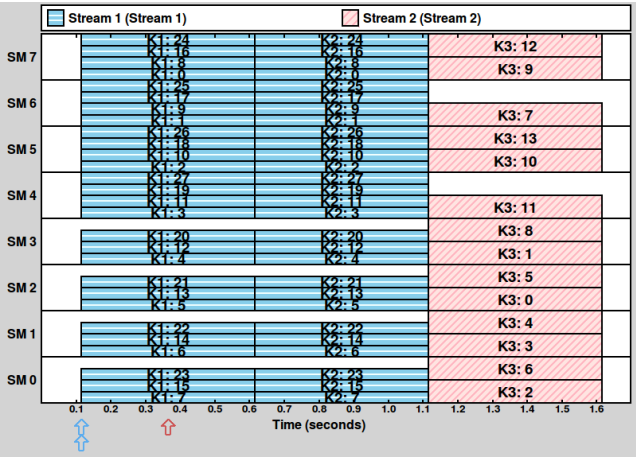


Fig. 5. EE queue benchmark trace on NVIDIA K5000 with thread block count adjusted to saturate the GPU

During the subsequent empirical investigation, it became clear that the validator correctly detected a rule violation. Kepler architecture GPUs do not follow the same rules as their successors. This peculiarity brings us back to Fig. 1 and Fig. 2.

Each graph plots GPU time on the horizontal axis for each SM plotted on the vertical axis. Every rectangle in the plot area represents a thread block running over some time period. The digit immediately prior to the colon in the label on each block indicates the block's kernel affiliation, and the number immediately after indicates the unique identification number of this thread block among all the kernel's thread blocks. Different colors indicate different CUDA streams, and the colored arrows along the horizontal axis indicate when kernels are released from our CPU process.

Now consider the simple plot presented in Fig. 1 generated from a benchmark trace on the Pascal-based TX2. To walk through the series of events represented by this plot, kernels K1 and K2 release into stream 1 before 0.1s. At this point, K1 and K2 are in the stream 1 queue and K2 is in the EE queue. K1 quickly dispatches all of its blocks, nearly occupies all of the GPU's resources, and leaves the EE queue. K3 then releases shortly after the 0.3s mark and immediately moves to the head of the EE queue. It goes on the EE queue before K2 because rule G4 has kept K2 blocked behind K1 in the stream 1 queue. (Fig. 6 illustrates this point in time.) Once K1 completes execution around 0.55s, it leaves stream 1 and allows K2 to enqueue on the EE queue behind K3. K3 then fully dispatches, saturates the GPU, and leaves the EE queue. At K3's completion point around 1.05s, K2 then has space for at least one of its thread blocks, begins execution, and runs to completion. In this plot, nothing unexpected occurs. All the rules hold and operate correctly.

Next, consider Fig. 2. We generated Fig. 2 using the same benchmark configuration as Fig. 1, but executed on the Quadro K5000 Kepler-architecture GPU. A similar pattern emerges up to just before timestamp 0.4s. As in the prior example, K3 is expected to be on the EE queue and K2 is

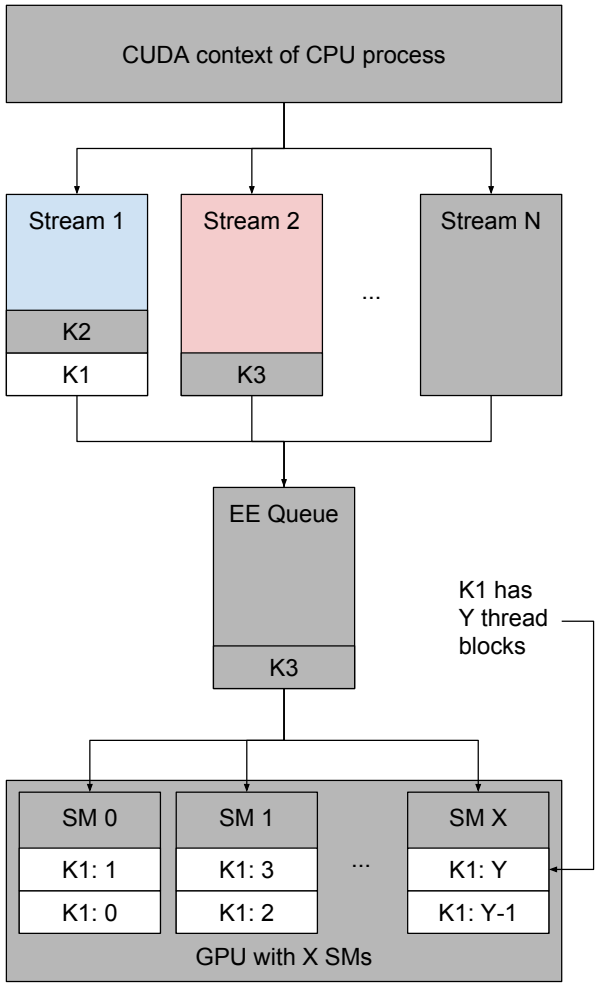


Fig. 6. Example of expected queue states for Fig. 1, Fig. 2, and Fig. 5 at time point 0.4s

expected to be blocked in the stream 1 queue. (See Fig. 6.) This is not consistent with what we observe on Kepler GPUs. If K3 were immediately moved to the head of the EE queue as we expect, it should start almost immediately on release due to availability of sufficient capacity for at least one thread block. Further analysis of execution traces reveal another interesting behavior that one would not observe simply by viewing a plot such as Fig. 2. At a nanosecond level, K2 starts before K3. This indicates that both rules G2 and G4 do not apply on Kepler GPUs.

