Proceedings of

OSPERT 2011

7th annual workshop on
Operating Systems Platforms for
Embedded Real-Time Applications

July 5th, 2011 in Porto, Portugal

in conjunction with the
23rd Euromicro Conference on Real-Time Systems
Portugal, July 6-8, 2011

Editors:
Gabriel Parmer
Thomas Gleixner

Contents

Message from the Chairs

We aim to continue the interactive emphasis for this 7th workshop on Operating Systems Platforms for Embedded Real-Time Applications. Toward this, we will have two discussion-based sessions. One is a discussion led by a panel of four experts to discuss the present and future of parallelism and real-time. Additionally, the conference will commence with a keynote by Gernot Heiser who will lend his success and experience in both academia and industry. OSPERT this year accepted 6 of 8 peer reviewed papers, and we have included an invited paper on resource management for GPUs. Given the quality and controversial nature of the papers, we expect a lively OSPERT.

We'd like to thank all of the people behind the scenes that were involved in making OSPERT what it is. Gerhard Fohler has made this workshop possible, and we appreciate the support and venue for the operating systems side of real-time systems. The advice and support from Stefan Petters and James Anderson has been invaluable; they have increased the quality of the workshop. The program committee has done wonderful work in fastidiously reviewing submissions, and providing useful feedback.

Most of all, this workshop will be a success based on the community of operating systems and real-time researchers that provide the excitement and discussion that defines OSPERT.

The Workshop Chairs,
Gabriel Parmer
Thomas Gleixner

Program Committee

Carsten Emde, *Open Source Automation Development Lab, Germany*
Peter Zijlstra, *Red Hat Linux, Netherlands*
Jim Anderson, *University of North Carolina, USA*
Rodolfo Pellizzoni, *University of Waterloo, Cananda*
Scott Brandt, *University of California, Santa Cruz, USA*
Kevin Elphinstone, *University of New South Wales/NICTA, Australia*
Neil Audsley, *University of York, UK*
Hermann Hartig, *TU Dresden, Germany*
Stefan Petters, *Instituto Superior de Engenharia do Porto, Portugal*

Keynote Talk

Toward an OS Platform for Truly Dependable Real-Time Systems

Gernot Heiser
University of New South Wales, NICTA

Many embedded systems are used in mission or even life-critical scenarios, and their dependability is paramount. The growing functionality, and resulting complexity, means that the traditional bare-metal approach is no longer feasible for such systems. This necessitates the use of spatial and temporal isolation, enforced by an operating system or hypervisor.

The dependability of the system then hinges on the dependability of that OS platform: it must ensure at least the integrity and timely execution of critical subsystems in the presence of malfunctions in non-critical parts. The talk presents our roadmap to such a platform, and discusses progress to date.

Biography:

Gernot Heiser is Scientia Professor and John Lions Chair of Operating Systems at the University of New South Wales (UNSW), and leads the Software Systems research group at NICTA, Australia's National Centre of Excellence for ICT Research. He joined NICTA at its creation in 2002, and before that was a full-time member of academic staff at UNSW from 1991. His past work included the Mungi single-address-space operating system (OS), several un-broken records in IPC performance, and the best-ever reported performance for user-level device drivers, and the world's first formal verification of a complete general-purpose OS kernel.

In 2006, Gernot with a number of his students founded Open Kernel Labs, now the market leader in secure operating-systems and virtualization technology for mobile wireless devices. The company's OKL4 operating system, a descendent of L4 kernels developed by his group at UNSW and NICTA, is deployed in more than 1.2 billion mobile phones. This includes the Motorola Evoke, the first (and to date only) mobile phone running a high-level OS (Linux) and a modem stack on the same processor core.

In a former life, Gernot developed semiconductor device simulators and models of device physics for such simulators, and pioneered the use of three-dimensional device simulation for the characterisation and optimisation of high-performance silicon solar cells.

Program

	Tuesday, July 5th 2011
8:30-9:00	Registration
9:00-10:30	Keynote Talk: Gernot Heiser
10:30-11:00	Coffee Break
11:00-12:30	Session 1: Parallelism in Real-Time Systems An efficient and scalable implementation of global EDF in Linux *Juri Lelli, Giuseppe Lipari, Dario Faggioli, Tommaso Cucinotta* A Comparison of Pragmatic Multi-Core Adaptations of the AUTOSAR System *Niko Bohm, Daniel Lohmann, Wolfgang Schroder-Preikschat* Operating Systems Challenges for GPU Resource Management *Shinpei Kato, Scott Brandt, Yutaka Ishikawa, Ragunathan (Raj) Rajkumar*
12:30-13:30	Lunch
13:45-15:30	Panel Discussion: The Present and Future of Parallelism in Real-Time *Panel members: Gernot Heiser, Thomas Gleixner, Shinpei Kato, and Andrea Bastoni*
15:30-16:00	Coffee Break
16:00-18:00	Session 2: Abstraction in Real-Time Systems Virtual Real-Time Scheduling *Malcolm Mollison and James Anderson* Temporal isolation in an HSF-enabled real-time kernel in the presence of shared resources *Martijn M. H. P. van den Heuvel, Reinder J. Bril, Johan J. Lukkien* Hard Real-time Support for Hierarchical Scheduling in FreeRTOS *Rafia Inam, Jukka Maki-Turja, Mikael Sjodin, Moris Behnam* RTOS-Based Embedded Software Development using Domain-Specific Language *Mohamed-El-Mehdi Aichouch, Jean-Christophe Prevotet, Fabienne Nouvel*
18:00-18:30	Discussion and Closing Thoughts

	Wednesday, 6th - Friday, 8th 2011
	ECRTS main proceedings.

An efficient and scalable implementation of global EDF in Linux *

Juri Lelli, Giuseppe Lipari, Dario Faggioli, Tommaso Cucinotta
{name.surname}@sssup.it
Scuola Superiore Sant'Anna - ITALY

Abstract

The increasing popularity of multi-core architectures is pushing researchers and developers to consider multi-cores for executing soft and hard real-time applications. Real-Time schedulers for multi processor systems can be roughly categorized into partitioned and global schedulers. The first ones are more adequate for hard real-time embedded systems, in which applications are statically loaded at start-up and rarely change at run-time. Thanks to automatic load balancing, global schedulers may be useful in open systems, where applications can join and leave the system at any time, and for applications with highly varying workloads.

Linux supports global and partitioned scheduling through its real-time scheduling class, which provides `SCHED_FIFO` and `SCHED_RR` fixed priority policies. Recently, the `SCHED_DEADLINE` policy was proposed that provides Earliest Deadline First scheduling with budget control. In this paper we propose a new implementation for global EDF scheduling which uses a heap global data structure to speed-up scheduling decisions. We also compare the execution time of the main scheduling functions in the kernel for four different implementations of global scheduling, showing that our implementation is as scalable and efficient as `SCHED_FIFO`.

*The research leading to these results has received funding from the European Community's Seventh Framework Programme n.248465 "S(o)OS – Service-oriented Operating Systems."

1 Introduction

Multi-processor and multi-core computing platforms are nowadays largely used in the vast majority of application domains, ranging from embedded systems, to personal computing, to server-side computing including GRIDs and Cloud Computing, and finally high-performance computing.

In embedded systems, small multi-core platforms are considered as a viable and cost-effective solution, especially for their lower power requirements as compared to a traditional single processor system with equivalent computing capabilities. The increased level of parallelism in these systems may be conveniently exploited to run multiple real-time applications, like found in industrial control, aerospace or military systems; or to support soft real-time Quality of Service (QoS) oriented applications, like found in multimedia, gaming or virtual reality systems.

Servers and data centres are shifting towards (massively) parallel architectures with enhanced maintainability, often accompanied by a decrease in the clock frequency driven by the increasing need for "green computing" [13]. Cloud Computing promises to move most of the increasing personal computing needs of users into the "cloud". This is leading to an unprecedented need for supporting a large number of interactive and soft real-time applications, often involving on-the-fly media streaming, processing and transformations with demanding performance and latency requirements. These applications usually exhibit nearly periodic workload patterns which often do not saturate the available computing power of a single (powerful) CPU. Therefore, there is a strong industrial interest in executing an increasing number

of applications of this type onto the same system, node, CPU and even core, whenever possible, in order to minimize the number of needed nodes (and reduce both power consumption and costs). In this context, a key role is played by real-time CPU scheduling algorithms for multi-processor systems. These can be roughly categorised into *global schedulers* and *partitioned schedulers*.

In *partitioning*, tasks are allocated to cores and will rarely (or never) move from one core to another one. Every core has its own private scheduling queue, and its private scheduler. In *closed systems*, the allocation algorithm is executed off-line and tasks are statically allocated to cores. In *open systems* tasks can dynamically join and leave the system: however task join/leave are rare event compared to the time granularity of the run-time events (i.e. the tasks' periods, or the scheduling tick). When a task joins the system, the allocation algorithm is executed, which may cause a re-allocation of existing tasks and hence a migration from one core to another (*load balancing*).

In *global scheduling*, ready tasks are enqueued in a logical global queue, and the M highest priority tasks are selected to run on the M cores. Therefore, a task can be suspended on one core and resume execution on another core. The number of migrations is much higher than in the previous case. *Clustered schedulers* reside in the middle, where the available processors are partitioned into clusters to which tasks are statically assigned, but in each cluster tasks are globally scheduled.

Global scheduling has the advantage of automatically performing load balancing, however it also raises many concerns about its practicality. Indeed, migrations may invalidate the cache, increasing the task execution time. For this reason, partitioned scheduling is mostly used in hard real-time embedded applications, where tasks are known at configuration time, and optimal allocation and load balancing can be performed off-line. On the other end, global scheduling seems more appropriate for dynamic open systems with highly varying workloads.

One concern of researchers and practitioners is the overhead of scheduling. In particular, the problem is to maintain data structures in the kernel to represent the global queue; such global structures are concurrently accessed by all cores and must therefore be appropriately protected with concurrency control mechanisms (lock-based or lock-free).

1.1 Contributions of this work

In this work, we present an implementation of a global EDF scheduler in Linux using a heap data structure to optimize access to the earliest deadline tasks. The implementation is an improvement over `SCHED_DEADLINE` [8]. After describing the base real-time scheduler of Linux (Section 3), and our implementation (Section 4), we compare its performance against the global POSIX-compliant fixed priority scheduler shipped with stock Linux and with the previous version of `SCHED_DEADLINE` (Section 6). The results show that using appropriate data structures it is indeed possible to build efficient and scalable global real-time schedulers. In Section 7, we also identify space for possible improvements.

All the code that has been developed during the experimental evaluation phase of this study can be downloaded by following the instructions at the very top of this page: `https://www.gitorious.org/sched_deadline/pages/Download`.

2 State of the art

When deciding which kind of scheduler to adopt in a multiple processor system, there are two main options: partitioned scheduling and global scheduling.

In partitioned scheduling, the placement of tasks among the available processors is a critical step. The problem of optimally dividing the workload among the various queues, so that the computing resources be well utilised, is analogous to the bin-packing problem, which is known to be NP-hard in the strong sense [10].This complexity is typically avoided using sub-optimal solutions provided by polynomial and pseudo-polynomial time heuristics, like First Fit, Best Fit, etc. [12, 11].

In **global scheduling**, tasks are ordered into a single logical queue and scheduled onto the available processors. Using a single logical queue, the load is thus intrinsically balanced, since no processor is idled

as long as there is a ready task in the global queue. A class of algorithms, called Pfair schedulers [2], is able to ensure that the full processing capacity can be used, but unfortunately at the cost of a potentially large run-time overhead. Migrative and non-migrative algorithms have been proposed modifying well-known solutions adopted for the single processor case and extending them to deal with the various anomalies [1] that arise on a parallel computing platform.

Complications in using a global scheduler mainly relate to the cost of inter-processor migration, and to the kernel overhead due to the necessary synchronisation among the processors for the purpose of enforcing a global scheduling strategy. As we will see in Section 3, in order to reduce the contention, the logical single queue can actually be implemented as a set of distributed queues plus some helper data structures to maintain consistency.

In addition to the above classes, there are also intermediate solutions, like **clustered**- and restricted-migration schedulers. A clustered scheduler [6] limits the number of processors among which a task can migrate. This method is more flexible than a rigid partitioning algorithm without migration.

Linux supports real-time scheduling with two scheduling policies, `SCHED_RR` and `SCHED_FIFO`. They are based on fixed priority and are compliant with the POSIX standard. The base scheduler is global, i.e. tasks can freely migrate across processors. By specifying task's processor affinity, it is possible to pin a task on one processor (partitioning) or to a set of processors (clustered scheduling). The implementation of such scheduler will be described in Section 3.

Recently, the `SCHED_DEADLINE` scheduling class has been proposed for Linux [8]. It mimics the fixed priority scheduling class, but provides Earliest Deadline First scheduling. Again, using affinity it is possible to implement global, partitioned and clustered scheduling. An overview of `SCHED_DEADLINE` can be found in Section 4.

The line of research closer to the approach of this paper is the one carried out by the Real-Time Systems Group at University of North Carolina at Chapel Hill, conducted by means of their $LITMUS^{RT}$ testbed [5] and investigating how real overheads affect analysis results. There are several works by such group going in this direction: in [5] Calandrino et al. studied the behaviour of some variants of global EDF and Pfair, but did not consider fixed-priority; in [4], Brandenburg et al. explored the scalability of a similar set of algorithms, while in [3] the impact of the implementation details on the performance of global EDF is analysed. In all these works, samples of the various forms of overhead that show up during execution on real hardware are gathered and are then plugged in schedulability analysis, trying to make it realistic, which is an important difference between their works and the present paper (that does not consider schedulability as a metric).

Furthermore, in this work we propose an efficient implementation of a global EDF scheduler that, althought shares basic principles with [3], perfectly fits into the stock Linux scheduler, providing experimental evidence of its usability on a large multi-core machine.

3 `SCHED_FIFO` scheduler

In the Linux kernel, schedulers are implemented inside *scheduling classes*. Stock Linux comes with two classes, one for fair scheduling of non-real-time activities (`SCHED_OTHER` policy) and one implementing fixed priority real-time scheduling (`SCHED_FIFO` or `SCHED_RR` policies), following the POSIX 1001.3b [9] specification. In this paper we will focus on the real-time scheduling policies.

The fixed priority scheduling class already supports global scheduling. However, to reduce memory contention and improve locality, the logical global queue is implemented using a set of distributed run-queues, one for each CPU. Tasks are migrated across CPUs using *push* and *pull* operations.

3.1 Run-queues, masks and locks

To keep track of active tasks, the kernel uses a data structure called *runqueue*. There is one runqueue for each CPU and they are managed separately in a distributed manner. Every runqueue is protected by

a spin-lock to guarantee correctness on concurrent updates. Runqueues are modular, in the sense that there is a separate sub-runqueue for each scheduling class. Key components of the fixed priority sub-runqueue are:

- a priority array on which tasks are actually queued;
- fields used for load balancing;
- fields to speed up decisions on a multiprocessor environment.

Tasks are *enqueued* on some runqueue when they wake up and are *dequeued* when they are suspended.

An additional data structure, called `cpupri`, is used to reduce the amount of work needed for a push operation. This structure tracks the priority of the highest priority task in each runqueue. The system maintains the state of each CPU with a 2 dimensional bitmap: the first dimension is for priority class and the second for CPUs in that class. Therefore a push operation can find a suitable CPU where to send a task in $O(1)$ time, since it has to perform a two bits search only (if we don't consider affinity restriction). Concurrent access to bitmap fields is protected by means of spinlocks in a fine-grained way. In particular, there is a different spinlock for each CPU of each class; concurrent tasks have to spin waiting only if they need to update data regarding the same CPU.

3.2 Push and pull operations

When a task is activated on CPU k, first the scheduler checks the local runqueue to see if the task has higher priority than the executing one. In this case, a preemption happens, and the preempted task is inserted at the head of the queue; otherwise the wakeup task is inserted in the proper runqueue, depending on the state of the system. In case the head of the queue is modified, a *push* operation is executed to see if some task can be moved to another queue. When a task suspends itself (due to blocking or sleeping) or lowers its priority on CPU k, the scheduler performs a *pull* operation: it looks at the other run-queues to see if some other higher priority tasks need to be migrated to the current CPU. Pushing or pulling a task entails modifying the state of the source and destination runqueues: the scheduler has to dequeue the task from the source and then enqueue it on the destination runqueues.

```
struct dl_rq {
    struct rb_root rb_root;
    struct rb_node *rb_leftmost;
    unsigned long dl_nr_running;

#ifdef CONFIG_SMP
    struct {
        /* two earliest tasks in queue */
        u64 curr;
        u64 next; /* next earliest */
    } earliest_dl;
    int overloaded;
    unsigned long dl_nr_migratory;
    unsigned long dl_nr_total;
    struct rb_root pushable_tasks_root;
    struct rb_node *pushable_tasks_leftmost;
#endif /* CONFIG_SMP */
};
```

Figure 2: `struct dl_rq` extended

4 SCHED_DEADLINE

Recently, a new scheduling class has been made available for the Linux kernel, called `SCHED_DEADLINE` [8]. It implements partitioned, clustered and global EDF scheduling with hard and soft reservations [1]. The approach used for the implementation is the same used in the Linux kernel for the fixed-priority scheduler. This is usually called *distributed run-queue*, meaning that each CPU maintains a private data structure implementing its own ready queue and, if global scheduling is to be achieved, tasks are migrated among processors when needed.

In more details:

- the tasks of each CPU are kept into a CPU-specific run-queue, implemented as a *red-black tree* ordered by absolute deadlines;
- tasks are migrated among run-queues of different

[1] Full source code available at: `http://gitorious.com/sched_deadline`.

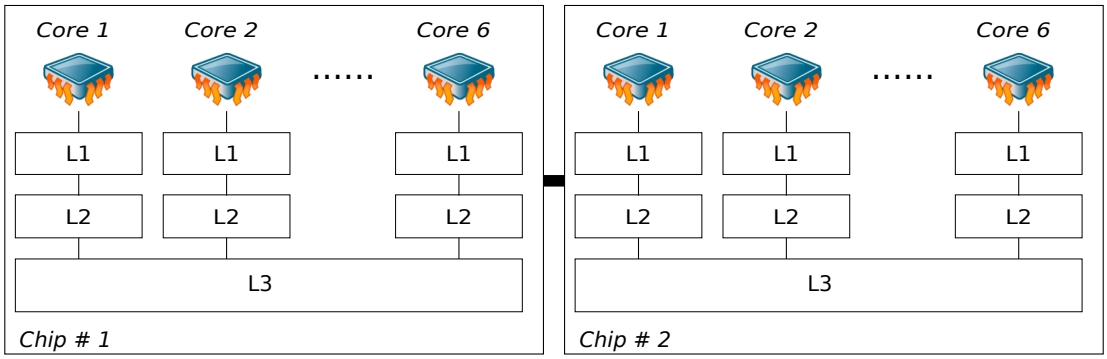

Figure 1: Architecture of a single processor (Multi Chip Module) of the Dell PowerEdge R815.

CPUs for the purpose of fulfilling the following constraints:

  - on m CPUs, the m earliest deadline ready tasks run;
  - the CPU affinity settings of all the tasks is respected.

Migration points are the same as in the fixed priority scheduling class. Decisions related to push and pull logic are taken considering deadlines (instead of priorities) and according to tasks affinity and system topology. The data structure used to represent the EDF ready queue of each processor has been modified, as shown in Figure 2 (new fields are the one inside the `#ifdef CONFIG_SMP` block).

- `earliest_dl` is a per-runqueue data structure used for "caching" the deadlines of the first two ready tasks, so to facilitate migration-related decisions;
- `dl_nr_migratory` and `dl_nr_total` represent the number of queued tasks that can migrate and the total number of queued tasks, respectively;
- `overloaded` serves as a flag, and it is set when the queue contains more than one task;
- `pushable_tasks_root` is the root of the red-black tree of tasks that can be migrated, since they are queued but not running, and it is ordered by increasing deadline;
- `pushable_tasks_leftmost` is a pointer to the node of `pushable_tasks_root` containing the task with the earliest deadline.

A *push* operation tries to move the first ready and not running task of an overloaded queue to a CPU where it can execute. The best CPU where to push a task is the one which is running the task with the latest deadline among the m executing tasks, considering also the constraints due to the CPU affinity settings. A *pull* operation tries to move the most urgent ready and not running tasks among all tasks on all overloaded queues in the current CPU.

4.1 Idle processor improvement

The push mechanism core is realized in a small function that finds a suitable CPU for a to-be-pushed task. The operation can be easily accomplished on a small multi-core machine (for example a quad-core) just by looking at all queues in sequence. The original `SCHED_DEADLINE` implementation realizes a complete loop through all cores for every push decision (pseudo-code on Figure 3). The execution time of such function increases linearly with the number of cores, therefore it does not scale well to systems with large number of cores.

A simple observation is that on systems with large number of processors and relatively light load, many CPUs are idle most of the time. Therefore, when a task wakes up, there is a high probability of finding an

```
cpu_mask push_find_cpu(task) {
    for_each_cpu(cpu, avail_cores) {
        mask = 0;
        if (can_execute_on(task, cpu) &&
            dline_before(task, get_curr(cpu)))
            mask |= cpu;
    }
    return mask;
}
```

Figure 3: Find CPU eligible for push.

```
cpu_mask push_find_cpu(task) {
    if (dlf_mask & affinity)
       return (dlf_mask & affinity);
    mask = 0;
    for_each_cpu(cpu, avail_cores) {
        if (can_execute_on(task, cpu) &&
            dline_before(task, get_curr(cpu)))
            mask |= cpu;
    }
    return mask;
}
```

Figure 4: Using idle CPU mask.

idle CPU. To improve the execution time of the push function, we can use a bitmask that stores the idle CPUs with a bit equal to 1. On a 64-bit architecture, we can represent the status of up to 64 processors by using a single word. Therefore, the code of Figure 3 can be rewritten as in Figure 4, where `dlf_mask` is the mask that represents idle CPUs, and the loop is skipped (returning all suitable CPUs to the caller) if is it possible to push the task on a free CPU.