The violation becomes much clearer with scrutiny of Fig. 5, which uses a similar configuration to Figs. 1 and 2, but with the number of thread blocks scaled up to compensate for the increased capacity of the K5000. This makes the flipped order of execution of K2 and K3 more visually apparent in contrast to Fig. 1.

The likely explanation for this behavior is that Kepler GPUs handle stream queues differently than later architectures. More specifically, the behavior can be explained if we

conclude that *rule G4 does not apply to Kepler* and if we change rule G2 to the following:

G2 (Kepler) A kernel is *dequeued from its stream queue* and enqueued on the EE queue when it reaches the head of its stream queue.

In essence, all the stream queues become aliases to only one single hardware queue. One can verify that these rules for Kepler work in at least some cases by stepping through Fig. 2 and Fig. 5. Each figure support our hypothesis by behaving as expected under the proposed rule variation.

V. Conclusion

A solid understanding of hardware scheduling behavior forms the essential foundation for any safety-critical system. To meet SWaP requirements, GPUs have emerged as one of the premier compute accelerators used in these platforms. Unfortunately, sufficient low-level documentation for these accelerators has not been forthcoming. Past solutions to this uncertainty have precluded parallelism via locking or introduced overheads without addressing questions about interference. Our recent work to cast light on GPU behavior rules has heavily depended on empirical observation. That approach quickly proves impractical and insufficient on large GPUs or in complicated test programs.

Our solution addresses that problem via an automated validation framework. By minimizing human input, our framework enables rigorous validation of scheduling rules across a multitude of complex platforms and workloads without the limitations of human error and inefficiency.

Future work could expand the state machine used for validation in this work to include the rules for priority streams, the NULL stream, copy operations, shared-memory blocking, and other yet-to-be codified rules. In the more distant future, one hopes that this framework could be modified to support traces from NVIDIA's native `nvprof` profiler, and thus be used to validate rule authority on execution traces from any CUDA program rather than just logs from the benchmarking framework.

Separately, we hope to further explore the irregularities causing complex tasks to fail validation. While we are reasonably confident that these unexpected results are being triggered by inaccurate timing information, we would like to rigorously confirm that there is no fundamental flaw in our rules.

The framework developed in this paper should enable future work relying on rule-based models of GPU behavior to both progress faster and yield more confident results. We need a comprehensive understanding of hardware to build safe autonomy, and this framework helps accelerate the assembly of that core foundation.

References

[1] T. Amert, N. Otterness, M. Yang, J. Anderson, and F. D. Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *RTSS 2017*.

[2] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed GPU simulator. In *ISPASS 2009*.

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

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

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

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

[7] Fred Lambert. Look inside Tesla's onboard Nvidia supercomputer for self-driving. *Electrek*, 2017.

[8] H. Lee and M. Abdullah Al Faruque. Gpu-evr: Run-time scheduling framework for event-driven applications on a GPU-based embedded system. In *TCAD '16*.

[9] NVIDIA. Autonomous car development platform from nvidia. Online at `https://www.nvidia.com/en-us/self-driving-cars/drive-px/`.

[10] NVIDIA. Fermi architecture whitepaper. Online at `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[11] N. Otterness, M. Yang, T. Amert, J. Anderson, and F.D. Smith. Inferring the scheduling policies of an embedded CUDA GPU. In *OSPERT '17*.

[12] N. Otterness, M. Yang, S. Rust, E. Park, J. Anderson, F.D. Smith, A. Berg, and S. Wang. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *RTAS '17*.

[13] S. Shalev-Shwartz, S. Shammah, and A. Shashua. On a Formal Model of Safe and Scalable Self-driving Cars. *ArXiv e-prints*, August 2017.

[14] Tesla. An update on last week's accident. Online at `https://www.tesla.com/blog/update-last-week%E2%80%99s-accident`, 2018.

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

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

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

[18] Daisuke Wakabayashi. Self-driving uber car kills pedestrian in Arizona, where robots roam. *New York Times*, 2018.

[19] Y. Xu, R. Wang, T. Li, M. Song, L. Gao, Z. Luan, and D. Qian. Scheduling tasks with mixed timing constraints in GPU-powered real-time systems. In *ICS '16*.

[20] M. Yang, N. Otterness, T. Amert, J. Bakita, J. Anderson, and F.D. Smith. Avoiding pitfalls when using nvidia gpus for real-time tasks in autonomous systems. In *ECRTS '18 (to appear)*.

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

Evaluating the Memory Subsystem of a Configurable Heterogeneous MPSoC

Ayoosh Bansal*, Rohan Tabish*,
Giovani Gracioli‡, Renato Mancuso†, Rodolfo Pellizzoni‡, Marco Caccamo*
*University of Illinois at Urbana-Champaign, USA, {*ayooshb2, rtabish, mcaccamo*}*@illinois.edu*
†Boston University, USA, *rmancuso@bu.edu*
‡University of Waterloo, Canada, {*g2gracio, rpellizz*}*@uwaterloo.ca*

***Abstract*—This paper presents the evaluation of the memory subsystem of the Xilinx Ultrascale+ MPSoC. The characteristics of various memories in the system are evaluated using carefully instrumented micro-benchmarks. The impact of micro-architectural features like caches, prefetchers and cache-coherency are measured and discussed. The impact of multi-core contention on shared memory resources is evaluated. Finally, proposals are made for the design of mixed-criticality real-time applications on this platform.**

I. Introduction

The design of efficient computing platforms is essential to achieve real-time guarantees at low power consumption and cost in current and future real-time applications, from complex cyber-physical systems to mobile systems, and to ensure high-performance with acceptable quality of service (QoS) [1]. For instance, in autonomous vehicles, tasks such as steering control, fuel injection, and brake handling, are critical and have hard real-time requirements. Multimedia infotainment systems, however, demand high-performance and tolerate large variations in QoS (*i.e.,* best-effort requirements). Finally, vision-based driver assistance and navigation have become too complex to fit within the traditional development cycle of critical embedded systems, yet they cannot be handled as best-effort software components. Such tasks demand high-processing power and predictability at the same time [1].

Multi-Processor System-on-a-Chip (MPSoC) architectures provide an ideal trade-off between performance and cost to meet such requirements found in complex cyber-physical systems. The considered family of MPSoC architectures is composed of several heterogeneous processing elements with specific functionalities: general-purpose multi-core processors, DSPs, specialized processing cores, GPUs, and FPGA. They also feature a rich memory hierarchy, comprised of scratch-pads, DRAMs, Block RAM, and multiple levels of cache. A similarly rich I/O subsystem, with a number of interfaces, embedded devices, Direct-Memory Access (DMA) engines, shared buses and interconnects completes the picture.

It follows that on the one hand the considered MPSoC platforms provide a vast number of configuration options. On the other hand, however, they also make it difficult to design basic software components (real-time operating system – RTOS and hypervisor), and to understand all the sources of unpredictability. The most relevant sources of unpredictability in MPSoCs are:

- **Shared Memory Hierarchy:** several latency hiding mechanisms, including caches, buffers, scratchpads, and FIFOS are placed among the main memory, processors, and I/O devices. Such mechanisms enable latency and bandwidth demands to coexist in a hierarchy at the price of poor predictability [1]. Techniques such as private memory and cache coherency increase performance, but suffer from limitations in scalability, energy efficiency, and timing [1]. Thus, such techniques become the primary sources of unpredictability in modern MPSoCs [1, 2]. DRAM itself improves the average case performance by using row open arbitration policies or bank level interleaving but these in turn introduce further unpredictability.
- **Shared I/O Subsystem:** latency hiding mechanisms are also used in I/O subsystems. I/O subsystems deliver lower throughput compared to those designed to feed data-hungry CPUs [1]. Many systems are designed assuming that just a few I/O devices will be active at any given time, which is often a wrong assumption for large MPSoCs [1]. Then, delays and deadline misses can occur due to the contention in the I/O subsystem and the increased variation in the response time [1].
- **Shared Buses:** Multi-Processor systems use limited number of shared buses to communicate with the memory subsystems. These buses frequently become a hot spot for contention. The memory bandwidth available to a processor at any instant is affected by activity of other processors. Variable memory latency due to other processors running independent applications can cause any number of deadline violations for a processor. For this problem, various solutions have been proposed [3, 4] and analyzed [5, 6].

In this paper, we provide a benchmark-based analysis of a modern MPSoC considering the main sources of unpredictability and, based on the obtained results, we propose a basic software architecture to improve the predictability of real-time applications running on a MPSoC platform. In summary, the main contributions of this paper are:

- We benchmark the memory types available in a modern

heterogeneous MPSoC platform. We conclude that various memories exhibit varying characteristics and sensitivity to multi-core contention. We use this information to propose an architectural design paradigm in Section V.
- We propose a software/hardware architecture to improve the predictability in the modern MPSoC platforms. Our software architecture relies on a partitioning hypervisor, an RTOS, and several OS-related techniques, such as cache memory partitioning, hardware performance counters, memory bandwidth regulation, and DRAM bank-aware memory allocation.

II. Platform Overview

The selected platform ZCU102 [7] contains a Xilinx Ultrascale+ MPSoC [8]. The main components of this platform are:

1) Application Processing Unit, ARM Cortex A-53 [9]
   - Quad Core ARMv8-A Architecture
   - 32 KB each Private L1 Instruction and Data Cache per core
   - 1 MB Shared L2 cache
2) Real-Time Processing Unit, ARM Cortex-R5
   - Dual-Core ARMv7-R Architecture
   - 32 KB combined Private Instruction and Data Cache per core
   - 128 KB Tightly Coupled Memory (TCM) per core
3) Programmable Logic (PL)
4) Memory
   - *OCM*: 256 KB On-Chip Memory
   - *PS DRAM*: 4 GB DDR4 Kingston KVR21SE15S8/4
   - *PL DRAM*: 512 MB DDR4 Micron MT40A256M16GE-075E connected to Programmable Logic
   - *PL BRAM*: Block RAM in Programmable Logic
5) ARM Mali-400 Based GPU

Figure 1 presents a simplified block diagram of the targeted Ultrascale+ MPSoC. Note that the programmable logic can provide a DRAM controller to access a 512 MB DDR4 memory (here called as PL-DRAM) and a Block RAM (B-RAM). The OCM memory is accessed by the A-53 cores through two buses, and so is the 4 GB DDR4 memory (PS-DRAM). Block RAM (BRAM) [10] are embedded memory elements instantiated in the FPGA which are being used as RAM. We use up to 2 MB of BRAM in the experiments.