This simple data structure introduces little or no overhead for the scheduler and significantly improves performance figures in large multi-core systems (more on this later). Updates on `dlf_mask` are performed in a thread-safe way: we use a low level `set_bit()` provided in Linux which performs an atomic update of a single bit of the mask.

5 Heap Data structure

When the system load is relatively high, idle CPUs tend to be scarce. Therefore, we introduce a new data structure to speed-up the search for a destination CPU inside a push operation. The requirements for the data structure are: $O(1)$ complexity for searching the best CPU; and less-than-linear complexity for updating the structure. The classical heap data structure fulfils such requirements as it presents $O(1)$ complexity for accessing to the first element, and $O(\log n)$ complexity for updating (if contention is not considered). Also, it can be implemented using a simple array. We developed a *max heap* to keep track of deadlines of the earliest deadline tasks currently executing on each runqueue. Deadlines are used as keys and the *heap-property* is: if B is a child node of A, then $deadline(A) \geq deadline(B)$. Therefore, the node in the root directly represent the CPU where the task need to be pushed.

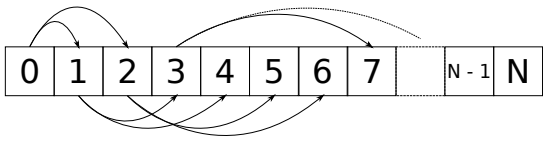

Figure 5: Heap implementation with a simple array.

A node of the heap is a simple structure that contains two fields: a deadline as key and an `int` field representing the associated CPU (we will call it `item`). The whole heap is then self-contained in another structure as described in Figure 6:

- `elements` contains the heap; `elements[0]` contains the root and the node in `elements[i]` has its left child in `elements[2*i]`, its right child in `elements[2*i+1]` and its parent in `elements[i/2]` (see Figure 5);

```
struct dl_heap {
    spinlock lock;
    int size;
    int cpu_to_idx[NR_CPUS];
    item elements[NR_CPUS];
    bitmask free_cpus;
};
```

Figure 6: Heap structure.

- `size` is the current heap size (number of non idle CPUs);
- `cpu_to_idx` is used to efficiently update the heap when runqueues state changes, since with this array we keep track of where a CPU resides in the heap;
- `free_cpus` accounts for idle CPUs in the system.

Special attention must be given to the `lock` field. Consistency of the heap must be ensured on concurrent updates: every time an update operation is performed, we force the updating task to spin, waiting for other tasks to complete their work on the heap. This kind of coarse-grained lock mechanism simplifies the implementation but it increases contention and overhead. In the future, we will look for alternative lock-free implementation strategies.

Potential points of update for the heap are enqueue and dequeue functions. If something changes at the top of a runqueue, a new task starts executing becoming the so-called *curr*, or the CPU becomes idle, the heap must be updated accordingly. We argued, and then experimented, an increase in overhead for the aforementioned operations, but we will show in Section 6 data that suggest this price is worth paying in comparison with push mechanism performance improvements.

With the introduction of the heap, code in Figure 4 can be changed as in Figure 7, where `maximum(...)` returns the heap root. As we can see from the pseudo-code we first try to push a task to idle CPUs, then we try to push it on the *latest deadline* CPU; if both operations fail, the task is not pushed away.

```
cpu_mask push_find_cpu(task) {
    if (dl_heap->free_cpus & affinity)
        return (dl_heap->free_cpus & affinity);
    if (maximum(dl_heap) & affinity)
        return maximum(dl_heap);
    mask = 0;
    for_each_cpu(cpu, avail_cores) {
        if (can_execute_on(task, cpu) &&
            dline_before(task, get_curr(cpu)))
            mask |= cpu;
    }
    return mask;
}
```

Figure 7: Find eligile CPU using a heap.

This kind of functioning is compliant with classical global scheduling, as it performs continuous load balancing across cores: rather than compacting all tasks on few cores we prefer every core share an (as much as possible) equal amount of real-time activities.

6 Evaluation

6.1 Experimental setup

The aim of the evaluation is to measure the performance of the new data structures compared with the reference Linux implementation (`SCHED_FIFO`) and the original `SCHED_DEADLINE` implementation. Since all mechanisms described so far share the same structure (i.e. distributed runqueues, and push and pull operations for migrating tasks), we measures the average number of cycles of the main operations of the scheduler: to enqueue and dequeue a task from one of the runqueues; the push and pull operations.

We conducted our experiments on a Dell PowerEdge R815 server equipped with 64GB of RAM, and 4 AMD^R Opteron^{TM} 6168 12-core processors (running at 1.9 GHz), for a total of 48 cores. The memory is globally shared among all the cores, and the cache hierarchy is on 3 levels (see Figure 1), private per-core 64 KB L1D and 512 KB L2 caches, and a global 10240 KB L3 cache. The R815 server was configured with a Debian Sid distribution running a patched 2.6.36 Linux kernel.

In the following we will refer the three patches we developed as:

- `original`, the original `SCHED_DEADLINE` implementation;
- `fmask`, `SCHED_DEADLINE` plus changes described in Section 4.1;
- `heap`, `SCHED_DEADLINE` plus the heap described in Section 5.

The reference Linux scheduler is denoted with `SCHED_FIFO`.

6.2 Task set generation

The algorithm for generating task sets used in the experiments works as follows. We generate a number of tasks $N = x \cdot m$, where m is the number of processors (see below), and x is set equal to 3. Similar overhead figures have been obtained with a higher number of tasks (results omitted for the sake of brevity).

The overall utilisation U of the task set is set equal to $U = R \cdot m$ where R is 0.6, 0.7 and 0.8. To generate the individual utilisation of each task, the `randfixedsum` algorithm [7] has been used, by means of the implementation publicly made available by Paul Emberson[2]. The algorithm generates N randomly distributed numbers in $(0, 1)$, whose sum is equal to the chosen U. Then, the periods are randomly generated according to a log-uniform distribution in $[10ms, 100ms]$. The (worst-case) execution times are set equal to the task utilisation multiplied by the task period.

We generated 20 random task sets considering 2, 4, 8, 16, 24, 32, 40 and 48 processors. Than we ran each task set for 10 seconds using a synthetic benchmark (that lets each task execute for its WCET every period). We varied the number of active CPUs using Linux CPU hotplug feature. We collected scheduler statistics through `sched_debug` as to maintain measuring overhead at a minimum value.

6.3 Results

In Figures 8 and 9 we show the number of clock cycles required by a push operation in average, depending on the number of active cores. In Figure 8, we considered an average load per processor equal to $U = 0.6$, while in Figure 9 the load was increased to $U = 0.8$. We measured the 95% confidence interval of each average point, and it is always very low (in the order of a few tens of cycles), so we did not report it in the graphs for clarity.

From the graphs it is clear that the overhead of the original implementation of `SCHED_DEADLINE` increases linearly with the number of processors, as expected, both for light load and for heavier load.

[2]More information is available at: `http://retis.sssup.it/waters2010/tools.php`.

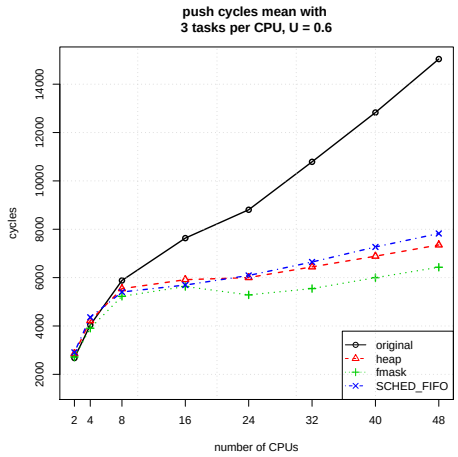

Figure 8: Number of push cycles for average loads of 0.6.

In `fmask`, we added the check for idle processors. Surprisingly, this simple modification substantially decreases the overhead for both types of loads, and it becomes almost constant in the number of processors. For light load, `fmask` is actually the one with lowest average number of cycles; this confirms our observation that for light loads the probability of finding an idle processor is high. For heavier loads, the probability of finding an idle processor decreases, so the `SCHED_FIFO` and the heap implementations are now the ones with lowest average overhead. Notice also that the latter two show very similar performance. This means that the overhead of implementing global EDF is comparable (and sometimes even lower) than implementing global Fixed Priority.

To gain a better understanding of these performance figures, it is also useful to analyse the overhead of two basic operations, enqueue and dequeue. Please remind that push and pull operations must perform at least one dequeue and one enqueue to migrate a task.

The number of cycles for enqueue operations for the four implementations is shown in Figure 10 for light load, and in Figure 11 for higher loads. The implementation with the lower enqueue overhead is

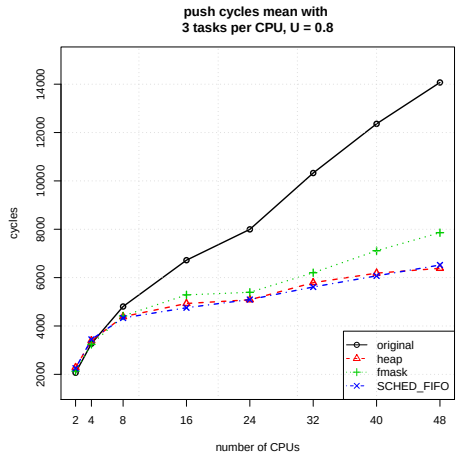

Figure 9: Number of push cycles for average loads of 0.8.

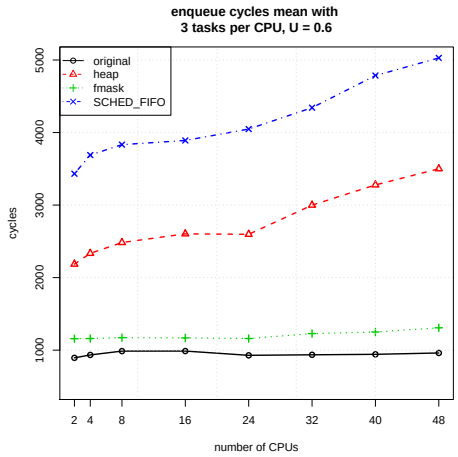

Figure 10: Number of enqueue cycles for average loads of 0.6.

`original`, because it is the one that requires the least locking and contention on shared data structures: it only requires to lock the runqueue of the CPU where the task is being moved to. `fmask` has a slightly higher overhead, as it also require to update the idle CPU mask with an atomic operation. Heap and `SCHED_FIFO` require the higher overhead as they must lock and update also global data structures (heap in the first case and priority mask in the second case). Updating the heap takes less time probably because it is a small data structure guarded by one single coarse-grain lock, whereas the prioriy mask is a complex and larger data structure with fine-grained locks. Dequeue operations have very similar performance figures and are not shown here for lack of space.

Pull operations do not take advantage of any dedicated data structure. In all four schedulers, the pull operation always looks at all runqueues in sequence to find the tasks that are eligible for migration. Therefore, the execution cycles are very similar to each other. As a future work, we plan to optimize pull operations using dedicated data structures.

7 Conclusions and future work

In this paper we presented an efficient implementation of global EDF seamless integrated with the Linux scheduler. We also compared our implementation with the `SCHED_FIFO` Linux scheduler and with our previous implementation of `SCHED_DEADLINE`. Our implementation is scalable, and its performance is very close to the one of `SCHED_FIFO`.

Is there space for improvements? We believe that it is possible to work along two different directions. First, our work was directed toward improving push operations only. However, migrations may happen also through pull operations. By using appropriate data structures for pulling out tasks from queues, we could speed up also this phase.

Second, locking is a costly operation in modern multi-core processors, as it is evident from Figures 10 and 11. We will acquire data on time wasted in spinlocks and then investigate the use of lock-free techniques for protecting access to the heap, hoping for reducing the number of cycles required by an enqueue/dequeue.

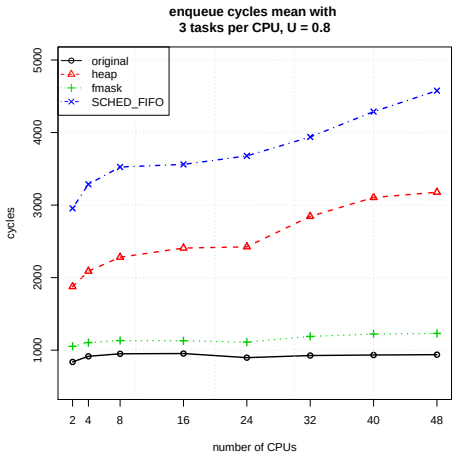

Figure 11: Number of enqueue cycles for average loads of 0.8.

References

[1] B. Andersson and J. Jonsson. Preemptive multiprocessor scheduling anomalies. In *Proceedings of the 16th International Symposium on Parallel and Distributed Processing*, pages 12–, Washington, DC, USA, 2002. IEEE Computer Society.

[2] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 345–354, New York, NY, USA, 1993. ACM.

[3] B. B. Brandenburg and J. Anderson. On the implementation of global real-time schedulers. In *Proc. of the* 30^{th} *IEEE Real-Time Systems Symposium (RTSS 2009)*, Washington D.C., USA, December 2009.

[4] B. B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proc. of the* 29^{th} *IEEE Real-Time Systems Symposium (RTSS 2008)*, Barcelona, Spain, December 2008.

[5] J. Calandrino, H. Leontyev, A. Block, U. Devi, , and J. Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proc. of the* 27^{th} *IEEE Real-Time Systems Symposium (RTSS 2006)*, Rio de Janeiro, Brazil, December 2006.

[6] John M. Calandrino, James H. Anderson, and Dan P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 247–258, Washington, DC, USA, 2007. IEEE Computer Society.

[7] Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, Brussels, Belgium, July 2010.

[8] Dario Faggioli, Fabio Checconi, Michael Trimarchi, and Claudio Scordino. An EDF scheduling class for the Linux kernel. In *Proceedings of the Eleventh Real-Time Linux Workshop*, Dresden, Germany, September 2009.

[9] IEEE. *Information Technology - Portable Operating System Interface - Part 1: System Application Program Interface Amendment: Additional Realtime Extensions.* 2004.

[10] Pramote Kuacharoen, Mohamed A. Shalan, and Vincent J. Mooney III. A configurable hardware scheduler for real-time systems. In *in Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 96–101. CSREA Press, 2003.

[11] Sylvain Lauzac, Rami Melhem, and Daniel Mossé. An improved rate-monotonic admission control and its applications. *IEEE Trans. Comput.*, 52:337–350, March 2003.

[12] Jörg Liebeherr, Almut Burchard, Yingfeng Oh, and Sang H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. Comput.*, 44:1429–1442, December 1995.

[13] Ernst von Weizsaecker, Karlson Hargroves, Michael Smith, Cheryl Desha, and Peter Stasinopoulos. *Factor Five – Transforming the Global Economy through 80% Improvements in Resource Productivity.* Earthscan, November 2009.

A Comparison of Pragmatic Multi-Core Adaptations of the AUTOSAR System

Niko Böhm, Daniel Lohmann, Wolfgang Schröder-Preikschat
Department of Computer Science 4
FAU Erlangen-Nuremberg
Erlangen, Germany
{*boehm,lohmann,wosch*}*@cs.fau.de*

***Abstract*—AUTOSAR is the dominating system standard in the automotive domain. Founded in 2003, the AUTOSAR partnership, however, only specifies a single processor system. Multi-core support has recently been added as an extension to "legacy" AUTOSAR by forcing most of the code base to run on a single core. The performance of this approach is disillusioning at best, with the single-core system outperforming the multi-core system in a full-load scenario.**

We propose a classic alternative: protect the legacy code base with a single lock. Our measurements show that even this coarse approach performs significantly better than the AUTOSAR approach. Additionally, the future prospects of getting more sophisticated multi-core support for the AUTOSAR system are presented.

I. Introduction

Today, multi-core processors are common in the desktop area and are moving on to other areas such as embedded computing. Automotive OEMs have recently become interested in multi-core processors for use in automotive *Electronic Control Units* (or *ECUs* for short). The main motivations in the automotive domain are **Functional Safety** and **Performance**.

Functional safety, as specified by industry standards such as ISO 26262, requires (or at least strongly recommends) redundancy of critical system components. An easy way to achieve this is the application of a multi-core processor in the so-called *lock-step mode*: In this mode, the system is a single processor from the outside view, so it can be operated with existing software.

Performance is desired for a couple of reasons. The first one is to run more applications on an ECU. There are also applications – especially in the power train and engine control areas – that get more and more complex in order to improve gas mileage and to comply with increasingly restrictive emission standards. Another trend is to reduce the total number of ECUs built into a single car. Without reduction of features, this is only possible with ECUs that provide enough processing power to run the additional applications. In a simple example this would mean that two previously separated single-core ECUs may be combined to a single ECU powered by a dual-core processor.

Another recent development in the automotive industry is *AUTOSAR* (short for *Automotive Open System Architecture* [3]). AUTOSAR is an international partnership of automotive OEMs and suppliers. The main purpose of AUTOSAR is the specification of a modern system software for automotive ECUs. The term AUTOSAR is commonly used to refer to the organization as well as the standard they produce. It is important to notice that AUTOSAR added support for multi-core ECUs only in December 2009 with release 4.0 of the AUTOSAR system specification. Consequently, all AUTOSAR systems prior to this date and – due to the length of the development cycles in the automotive industry – every AUTOSAR system on the streets is a single-core system. The challenge the automotive software developers are now facing is the adaption of their legacy AUTOSAR systems (prior to 4.0) to multi-core processors.

The remainder of this paper is structured as follows: Section II gives an overview of the AUTOSAR system, the central object of our studies. We evaluate the AUTOSAR multi-core system in Section III and propose our alternative in Section IV. In Section V we compare our approach to AUTOSAR. The results are discussed in Section VI. Our plans for future work are presented in Section VII. Finally, we conclude this paper with a summary in Section VIII.

II. The AUTOSAR System

The main concept and – considering the automotive domain – also main novelty of AUTOSAR is a middleware-like communication concept. The application is split up into so-called *Software Components* (*SW-Cs*). Each of those SW-Cs specifies its input and output ports and is only allowed to communicate through these ports, thus conceptually making the SW-C independent of the ECU. All services used by the SW-Cs are provided by the AUTOSAR *Run-Time Environment* or *RTE* for short. The RTE is specifically generated for every ECU depending on the SW-Cs present. The RTE itself uses the AUTOSAR Operating System (OS) and *Basic Software* (BSW). Figure 1(a) depicts this architecture [1].

The BSW provides a standardized, highly-configurable set of services, such as communication over various physical interfaces, NVRAM[1] access, management of run-time errors, amongst others. The BSW forms the biggest part of the standardized AUTOSAR environment. It is organized

[1] *Non-Volatile RAM*

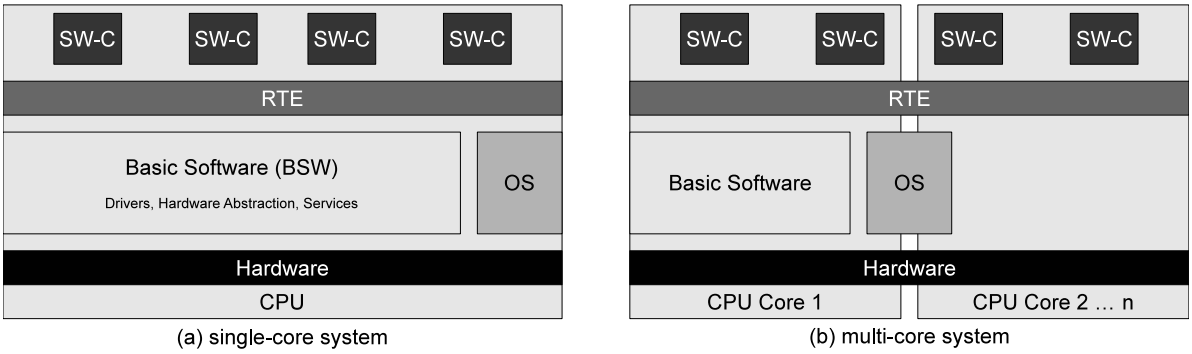

Figure 1. The AUTOSAR System Architecture

in more than forty individual modules – which may be combined from different vendors – and its size is some ten thousand lines of code.

The AUTOSAR OS is an event-triggered real-time operating system, that schedules the tasks and interrupt service routines (ISRs) of the system. Tasks are strictly priority scheduled, though there is a mechanism called *Schedule Tables* that can activate tasks at pre-defined time offsets. The OS furthermore provides a deadlock-free synchronization mechanism called *Resources*, based on the Stack-Based Priority Ceiling Protocol [4].

The AUTOSAR system targets relatively small 16- and 32-bit microcontrollers. In this domain, systems with 128 KiB RAM and less are quite common. There is no dynamic memory allocation used: the whole system is statically configured. Since the memory usage is derived directly from the configuration during compilation, the system is guaranteed not to require more memory than available at run time. The system and all the user applications are compiled and linked into a single binary and the static configuration is fixed in the binary[2]. As an example, the number of tasks, along with their priorities and – in the multi-core system – core-bindings cannot be changed at run time. Other fixed parameters include NVRAM use, as well as all communication partners of the ECU.

As mentioned above, AUTOSAR was devised as a single-core system. Multi-core support has been added recently and to our knowledge there is currently no software vendor that already has a multi-core implementation which is considered stable for production use. The reasons for this are quite simple: Due to its single-core roots, the less-than-five-years-old system already has to be considered *legacy software* and its integration into a multi-core system is a nontrivial task. AUTOSAR has foreseen this problems and devised a pragmatic solution in the standardization document of its multi-core approach.

[2]There is the possibility to have multiple configuration sets linked into the binary and select one of them during start-up.