The Programmable Logic (PL) communicates with the A-53/R-5 cores and DRAM in the Processing System (PS) via AXI-4 [11] buses. The PS side interface contains 3 AXI Masters and 3 AXI Slaves which can be individually enabled and configured. In our experiments we use 2 AXI Masters on the PS side which connect to AXI Interconnects on the PL which provide the corresponding AXI Slave ports. AXI Master ports on these interconnects are connected to AXI Slave ports on PL DRAM and PL BRAM controllers respectively.

III. Benchmarks

Platform evaluation is performed using user space benchmarks available here [12]. The benchmarks create carefully controlled memory traffic and use timing information for those accesses to deduce platform characteristics.

A. Memory Mapping

Various memories are available on this platform as described in Section II. To benchmark specific memory from linux user space the benchmarks use the `/dev/mem` [13] interface which exposes the physical memory as a file. The `mmap` system call [14] is used to map the physical address space from `/dev/mem` to the virtual address space of the benchmark. The `mmap` system call in the kernel was modified to explicitly control the cacheability of the mapped memory. The mapped memory could be made cacheable or non-cacheable as desired. Due to the small size in the same order of magnitude as L2 Cache, PL Block RAM is always mapped as non-cacheable in all experiments.

B. Memory Latency

Memory Latency is defined here as the time difference between the processor issuing a read request and receiving the data. A strict data dependence is created between each load used to evaluate the latency. This effectively eliminates any parallelization that could be introduced by the compiler or processor and skew this metric. The average latency is calculated over a large number of such loads.

The behavior of this benchmark was verified by inspecting the assembly code generated by the compiler and using perf utility [15] at runtime. The benchmark was compiled with gcc -O2 optimizations.

C. Bandwidth

This benchmark evaluates the read or write bandwidth available to the processor for specific physical memory address ranges. The benchmark accesses the memory range under evaluation in a sequential manner with the corresponding type of access (read or write). This is done for *5 seconds* and the average bandwidth is calculated. The benchmark evaluates the processors' ability to read or write sequential address ranges. Every access made to the memory is 64 bits wide. The benchmark was compiled with gcc -O2 optimizations.

D. Cache Coherence

The effect of cache coherence on memory access time is also evaluated. The benchmark considered is similar to the one used in [16]. The benchmark in [16] uses two tasks. Each task accesses a fixed memory range with reads and writes in a sequential manner. The two tasks can be arranged with respect to each other in the following arrangements:

- *Sequential*: The two tasks are run one after the other on the same processor;
- *Parallel*: The two tasks run on two different processors but access private data only. There is no coherence dependence between the tasks;

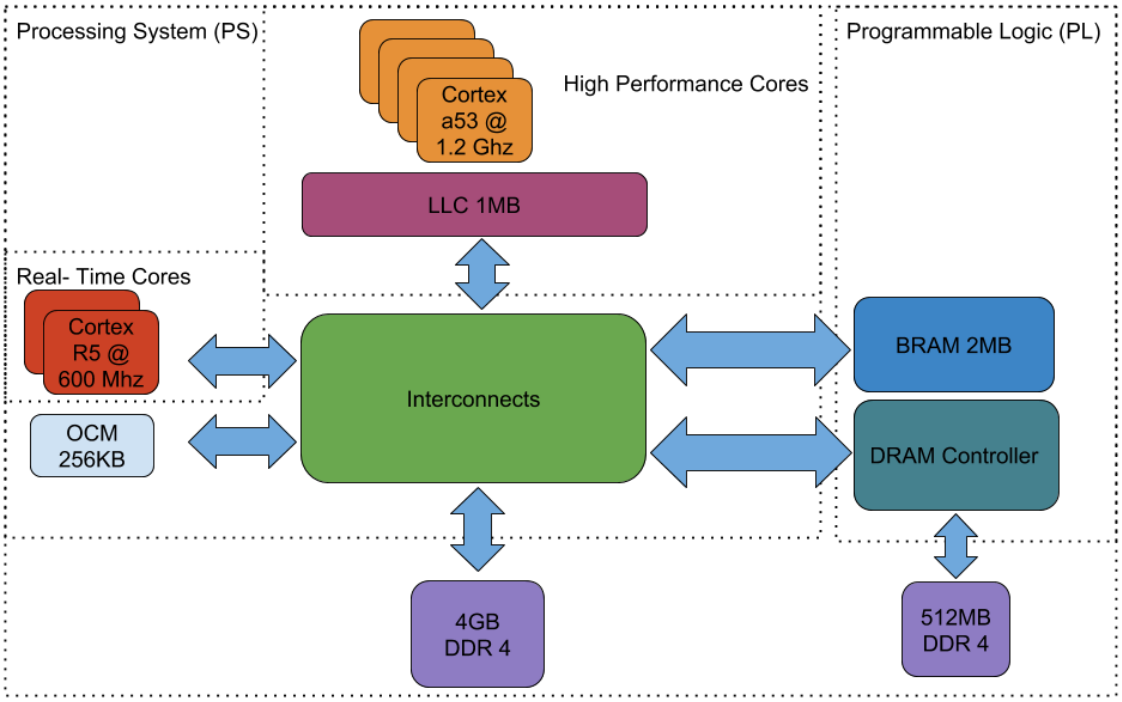


Fig. 1. Simplified Block Diagram of the UltraScale+ MPSoC

- *Concurrent*: The two tasks run on two different processors and access shared data leading to overheads due to coherence traffic.

IV. Platform Evaluation

This section provides a summary of the evaluation results.

A. Measuring Latency and Bandwidth

Using the latency benchmark as described earlier in Section III-B, we measured latency of different memory subsystems. The results of the experiments for serialized versus random access pattern to measure latency for PS DRAM, PL DRAM and PL Block RAM are shown in Figure 2. Memory accesses in this experiment bypass caches as described in Section III-A. The experimental results reveal that both PS-DRAM and PL-DRAM show less latency in serialized access compared to random access. The PL-BRAM does not exhibit any latency difference between serialized versus random access. BRAM accesses latency is independent of access pattern as it lacks constructs like banks and row buffers that are common in DRAMs.

We also ran the latency benchmark with caching enabled for varying working set sizes. Figure 3 shows the results. At the lowest working set size of 16 KB, all accesses hit in private L1 d-Cache of the processor. The access latency for the L1 d-Cache is hence around **3ns**. The shared L2 Cache has a capacity of 1 MB. Until the working set is increased beyond the 1 MB mark, the majority of memory accesses hit in L1 or L2 cache. The sharp latency increase for working sets larger than 1 MB are due to actual DRAM accesses. L2 cache latency

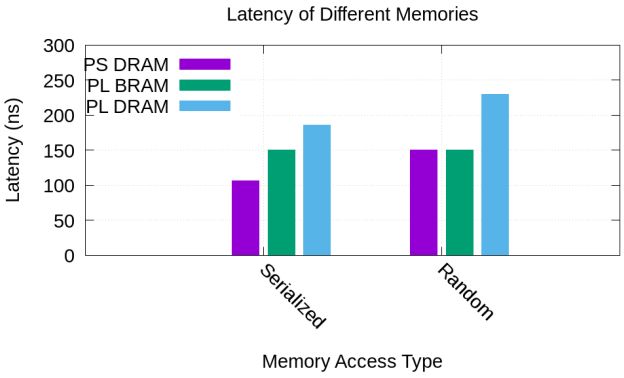


Fig. 2. Stress Results of PL Versus PS DRAM

is hence around **20ns**. Serialized read latency is substantially lower than random read latency. Additionally, note that the read latency for serialized memory accesses, even for large working sets, is comparable to L2 cache latency. This is the impact of speculative prefetching. Recall that the results in Figure 2 were obtained by defining non-cacheable buffers. At large working set sizes the latency for randomized accesses to cacheable memory (see Figure 3) converges to the latency observed for non-cached memory (Figure 2), as all accesses miss in L1/L2 cache.

Similar to the latency benchmark, we run the bandwidth benchmark described in Section III-C on A-53 core to measure