A. The AUTOSAR Approach to Multi-Cores

In AUTOSAR 4.0, multi-core support for the system is still optional. As a consequence the multi-core architecture is described in a document of its own [2]. This document mainly describes the multi-core extensions to the operating system and gives a few hints to the overall architecture (depicted in Figure 1(b)). The key point of this architecture is to have a minimal changes to the single-core system. This is achieved by declaring the BSW safe for single-core use only, and force the system to schedule all[3] activations of the BSW always on the same core, called the *BSW core*. The sheer size of the BSW is a significant factor in this decision. A consequent re-factoring of the BSW would be an enormous effort that is hardly acceptable for an optional development feature. Since SW-Cs are independent of the ECU, they are also independent of the core in a multi-core system. Hence the application does not require special adaption.

B. The Problem

This architecture has a problem: Every access to the BSW from a non-BSW core requires a cross-core communication within the RTE. Such a cross-core communication is expensive: the current task of the BSW-core is interrupted by an inter-processor-interrupt which activates a task on the BSW core to perform the cross-core operation. This consumes execution time on the BSW core. At the same time the issuing non-BSW core may be idle, if the desired call is synchronous. This is in fact the common case since most BSW-calls in AUTOSAR return at least a status value. By comparison with similar system designs used in the past, we strongly assume that the BSW core will turn out to be a bottleneck.

[3]with exception of the ECU State Manager, which has to perform system start-up on all cores

III. Evaluating the AUTOSAR Multi-Core System

In order to prove this hypothesis, we performed measurements running a real multi-core AUTOSAR system on an automotive dual-core microcontroller.

A. Evaluation Setup

For evaluation purposes, we had access to an industrial AUTOSAR-solution containing RTE, OS and most of the BSW. Evaluation took place on an SPC56EL microcontroller, which contains two PowerPC e200z4 cores, 128 KiB shared RAM and 1 MiB flash memory. Our target was clocked at 80MHz. Timing measurements were always started and stopped on the same core, so the *time base* of the CPU core was used. The time base is a free-running timer which is incremented every clock cycle. It is 64 bits wide and set to zero on system start-up, so it is guaranteed not to overflow during our measurements.

B. Evaluation Scenario

Our evaluation scenario used an AUTOSAR-system with two identical SW-Cs, named *SWC1* and *SWC2*. SWC1 is always bound to the first core, whereas SWC2 is bound to the second core when the multiple cores are used. Each SW-C is triggered by an external message. It then reads the received signal from the BSW, and uses the value as input for a simple busy loop simulating a workload. Afterwards it sends the result back to an external receiver; this is the second BSW invocation.

The external messages are simulated CAN-messages. Instead of using a real CAN-bus, we replaced the CAN-driver with a small stub of our own. This stub can simulate the reception of a message to the BSW and it is called by the BSW when a CAN message shall be transmitted. Besides those two SW-Cs, the system contains a high-priority periodic task, which is required by a BSW module. This task is activated every ten milliseconds.

C. Measurements Performed

Using this scenario, we performed two types of measurements. First a **timing measurement**, where we measured the round-trip time from the simulation of the incoming CAN-message to the moment our CAN-driver stub is called to send the reply. We deliberately schedule the tasks with an adequate amount of idle-time surrounding them. This ensures that they neither interfere with each other, nor with any other system activity that could disturb the measurements. The times presented in the graphs and tables are mean times of ca. 1450 runs each. This number was determined by the amount of RAM available to hold the results. The relative standard error for all runs has been less than 1 percent.

The second kind of measurement was to measure the maximal **activation rate** of each SW-C, that is, how often can every SW-C be triggered per second when the system is under full load. To perform this measurement, our CAN driver stub initially simulates the reception of both CAN messages. At the moment it is called to send a reply, it confirms the reply and immediately triggers the next incoming message again. This full-load situation is bound to lead to heavy contention on shared resources, which – depending on the synchronization method used – causes more or less delay. This delay is main motivation for this kind of measurement, because it causes the rates to (sometime severely) deviate from the optimal (timing measurement)$^{-1}$. The generated figures are intended to tell us how good a synchronization approach works under worst-case conditions. The numbers presented in the graphs and tables are the mean activation rates in Hz, calculated from 10 runs of 15 seconds length each. The measurement was split into multiple runs to give more weight to the startup-phase, the 150 seconds total were chosen to get relative standard error below 1 percent in most of the cases[4]

Both kinds of measurement were repeated for different values for the number of loop cycles in the simulated workload.

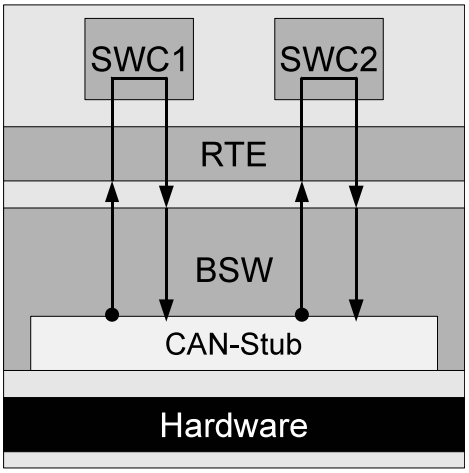

Figure 2. The evaluation scenario

D. Performance Comparison of Single- and Multi-Core AUTOSAR

Table I shows the timing measurement results without simulated workload, so nearly all the time measured is spent in the OS, RTE and BSW. The single-core times are nearly identical. There is a slight difference because though they are identical in implementation, they use different resources (such as OS tasks, RTE signals, etc.) at runtime. Access to those resources cannot always be performed in $O(1)$, hence the difference in run time.

Looking at the multi-core figures, we notice that already SWC1 shows some overhead, despite having the same con-

[4]It is less than 5% for all test runs

	SWC1	SWC2
single-core	79.59	80.28
multi-core	87.26	191.62
overhead	10%	139%

Table I
AVERAGE ROUND TRIP TIME IN μS WITHOUT SIMULATED WORKLOAD (AUTOSAR MULTI-CORE)

	SWC1	SWC2	sum
single-core	3840	3840	7680
multi-core	0	5367	5367
multi-core gain			-30%

Table II
AVERAGE NUMBER OF ACTIVATIONS PER SECOND WITHOUT SIMULATED WORKLOAD (AUTOSAR MULTI-CORE)

trol flow as the single-core variant. This overhead is really an optimization of the OS, when used in single-core mode: Most of the OS's internal state – most notably the ready queue – is replicated per core. So whenever the OS needs to access such a variable, it has to determine on which core it has been invoked to identify the correct set of variables to operate on. If the OS is compiled for single-core use, then these core-lookups are optimized away.

Accesses to SWC2 – located on the second core – show a really huge overhead. Since the code of the SW-C itself is identical to SWC1 this overhead of an additional 104 μs shows the *cross-core penalty* of the AUTOSAR system approach. This penalty is a constant sum of the two cross-core BSW accesses of the SW-C (*read signal* and *write signal*) and the overhead in the OS for cross-core task activation and task unblocking.

The results of the activation rate measurements are shown in Table II. On the single-core OS, the number of activations for both SWCs is identical. This has been expected, since we configured the related tasks to equal priorities and as a consequence they are scheduled in FIFO order. We were, however, surprised when we saw the results for the multi-core system. Despite having twice the computation power at hand, there were actually fewer activations processed in the overall system. Even more interesting: all of those activations were counted on the second core. This is caused by the method used for cross-core BSW accesses. There is a *proxy task* on the first core to carry out the BSW accesses for the SW-Cs from the second core. This task had a higher priority than the task running SWC1. In our scenario without simulated workload, both SW-Cs nearly only do BSW-accesses. It turns out that in this scenario the first core never gets to execute the task running SWC1. We did some experiments with the relevant tasks set to equal priority. These are described in more detail in Section VI.

IV. AN ALTERNATIVE SOLUTION

As our analysis has shown, the biggest and crucial part of the overhead is caused by dispatching BSW requests to the BSW core. The question is now: why can the BSW not be executed on every core? The answer lies in the fact that the BSW was specified and developed for single-core systems. So all synchronization measures it contains were designed only to protect against preemption. Critical sections (called *exclusive areas* in AUTOSAR) are synchronized in a non-preemptive semantic using the services of a dedicated module, the *BSW Scheduler* (abbreviated *SchM*). The configuration of this module can assign different synchronization methods to each exclusive area. These are most notably: AUTOSAR Resources, Interrupt Locking, and no protection at all. The latter one makes sense if an analysis of the system configuration concludes that there will be no contention on this specific exclusive area. The other synchronization methods are only applicable if the software runs on a single core.

We propose another pragmatic solution: Instead of executing the BSW on a single core, we allow it to run on any core, but assure that it executes only on a single core *at a time*, so all single-core synchronization mechanisms are still valid. This can be achieved with a simple locking mechanism. Every time when some thread of execution wants to access the BSW it acquires a lock first and releases it afterwards. This technique is not new. It has effectively been used for decades when adapting legacy operating systems to multiprocessing environments. A well-known example for this is the "Big Kernel Lock" in Linux (prior to Linux 2.4) [5]. Though the BSW is not part of the operating system in AUTOSAR, this comparison is still applicable, because in AUTOSAR the RTE is the interface to the application, not the OS. So everything below the RTE resembles what a "kernel" is in traditional monolithic systems. Consequently, we call our synchronization mechanism the *Big BSW Lock* (BBL), following the Linux naming.

A. Implementing the Big BSW Lock

We implemented the BBL as a simple spin-lock which needs to be taken whenever the BSW is entered. Most entries to the BSW are in the generated RTE methods. So the RTE generator needs to generate the additional code to acquire and release the lock surrounding the BSW calls. The next entry point we identified were cyclic BSW tasks, generated by the SchM. Since those tasks are also generated, again only an adaption of the according generator is necessary. The last entry point are interrupt service routines from the hardware drivers. These need to be adapted manually when they access shared data or functions from the BSW. An alternative method would be the introduction of ISR wrappers which acquire the lock and then call the original ISR. This would, however, create additional overhead.

	SWC 1	SWC2
single-core	79.59	80.28
multi-core (BBL)	106.25	107.66
overhead	33%	34%

Table III
AVERAGE ROUND TRIP TIME IN μS WITHOUT SIMULATED WORKLOAD (BIG BSW LOCK)

	SWC1	SWC2	sum
single-core	3840	3840	7680
multi-core (BBL)	3928	4003	7931
multi-core gain			3.3%

Table IV
AVERAGE NUMBER OF ACTIVATIONS PER SECOND WITHOUT SIMULATED WORKLOAD (BIG BSW LOCK)

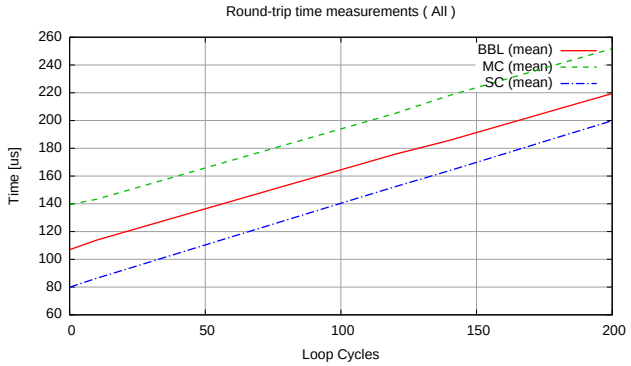

Figure 3. Timing measurements with simulated workload

B. Performance of the Big BSW Lock

We implemented the BSW Lock in the same measurement environment as described in Section III. Since we had no access to the generator sources, we manually adapted the code after generation. Table III shows the results of the timing measurements in comparison to the single-core OS. The time for SWC1 is in fact even higher than it is with AUTOSAR Multicore. The additional overhead can be explained with the lock handling. On the other hand, we see that the run time of SWC2 is nearly identical to that of SWC1. This is caused by it running completely on the second core, with no cross-core overhead occurring. This is a big improvement over the AUTOSAR approach.

We performed the activation rate measurements next and were finally able to see an improvement over the single-core case (see Table IV). The gain is not really significant, but this comes not as a surprise, since these measurements were also performed without simulated workload. Most of the time is spent in the BSW, so that the lock is nearly permanently owned by one of the processors. The first core can perform a little less activations, because the cyclic BSW task is scheduled to run on this core.

V. COMPARISON OF SINGLE-CORE, MULTI-CORE AND BIG-BSW-LOCK AUTOSAR

Following the overhead measurements we slowly increased the simulated work load. Figure 3 shows the absolute time in dependence of the number of loop cycles. The times shown are mean times of SWC1 and SWC2. The time measurements showed a linear increase with the number of loop cycles, which meets our expectations.

The activation rate measurements are more interesting. Figure 4 shows the activation rates of the three system designs. Each line in the graph represents the sum of the activations of both SWC1 and SWC2. We see that in general the rates drop as the workload increases. The exception is at the beginning of the BBL-line. Here we see that the rates are even raising a bit and then stay nearly constant for a certain amount of workload. This means that the system shows higher performance as contention on the lock decreases. The AUTOSAR multi-core system lies significantly below the single-core system, but the downward slope is less pronounced, so we can expect that the lines cross eventually.

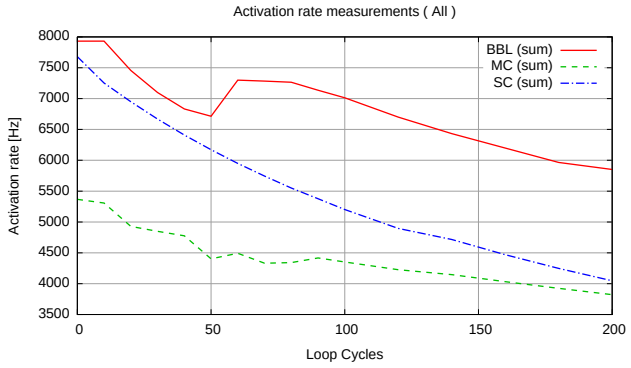

Figure 4. Activation rate measurements with simulated workload

From the activation rates measured, we derived what we call the *multicore gain*, defined as the surplus of activations that is gained by using a second core. The resulting graph is shown in Figure 5. From this graph it is easier to point out that the multicore-gain is increasing when the workload is higher, because contention on the BSW decreases.

VI. DISCUSSION

Our figures back up our assumption that the AUTOSAR multi-core system performs badly in a full-load scenario. Despite having twice the CPU resources, our system performed even worse than the single-core system with the same workload. Our timing measurements have also shown significant cross-core overhead in a concurrency-free scenario, which applies to every application running on a non-BSW core.

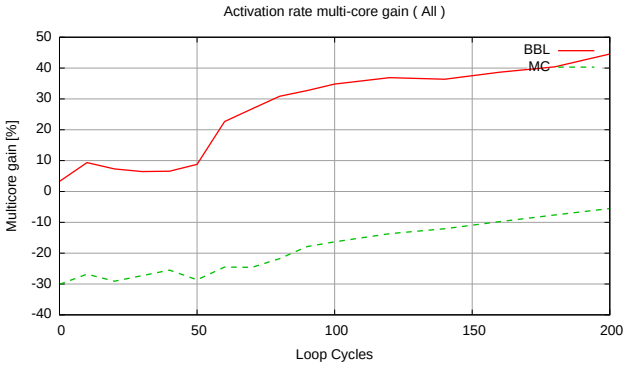

Figure 5. Multi-Core performance gain with simulated workload

A full-load scenario as used for our activation rate measurements is admittedly not likely to occur in real-world applications where periodic processing dominates. Since we had no access to real-world automotive applications, we were not able to perform an evaluation on a more realistic scenario. We consider our scenario being the worst-case for this kind of system. We can therefore conclude that multi-core AUTOSAR primarily makes sense for long-running, calculation-intensive applications that have limited cross-core communication.

We have furthermore shown, that a global BSW lock provides a simple and relatively inexpensive modification to the system, which drastically improves the system performance. The key behind this is reducing the number of cross-core accesses.

So the question is, why did AUTOSAR itself not consider some kind of BSW lock? We need to point out here, that AUTOSAR does not forbid this. Implementers are free to adapt the BSW to multi-core environments by any means they find appropriate. On the other hand, the big BSW lock we have implemented does not come without disadvantages. The locking of the BSW on one core creates possibly big latencies on another core that wants to use the BSW. Such latencies should be avoided in real-time systems, because they are hard to estimate, yet they need to be taken into account during the static timing analysis of the system. If those latencies are considered, they can significantly reduce the possible utilization of the system. This is reflected in our figures, where we see that the performance of the BBL on the dual-core system – though significantly better than the AUTOSAR approach – is far from being twice the performance of the single-core system. Additionally, we have to consider that the potential of the AUTOSAR approach may not yet be fully exploited in the available implementation.

Proxy Task Priority

In reaction to the somewhat strange results of the activation rate measurements with the AUTOSAR approach, we also analyzed the effect of the proxy task priority on the system performance. Since SWC1 and SWC2 are identically configured, the tasks which execute them have the same priority. This leads to the observed FIFO-scheduling when both tasks execute on the same core. When using the AUTOSAR-approach SWC2 (running on the second core) is supported by a proxy task running on the BSW core. The priority of the proxy task may be chosen either equal or higher than the priority of the tasks running the SW-Cs. Depending on this choice, different blocking occurs when SWC2 issues a BSW call while SWC1 is running:

- **Equal priority** The SWC2 proxy task and the SWC1 task are scheduled in FIFO order. The proxy task needs to wait until the SWC1 task finishes executing. This waiting time adds to the blocking time of SWC2. The duration of the waiting time is variable depending on the remaining time SWC1 is running.
- **Higher priority** When the proxy task gets activated, SWC1 gets preemption-blocked. The blocking time spans the runtime of the proxy task, that is the execution of the BSW call issued by SWC2. The blocking duration can assumed to be nearly constant for identical BSW calls.

Figure 6 shows the results of the activation rate measurements for both options. For higher values of simulated workload the higher priority proxy task shows better performance, due to the advantage of the near-constant blocking time. Since our overall measurements show that the AUTOSAR approach is best suited for long running SW-Cs, we chose this approach for the presented measurements. Please note that the absolute numbers of these measurements differ from those presented in Section III-D, because they were performed using a slightly different system software revision[5].

VII. Ongoing Research

We plan to evaluate more alternative approaches to design an AUTOSAR multi-core system. In contrary to AUTOSAR, we do not restrict ourselves to solutions that leave the BSW unchanged. We know that every adaption in the BSW is a significant cost-factor for automotive software-vendors, but our goal is to provide a cost-benefit analysis of different system designs that may be an improvement to

Our next step is to introduce **Fine-Grained Locks** as a synchronization mechanism in the BSW to allow (mostly) concurrent execution. Since it is very time-consuming to apply this to the whole AUTOSAR BSW, we chose to apply this to a single BSW-module only. The other modules

[5]most notably a different cache configuration

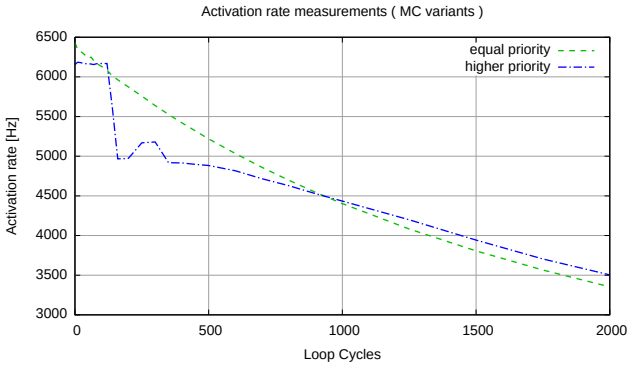

Figure 6. AUTOSAR Multi-Core performance for different priorities of the BSW proxy task

will not be adapted, but replaced by a minimalistic re-implementation that provides only the features required for our evaluation scenario. We have already implemented these mini-modules and a first step of the fine-grained locks. The results look promising, but some need additional analysis before they can be published.

A. Future work

After finishing the evaluation of the fine-grained lock synchronization, we will move on to evaluate the following alternatives:

1) **Non-Blocking Synchronization**. This approach synchronizes shared memory structures using non-blocking mechanisms. Especially wait-free algorithms as defined by Herlihy [6] would ease the real-time analysis of the resulting system. All non-blocking algorithms are not trivial in their application and most of them depend on special atomic instructions such as compare-and-swap in the underlying hardware's instruction set. This dependence on certain assembler instructions make non-blocking synchronization mechanisms difficult to port across different platforms
2) **Partitioning**. A completely different approach is to run a single-core AUTOSAR system on every core and partition the MCU's resources like RAM, ROM, peripherals, etc. so that those systems are independent of each other. This promises some advantages: nothing of the AUTOSAR-system needs to be specifically adapted to multi-core environments, and all real-time analysis that can be performed on single-core systems still apply. A foreseeable drawback is of course the multiplication of the system footprint (this is why AUTOSAR dropped this approach) and more expensive cross-core communication (because it needs to use an external bus).

In addition we will analyze the real-time properties of all approaches and perform more measurements with still increased simulated workload to determine the point at which the AUTOSAR multi-core system will be profitable in a way that it provides significantly more performance than the single-core system.

VIII. Summary

In this paper we presented the AUTOSAR multi-core system and set up a simple evaluation scenario on a dual-core processor to measure its performance against the single-core system. Plain run-time measurements showed a significant overhead for software components running on the second core. To our surprise, a full-load scenario showed that the dual-core system provides even less throughput in total than the single-core system. We analyzed this and found that most of the overhead occurs on the first core, whereas the second core idles most of the time. We proposed a pragmatic alternative of a "Big Kernel Lock" to distribute the load better between the two processors. Our measurements prove that this is significantly more efficient than the AUTOSAR approach. Our next step is to implement and evaluate more sophisticated alternatives that promise even more efficiency. Since the work effort to implement such an alternative is an important factor for the automotive software industry, we will combine these performance evaluations with an effort evaluation to create a cost-benefit analysis.

IX. Availability

The source code of the applications used for the measurements is available at `http://www4.cs.fau.de/~boehm/Research/OSPERT11_Eval_Application.tar.bz2`

The AUTOSAR system used is a commercial product and is not freely available.

References

[1] AUTOSAR. Layered Software Architecture (Version 2.2.2). Technical report, Automotive Open System Architecture GbR, August 2008.