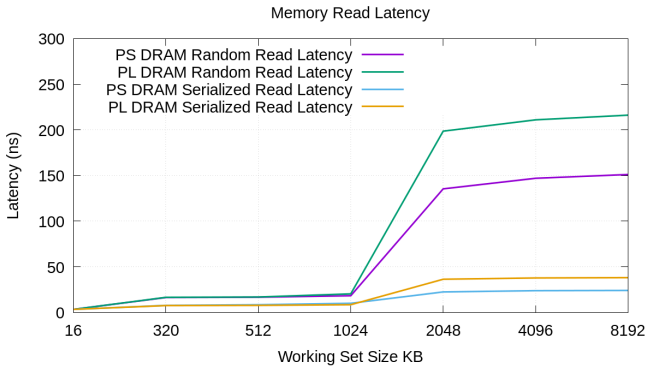


Fig. 3. Random and Serial Read Latency With Different Working Set Size

the bandwidth of different memory sub-systems as reported in Table I. From these results we draw several conclusions. In general PS DRAM is better then both PL DRAM and PL Block RAM. This is due to shorter line distance to the PS DRAM and higher clock rates in the PS subsystem. Reads with caching enabled are boosted by speculative prefetching as the accesses are strictly serial. Multiple loads are issued per cache line leading to further boost in Read bandwidth with caches as compared to without caches. Reads without caches fetch data from underlying memory on every access and hence suffer a low bandwidth. Writes without caching return asynchronously i.e. the store instruction returns without waiting for the data to be committed to the underlying memory. Without caching there is not a requirement to allocate a cache line to complete a store. In case of writes with caches enabled, stores frequently lead to dirty cache line evictions and cache line allocate for the first write to a cache line (write-allocate policy). Hence we see the large write bandwidth when caches are disabled but a low write bandwidth with caching enabled. PL Block RAM is only accessed with caches disabled. Read bandwidth from PL Block Ram is greater than PL DRAM as the logic to reach Block RAM in Programmable Logic is smaller than that to reach PL DRAM. Block RAM is also inherently faster than DRAM for single access latency which is a good approximation for the traffic pattern of the read bandwidth benchmark. On the other hand, write bandwith benchmark without caches bombards the underlying memory with write requests. In this case PL Block RAM provides a lower throughput than the PL DRAM. This is due to lack of parallelization of memory accesses and limited buffering in the access path to PL Block RAM, as compared to PL DRAM.

TABLE I
BANDWIDTH MEASUREMENTS FOR DIFFERENT MEMORIES

Access Type	PS DRAM (MB/s)	PL DRAM (MB/s)	PL BRAM (MB/s)
Write With Cache	1881	880	xx
Read With Cache	2493	1414	xx
Write W/O Cache	12000	5440	4568
Read W/O Cache	556	320	406

B. Measuring Latency Under Stress

In this section we report the memory latency seen by a core under analysis running the latency benchmark when other cores are stressing the same memory as core under analysis using the bandwidth benchmark. The bandwidth benchmark on other cores is configured to stress with write, whereas the latency is configured to perform read. In Figure 4, we show the amount of read latency seen by the core under analysis on PS DRAM and PL DRAM as we increase the stressing cores from one to up to three. Compared to solo case, the stress case of three cores shows a slow down of 1.85 times for the PS DRAM and a slow down of 5.37x for the PL DRAM. This slow-down can be explained by the DRAM specs of the PL and PS DRAM and the interconnect between the two. We also note that BRAM access latency is largely unaffected by the increasing contending traffic.

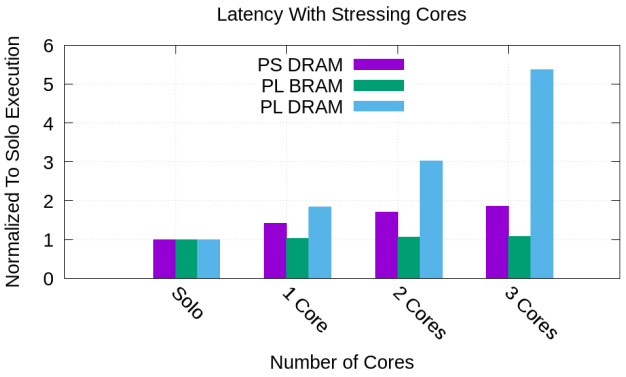


Fig. 4. Stress Results of PL BRAM/DRAM Versus PS DRAM

C. Evaluating Cache Coherence

Figure 5 shows the results of cache contention experiments as described in section III-D. We can clearly note the effects of the cache coherence protocol on the performance. The concurrent benchmark version, which runs two threads in different cores at the same time accessing the same data array, is about **3.6** times slower than the parallel version. When the second thread accesses the shared data, it gets an invalid access and must ask (snoop request) for the most recent copy of the data or recover it from a higher memory level [16]. Whenever a snoop request must be completed, it takes time, which may lead to unexpected increase of the task's execution time and deadline misses [2, 16]. According to [17], the time to complete a snoop request is considerably slow (comparable to access the off-chip RAM).

ARM Cortex-A53 processor uses the MOESI protocol to maintain data coherency between multiple cores [9]. Coherency is maintained between the cores, cache, I/O master, PL, and DRAM using the cache coherence interconnect (CCI).

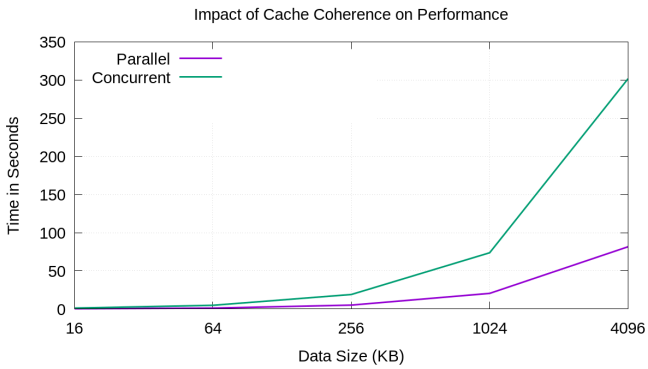


Fig. 5. Cache Coherence Results

V. Proposed Software/Hardware Predictable Architecture

In order to provide strong temporal isolation among high performance cores on the considered heterogeneous MPSoC, we propose the software architecture as shown in Figure 6.

Our proposed architecture uses Jailhouse hypervisor [18] that provides physical isolation of hardware devices including processors, among the different OSes. We propose to use a general-purpose OS such as Linux for non-critical tasks on one of the high performance core and a Real-Time operating system (RTOS) such as Erika [19] for safety-critical tasks. Like any hypervisor, the communication between different OSes running on different cores is achieved using Jailhouse. We propose prohibiting direct access to the shared resources from different cores. This eliminates unbounded contention which could make the system unpredictable.

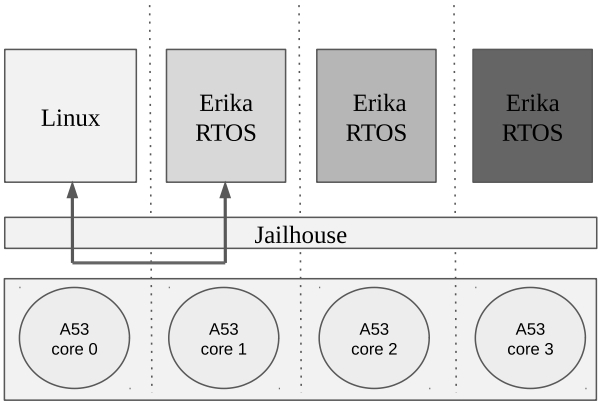


Fig. 6. Overview of the envisioned software architecture.

From our experimental results, it is clear that the main sources of contention in our system are shared memory resources such as LLC and the main memory such as PS DRAM. We propose to partition the LLC using page coloring with the help of Jailhouse. This removes the contention that can be introduced by the LLC. In order to avoid the contention at the DRAM, we propose the use of DRAM bank-aware memory allocator (PALLOC) [20]. Using cache partitioning and PALLOC we can assign a specific amount of cache and dedicate DRAM banks to a specific core and enforce strong isolation between the OSes running on different cores. For shared memory, we propose to use PL block RAM (BRAM). This is because, as shown by the experimental results in Figure 4, the BRAM does not suffer any contention when accessed using different cores.

VI. Related Work

Our proposed software architecture is similar to one proposed in [21]. However, in our proposal, Jailhouse would be responsible for providing cache partitioning (possibly through page coloring) and also DRAM bank-aware memory allocator (through PALLOC [20]). Modica et al. also proposed a similar hypervisor-based architecture targeting critical systems [22]. Cache partitioning is used to provide spatial isolation, while a DRAM bandwidth reservation mechanism provides temporal isolation. Both cache partitioning and memory reservation mechanisms were implemented in the XVISOR open-source hypervisor [23] and tested in a quad-core ARM A7 processor. Our proposed hypervisor-based approach, instead, uses a MPSoC platform, which gives the possibility to explore other features, such as specific FPGA DMA blocks (to handle data transfer between PS and PL sides for instance) and data prefetching. Another difference is that our approach will also use DRAM bank-aware memory allocator, which can provide better predictability in terms of main memory accesses.

MARACAS addresses shared cache and memory bus contention through multicore scheduling and load-balancing on top of the Quest OS [24]. MARACAS uses hardware performance counters information to throttle the execution of threads when memory contention exceeds a certain threshold. The counters are also used to derive an average memory request latency to reduce bus contention. vCAT uses the Intel's Cache Allocation Technology (CAT) to achieve core-level cache partitioning for the hypervisor and virtual machines running on top of it [25]. vCAT was implemented in Xen and $LITMUS^{RT}$. Although interesting, this approach is architecture dependent and uses non real-time basic software support (Linux and Xen).

Kim and Rajkumar proposed a predictable shared cache framework for multicore real-time virtualization systems [26]. The proposed framework introduces two hypervisor techniques (vLLC and vColoring) that enables cache-aware memory allocation for individual tasks running running in a virtual machine. CHIPS-AHOy is a predictable holistic hypervisor [1]. It integrates shared hardware isolation mechanism, such as memory partitioning, with an observe-decide-adapt loop to achieve predictability and energy, thermal, and wearout management.

VII. Conclusions

In this paper we have evaluated the different memory subsystems of the Xilinx Ultrascale+ platform. The results of the experiments show that the platform has significant

contention at LLC, PS DRAM and PL DRAM. Therefore, it cannot be used as is for multi-core applications requiring hard-real time guarantees. To provide strong isolation among the cores, we propose the use of cache coloring using JailHouse (a hypervisor) and DRAM bank partitioning using PALLOC. With strict partitioning of shared resources we can run Real Time OS on any core unaffected by application running on other cores.

REFERENCES

[1] T. Mück, A. A. Fröhlich, G. Gracioli, A. Rahmani, and N. Dutt. Chips-ahoy: A predictable holistic cyber-physical hypervisor for mpsocs. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 1–8, Samos Island, Greece, 2018.

[2] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 48(2), 2015.

[3] 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 *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, April 2013.

[4] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 299–308, July 2012.

[5] R. Mancuso, R. Pellizzoni, N. Tokcan, and M. Caccamo. WCET Derivation under Single Core Equivalence with Explicit Memory Budget Assignment. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:23, Dagstuhl, Germany, 2017.

[6] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun. WCET(m) estimation in multi-core systems using single core equivalence. In *Proceedings of the 2015 27th Euromicro Conference on Real-Time Systems*, ECRTS '15, pages 174–183, Washington, DC, USA, 2015. IEEE Computer Society.

[7] Inc. Xilinx. Ultrascale+ MPSoC ZCU102. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html, 2018. [Online; accessed 16-April-2018].