[2] AUTOSAR. Specification of Multi-Core OS Architecture (Version 1.0.0). Technical report, Automotive Open System Architecture GbR, November 2009.

[3] AUTOSAR homepage. http://www.autosar.org/, visited 2009-03-26.

[4] Theodore P. Baker. A Stack-Based Resource Allocation Policy for Realtime Processes. In *Proceedings of the 12th International Conference on Real-Time Systems (RTSS '91)*, pages 191–200. IEEE Computer Society Press, 1991.

[5] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001.

[6] Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 11:124–149, 1991.

Operating Systems Challenges for GPU Resource Management *

Shinpei Kato and Scott Brandt
University of California, Santa Cruz

Yutaka Ishikawa
University of Tokyo

Ragunathan (Raj) Rajkumar
Carnegie Mellon University

Abstract

The graphics processing unit (GPU) is becoming a very powerful platform to accelerate graphics and data-parallel compute-intensive applications. It significantly outperforms traditional multi-core processors in performance and energy efficiency. Its application domains also range widely from embedded systems to high-performance computing systems. However, operating systems support is not adequate, lacking models, designs, and implementation efforts of GPU resource management for multi-tasking environments.

This paper identifies a GPU resource management model to provide a basis for operating systems research using GPU technology. In particular, we present design concepts for GPU resource management. A list of operating systems challenges is also provided to highlight future directions of this research domain, including specific ideas of GPU scheduling for real-time systems. Our preliminary evaluation demonstrates that the performance of open-source software is competitive with that of proprietary software, and hence operating systems research can start investigating GPU resource management.

1 Introduction

Performance and energy are major concerns for today's computer systems. In the early 2000s, chip manufacturers had competed on processor clock rate to continue performance improvements in their product lines. For instance, the Intel Pentium 4 processor was the first commercial product that exceeded a clock rate of 3 GHz in 2002. This performance race on clock rate, however, came to end due to power and heat problems that prevent the chip design from increasing clock rate in classical single-core technology. Since the late 2000s, performance improvements have continued to come through innovations in multi-core technology, rather than clock-rate increases. This paradigm shift was a breakthrough to achieve high-performance with low-energy. Today, we are getting into the "many-core" era, in order to meet the further performance requirements of emerging data-parallel and compute-intensive applications. Not only high-performance computing (HPC) applications but also embedded applications, such as autonomous vehicles [39, 41] and robots [26], benefit from the power of many-core processors to process a large amount of data obtained from their operating environments.

*This work is supported by the fund of Research Fellowships of the Japan Society for the Promotion of Science for Young Scientists.

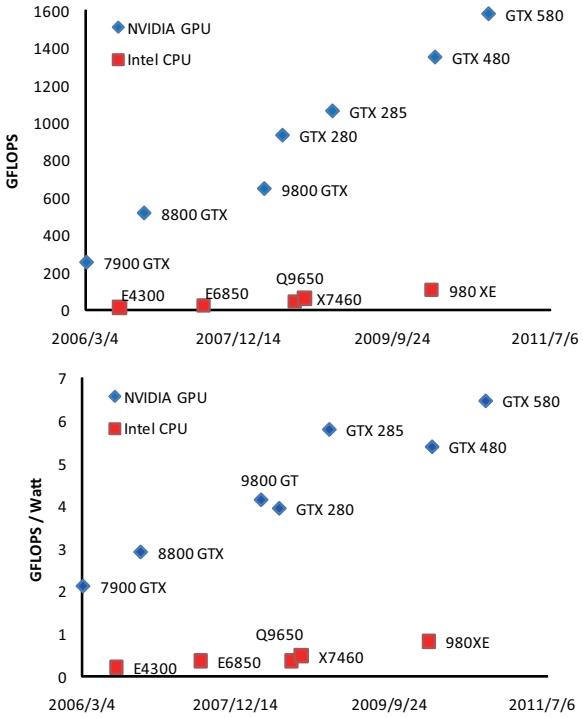

Figure 1. Performance trends on the well-known GPU and CPU architectures.

The graphics processing unit (GPU) has become one of the most powerful platforms embracing the concept of many-core processors. Figure 1 illustrates recent performance trends on the well-known GPU and CPU architectures from NVIDIA and Intel. The single-chip peak performance of the state-of-the-art GPU architecture exceeds 1500 GFLOPS, whereas that of a traditional microprocessor is around 100 GFLOPS at best. This non-trivial performance advantage of the GPU comes from hundreds of processing cores integrated on a chip. The GPU is also more preferable than the CPU in performance per watt. Specifically, the GPU is about 7 times more energy-efficient than the CPU today.

A recent announcement from the TOP500 supercomputing sites disclosed [40] that three of the top five supercomputers comprise GPU clusters, and significant performance improvements are provided by such GPU clusters for scientific applications [38]. Large-scale storage systems also benefit from the GPU [1, 9, 19]. In fact, the Amazon EC2 cloud computing ser-

vice leverages GPU clusters to build their data centers. In the embedded systems domain, a new version of Carnegie Mellon's autonomous vehicle [41] equips four NVIDIA's GPUs to enhance its computing power required for autonomous driving tasks, including vision-based perception and motion planning. A case study from Stanford [39] revealed that the GPU can speed up computer vision applications for autonomous driving by 40 times compared to CPU execution. Such a rapid growth of general-purpose computing on GPUs, also known as *GPGPU*, is supported by recent advances in programming technology enabling the GPU to be used easily for general "compute" problems.

Despite the success of GPU technology, operating systems support in commodity software [6, 28, 33] is very limited for GPU resource management. Multi-tasking concepts, such as fairness, prioritization, and isolation, are not supported at all. The research community has developed several approaches to GPU resource management recently. In particular, notable contributions include TimeGraph [17] providing capabilities of prioritization and isolation, and GERM [2] with fairness support, for multi-tasking GPU applications. Despite the very limited information of GPU hardware details available to the public, these studies made efforts to develop GPU resource management primitives at the device-driver level. However, their functionality is limited to specific workloads. There are also other research projects on GPU resource management provided on the layers above the device driver, including CPU schedulers [8], virtual machine monitors [7, 11, 12, 20], and user-space programs [3, 10, 36], but their basic performance and capabilities are limited to underlying commodity software. We believe that operating systems research must explore and address GPU resource management problems to enable GPU technology in multiple application domains.

This paper identifies several directions towards operating systems challenges for GPU resource management. Currently, we lack even a fundamental GPU resource management model that could underlie prospective research efforts. The missing information of open-source software is particularly a critical issue to explore systems design and implementation of GPU resource management. In this paper, we present initial ideas and potential solutions to these open problems. We also demonstrate that open-source software is now ready to be used reliably for research.

The rest of this paper is organized as follows. Assumptions behind this paper are described in Section 2. Section 3 presents the state-of-the-art GPU programming model, and Section 4 introduces a basic GPU resource management model for the operating system. Section 5 provides operating systems challenges for GPU resource management, including a preliminary evaluation of existing open-source software. The concluding remarks of this paper is presented in Section 6.

2 Our System Assumptions

This paper considers heterogeneous systems composed of multi-core CPUs and GPUs. Several GPU architectures exist today. While many traditional microprocessors designed based on the X86 CPU architecture compatible across many generations over decades, GPU architectures tend to change in years. NVIDIA has released the Fermi architecture [30] as of 2011, supporting both compute and graphics programs. This paper focuses on the Fermi architecture, but the concept is also applicable to other architectures. We also assume an *on-board* GPU. Although Intel provides a new X86-based architecture, called Sandy Bridge [13], which integrates the GPU on a chip, GPUs on a board are still more popular today. In future work, however, we will study implications of on-board and on-chip GPUs from the operating systems point of view.

Given an on-board GPU model, we assume that the CPU and the GPU operate asynchronously. In other words, CPU contexts and GPU contexts are separately processed. Once user programs launch a piece of code onto the GPU to get accelerated, it is offloaded from the CPU. The user programs may continue to execute on the CPU while this piece of code is processed on the GPU, and may even launch another piece of code onto the GPU before the completion of preceding GPU code. The GPU queues this launch request, and executes the code later when it is available.

3 Programming Model

The GPU is a device to accelerate particular program code rather than a control unit like the CPU. User programs hence start execution on the CPU, and launch pieces of code, often referred to as *GPU kernels*[1], onto the GPU to get accelerated. There are *at least* three major steps for user programs to take to accelerate on the GPU.

1. **Memory Allocation**: First of all, user programs must be allocated memory spaces on the host and device memory that are required for computation. There are several types of memory for the GPU: *shared*, *local*, *global*, *constant*, and *heap*.
2. **Data Copy**: Input data must be copied from the host to the device memory before the GPU kernel starts on the GPU. Usually output data is also copied back from the device to the host memory to return the computed result to user programs.
3. **Kernel Launch**: GPU-accelerated program code must be launched from the CPU to the GPU at runtime, as the GPU itself is not a control unit.

The memory-allocation phases do not likely access the GPU, and must manage the device memory address regions available for each request. The data-copy and kernel-launch phases, on the other hand, need to access the GPU to move data between the host and the device memory, and launch GPU program code. Figure 2 illustrates an example showing a brief execution flow of GPU-accelerated matrix multiplication, i.e., $A[] \times B[] = C[]$. The GPU kernel image must be loaded on the

[1] To avoid misunderstandings, a term "kernel" always indicates GPU program code and never points to the operating system kernel in this paper.

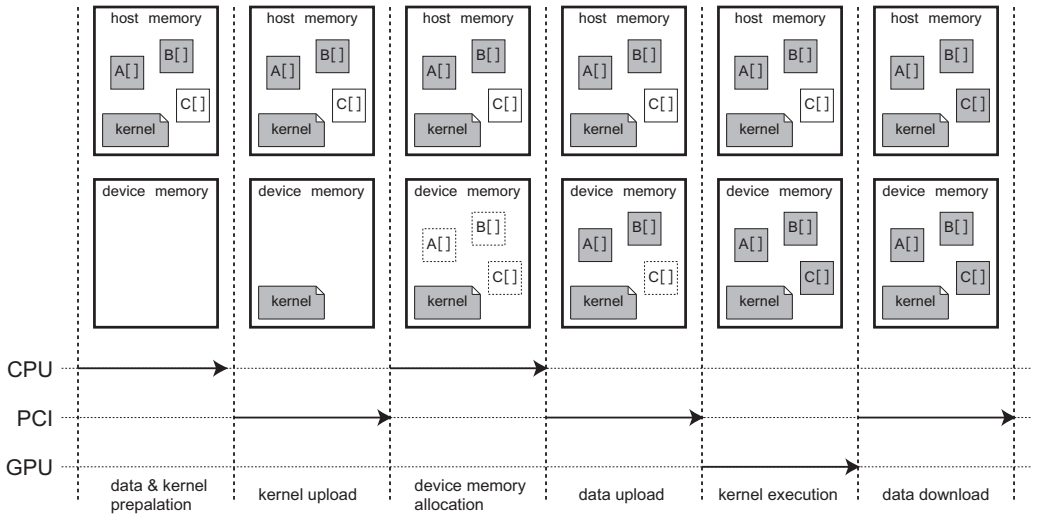

Figure 2. Example of an execution flow of matrix multiplication $A[] \times B[] = C[]$.

host memory. Two input buffers, $A[]$ and $B[]$, must also hold valid values for computation, while an output buffer $C[]$ may be empty. The kernel image is usually uploaded at the beginning. Since the GPU uses the device memory for data access, the data spaces must be allocated on the device memory. The input buffers are then copied onto these allocated data spaces on the device memory via the PCI bus. After the input data are ready on the device memory, the GPU kernel starts execution. The output data are usually copied back onto the host memory. This is a generic flow to accelerate the program on the GPU.

GPU programming requires the device and the host parts. The device part contains kernels coded by GPU instructions, while the host part is more like a main thread running on the CPU to control data copies and kernel launches. The host part can be written in an existing programming language, such as C and C++, but must be aligned with an application programming interface (API) defined by the programming framework to communicate with the device part. The following are well-known GPU programming frameworks:

- **Open Graphics Language (OpenGL)** provides a set of library functions that allow user-space applications to program GPU shaders and upload it to the GPU to accelerate 2-D/3-D graphics processing.
- **Open Computing Language (OpenCL)** is a C-like programming language with library functions support. It can parallelize programs conceptually on any device, such as GPUs, Cell BE, and multi-core CPUs.
- **Compute Unified Device Architecture (CUDA)** is also a C-like programming language with library functions support. It can parallelize programs like OpenCL, but is dedicated to the GPU.
- **Hybrid Multicore Parallel Programming (HMPP)** is a compiler pragma extension to parallelize programs conceptually on any device. OpenMP also employs this programming style but is dedicated to multi-core CPUs.

Graphics processing is typically more complicated than general computing. A graphics pipeline comprises dozens of stages, where vertex data come in from one end of the pipeline, got processed in each stage, and the rendered frame comes out the other end. Some stages are programmable and others are not. For example, vertex and pixel shaders are programmable, but rasterization and format conversion are fixed functions. General computing, meanwhile, uses only shader units, also known as compute (or CUDA) cores. It depends on user-space programs to upload GPU code. The GPU code can be compiled at runtime or offline. Compute programs are often compiled offline as just parallelized on compute cores, while graphics programs would need runtime compilation, since they use many shaders for graphics operations. Once compiled as GPU code binaries, however, they are loaded onto the GPU at runtime in the same manner. Hence, the software stack needs no modification in the operating system.

4 Resource Management Model

In this section, we present a basic model of GPU resource management, particularly along with the Linux system stack, accommodating NVIDIA's proprietary driver [28], PathScale's open-source driver [33], and Linux's open-source driver [6]. Windows Display Driver Model (WDDM) [35] may also be applicable to our model, since NVIDIA could share about 90% of code between Linux and Windows [34]. As mentioned in Section 2, the following discussion assumes the NVIDIA's Fermi architecture [30], but is also conceptually applicable to most of today's GPU architectures.

4.1 System Stack

The GPU architecture defines a set of *GPU commands* to enable the device driver and the user-space runtime engine to control data copies and kernel launches. The device driver provides primitives for user-space programs to send GPU com-

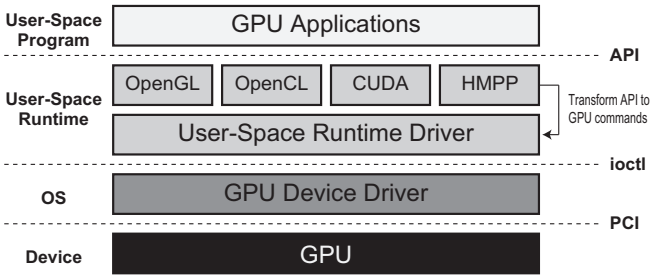

Figure 3. System stack for GPU processing.

mands to the GPU, and the user-space runtime engine provides a specific API to write user programs, abstracting the low-level primitives at the device driver. An `ioctl` system call is often used to interface between the device driver and the runtime engine. GPU commands are different GPU instructions that code GPU kernels, and there are many types of GPU commands. The device driver sends GPU commands to manage GPU resources including GPU contexts and device memory management units, while the runtime engine generates GPU commands to control the execution flows of user programs, e.g., data copies and kernel launches.

Figure 3 illustrates the system stack in our GPU resource management model. Applications call API library functions provided by the runtime engine. The front-end of the runtime engine is dependent on the programming framework, which transforms the API calls to GPU commands that are executed by the GPU. The runtime driver is the back-end of the runtime engine that abstracts the `ioctl` interface at the device driver, simplifying the development of programming frameworks. Performance optimization of the runtime engine can be unified in this layer. The device driver is responsible for submitting the GPU commands, received through the `ioctl` system call, to the GPU via the PCI bus.

4.2 GPU Channel Management

The device driver must manage GPU channels. The GPU channel is an interface that bridges across the CPU and the GPU contexts, especially when sending GPU commands from the CPU to the GPU. It is directly attached to the dispatch unit inside the GPU, which passes incoming GPU commands to the compute (or rendering) unit where GPU code is executed. The GPU channel is the only way to send GPU commands to the GPU. Hence, user programs must be allocated GPU channels. Multiple channels are supported in most GPU architectures. For instance, the NVIDIA's Fermi architecture supports 128 channels. Since each context requires at least one channel to use the GPU, at most 128 contexts are allowed to exist at the same time. GPU channels are independent of each other, and represent separate address spaces.

Figure 4 illustrates how to submit GPU commands to the GPU within a channel. The GPU channel uses two types of buffers in the operating-system address space to store GPU commands. One is memory-mapped onto the user-space

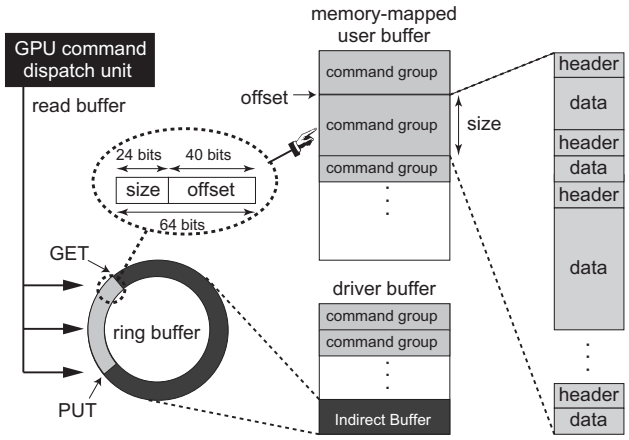

Figure 4. GPU command submissions.

buffer, into which the user-space runtime engine pushes GPU commands. The other buffer is directly used by the device driver to send specific GPU commands that control the GPU, such as status initialization, channel synchronization, and mode setting. GPU commands are usually grouped into multiple units, often referred to as GPU command groups. There are no constraints on how to compose GPU command groups. A single GPU command group may contain only one GPU command or numbers in the thousands, as far as the memory space allows. Regardless of how many GPU command groups are submitted, the GPU treats them as just a sequence of GPU commands. However, the number of GPU command groups could affect throughput, since GPU commands in each group are dispatched by the GPU in a burst manner. The more GPU commands are included in a group, the less communication is required between the CPU and the GPU. Instead, it could cause long blocking durations, since the device driver cannot *directly* preempt the executions of GPU contexts launched by preceding sets of GPU commands.

While the runtime engine pushes GPU commands into the memory-mapped buffer, it also writes *packets*, each of which is a (*size* and *address*) tuple to locate the corresponding GPU command group, into a specific ring buffer provided in the operating-system buffer, often referred to as the indirect buffer. The device driver configures the GPU command dispatch unit to read this ring buffer to pull GPU commands. This ring buffer is controlled by `GET` and `PUT` pointers. The pointers start from the same place. Every time packets are written to the buffer, the device driver moves the `PUT` pointer to the tail of the packets, and sends a signal to the GPU command dispatch unit to pull the GPU command groups located by the packets between the `GET` and `PUT` pointers. Afterward, the `GET` pointer is automatically updated to the same place as the `PUT` pointer. Once these GPU command groups are submitted to the GPU, the device driver does not manage them any longer, and just continues to submit the next set of GPU command groups, if any. As a consequence, this ring buffer plays a role of a command queue for the device driver.

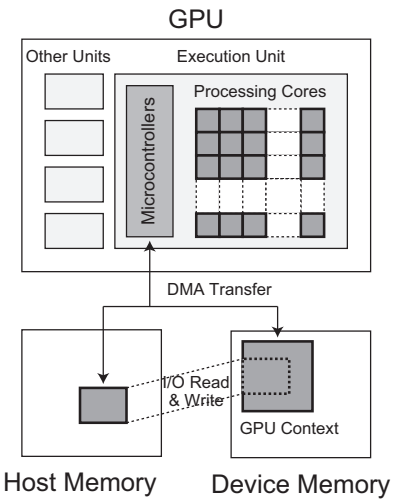

Figure 5. GPU context management model.

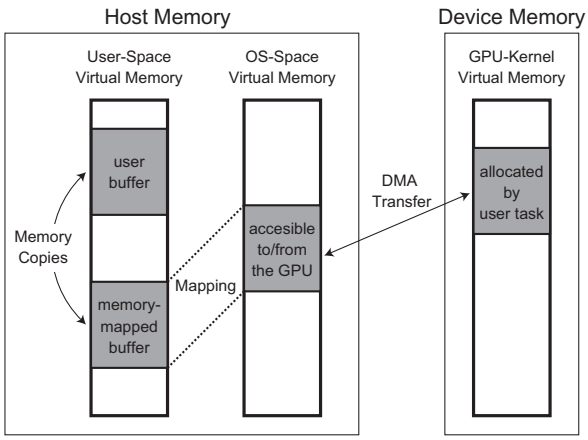

Figure 6. Host-device data copy model.

Each GPU command group may include multiple GPU commands. Each GPU command is composed of the header and data. The header contains *methods* and the data size, while the data contain the values being passed to the methods. Some methods are shared between compute and graphics, and others are specific to each. GPU command execution is out-of-order within the same GPU channel, and GPU channels could be switched implicitly by the GPU itself. It should be noted that the GPU channel is a path to send GPU commands and some other structures to control the GPU. GPU kernel images and their data buffers are uploaded onto the device memory through direct memory access (DMA), while DMA operation itself is controlled by GPU commands.

4.3 GPU Context Management

GPU contexts consist of memory-mapped I/O registers and hardware registers, which must be initialized by the device driver at the beginning. While memory-mapped I/O registers can be directly read and written by the device driver through the PCI bus, hardware registers need GPU sub-units to be read and written. There are multiple ways provided to access the context values. Reading from and writing to memory-mapped I/O registers on the CPU is the most straightforward way, but PCI bus communications are generated for all such operations. The GPU alternatively provides several hardware units to transfer data among the host memory, the device memory, and GPU registers in a burst manner.