[8] Inc. Xilinx. Ultrascale+ MPSoC Overview. https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf, 2018. [Online; accessed 16-April-2018].

[9] Arm Holdings. Cortex A-53. https://developer.arm.com/products/processors/cortex-a/cortex-a53, 2018. [Online; accessed 16-April-2018].

[10] Xilinx. Block RAM. https://www.xilinx.com/html_docs/xilinx2018_1/sdsoc_doc/ond1517252572114.html, 2018. [Online; accessed 22-May-2018].

[11] AXI. https://www.arm.com/products/system-ip/amba-specifications, 2018. [Online; accessed 24-April-2018].

[12] Heechul Yun. Benchmarks. https://github.com/heechul/misc, 2018. [Online; accessed 20-April-2018].

[13] mem. http://man7.org/linux/man-pages/man4/mem.4.html, 2018. [Online; accessed 24-April-2018].

[14] mmap. http://man7.org/linux/man-pages/man2/mmap.2.html, 2018. [Online; accessed 24-April-2018].

[15] perf. https://perf.wiki.kernel.org/index.php/Main_Page, 2018. [Online; accessed 24-April-2018].

[16] G. Gracioli and A. A. Fröhlich. On the influence of shared memory contention in real-time multicore applications. In *2014 Brazilian Symposium on Computing Systems Engineering*, pages 25–30, Nov 2014.

[17] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[18] Siemens. Jailhouse. https://github.com/siemens/jailhouse, 2018. [Online; accessed 22-May-2018].

[19] Erika Enterprise. Erika Enterprise RTOS v3. http://www.erika-enterprise.com/, 2018. [Online; accessed 22-May-2018].

[20] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, April 2014.

[21] Alessandro Biondi, Mauro Marinoni, Giorgio Buttazzo, Claudio Scordino, and Paolo Gai. Challenges in virtualizing safety-critical cyber-physical systems. In *Proceedings of Embedded World Conference 2018*, pages 1–5, Feb 2018.

[22] Paolo Modica, Alessandro Biondi, Giorgio Buttazzo, and Anup Patel. Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms. In *Proceedings of the IEEE International Conference on Industrial Technology (ICIT 2018)*, pages 1–7, Feb 2018.

[23] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi. Embedded hypervisor xvisor: A comparative analysis. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 682–691, March 2015.

[24] Y. Ye, R. West, J. Zhang, and Z. Cheng. Maracas: A real-time multicore vcpu scheduling framework. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 179–190, Nov 2016.

[25] M. Xu, L. Thi, X. Phan, H. Y. Choi, and I. Lee. vcat: Dynamic cache management using cat virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 211–222, April 2017.

[26] Hyoseung Kim and Ragunathan (Raj) Rajkumar. Predictable shared cache management for multi-core real-time virtualization. *ACM Trans. Embed. Comput. Syst.*, 17(1):22:1–22:27, December 2017.

Notes

OSPERT 2018 Program

	Tuesday, July 3 2018
8:00 – 9:00	Registration
9:00	Welcome
9:05 – 10:00	Keynote talk: *Mastering Security and Resource Sharing with future High Performance Controllers: A perspective from the Automotive Industry* *Dr. Kai Lampka, Elektrobit Automotive GmbH*
10:00 – 10:30	Coffee Break
10:30 – 12:00	Session 1: RTOS Implementation and Evaluation
	Deterministic Futexes Revisited *Alexander Züpke and Robert Kaiser*
	Implementation and Evaluation of Multi-Mode Real-Time Tasks under Different Scheduling Algorithms *Anas Toma, Vincent Meyers and Jian-Jia Chen*
	Jitter Reduction in Hard Real-Time Systems using Intra-task DVFS Techniques *Bo-Yu Tseng and Kiyofumi Tanaka*
	Examining and Support Multi-Tasking in EV3OSEK *Nils Hölscher, Kuan-Hsun Chen, Georg von der Brüggen and Jian-Jia Chen*
12:00 – 13:30	Lunch
13:30 – 14:30	Keynote talk: *On safety and real-time in embedded operating systems using modern processor architectures in different safety-critical applications* *Dr. Michael Paulitsch, Intel*
14:30 – 15:00	Session 2: Best Paper
	Levels of Specialization in Real-Time Operating Systems *Björn Fiedler, Gerion Entrup, Christian Dietrich and Daniel Lohmann*
15:00 – 15:30	Coffee Break
15:30 – 17:00	Session 3: Shared Memory and GPU
	Verification of OS-level Cache Management *Renato Mancuso and Sagar Chaki*
	The case for Limited-Preemptive scheduling in GPUs for Real-Time Systems *Roy Spliet and Robert Mullins*
	Scaling Up: The Validation of Empirically Derived Scheduling Rules on NVIDIA GPUs *Joshua Bakita, Nathan Otterness, James H. Anderson and F. Donelson Smith*
	Evaluating Memory Subsystem of Configurable Heterogeneous MPSoC *Ayoosh Bansal, Rohan Tabish, Giovani Gracioli, Renato Mancuso, Rodolfo Pellizzoni and Marco Caccamo*
17:00 – 17:05	Wrap-up

	Wednesday, July 4th – Friday, July 6th 2018
	ECRTS main conference.