Figure 5 illustrates a conceptual model of how to manage the GPU context. Generally, the GPU context is stored on the device memory. It could also be stored on the host memory, as the GPU can access both the host and device memory, but the device memory is strongly recommended due to performance issues, i.e., the GPU accesses the device memory much faster than the host memory. The host memory is often used to store one-time accessed data, such as the firmware program image of microcontrollers. Some parts of the GPU context are not directly accessible to the host memory. DMA transfers are needed to manage such parts of the GPU context through the microcontroller. It should be noted that the microcontroller also contains memory spaces where some data accessed by the GPU context are stored. Hence, DMA transfers are also needed to manage such data. The GPU has many hardware units other than the execution unit, some of which are used by the GPU context, but we do not describe details.

4.4 Memory Management

Memory management for GPU applications is associated with at least three address spaces. Given that a user program starts on the CPU first, the user buffer is created within the user-space virtual memory on the host memory. This buffer must be copied to the operating-system virtual memory, since the device driver must access it to transfer data onto the device memory. The destination of the data transfer on the device memory must match the address space allocated by the user program beforehand for the corresponding GPU kernel program. As a consequence, there are three address spaces associated with memory management: (i) the user-space virtual memory, (ii) the operating-system virtual memory, and (iii) GPU-kernel virtual memory.

Figure 6 depicts how to copy data from the user buffer on the host memory to the allocated buffer on the device memory. The user buffer must be first copied to the operating-system virtual memory space accessible to the device driver. A memory-mapped buffer may be used for this purpose, which is supported by the POSIX standard as the `mmap` system call. Once the buffer is memory-mapped, the user program can use it quite flexibly. Another approach to this data copy is that the user program communicates with the device driver, using I/O system calls, given that most operating systems provide a function to copy data from the user buffer to the operating-system virtual memory. Figure 6, however, assumes the first approach, which is adopted by Nouveau and PSCNV (and perhaps NVIDIA's proprietary driver). Once the buffer

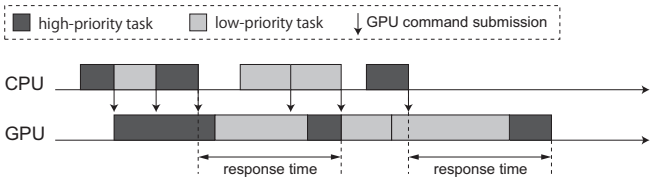

Figure 7. GPU processing without scheduling.

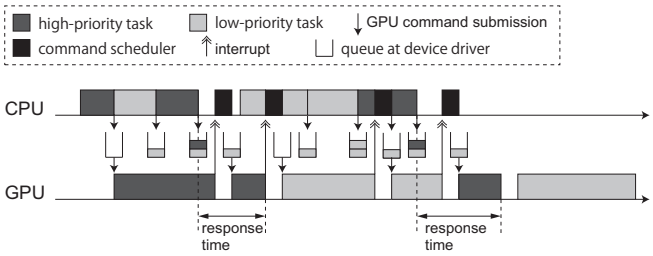

Figure 8. GPU processing with scheduling.

is copied to the operating-system virtual memory, the device driver transfers it to the device memory space allocated by the user program. Usually, the GPU provides virtual memory for each GPU channel to support multi-tasking, and the address of this device virtual memory is visible to user-space so that they can handle where to send data for computation.

5 Operating Systems Challenges

In this section, we consider operating systems challenges for GPU resource management. The following discussion is based on our experience and favorite research perspective, and does not cover the complete area of GPU resource management. The list of challenges provided herein must extend to advance further operating systems research for GPU technology. We however believe that the following discussion will lead to ideas of where we are at and where to go.

5.1 GPU Scheduling

GPU scheduling is perhaps the most important challenge to leverage the GPU in multi-tasking environments. Without GPU scheduling, GPU kernel programs are launched in first-in-first-out (FIFO) fashion, since the GPU command dispatch unit pulls GPU command groups in their arrival order. Hence, GPU processing becomes non-preemptive in a strong sense. Figure 7 illustrates a response-time problem caused due to the absence of GPU scheduling support, where two tasks with different priorities launch GPU code three times each as depicted on the CPU time line. The first launch of the high-priority task is serviced immediately, since the GPU has been idle. However, this is not always the case in multi-tasking environments. For instance, the second and the third launches of the high-priority task are blocked by the preceding executions of GPU contexts launched by the low-priority task. This blocking problem appears due to the nature of FIFO dispatching. Thus, tasks accessing the GPU need to be scheduled appropriately to avoid interference on the GPU.

Design Concept: The design of GPU schedulers falls into two categories. One approach implements a scheduler at the device-driver level to reorder GPU command groups submissions, given that the executions of GPU contexts are launched by GPU commands. GPU schedulers in the state of the art [2, 17] are designed based on this approach. For instance, TimeGraph [17] queues GPU command groups in the device driver space, and configures the GPU to generate interrupts to the CPU from the GPU upon completions of GPU code launched by prior GPU command groups so that the scheduler can be invoked to dispatch the next GPU command groups. Scheduling points are hence created at GPU command group boundaries. TimeGraph particularly dispatches GPU command groups according to task priorities, as depicted in Figure 8. The high-priority task can thereby respond quickly on the GPU whereas introducing additional overhead for the scheduling process. This is a trade-off, and it was demonstrated that this overhead is inevitable to protect important GPU applications from performance interference [16, 17].

Unfortunately, this device-driver approach still suffers from the non-preemptive nature of GPU processing. Specifically, the device driver can reorder GPU command groups submissions, but cannot directly preempt GPU contexts. For instance, the example in Figure 8 shows that the third launch of the high-priority task needs to wait for the completion of the second launch of the low-priority task. To make the GPU fully preemptive, GPU context switching needs to be supported at the microcontroller level, as mentioned in [17]. The design and implementation of such microcontrollers firmware, however, are very challenging issues. Some ideas could be extended from satellite kernels [27].

There are several other approaches to GPU scheduling. As studied in [8], the CPU scheduler is still effective in controlling GPU execution, since GPU code is launched from the CPU, and interrupts from the GPU are received on the CPU. However, GPU resource management is inevitably coarse-grained with this approach, due to the fact that the CPU scheduler is not aware of GPU command submissions and GPU contexts. We may also use compile-time and application-programming approaches [3, 10, 36], if modifications and recompilations of programs, using specific compilers, APIs, and algorithms, are acceptable. GPU resource management at the device-driver level, on the other hand, is finer-grained, and there is no need to compromise the generality of programming frameworks.

Scheduling Algorithm: Assuming that GPU schedulers are provided, the next question is: what scheduling algorithms are desired? We believe that at least two scheduling algorithms are required due to the hierarchy of the CPU and the GPU.

First, we insist that the CPU scheduling algorithm should consider the presence of the GPU. Particularly for real-time systems, classical deadline-driven algorithms, such as Earliest Deadline First (EDF) [23], are not effective as they are. Figure 9 shows an example where two tasks, one accesses the

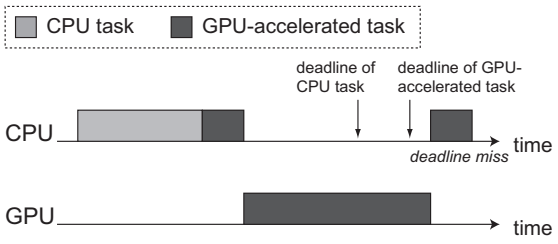

Figure 9. Deadline-driven CPU scheduling in the presence of the GPU.

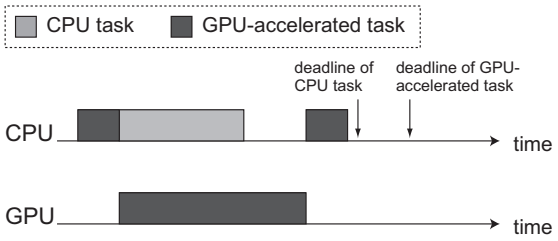

Figure 10. GPU-aware CPU scheduling in the presence of the GPU.

GPU ("GPU-accelerated task") and the other does not ("CPU task"), are scheduled on the CPU using the EDF algorithm. Consider that the CPU task is assigned a higher priority at some time, while the GPU-accelerated task has a large computation time on the GPU. Since GPU code is never launched until the CPU task is completed, the GPU remains idle while the CPU task is executing, and the CPU remains idle while the GPU-accelerated task is executing on the GPU. A more efficient algorithm should be developed to generate such a schedule that is depicted in Figure 10, where CPU and GPU times are effectively overlapped. Laxity-driven algorithms, such as Earliest Deadline Zero Laxity (EDZL) [4] and Earliest Deadline Critical Laxity (EDCL) [18], could alternate EDF, but an in-depth investigation is desirable.

Scheduling parallel tasks on the GPU is another issue of concern. Gang scheduling and co-scheduling [31] are well-known concepts for such a parallel multi-tasking model. The real-time systems community has also explored these scheduling methods recently [14, 21]. We believe that the concepts of gang scheduling and co-scheduling are useful to design the GPU scheduling algorithm. However, we must consider the constraints of the GPU. In general, the GPU is designed to execute threads in some group, also known as a warp in the Fermi architecture, simultaneously. Hence, the minimum unit of scheduling is a block of threads instead of a single thread. We still believe that existing concepts of parallel job scheduling are applicable, while algorithms implementation is a very challenging issue.

Data-copy Scheduling: As shown in Figure 2, data must be copied on to the device memory before GPU code is executed, and it is often required to copy the computation result

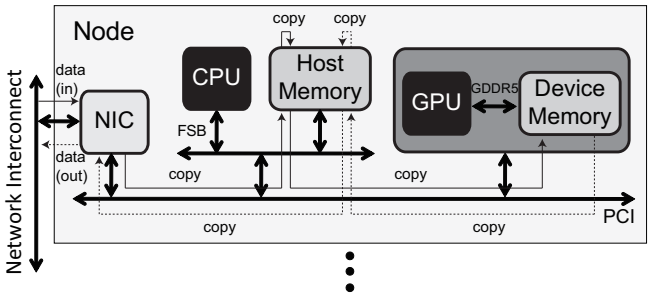

Figure 11. Data communication with the GPU.

back on to the host memory. The amount of time taken to perform the data copy depends on the data size. Furthermore, data-copy operations generate non-preemptive regions, which affect the performance and responsiveness of involved applications. Specifically, since data copies between host and device memory spaces are performed by DMA, as shown in Figure 6, and the device driver provides no way to preempt these DMA transfers once they are launched. Therefore, GPU scheduling must also be involved in data copies as well as the executions of GPU contexts.

5.2 GPU Clustering

Further research challenges include support for clustered multiple GPUs. GPU clustering is a key technology to use GPUs for HPC applications. Currently, we are developing this technology based on our GPU resource management model and GPU scheduling schemes to provide first-class support for GPU-based interactive data centers, supercomputers, and cyber-physical systems. These applications are data-intensive as well as compute-intensive. Hence, performance is affected by data communications across multiple GPUs.

GPU clusters are generally hierarchical. Each node is composed of a small number of GPUs clustered on a board. Many such nodes are further clustered as a system. Since these two types of clustering use different technologies [37], operating systems support must be provisioned differently.

On-board GPU clusters: The management of multiple GPUs on a board may be either in the user-space runtime or the operating system. It is usually an application task that determines which GPU to use for computation. Data copies between GPUs can also be handled in user-space by copying data via the host memory. However, the operating system is responsible to configure the GPUs, if fast direct data copies between two different device memory spaces via the PCI bus or the SLI interface are required. For instance, the CUDA 4.0 specification requires such a data communication interface, often referred to as *GPU Direct*. In GPU clustering, hence, scheduling must be involved in device-to-device data copies in addition to the executions of GPU contexts and data copies between the host and the device memory.

Networked GPU clusters: The management of multiple GPUs connected over the network is more challenging, as it in-

volves networking. GPU-based HPC applications could scale to use thousands of nodes [38]. We believe that data communications over the network will be a bottleneck to scale the performance of GPU clusters in the number of nodes. Figure 11 illustrates how to send and receive data between the network interface card (NIC) and the GPU in a basic model. Generally, when the NIC receives data, the NIC device driver transfers this data to the host (main) memory via DMA. The GPU device driver next needs to copy this data to the device memory, but the address spaces visible to the NIC and the GPU are different, as these device drivers are usually developed individually. Therefore, the GPU device driver needs to copy the data to another space on the host memory accessible to the GPU. The same data-copy path is used from the opposite side, when the GPU sends data to the NIC. The NVIDIA's GPU Direct technology enabled this data communication stack to skip host-to-host data copy [24], but non-trivial overheads caused by data copies among the NIC, the host memory, and the device memory are still imposed on networked GPU clusters. Coordination of the NIC and the GPU device drivers is needed to reduce such communication overhead.

The same performance issue would appear in distributed systems exploiting GPUs. For example, autonomous vehicles using GPUs and camera sensors need to get data from the camera sensors to the GPUs. In storage systems, data may also come through the network to the GPUs. Coordination of heterogeneous devices and resources is therefore an important problem for future work.

5.3 GPU Virtualization

Virtualization is a useful technique widely adopted in many application domains to isolate clients in the system, and make the system compositional and dependable. Virtualizing GPUs hence provides the same benefits for GPU-accelerated systems. GPU virtualization support has been provided by runtime engines [20], VMMs [11, 12], and I/O managers [7] in the literature. We however believe that there is a problem space for operating systems to support GPU virtualization. In fact, VMs eventually access the GPU via the device driver in the host operating system. Hence, GPU resource management at the device-driver level plays a vital role for GPU virtualization as well. For instance, prioritization and isolation capabilities provided at the device driver level [17] could be very powerful to run GPU VMs.

Our major concern for GPU virtualization appears when different guest operating systems are installed in VMs. The GPU is typically controlled by microcontrollers as depicted in Figure 5. These microcontrollers require firmware to operate correctly, which must be uploaded by the device driver. The firmware image must match the assumption of the device driver. However, GPU device drivers in different guest operating systems may use different firmware images and assumptions. For instance, one guest operating system may provide firmware with context switching support while another may provide that with power management support. In such a case, the device driver in the host operating system needs to switch firmware accordingly among these different guest operating systems, or provide "all-in-one" firmware that provides all necessary functions.

5.4 GPU Device Memory Management

Device memory spaces allocated by user programs are typically pinned. They never become available for different programs unless freed explicitly, resulting in the allocatable memory size limited to the device memory size. This is not an efficient memory management model. The GPU often supports virtual memory to isolate address spaces among GPU channels (contexts). Operating systems should utilize this virtual memory functionality to expand the allocatable device memory spaces, just as they support it for the host memory through memory management units (MMUs). We could establish a hierarchical model of device memory, host memory, and storage to virtualize the device memory as a nearly infinite size of memory, which significantly improve the flexibility and availability of GPU programming.

GPU applications are often very data-intensive. Hence, virtual memory management should cope with frequent data swapping among the device memory, host memory, and storage to maintain performance and interactivity. We believe that prior memory management models [15, 42] are applicable to GPU device memory management to hide the penalty of data swapping. In the presence of multiple GPUs, especially, distributed shared memory (DSM) [22] systems could also be beneficial to transparently increase the available memory space for GPU programming.

5.5 Coordination with Runtime Engines

GPU operations are controlled by GPU command groups issued from user-space programs. For instance, GPU kernel launches and data copies between the host and the device memory are triggered by a specific sets of GPU commands. However, the operating system does not recognize what types of GPU commands are issued from user-space programs. It just controls the ring buffer pointing to the memory location where GPU command groups are stored by the user-space runtime engine so that the GPU can dispatch them, as illustrated in Figure 4. Prior work [2, 16, 17] hence queue and dispatch *all* GPU command groups. However, there are many trivial GPU command groups that do not generate GPU workload. Since the overhead for queuing and dispatching GPU command groups is sometimes significant [17], the operating system should select an appropriate set of GPU command groups to queue and dispatch, including those related to GPU kernel executions and data copies. To do so, the operating system must support an interface for the user-space runtime engine to specify what types of GPU command groups are submitted.

In fact, there are many pieces of useful information to share between the operating system and the user-space runtime engine. To support real-time systems, for instance, it is preferable to know execution times consumed on the GPU before

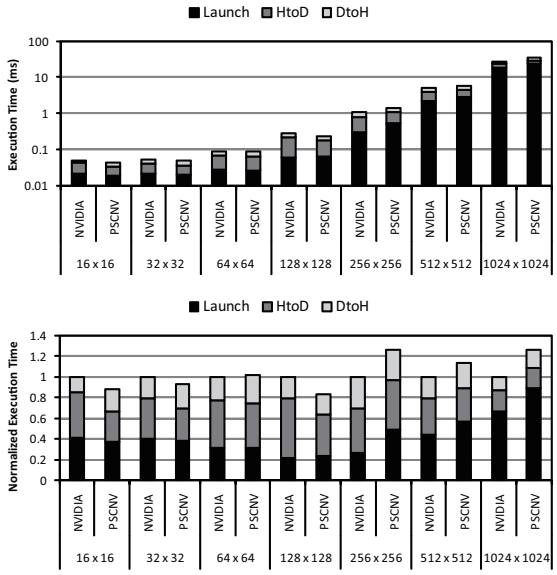

Figure 12. Performance comparison of NVIDIA's proprietary driver and PSCNV.

launching GPU kernels to account and enforce the executions of GPU contexts. It is possible to predict execution times based on a sequence of GPU command groups, but the predicted execution times could be imprecise depending on GPU workload [17]. The operating system should therefore preferably provide an interface to obtain such information from user-space programs.

5.6 Open-Source Implementation

Developing open-source tools is an essential duty to share ideas about systems implementation and facilitate research. Linux, for instance, is a well-known open-source software used in operating systems research. Nouveau [6] and PSCNV [33] are open-source GPU device drivers, available with Linux, for NVIDIA's GPUs. Our previous studies on TimeGraph [16, 17] particularly used Nouveau to implement and evaluate a new real-time GPU command scheduler.

Nouveau is often used in conjunction with an open-source OpenGL runtime engine, Gallium3D [25], for 3-D graphics applications. It should be noted that the performance of this open-source software stack is even competitive with NVIDIA's proprietary software [5], though its usage is limited to graphics applications for now.

PSCNV forked from Nouveau to support general compute programs, which is managed by PathScale Inc., as part of its GPGPU software solution [32]. We are currently involved in its development. Their corresponding runtime engine supports HMPP and CUDA for now, and OpenCL support is also in consideration. In fact, this PathScale's runtime engine can be used in conjunction with NVIDIA's proprietary driver as well. Therefore, fair performance comparisons of these proprietary and open-source device drivers can perform under the same runtime engine. The source code of this runtime engine is not yet open to the public, but it may be available upon request for the research community.

Figure 12 shows a performance comparison of NVIDIA's closed-source proprietary driver and the PSCNV open-source driver in integer matrix multiplication operation of variable sizes, using an NVIDIA GeForce GTX 480 graphics card, where "Launch" represents the execution time of the launched GPU kernel, and "HtoD" and "DtoH" represent data copy durations of host-to-device and device-to-host directions respectively. The GPU clock rate is set at maximum. The GPU kernel of matrix multiplication is compiled using NVIDIA CUDA Toolkit 3.2 [29]. Both drivers run under PathScale's runtime engine. According to our evaluation, the performance difference is very small for a small matrix, while NVIDIA's driver provides better performance for a large matrix. This is mainly attributed to the fact that this open-source device driver has not yet figured out how to activate "performance mode" that boosts the performance of the GPU aside from the clock rate. The performance difference, however, is limited to about 20% at most. Once we learn how to fully configure the GPU, such a performance difference would disappear. We hence believe that open-source software is now reliable enough to conduct operating systems research on GPU resource management.

6 Concluding Remarks

In this paper, we have presented the state of the art in GPU resource management. GPU technology is promising in many application domains due to its high performance and energy efficiency. Most current solutions, however, are focused on how to accelerate a single application task, and multi-tasking problems are not widely discussed. This paper identified core challenges for operating systems research to efficiently use the GPU in multi-tasking environments, and also provided some insights into their solutions. The identified list of challenges needs to expand as our understanding progresses. In particular, real-time systems need to address additional timing issues. We are fortunate to have open-source software that could underlie to explore such new domains of operating systems research. We believe that this paper will encourage research communities to further advance GPU resource management schemes for a grander vision of GPU technology.

Acknowledgment

We thank PathScale for sharing their technology and source code with us in this work.

References

[1] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing*, pages 165–174, 2008.

[2] M. Bautin, A. Dwarakinath, and T. Chiueh. Graphics Engine Resource Management. In *Proceedings of the Annual Multimedia Computing and Networking Conference*, 2008.

[3] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao. Dynamic Load Balancing on Single- and Multi-GPU Systems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2010.

[4] H. Cho, B. Ravindran, and E.D. Jensen. Efficient Real-Time Scheduling Algorithms for Multiprocessor Systems. *IEICE Transactions on Communications*, 85:807–813, 2002.

[5] Linux Open-Source Community. Nouveau Companion 44. `http://nouveau.freedesktop.org/`.

[6] Linux Open-Source Community. Nouveau Open-Source GPU Device Driver. `http://nouveau.freedesktop.org/`.

[7] M. Dowty and J. Sugeman. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[8] G. Elliott and J. Anderson. Real-Time Multiprocessor Systems with GPUs. In *Proceedings of the International Confrence on Real-Time and Network Systems*, 2010.

[9] A. Gharaibeh, S. Al-Kiswany, S. Gopalakrishnan, and M. Ripeanu. A GPU Accelerated Storage System. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing*, pages 167–178, 2010.

[10] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron. Enabling Task Parallelism in the CUDA Scheduler. In *Proceedings of the Workshop on Programming Models for Emerging Architectures*, pages 69–76, 2009.

[11] V. Gupta, A. Gavrilovska, N. Tolia, and V. Talwar. GViM: GPU-accelerated Virtual Machines. In *Proceedings of the ACM Workshop on System-level Virtualization for High Performance Computing*, pages 17–24, 2009.

[12] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[13] Intel. Intel Microarchitecture Codename Sandy Bridge. `http://www.intel.com/`.

[14] S. Kato and Y. Ishikawa. Gang EDF Scheduling of Parallel Task Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 459–468, 2009.

[15] S. Kato, Y. Ishikawa, and R. Rajkumar. CPU Scheduling and Memory Management for Interactive Real-Time Applications. *Real-Time Systems*, 2011.

[16] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar. Resource Sharing in GPU-accelerated Windowing Systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Aplications Symposium*, pages 191–200, 2011.

[17] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[18] S. Kato and N. Yamasaki. Global EDF-based Scheduling with Efficient Priority Promotion. In *Proceedings of the IEEE Embedded and Real-Time Computing Systems and Applications*, pages 197–206, 2008.

[19] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A.D. Nguyen, T. Kaldewey, V.W. Lee, S.A. Brandt, and P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD/PODS Conference*, 2010.

[20] H.A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara. VMM-Independent Graphics Acceleration. In *Proceedings of the ACM/Usenix International Conference on Virtual Execution Environments*, pages 33–43, 2007.

[21] K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling Parallel Real-Time Tasks on Multi-core Processors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 259–268, 2010.

[22] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

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

[24] Mellanox. NVIDIA GPUDirect Technology– Accelerating GPU-based Systems (Whitepaper). `http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf`.

[25] Mesa3D. `http://www.mesa3d.org/`.

[26] P. Michel, J. Chestnutt, S. Kagami, K. Nishiwaki, J. Kuffner, and T. Kanade. GPU-accelerated Real-Time 3D Tracking for Humanoid Locomotion and Stair Climbing. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 463–469, 2007.

[27] E.B. Nightingale, O. Hodson, R. Mcllory, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2009.

[28] NVIDIA. Linux X64 (AMD64/EM64T) Display Driver. `http://www.nvidia.com/`.

[29] NVIDIA. NVIDIA CUDA Toolkit Version. `http://developer.nvidia.com/cuda-toolkit-32-downloads`.

[30] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi (Whitepaper). `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[31] J.K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[32] PathScale. ENZO. `http://www.pathscale.com/`.

[33] PathScale. PSCNV GPU Device Driver. `https://github.com/pathscale/pscnv/`.

[34] Phoronix. NVIDIA Developer Talks Openly About Linux Support. `http://www.phoronix.com/scan.php?page=article&item=nvidia_qa_linux&num=2`.

[35] S. Pronovost, H. Moreton, and T. Kelley. Windows Display Driver Model (WDDM v2) And Beyond. In *Windows Hardware Engineering Conference*, 2006.

[36] A. Saba and R. Mangharam. Anytime Algorithms for GPU Architectures. In *Proceedings of the Analytic Virtual Integration of Cyber-Physical Systems Workshop*, 2010.

[37] D. Schaa and D. Kaeli. Exploring the Multiple-GPU Design Space. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2009.

[38] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T¿ Endo, A. Nukada, N. Maruyama, and S. Matsuoka. An 80-Fold Speedup, 15.0 TFlops, Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, 2010.

[39] S. Thrun. GTC Closing Keynote. `http://livesmooth.istreamplanet.com/nvidia100923/`, 2010.

[40] Top500 Supercomputing Sites. `http://www.top500.org/`.

[41] C. Urmson, J. Anhalt, H. Bae, D. Bagnell, C. Baker, R. Bittner, T. Brown, M. Clark, M. Darms, D. Demitrish, J. Dolan, D. Duggins, D. Ferguson, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. Howard, S. Kolski, M. Likhachev, B. Litkouhi, A. Kelly, M. McNaughton, N. Miller, J. Nickolaou, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, V. Sadekar, B. Salesky, Y-W. Seo, S. Singh, J. Snider, J. Struble, A. Stentz, M. Taylor, W. Whittaker, Z. Wolkowicki, W. Zhang, and J. Ziglar. Autonomous Driving in Urban Environments: Boss and the Urban Challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[42] T. Yang, T. Liu, E.D. Berger, S.F. Kaplan, and J.E-B. Moss. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 73–86, 2008.

Virtual Real-Time Scheduling

Malcolm S. Mollison and James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

We propose a new approach for the runtime scheduling of real-time workloads. This approach, which we call virtual scheduling*, decouples real-time scheduling from the underlying real-time operating system (RTOS) kernel. Such a decoupling provides for the use of scheduling algorithms on an RTOS platform that does not support them natively. This allows new scheduling functionality to be offered to industry practitioners without sacrificing the ideal of the stable, predictable, time-tested, and (mostly) bug-free RTOS kernel.*

1. Introduction

In recent years, significant research effort has been expended in the development of novel scheduling algorithms and synchronization protocols for multiprocessor real-time systems. Much attention has also been given to determining whether these techniques perform well on real hardware platforms, despite potential obstacles such as scheduling overheads. Overall, the results are promising: a number of interesting multiprocessor scheduling and synchronization techniques have proven viable. For example, recent work has shown that the *clustered earliest-deadline-first algorithm* (C-EDF) performs well on large multicore machines [3], and an asymptotically optimal locking protocol that can be used with this algorithm has been given [5].

Unfortunately, real-time operating systems (RTOSs) have not kept pace with such developments. Support for any real-time scheduling algorithm or locking protocol developed within the last twenty years[1] is practically nonexistent in both commercial and open-source RTOSs, at least without modification by the end user (which, in general, is impractical). This leaves industry practitioners without a means to make use of recent advances in the state-of-the-art, and thus unable to deploy the more sophisticated embedded and real-time applications that have otherwise been made possible.

Thus, in terms of real-time scheduling and synchronization, there is a large gap between technology (including theory) that exists in academia, and technology that is "consumable" by industry practitioners. This gap is likely only to widen in future years. Even as some advances are incorporated into consumable technology (if they are), new advances will be made by researchers.

We believe that RTOS vendors are strongly disincentivized from incorporating these kinds of advances into existing kernels, which would explain the gap described above. Any changes will alter the timing characteristics of the kernel, and could introduce new sources of timing indeterminism and new bugs. From the perspective of RTOS consumers whose immediate needs are met by existing kernels—that is, most likely, the vast majority of them—making changes to the kernel is a *bad* thing. Over time, RTOS customers may begin to call for new features. However, it is likely that only the most generic features—that is, those that can be used by customers spread across many different sectors of industry—will be incorporated. This runs counter to increasing interest in real-time scheduling for more niche domains, such as adaptive systems and cyber-physical systems.

RTOS consumers are also disincentivized from modifying existing RTOS kernels to achieve more advanced functionality. The complexity of any non-trivial operating system is such that any kernel modifications are hazardous, unless done by someone with specialized expertise in that particular kernel, and with very extensive testing.

Thus, we believe that it is likely that the gap described above will only continue to widen, unless a way around these problems can be found. Specifically, a technique to avoid these problems would allow real-time scheduling and synchronization to be modified or customized totally independently of the RTOS kernel.

Fundamentally, solving these problems means decoupling real-time scheduling and synchronization from the kernel. This implies the creation of a layer of middle-

Work supported by the NC Space Grant College and Fellowship Program; NC Space Grant Consortium; NSF grants CNS 0834270, CNS 0834132, and CNS 1016954; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

1. The stack resource protocol, commonly available in real-time operating systems (albeit under other names), was developed around 1990 [2].

ware software running "atop" the kernel that provides for real-time workloads to be scheduled according to new scheduling paradigms, while making use of existing kernel services, including the existing scheduler provided with a given RTOS kernel. Such a software layer would provide the effective illusion of the presence of a more capable kernel scheduler than actually exists in the system.

This classic pattern—in which a new layer of indirection allows for both better management of complexity and the addition of new features—is well known in software engineering and computer science. One ubiquitous example of this pattern is virtual memory, which provides the illusion to processes of access to a full memory address space and allows for new memory management features. Another example of this pattern is the use of process virtual machines (such as the Java Virtual Machine, or JVM), which gives the illusion of a standard hardware platform when none exists, and also provides features (such as garbage collection) not available natively. Because we draw inspiration from this pattern, and in light of existing terminology for describing it, we call our approach *virtual real-time scheduling*.[2]

The goal of this paper is to explore the viability of the virtual real-time scheduling approach. First, in Section 2, we explore the support for real-time workloads provided by generic, existing RTOS kernels, and cover necessary background material and related work. In Section 3, we show (by constructing an example) that a virtual scheduler supporting a wide variety of real-time scheduling algorithms and synchronization protocols can itself be supported atop a generic POSIX-compliant RTOS. In Section 4, we explain how to modify the virtual scheduler from the previous section to support memory protection (*i.e.*, space partitioning) between tasks. In Section 5, we describe future work. In Section 6, we conclude.

2. Foundations for Virtual Scheduling

In the following subsections, we describe the underlying RTOS functionality that will serve as a foundation for virtual scheduling. Interspersed with this discussion is necessary background material and information about related work.

2.1. OS Support for Multiprocessor Real-Time Scheduling and Synchronization

In this paper, we adopt a widely-used formalism for representing real-time *task systems* known as the *sporadic task model*. Under this model, each task T has an associated *worst-case execution time* (WCET), $T.e$, and *minimum separation time*, $T.p$. Each successive *job* of T is released at least $T.p$ time units after its predecessor, and a job released at time t must complete by its *deadline*, $t + T.p$. The first job of each task is released at an arbitrary time after the release of the task system. The *utilization*, or long-run processor share required by a task, is given by $T.u = T.e/T.p$.

A task system of n tasks is *schedulable* if, given a scheduling algorithm and m processors, the algorithm can schedule tasks in such a way that all its timing constraints are met. For *hard real-time* task systems, jobs must never miss their deadlines, while for *soft real-time* task systems, some deadline misses are tolerable. Specifically, we require here that the tardiness of jobs of soft real-time tasks be bounded by a (reasonably small) constant.

Scheduling algorithms. Approaches to scheduling real-time tasks on multicore systems can be categorized according to two fundamental (but related) dimensions: first, the choice of how tasks are mapped onto processing cores; and second, the choice of how tasks are prioritized. In each case, there are two common choices. Tasks are typically mapped onto cores either by *partitioning*, in which each task is assigned to a core at system design time and never migrates to another core; or by using a *migrating* approach, in which tasks are assigned to cores at runtime (and can be dynamically re-assigned). Tasks are typically prioritized using either *static priorities*, in which case priorities are chosen at design time and never change; or *dynamic priorities*, in which case tasks' priorities relative to one another change at runtime according to some criteria specified by the scheduling algorithm.

In this paper, migrating, dynamic-priority algorithms are of particular interest. An example is the clustered earliest-deadline-first (C-EDF) algorithm, which was mentioned in the Section 1. Under C-EDF, before system runtime, each core is assigned to one of c clusters, where $1 \leq c \leq m$; and each of the n tasks is assigned to one of the clusters.[3] Let d denote the cluster size, *i.e.*, m/c. At system runtime, within each cluster, the d eligible tasks with highest priority are scheduled on the d available cores. (The word "eligible" is used to exclude tasks that are waiting for some shared resource to become available.)

In contrast, almost all RTOSs support only static-priority scheduling. These include VxWorks [12], which is widely considered to be one of the industry leaders in terms of RTOS market share, and Linux, which can be used to run certain real-time workloads. Some existing RTOSs do support dynamic-priority scheduling, such as ERIKA

2. More pragmatically, our terminology serves to distinguish our approach from existing middleware software which supports real-time tasks *without* providing new functionality beyond that provided by the "real" (kernel) scheduler. This practice is detailed in Section 2 under "Related Work."

3. C-EDF partitions tasks, rather than migrating them, if and only if $c = m$.

Enterprise [7]. However, all existing RTOSs essentially limit their users to the particular scheduling algorithm(s) chosen by the RTOS implementers.

In this paper, our concern is to enable migrating, dynamic-priority scheduling algorithms to be run atop kernels that natively only support static-priority scheduling.

Synchronization protocols. A real-time *synchronization protocol* is used to arbitrate among tasks that share resources that cannot be simultaneously accessed by any number of tasks, such as a critical section of code or a shared hardware device. These protocols typically attempt to prevent *priority inversions*, in which lower-priority tasks are allowed to execute in favor of higher-priority tasks due to resource-sharing dependencies. The possibility of priority inversions in a system must be accounted for in schedulability analysis. To the best of our knowledge, no existing commercial RTOS supports any synchronization protocol more recent than the *stack resource protocol*, which was developed for uniprocessor systems around 1990 [2]. Since then, a number of other multiprocessor locking protocols have been developed. These include the *multiprocessor priority-ceiling protocol* (MPCP) [9], the *distributed priority-ceiling protocol* (DPCP) [10], the *flexible multiprocessor locking protocol* (FMLP) [4], and the *O(m) locking protocol* (OMLP) [5]. These protocols are designed for use with migrating, dynamic-priority scheduling algorithms such as those discussed in the previous subsection. The example virtual scheduler described in this paper can support these protocols.

2.2. OS Support for Real-Time Tasks

In this paper, we use the term "task mechanism" to denote a mechanism by which tasks—that is, separate segments of code, runnable independently of one another—can be supported. The range of task mechanisms available on a given system depends upon the underlying CPU architecture, and upon the abstractions made available for programmers by the operating system. These mechanisms are, at least for our purposes, similar across all modern systems. Not all of these mechanisms are accessible to the kernel scheduler, but all of them *are* accessible to the virtual scheduler. Any virtual scheduler implementation must take careful account of how these abstractions are leveraged, because this will have an impact upon both the performance characteristics of the virtual scheduler and the features it may offer.

Task mechanisms commonly available in modern computer systems are given in the list below. Note that terminology used to describe these mechanisms is far from unified; we attempt to use terminology that is specific enough to avoid and potentially confusing overlap with existing usage.

- A *function*. Tasks implemented using functions share the same stack. This prohibits arbitrarily preempting and switching between tasks. In particular, tasks must run in a nested order in order to preserve the stack. This seriously limits the scheduling algorithms and synchronization protocols that can be supported under this task mechanism.
- A processor *context*.[4] Each context has its own stack, allowing tasks to be preempted and switched among in an arbitrary order. A single context can persist over multiple function calls. In C, contexts can be stored and recalled in userspace using the **setjmp()** and **longjmp()** functions.
- A *kernel-level thread*. A kernel-level thread is defined to be an entity that is known to the operating system and schedulable by the kernel scheduler. The relationship between a kernel-level thread and a context is typically one-to-one, but could be one-to-many, or many-to-many; in other words, contexts can be shared between kernel-level threads. A kernel thread may or may not have memory protection from other kernel threads in the system.
- A *process*. A process is defined to be a group of one or more kernel-level threads that have memory protection from all kernel-level threads that are not in the group. The relationship between a process and a kernel-level thread is typically one-to-one, but could be one-to-many. In many older operating systems, the distinction between kernel-level threads and processes did not exist; all kernel-schedulable entities had independent memory protection. In such cases, the term "process" was generally preferred over other terms.

Making Use of Task Mechanisms. A *user threading library* implements tasks using *user-level threads*, *i.e.*, threads which the kernel is not directly aware of or able to schedule (in contrast to kernel-level threads). User threading libraries typically implement user-level threads by means of contexts. There exist many such libraries; GNU Portable Threads ("Pth") [8] is one example. User-level threading libraries support concurrency (*i.e.*, multiple tasks whose execution is interleaved); however, they do not support parallelism, because they are assumed to execute from within only a single kernel thread.

POSIX Threads ("Pthreads") is a POSIX standard that defines an API for creating and managing threads; a POSIX-compliant threading implementation can make use of either user-level threads or kernel-level threads. Most generic RTOSs support Pthreads, possibly alongside a

4. This term is equivalent, for our purposes, to the term *continuation*, which arises in the study of programming languages and denotes an abstract representation of the processor context that can be stored and recalled by the programmer.

vendor-defined API; these implementations, in turn, rely on kernel-level threads (with or without memory protection), which are then scheduled to run according to a static-priority scheduling algorithm, as discussed previously.

The virtual scheduler implementation proposed in this paper uses both user-level threads *and* kernel-level threads. This approach, known as *hybrid threading* (as opposed to "user threading" or "kernel threading"), has been used before in the parallel processing community. The highly concurrent, non-real-time Erlang programming language uses this approach [6]; however, Erlang does not offer enough fine-grained control of tasks to support our needs. On the other hand, a hybrid scheduling approach known as *scheduler activations* [1]—like Erlang, originally devised for large-scale parallel processing—does support many features that are desirable for virtual scheduling. However, making use of scheduler activations requires specialized kernel support. To the best of our knowledge, no RTOS offers support for scheduler activations.[5]

2.3. Related Work

Virtual scheduling is not the only approach that proposes to enhance practical real-time scheduling capabilities through software that runs in userspace. Other approaches include the following.

- In industry, *operating system abstraction layers* (OSALs) are occasionally used to provide portability for applications between different RTOSs.
- The *Real-Time Ada* programming language provides for a userspace runtime library that is intended to simplify and standardize the development of real-time applications, and particularly, to improve their portability across different operating system platforms.
- The *Real-Time Java* programming language is similar (for our purposes), though it is implemented using a process virtual machine (a specialized JVM) instead of a runtime library.

Real-Time Ada and Real-Time Java do support certain kinds of processor synchronization. However, to the best of our knowledge, they currently only support the same scheduling services offered by the underlying kernel scheduler—*i.e.*, fixed-priority scheduling.[6] Because these two languages rely on userspace implementations, enhancing the facilities they currently offer with the addition of a virtual scheduler (*i.e.*, a scheduler that supports services *not* present in the underlying scheduler) would be a natural fit.

5. Support for scheduler activations is present in certain versions of the Solaris operating system, and (surprisingly) in Windows 7.

6. There has been some discussion of extending Ada to support earliest-deadline-first scheduling [11]. This is comparable to the overall topic of this paper, but has a different goal.

3. Constructing a Virtual Scheduler

In this section, we first determine the functionality that should be supported by our virtual scheduler. We then describe our proposed implementation. In describing this implementation, our intention is, first, two show that a generic, POSIX-compliant RTOS offers sufficient features to virtualize a broad variety of scheduling algorithms; and, second, to provide a starting point for investigating more refined virtual scheduler implementations. Thus, we attempt to avoid trivial optimizations in favor of a more generic and easily-understood model.

3.1. Functional Requirements

Earlier, we classified scheduling algorithms according to whether the priorities of tasks can change, and whether tasks can migrate. In effect, this creates four classes of algorithms, which (for convenience) we label as follows.

(P-SP) partitioned, static-priority algorithms
(P-DP) partitioned, dynamic-priority algorithms
(M-SP) migrating, static-priority algorithms
(M-DP) migrating, dynamic-priority algorithms

Any scheduler (either virtual or kernel-based) that supports arbitrary P-DP algorithms supports arbitrary P-SP algorithms. Any scheduler that supports arbitrary M-DP algorithms supports arbitrary M-SP algorithms. Finally, any scheduler that supports arbitrary M-DP algorithms supports arbitrary P-DP algorithms. Thus, any scheduler that can support arbitrary M-DP algorithms can support all four classes. Furthermore, in each of the classes, it is possible to define algorithms that support either preemptive or non-preemptive scheduling; preemptive scheduling is more general. To the best of our knowledge, these classifications cover all of the multiprocessor scheduling algorithms for sporadic real-time task systems that are currently of interest in the research community.

The virtual scheduler proposed in this paper supports one scheduling algorithm from the preemptive M-DP class. To aid in understanding of the proposed implementation, we did not attempt to support arbitrary algorithms from the preemptive M-DP class, which would (in turn) allow any scheduling algorithm from any of the given classes to be supported. However, we conjecture that extending our specific implementation to support arbitrary M-DP algorithms is possible. Thus, we hope to convince the reader that our proposed virtual scheduler can easily be extended to support all scheduling algorithms of interest.

In contrast to the algorithmic flexibility suggested by the virtual scheduler implementation proposed in this section, our proposed implementation supports only partial memory protection between tasks. Memory protection is a relatively new feature for some widely-used RTOSs; for example, the

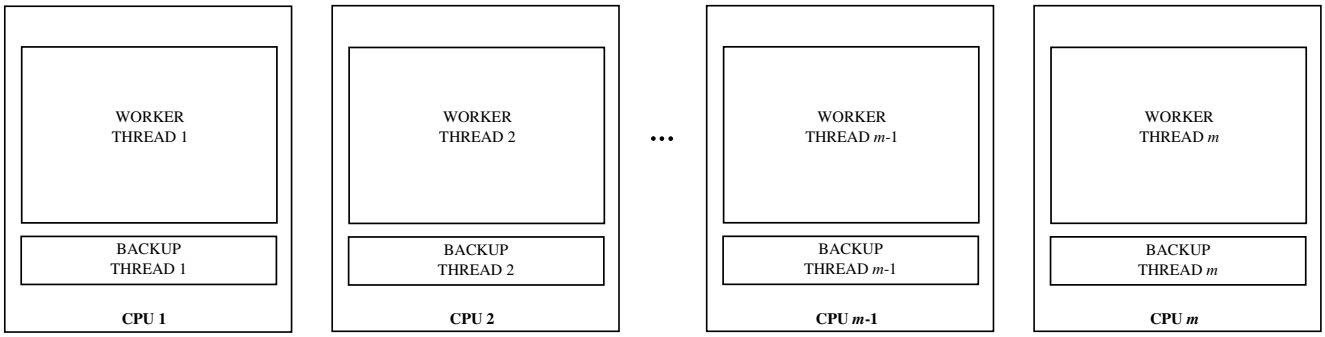

Figure 1: Assignment of kernel-level threads to CPUs.

first edition of VxWorks to support memory protection was the 6.0 series, which was released in 2004. Nonetheless, memory protection can be expected to play a larger role in real-time systems over time, as these systems grow more complex. These issues are explored in depth in Section 4, where we describe how our proposed implementation could be altered to support full memory protection between tasks, at the expense of increased runtime overhead.

3.2. Proposed Implementation

Our approach uses hybrid scheduling—that is, a mixture of user-level threads and kernel-level threads. For ease of explanation, the implementation provided here only supports the *global earliest-deadline-first* (G-EDF) algorithm. G-EDF is equivalent to C-EDF with cluster size m. As mentioned previously, we conjecture that our implementation can be extended to support arbitrary scheduling algorithms in the M-DP class.

Task mechanism. Our implementation supports real-time tasks as contexts. This choice is preferable over more heavy-weight task mechanisms because user-level context switching is generally considered to have at least an order-of-magnitude speed advantage over switching between kernel threads [1]. On the other hand, a more light-weight mechanism—supporting tasks in a manner that maps multiple tasks to one context—would not allow for switching between tasks in arbitrary order, as explained in Section 2. This rules out using mapping the n tasks to fewer than n separate contexts. Nonetheless, each task *is* encapsulated inside a function call; when this function call is executed, a job of the task begins executing, and when a job completes, the function returns. Our implementation mechanisms will ensure that each task receives its own context. These mechanisms are implemented as a runtime library that merely needs to be included with the tasks' code as a header file.

Task initialization. A special *initialization thread* is responsible for setting up all state needed by the virtual scheduler for the real-time workload to begin executing. This includes initializing the real-time tasks. To initialize a task, the initialization thread calls the **setjmp()** function immediately before a conditional call to the task's function, storing the result—the newly-initialized context—in a variable. The conditional call, starting the execution of the task, only executes when **setjmp()** returns after a context switch—hence, *not* during the initialization phase.

Data structure initialization. After initializing the tasks, the initialization thread creates the following key data structures.

- The release queue, a priority queue that holds the contexts for tasks that have no currently eligible job, but will experience a job release at a known time in the future.
- The ready queue, a priority queue that holds the contexts for tasks that are currently eligible, but not running.
- The lowest-priority indicator. During runtime, this variable indicates the CPU of the lowest-priority currently-running task. This will allow for the proper task to be interrupted when a new task that has sufficient priority to run becomes eligible. Ties are broken arbitrarily.

Additional data structures created by the initialization thread are described later in the paper, when they can be understood more easily.

Kernel-level thread initialization. Finally, the initialization thread creates a number of kernel-level threads. This includes n *worker threads* and m *backup threads*. Each of the m cores has exactly one of the worker threads statically assigned to it; these are called *active* worker threads. The non-active worker threads and the backup threads are only relevant in a specialized case, which is explained later. Until then, they can be safely ignored. See Figure 1 for a

diagram showing the described kernel threads.

Overview of runtime scheduling. The active worker threads perform the role of switching between tasks and executing them. Under our G-EDF example, at any time, the m highest-priority eligible tasks are executed by the m active worker threads; if there are less than m eligible tasks, then one or more of the active worker threads is idle.

Task switching. An active worker thread can switch tasks by saving the current task context using **setjmp()** and appending it to a queue (or other shared data structure) where it can be retrieved later, and executing **longjmp()** on a task context that it has obtained from a queue (or other shared data structure) and is to begin executing. Whenever task switching occurs, the lowest-priority indicator variable is updated.

In summary, the basic technique used by the virtual scheduler is to execute the n tasks on m active worker threads, switching execution between tasks as appropriate using **setjmp()** and **longjmp()**. Now, we can move on to an explanation of how job completions, job releases, and task synchronization are handled.

Job completions. When a job completes, the active worker thread executing the task switches to a new task drawn from the ready queue (if any is available). If the completed task is released periodically—*i.e.*, at a known offset from its previous release— it is added to either the release queue or ready queue, depending on whether the release time of the next job has already been reached. Note that adding a task to the release queue implies updating the release timer under certain circumstances, as explained below. On the other hand, if the task is released in response to some stimulus, it is instead saved in a data structure reserved for tasks of this kind.

Responding to asynchronous stimuli. Tasks that are released in response to external stimuli can be accommodated easily. The virtual scheduler implementation merely needs to define a signal handler for signals indicating that a task of this kind needs to be added to the ready queue. Such a signal could originate from a task that receives network packets, for example, or directly from a device driver.

Aside: POSIX timers and signals. The rest of our explanation requires a basic understanding of POSIX timers and signals. A POSIX timer is armed with an associated expiry time and notification thread. (The notification thread must be one created using the POSIX API). When the expiry time is reached, the timer fires, sending a POSIX timer-expired signal to the notification thread.

Maintaining the release queue timer. Throughout task system execution, a POSIX timer is used to mark the time of the next job release in the system. In our virtual scheduler, this timer is set (or reset) whenever a task is added to the release queue, if no task with an earlier release time exists in the queue. The low-priority indicator variable is used as a basis for selecting the proper notification thread (*i.e.*, one of the worker threads).

Responding to a timer expiry signal. When a worker thread receives the timer expiry signal, its execution jumps to an associated signal handler (provided in our virtual scheduler's runtime library). In this signal handler, the worker thread moves the next-to-be-released task to the ready queue, and updates the release queue timer. If the moved task is of sufficiently high priority, some worker thread will begin executing it. This could be the same worker thread that just released it; otherwise, a signal is sent to the appropriate worker thread by the thread that released it. The appropriate thread is determined by the current status of the lowest-priority indicator variable.

Synchronization protocols. The virtual scheduler runtime library must provide an API that can be used by a task to request access to a shared resource. If the task is to be allowed to acquire the resource without waiting, the API returns immediately. Otherwise, the task's context is saved and stored in a data structure used specifically to hold tasks blocked in such a case. When a task is finished with the resource, it must perform another API call to release it. At this point, any task held in the data structure referenced above that is now eligible (*i.e.*, can obtain the resource) is moved onto the ready queue. If the newly-unblocked task has sufficient priority, a signal is then sent to the appropriate worker thread (as indicated via the lowest-priority indicator variable) so that the task will be removed from the ready queue and executed.

Blocking in the kernel. We have laid most of the essential groundwork for our virtual scheduler. Only one key issue remains: What if a worker thread blocks while running in kernelspace? Such a scenario could arise due to a system call or a page fault. We solve this problem using the backup threads that were mentioned earlier. Recall that there are exactly m backup threads, each affixed to one of the m processors. The backup threads are assigned a lower priority than the active worker threads; thus, a backup thread only runs on a core if the worker thread on that core blocks in the kernel. In such a case, the backup thread causes one of the non-active workers to become the new active worker thread on that core. (The non-active workers are kept at a priority below the backup threads, so that none of them will execute until it has been selected to become an active worker, has been migrated to the proper core, and has had its priority adjusted.) Just before adjusting the

priority of the new active worker and giving up its control of the processor, the backup thread boosts the priority of the blocked worker thread to be above that of all worker threads, and sends it a signal to indicate that it has blocked in the kernel. As soon as the blocked worker finishes blocking in the kernel, it will receive the signal indicating that it has blocked. This worker adds its task to the release queue, sends a signal to the worker thread indicated by the lowest-priority indicator variable (to potentially trigger a context switch to the task that blocked in the kernel), and becomes a non-active worker. Note that this process happens immediately when the task finishes blocking in the kernel, since the priority of the blocking thread has been boosted above that of the active worker threads.

4. Supporting Memory Protection

One hazard of any multitasking computer system is the possibility of one task corrupting the memory of another task. As real-time systems have become more complex, this problem has become a growing concern. In this section, we discuss the degree of memory protection provided by the virtual scheduler implementation described in Section 3, and also explain how full memory protection between tasks can be provided, at the expense of additional runtime overhead.

Note that, in this paper, we are *not* concerned with the difficult problem of preventing code created with malicious intent from damaging the system, which is typically only a concern in a narrow segment of highly-critical real-time systems (such as military weapon systems and nuclear power plants). Rather, we are interested in preventing widespread failures caused by programmer error and in the absence of attacks by adversaries.

4.1. Existing Properties

The implementation given in Section 3 can be made relatively robust to these kinds of errors with just a little bit of effort. Memory protection for kernel-level threads is now a common feature, even in RTOS kernels. If each kernel-level worker thread has memory protection from all other threads (*i.e.*, is treated as a process), the only possibility for one task to corrupt another is by corrupting the data structures shared between worker threads (such as the release queue and the ready queue).

Furthermore, if this kind of corruption does not occur, the implementation is already robust to process failures (for example, segfaults). Under such a failure, the worker thread and the real-time task being executed inside it would be lost; the backup thread running on that core would then run and activate a non-active worker thread in its place, in accordance with the specification given in Section 3.

4.2. Achieving Complete Memory Protection

Nonetheless, the implementation given in Section 3 leaves critical shared scheduling data structures vulnerable to corruption. If one of these data structures were to become corrupted, it could cause the failure of all tasks in the system. The only way around this problem is to prevent any task application code from being able to write to these shared scheduling data structures. (We assume that the virtual scheduler runtime library code is trusted not to cause corruption; ultimately, some code in the system must be trusted to update scheduling data structures.) Below, we outline how the implementation from Section 3 can be modified to achieve this property. (There are many specific implementation tradeoffs that may be worth investigating in future work. Here, our concern is simply to show that our virtual scheduler mechanism can be modified to enable memory protection.)

Modifications for full memory protection. Rather than using contexts as the task mechanism, kernel-level threads are used. When the system is initialized, n such threads are created, all with a priority lower than that of the backup threads. In this scheme, the backup threads play a more important role than before. The backup threads share access to scheduling data structures. They select tasks to run on each core, and cause them to do so by forcing them to migrate to the appropriate core and setting their priority to be higher than that of the backup threads. Task completion is carried out when a task sets its priority back to the lower setting, returning control to the backup thread on that core. Timer-driven signals for releasing tasks are set up in a way that causes them to be delivered to the task running on the relevant core. When a task receives such a signal, it returns control to the backup thread on that core by lowering its priority.

5. Future Work

Initially, we would like to make a more elaborate examination of the likelihood of the virtual scheduling approach to yield useful results. Such an examination would most likely focus on measuring the performance of several virtual scheduler implementations and comparing their performance to that achieved by native RTOS schedulers.

Given the success of such an examination, we would ultimately like to conduct a more comprehensive study of virtual schedulers. Such a study would first catalog relevant classes of real-time systems, as distinguished by characteristics like hardware platform (including number of cores and caching hierarchy); workload properties (such

as the number of tasks and the maximum per-task utilization); performance requirements (such as types of timing constraints or requirements for adaptivity); and robustness requirements (such as fault isolation and security protection). Then, the study would determine, as best as possible, the most effective virtual scheduling implementation for each class, in terms of characteristics like runtime overhead. Such a study would allow industry practitioners to choose intelligently between native RTOS scheduling and virtual scheduling for applications of interest, and would (hopefully) open up avenues for entirely new classes of real-time systems to be deployed.

In this study, we do not wish to rule out implementation approaches necessitating basic RTOS kernel modifications, in return for a significant boost in the capabilities of the virtual scheduler. Such approaches would be of interest if the needed modifications were generic and straightforward enough that one could reasonably hope for commercial RTOS vendors to eventually adopt them.

If the virtual scheduling approach proves viable, we are eager to apply it to novel topics currently being studied by the real-time systems research community, such as adaptive systems, mixed-criticality systems, hierarchical scheduling systems, and secure real-time systems. We believe virtual scheduling may enable significant practical advances in these areas.

6. Conclusion

In this paper, we proposed a new approach for the runtime scheduling of real-time workloads, *virtual scheduling*, which decouples real-time scheduling from the underlying real-time operating system (RTOS) kernel. This decoupling provides for the use of scheduling algorithms on an RTOS platform that does not support them natively. If it proves to be viable, this approach will allow new scheduling functionality to be offered to industry practitioners without sacrificing the ideal of the stable, predictable, time-tested, and (mostly) bug-free RTOS kernel. However, the best way to go about exploring this concept is far from obvious. We are eager to receive feedback from real-time kernel developers and academic researchers on how to make the most of this approach, considering the intricacies of real-time operating systems and the complexity of real-time scheduling algorithms.

References

[1] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for user-level management of parallelism. In *ACM Transactions on Computer Systems, vol. 10, no. 1*, pages 53–79, 1992.

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

[3] A. Bastoni, B. Brandenburg, and J. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, pages 14–24, 2010.

[4] A. Block, B. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–57, 2007.

[5] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, pages 49–60, 2010.

[6] Erlang Web site. http://www.erlang.org/.

[7] Evidence Web site. http://www.evidence.eu.com.

[8] GNU Portable Threads Web site. http://www.gnu.org/software/pth.

[9] R. Rajkumar. Real-time synchronization protocols for shared-memory multiprocessors. In *Proceedings of the 10th International Conferece on Distributed Computing Systems*, pages 116–123, 1990.

[10] R. Rajkumar, L. Sha, and J. Lehockzy. Real-time synchronization protocols for multiprocessors. In *Proceedings of the 9th Real-Time Systems Symposium*, pages 259–269, 1988.

[11] A. Wellings and A. Burns. Generalizing the edf scheduling support in ada 2005. In *Ada Letters, vol. 30*, pages 116–124, 2010.

[12] Wind River Web site. http://www.windriver.com.

Temporal isolation in an HSF-enabled real-time kernel in the presence of shared resources

Martijn M. H. P. van den Heuvel, Reinder J. Bril and Johan J. Lukkien
Department of Mathematics and Computer Science
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5600 AZ Eindhoven, The Netherlands

***Abstract*—Hierarchical scheduling frameworks (HSFs) have been extensively investigated as a paradigm for facilitating temporal isolation between components that need to be integrated on a single shared processor. To support resource sharing within two-level, fixed priority scheduled HSFs, two synchronization protocols based on the stack resource policy (SRP) have recently been presented, i.e. HSRP [1] and SIRAP [2]. In the presence of shared resources, however, temporal isolation may break when one of the accessing components executes longer than specified during global resource access. As a solution we propose a SRP-based synchronization protocol for HSFs, named Basic Hierarchical Synchronization protocol with Temporal Protection (B-HSTP). The schedulability of those components that are independent of the unavailable resource is unaffected.**

This paper describes an implementation to provide HSFs, accompanied by SRP-based synchronization protocols, with means for temporal isolation. We base our implementations on the commercially available real-time operating system μC/OS-II, extended with proprietary support for two-level fixed priority preemptive scheduling. We specifically show the implementation of B-HSTP and we investigate the system overhead induced by its synchronization primitives in combination with HSRP and SIRAP. By supporting both protocols in our HSF, their primitives can be selected based on the protocol's relative strengths[1].

I. Introduction

The increasing complexity of real-time systems demands a decoupling of (*i*) development and analysis of individual components and (*ii*) integration of components on a shared platform, including analysis at the system level. Hierarchical scheduling frameworks (HSFs) have been extensively investigated as a paradigm for facilitating this decoupling [3]. A component that is validated to meet its timing constraints when executing in isolation will continue meeting its timing constraints after integration (or admission) on a shared platform. The HSF therefore provides a promising solution for current industrial standards, e.g. the AUtomotive Open System ARchitecture (AUTOSAR) [4] which specifies that an underlying OSEK-based operating system should prevent timing faults in any component to propagate to different components on the same processor. The HSF provides temporal isolation between components by allocating a *budget* to each component, which gets mediated access to the processor by means of a *server*.

An HSF without further resource sharing is unrealistic, however, since components may for example use operating system services, memory mapped devices and shared communication devices which require mutually exclusive access. Extending an HSF with such support makes it possible to share logical resources between arbitrary tasks, which are located in arbitrary components, in a mutually exclusive manner. A resource that is used in more than one component is denoted as a *global shared resource*. A resource that is only shared by tasks within a single component is a *local shared resource*. If a task that accesses a global shared resource is suspended during its execution due to the exhaustion of its budget, excessive blocking periods can occur which may hamper the correct timeliness of other components [5].

Looking at existing industrial real-time systems, fixed-priority preemptive scheduling (FPPS) is the de-facto standard of task scheduling, hence we focus on an HSF with support for FPPS within a component. Having such support will simplify migration to and integration of existing legacy applications into the HSF. Our current research efforts are directed towards the conception and realization of a two-level HSF that is based on (*i*) FPPS for both *global scheduling* of servers allocated to components and *local scheduling* of tasks within a component and (*ii*) the Stack Resource Policy (SRP) [6] for both local and global resource sharing.

To accommodate resource sharing between components, two synchronization protocols [1], [2] have been proposed based on SRP for two-level FPPS-based HSFs. Each of these protocols describes a run-time mechanism to handle the depletion of a component's budget during global resource access. In short, two general approaches are proposed: (*i*) *self-blocking* when the remaining budget is insufficient to complete a critical section [2] or (*ii*) *overrun* the budget until the critical section ends [1]. However, when a task exceeds its specified worst-case critical-section length, i.e. it *misbehaves* during global resource access, temporal isolation between components is no longer guaranteed. The protocols in [1], [2] therefore break the temporal encapsulation and fault-containment properties of an HSF without the presence of complementary protection.

A. Problem description

Most off-the-shelf real-time operating systems, including μC/OS-II [7], do not provide an implementation for SRP nor hierarchical scheduling. We have extended μC/OS-II with support for idling periodic servers (IPS) [8] and two-level FPPS. However, existing implementations of the synchroniza-

[1]The work in this paper is supported by the Dutch HTAS-VERIFIED project, see http://www.htas.nl/index.php?pid=154. Our μC/OS-II extensions are available at http://www.win.tue.nl/~mholende/relteq/.

tion protocols in our framework [9], [10], as well as in the framework presented in [11], do not provide any temporal isolation during global resource access.

A solution to limit the propagation of temporal faults to those components that share global resources is considered in [12]. Each task is assigned a dedicated budget per global resource access and this budget is synchronous with the period of that task. However, in [12] they allow only a single task per component.

We consider the problem to limit the propagation of temporal faults in HSFs, where multiple concurrent tasks are allocated a shared budget, to those components that share a global resource. Moreover, we present an efficient implementation and evaluation of our protocol in μC/OS-II. The choice of operating system is driven by its former OSEK compatibility[2].

B. Contributions

The contributions of this paper are fourfold.

- To achieve temporal isolation between components, even when resource-sharing components misbehave, we propose a modified SRP-based synchronization protocol, named *Basic Hierarchical Synchronization protocol with Temporal Protection* (B-HSTP).
- We show its implementation in a real-time operating system, extended with support for two-level fixed-priority scheduling, and we efficiently achieve fault-containment by disabling preemptions of other tasks within the same component during global resource access.
- We show that B-HSTP complements existing synchronization protocols [1], [2] for HSFs.
- We evaluate the run-time overhead of our B-HSTP implementation in μC/OS-II on the OpenRISC platform [13]. These overheads become relevant during deployment of a resource-sharing HSF.

C. Organization

The remainder of this paper is organized as follows. Section II describes related works. Section III presents our system model. Section IV presents our resource-sharing protocol, B-HSTP, which guarantees temporal isolation to independent components. Section V presents our existing extensions for μC/OS-II comprising two-level FPPS-based scheduling and SRP-based resource arbitration. Section VI presents B-HSTP's implementation using our existing framework. Section VII investigates the system overhead corresponding to our implementation. Section VIII discusses practical extensions to B-HSTP. Finally, Section IX concludes this paper.

II. Related work

Our basic idea is to use two-level SRP to arbitrate access to global resources, similar as [1], [2]. In literature several alternatives are presented to accommodate task communication in reservation-based systems. De Niz et al. [12] support resource sharing between reservations based on the immediate priority ceiling protocol (IPCP) [14] in their FPPS-based Linux/RK resource kernel and use a run-time mechanism based on resource containers [15] for temporal protection against misbehaving tasks. Steinberg et al. [16] showed that these resource containers are expensive and efficiently implemented a capacity-reserve donation protocol to solve the problem of priority inversion for tasks scheduled in a fixed-priority reservation-based system. A similar approach is described in [17] for EDF-based systems and termed bandwidth-inheritance (BWI). BWI regulates resource access between tasks that each have their dedicated budget. It works similar to the priority-inheritance protocol [14], i.e. when a task blocks on a resource it donates its remaining budget to the task that causes the blocking. However, all these approaches assume a one-to-one mapping from tasks to budgets, and inherently only have a single scheduling level.

In HSFs a group of concurrent tasks, forming a component, are allocated a budget [18]. A prerequisite to enable independent analysis of interacting components and their integration is the knowledge of which resources a task will access [2], [19]. When a task accesses a global shared resource, one needs to consider the priority inversion between components as well as local priority inversion between tasks within the component. To *prevent budget depletion* during global resource access in FPPS-based HSFs, two synchronization protocols have been proposed based on SRP [6]: HSRP [1] and SIRAP [2]. Although HSRP [1] originally does not integrate into HSFs due to the lacking support for independent analysis of components, Behnam et al. [19] lifted this limitation. However, these two protocols, including their implementations in [9], [10], [11], assume that components respect their timing contract with respect to global resource sharing. In this paper we present an implementation of HSRP and SIRAP protocols that limits the unpredictable interferences caused by contract violations to the components that share the global resource.

III. Real-time scheduling model

We consider a two-level FPPS-scheduled HSF, following the periodic resource model [3], to guarantee processor allocations to components. We use SRP-based synchronization to arbitrate mutually exclusive access to global shared resources.

A. Component model

A system contains a set $\mathcal{R}$ of M global logical resources $R_1, R_2, \ldots, R_M$, a set $\mathcal{C}$ of N components $C_1, C_2, \ldots, C_N$, a set $\mathcal{B}$ of N budgets for which we assume a periodic resource model [3], and a single shared processor. Each component C_s has a dedicated budget which specifies its periodically guaranteed fraction of the processor. The remainder of this paper leaves budgets implicit, i.e. the timing characteristics of budgets are taken care of in the description of components. A server implements a policy to distribute the available budget to the component's workload.

The timing characteristics of a component C_s are specified by means of a triple $< P_s, Q_s, \mathcal{X}_s >$, where $P_s \in \mathbb{R}^+$ denotes its period, $Q_s \in \mathbb{R}^+$ its budget, and $\mathcal{X}_s$ the set of

[2]Unfortunately, the supplier of μC/OS-II, Micrium, has discontinued the support for the OSEK-compatibility layer.

maximum access times to global resources. The maximum value in $\mathcal{X}_s$ is denoted by X_s, where $0 < Q_s + X_s \leq P_s$. The set $\mathcal{R}_s$ denotes the subset of M_s global resources accessed by component C_s. The maximum time that a component C_s executes while accessing resource $R_l \in \mathcal{R}_s$ is denoted by X_{sl}, where $X_{sl} \in \mathbb{R}^+ \cup \{0\}$ and $X_{sl} > 0 \Leftrightarrow R_l \in \mathcal{R}_s$.

B. Task model

Each component C_s contains a set $\mathcal{T}_s$ of n_s sporadic tasks τ_{s1}, τ_{s2}, ..., τ_{sn_s}. Timing characteristics of a task $\tau_{si} \in \mathcal{T}_s$ are specified by means of a triple $< T_{si}, E_{si}, D_{si} >$, where $T_{si} \in \mathbb{R}^+$ denotes its minimum inter-arrival time, $E_{si} \in \mathbb{R}^+$ its worst-case computation time, $D_{si} \in \mathbb{R}^+$ its (relative) deadline, where $0 < E_{si} \leq D_{si} \leq T_{si}$. The worst-case execution time of task τ_{si} within a critical section accessing R_l is denoted c_{sil}, where $c_{sil} \in \mathbb{R}^+ \cup \{0\}$, $E_{si} \geq c_{sil}$ and $c_{sil} > 0 \Leftrightarrow R_l \in \mathcal{R}_s$. All (critical-section) execution times are accounted in terms of processor cycles and allocated to the calling task's budget. For notational convenience we assume that tasks (and components) are given in priority order, i.e. τ_{s1} has the highest priority and τ_{sn_s} has the lowest priority.

C. Synchronization protocol

Traditional synchronization protocols such as PCP [14] and SRP [6] can be used for *local* resource sharing in HSFs [20]. This paper focuses on arbitrating *global* shared resources using SRP. To be able to use SRP in an HSF for synchronizing global resources, its associated ceiling terms need to be extended and excessive blocking must be prevented.

1) Resource ceilings: With every global resource R_l two types of resource ceilings are associated; a *global* resource ceiling RC_l for global scheduling and a *local* resource ceiling rc_{sl} for local scheduling. These ceilings are statically calculated values, which are defined as the highest priority of any component or task that shares the resource. According to SRP, these ceilings are defined as:

$$RC_l = \min(N, \min\{s \mid R_l \in \mathcal{R}_s\}), \quad (1)$$

$$rc_{sl} = \min(n_s, \min\{i \mid c_{sil} > 0\}). \quad (2)$$

We use the outermost $\min$ in (1) and (2) to define RC_l and rc_{sl} in those situations where no component or task uses R_l.

2) System and component ceilings: The system and component ceilings are dynamic parameters that change during execution. The system ceiling is equal to the highest global resource ceiling of a currently locked resource in the system. Similarly, the component ceiling is equal to the highest local resource ceiling of a currently locked resource within a component. Under SRP a task can only preempt the currently executing task if its priority is higher than its component ceiling. A similar condition for preemption holds for components.

3) Prevent excessive blocking: HSRP [1] uses an overrun mechanism [19] when a budget depletes during a critical section. If a task $\tau_{si} \in \mathcal{T}_s$ has locked a global resource when its component's budget Q_s depletes, then component C_s can continue its execution until task τ_{si} releases the resource. These budget overruns cannot take place across replenishment boundaries, i.e. the analysis guarantees $Q_s + X_s$ processor time before the relative deadline P_s [1], [19].

SIRAP [2] uses a self-blocking approach to prevent budget depletion inside a critical section. If a task τ_{si} wants to enter a critical section, it enters the critical section at the earliest time instant so that it can complete the critical section before the component's budget depletes. If the remaining budget is insufficient to lock and release a resource R_l before depletion, then (*i*) the task blocks itself until budget replenishment and (*ii*) the component ceiling is raised to prevent tasks $\tau_{sj} \in \mathcal{T}_s$ with a priority lower than the local ceiling rc_{sl} to execute until the requested critical section has been finished.

The relative strengths of HSRP and SIRAP have been analytically investigated in [21] and heavily depend on the chosen system parameters. To enable the selection of a particular protocol based on its strengths, we presented an implementation supporting both protocols with transparent interfaces for the programmer [9]. In this paper we focus on mechanisms to extend these protocols with temporal protection and merely investigate their relative complexity with respect to our temporal-protection mechanisms.

IV. SRP WITH TEMPORAL PROTECTION

Temporal faults may cause improper system alterations, e.g. due to unexpectedly long blocking or an inconsistent state of a resource. Without any protection a self-blocking approach [2] may miss its purpose under erroneous circumstances, i.e. when a task overruns its budget to complete its critical section. Even an overrun approach [1], [19] needs to guarantee a maximum duration of the overrun situation. Without such a guarantee, these situations can hamper temporal isolation and resource *availability* to other components due to unpredictable blocking effects. A straightforward implementation of the overrun mechanism, e.g. as implemented in the ERIKA kernel [22], where a task is allowed to indefinitely overrun its budget as long as it locks a resource, is therefore not *reliable*.

A. Resource monitoring and enforcement

A common approach to ensure temporal isolation and prevent propagation of temporal faults within the system is to group tasks that share resources into a single component [20]. However, this might be too restrictive and lead to large, incoherent component designs, which violates the principle of HSFs to independently develop components. Since a component defines a coherent piece of functionality, a task that accesses a global shared resource is critical with respect to all other tasks in the same component.

To guarantee temporal isolation between components, the system must *monitor* and *enforce* the length of a global critical section to prevent a malicious task to execute longer in a critical section than assumed during system analysis [12]. Otherwise such a misbehaving task may increase blocking to components with a higher priority, so that even independent components may suffer, as shown in Figure 1.

To prevent this effect we introduce a *resource-access budget* q_s in addition to a component's budget Q_s, where budget q_s

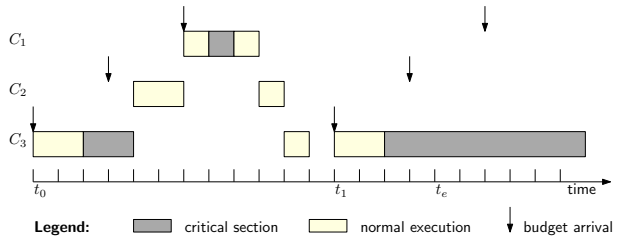

Fig. 1. Temporal isolation is unassured when a component, C_3, exceeds its specified critical-section length, i.e. at time instant t_e. The system ceiling blocks all other components.

is used to enforce critical-section lengths. When a resource R_l gets locked, q_s replenishes to its full capacity, i.e. $q_s \leftarrow X_{sl}$. To monitor the available budget at any moment in time, we assume the availability of a function $Q_s^{\texttt{rem}}(t)$ that returns the remaining budget of Q_s. Similarly, $q_s^{\texttt{rem}}(t)$ returns the remainder of q_s at time t. If a component C_s executes in a critical section, then it consumes budget from Q_s and q_s in parallel, i.e. depletion of either Q_s or q_s forbids component C_s to continue its execution. We maintain the following invariant to prevent budget depletion during resource access:

Invariant 1: $Q_s^{\texttt{rem}}(t) \geq q_s^{\texttt{rem}}(t)$.

The way of maintaining this invariant depends on the chosen policy to prevent budget depletion during global resource access, e.g. by means of SIRAP [2] or HSRP [1].

1) Fault containment of critical sections: Existing SRP-based synchronization protocols in [2], [19] make it possible to choose the local resource ceilings, rc_{sl}, according to SRP [6]. In [23] techniques are presented to trade-off preemptiveness against resource holding times. Given their common definition for local resource ceilings, a resource holding time, X_{sl}, may also include the interference of tasks with a priority higher than the resource ceiling. Task τ_{si} can therefore lock resource R_l longer than specified, because an interfering task τ_{sj} (where $\pi_{sj} > rc_{sl}$) exceeds its computation time, E_{sj}.

To prevent this effect we choose to disable preemptions for other tasks within the same component during critical sections, i.e. similar as HSRP [1]. As a result X_{sl} only comprises task execution times within a critical section, i.e.

$$X_{sl} = \max_{1 \leq i \leq n_s} c_{sil}. \tag{3}$$

Since X_{sl} is enforced by budget q_s, temporal faults are contained within a subset of resource-sharing components.

2) Maintaining SRP ceilings: To enforce that a task τ_{si} resides no longer in a critical section than specified by X_{sl}, a resource $R_l \in \mathcal{R}$ maintains a state *locked* or *free*. We introduce an extra state *busy* to signify that R_l is locked by a misbehaving task. When a task τ_{si} tries to exceed its maximum critical-section length X_{sl}, we update SRP's *system ceiling* by mimicking a resource unlock and mark the resource *busy* until it is released. Since the system ceiling decreases after τ_{si} has executed for a duration of X_{sl} in a critical section to resource R_l, we can no longer guarantee absence of deadlocks. Nested critical sections to global resources are therefore unsupported. One may alternatively aggregate global resource accesses into a simultaneous lock and unlock of a single artificial resource [24]. Many protocols, or their implementations, lack deadlock avoidance [11], [12], [16], [17].

Although it seems attractive from a schedulability point of view to release the *component ceiling* when the critical-section length is exceeded, i.e. similar to the system ceiling, this would break SRP compliance, because a task may block on a busy resource instead of being prevented from starting its execution. Our approach therefore preserves the SRP property to share a single, consistent execution stack per component [6]. At the global level tasks can be blocked by a depleted budget, so that components cannot share an execution stack anyway.

B. An overview of B-HSTP properties

This section presented a basic protocol to establish hierarchical synchronization with temporal protection (B-HSTP). Every lock operation to resource R_l replenishes a corresponding resource-access budget q_s with an amount X_{sl}. After this resource-access budget has been depleted, the component blocks until its normal budget Q_s replenishes. We can derive the following convenient properties from our protocol:

1) as long as a component behaves according to its timing contract, we strictly follow SRP;
2) because local preemptions are disabled during global resource access and nested critical sections are prohibited, each component can only access a single global resource at a time;
3) similarly, each component can at most keep a single resource in the busy state at a time;
4) each access to resource R_l by task τ_{si} may take at most X_{sl} budget from budget Q_s, where $c_{sil} \leq X_{sl}$.
5) after depleting resource-access budget q_s, a task may continue in its component normal budget Q_s with a decreased system ceiling. This guarantees that independent components are no longer blocked by the system ceiling;
6) when a component blocks on a busy resource, it discards all remaining budget until its next replenishment of Q_s. This avoids budget suspension, which can lead to scheduling anomalies [25].

As a consequence of property 2, we can use a simple non-preemptive locking mechanism at the local level rather than using SRP. We therefore only need to implement SRP at the global level and we can use a simplified infrastructure at the local level compared to the implementations in [9], [10], [11].

V. μC/OS-II AND ITS EXTENSIONS RECAPITULATED

The μC/OS-II operating system is maintained and supported by Micrium [7], and is applied in many application domains, e.g. avionics, automotive, medical and consumer electronics. Micrium provides the full μC/OS-II source code with accompanying documentation [26]. The μC/OS-II kernel provides preemptive multitasking for up to 256 tasks, and the kernel size is configurable at compile time, e.g. services like mailboxes and semaphores can be disabled.

Most real-time operating systems, including μC/OS-II, do not include a reservation-based scheduler, nor provide means for hierarchical scheduling. In the remainder of this section we outline our realization of such extensions for μC/OS-II, which are required basic blocks to enable the integration of global synchronization with temporal protection.

A. Timed Event Management

Intrinsic to our reservation-based component scheduler is timed-event management. This comprises timers to accommodate (*i*) periodic timers at the global level for budget replenishment of periodic servers and at the component level to enforce minimal inter-arrivals of sporadic task activations and (*ii*) virtual timers to track a component's budget. The corresponding timer handlers are executed in the context of the timer interrupt service routine (ISR).

We have implemented a dedicated module to manage *relative timed event queues* (RELTEQs) [27]. The basic idea is to store events relative to each other, by expressing the expiration time of an event relative to the arrival time of the previous event. The arrival time of the head event is relative to the current time, see Figure 2

A *system queue* tracks all server events. Each server has its own *local queue* to track its tasks' events, e.g. task arrivals. When a server is suspended its local queues are deactivated to prevent that expiring events interfere with other servers. When a server resumes, its local queues are synchronized with global time. A mechanism to synchronize server queues with global time is implemented by means of a *stopwatch queue*, which keeps track of the time passed since the last server switch.

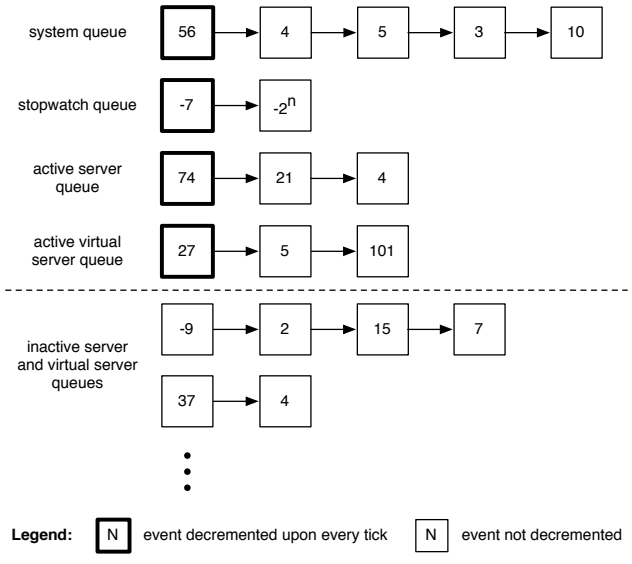

Fig. 2. RELTEQ-based timer management for two-level HSFs.

A dedicated server queue provides support for *virtual timers* to trigger timed events *relative to the consumed budget*. Since an inactive server does not consume any of its budget, a virtual timer queue is not synchronized when a server is resumed. We consider only budget depletion as a virtual event, so that a component can inspect its virtual-timer in constant time.

B. Server Scheduling

A *server* is assigned to each component to distribute its allocated budget to the component's tasks. A global scheduler is used to determine which server should be allocated the processor at any given time. A local scheduler determines which of the chosen server's tasks should actually execute. Although B-HSTP is also applicable to other server models, we assume that a component is implemented by means of an *idling periodic server* (IPS) [8]. Extending μC/OS-II with basic HSF support requires a realization of the following concepts:

1) Global Scheduling: At the system level a RELTEQ queue is introduced to keep track of server periods. We use a bit-mask to represent whether a server has capacity left. When the scheduler is called, it traverses the RELTEQ and activates the *ready* server with the earliest deadline in the queue. Subsequently, the μC/OS-II fixed-priority scheduler determines the highest priority ready task within the server.

2) Periodic Servers: Since μC/OS-II tasks are bundled in groups of sixteen to accommodate efficient fixed-priority scheduling, a server can naturally be represented by such a group. The implementation of periodic servers is very similar to implementing periodic tasks using our RELTEQ extensions [27]. An idling server contains an idling task at the lowest, local priority, which is always ready to execute.

3) Greedy Idle Server: In our HSF, we reserve the lowest priority level for a idle server, which contains μC/OS-II's idle task at the lowest local priority. Only if no other server is eligible to execute, then the idle server is switched in.

C. Global SRP implementation

The key idea of SRP is that when a component needs a resource that is not available, it is blocked at the time it attempts to preempt, rather than later. Nice properties of SRP are its simple locking and unlocking operations. In turn, during run-time we need to keep track of the system ceiling and the scheduler needs to compare the highest ready component priority with the system ceiling. Hence, a preemption test is performed during run time by the scheduler: A component cannot preempt until its priority is the highest among those of all ready components *and* its priority is higher than the *system ceiling*. In the original formulation of SRP [6], it introduces the notion of preemption-levels. This paper considers FPPS, which makes it possible to unify preemption-levels with priorities.

The system ceiling is a dynamic parameter that changes during execution. Under SRP, a component can only preempt the currently executing component if its priority is higher than the system ceiling. When no resources are locked the system ceiling is zero, meaning that it does not block any tasks from preempting. When a resource is locked, the system ceiling is adjusted dynamically using the resource ceiling, so that the system ceiling represents the highest resource ceiling of a currently locked resource in the system. A run-time

mechanism for tracking the system ceiling can be implemented by means of a stack data structure.

1) SRP data and interface description: Each resource accessed using an SRP-based mutex is represented by a `Resource` structure. This structure is defined as follows:

```
typedef struct resource{
    INT8U  ceiling;
    INT8U  lockingTask;
    void*  previous;
} Resource;
```

The `Resource` structure stores properties which are used to track the system ceiling, as explained in below. The corresponding mutex interfaces are defined as follows:

- Create a SRP mutex:
  `Resource* SRPMutexCreate(INT8U ceiling, INT8U *err);`
- Lock a SRP mutex:
  `void SRPMutexLock(Resource* r, INT8U *err);`
- Unlock a SRP mutex:
  `void SRPMutexUnlock(Resource* r);`

The lock and unlock operations only perform bookkeeping actions by increasing and decreasing the system ceiling.

2) SRP operations and scheduling: We extended μC/OS-II with the following SRP rules at the server level:

a) Tracking the system ceiling: We use the `Resource` data-structure to implement a *system ceiling stack*. `ceiling` stores the resource ceiling and `lockingTask` stores the identifier of the task currently holding the resource. From the task identifier we can deduct to which server it is attached. The `previous` pointer is used to maintain the stack structure, i.e. it points to the previous `Resource` structure on the stack. The `ceiling` field of the `Resource` on top of the stack represents the current system ceiling.

b) Resource locking: When a component tries to lock a resource with a resource ceiling higher than the current system ceiling, the corresponding resource ceiling is pushed on top of the system ceiling stack.

c) Resource unlocking: When unlocking a resource, the value on top of the system ceiling stack is popped. The absence of nested critical sections guarantees that the system ceiling represents the resource to be unlocked. The scheduler is called to allow for scheduling ready components that might have arrived during the execution of the critical section.

d) Global scheduling: When the μC/OS-II scheduler is called it calls a function which returns the highest priority ready component. Accordingly to SRP we extend this function with the following rule: when the highest ready component has a priority lower than or equal to the current system ceiling, the priority of the *task* on top of the resource stack is returned. The returned priority serves as a task identifier, which makes easily allows to deduct the corresponding component.

e) Local scheduling: The implementations of two-level SRP protocols in [9], [10], [11] also keep track of component ceilings. We only have a binary local ceiling to indicate whether preemptions are enabled or disabled, because we explicitly chose local resource ceilings equal to the highest local priority. During global resource access, the local scheduler can only select the resource-accessing task for execution.

VI. B-HSTP IMPLEMENTATION

In this section we extend the framework presented in Section V with our proposed protocol, B-HSTP. In many microkernels, including μC/OS-II, the only way for tasks to share data structures with ISRs is by means of disabling interrupts. We therefore assume that our primitives execute non-preemptively with interrupts disabled.

Because critical sections are non-nested and local preemptions are disabled, at most one task τ_{si} at a time in each component may use a global resource. This convenient system property makes it possible to multiplex both resource-access budget q_s and budget Q_s on a single budget timer by using our virtual timer mechanism. The remaining budget $Q_s^{\mathtt{rem}}(t)$ is returned by a function that depends on the virtual timers mechanism, see Section V-A. A task therefore merely blocks on its component's budget, which we implement by adjusting the single available budget timer $Q_s^{\mathtt{rem}}(t)$.

1) Resource locking: The lock operation updates the local ceiling to prevent other tasks within the component from interfering during the execution of the critical section. Its pseudo-code is presented in Algorithm 1.

In case we have enabled SIRAP, rather than HSRP's overrun, there must be sufficient remaining budget within the server's current period in order to successfully lock a resource. If the currently available budget $Q_s^{\mathtt{rem}}(t)$ is insufficient, the task will spinlock until the next replenishment event expires. To avoid a race-condition between a resource unlock and budget depletion, we require that $Q_s^{\mathtt{rem}}(t)$ is strictly larger than X_{sr} before granting access to a resource R_r.

Algorithm 1 void HSF_lock(`Resource*` r);

```
1: updateComponentCeiling(r);
2: if HSF_MUTEX_PROTOCOL == SIRAP then
3:     while X_sr >= Q_s^rem(t) do {apply SIRAP's self-blocking}
4:         enableInterrups;
5:         disableInterrups;
6:     end while
7: end if
8: while r.status = busy do {self-suspend on a busy resource}
9:     setComponentBudget(0);
10:    enableInterrups;
11:    Schedule();
12:    disableInterrups;
13: end while
14: Q_s^▽ ← Q_s^rem(t);
15: setComponentBudget(X_sr);
16: C_s.lockedResource ← r;
17: r.status ← locked
18: SRPMutexLock(r);
```

A task may subsequently block on a busy resource, until it becomes free. When it encounters a busy resource, it suspends the component and discards all remaining budget. When the resource becomes free and the task which attempted to lock the resource continues its execution, it is guaranteed that there is sufficient budget to complete the critical section (assuming that it does not exceed its specified length, X_{sr}). The reason for this is that a component discards its budget when it blocks on a busy resource and can only continue with a fully replenished budget. This resource holding time X_{sr} defines

the resource-access budget of the locking task and component. The component's remaining budget is saved as $Q_s^{\triangledown}$ and reset to X_{sl} before the lock is established.

$setComponentBudget(0)$, see line 9, performs two actions: (*i*) the server is blocked to prevent the scheduler from rescheduling the server before the next replenishment, and (*ii*) the budget-depletion timer is canceled.

2) Resource unlocking: Unlocking a resource means that the system and component ceilings must be decreased. Moreover, the amount of consumed budget is deducted from the components stored budget, $Q_s^{\triangledown}$. We do not need to restore the component's budget, if the system ceiling is already decreased at the depletion of its resource-access budget, i.e. when a component has exceeded its specified critical-section length. The unlock operation in pseudo-code is as follows:

Algorithm 2 void HSF_unlock(`Resource*` r);

1: $updateComponentCeiling(r)$;
2: $r.status \leftarrow free$;
3: $C_s.lockedResource \leftarrow 0$;
4: **if** $System\_ceiling == RC_r$ **then**
5: $\quad setComponentBudget(\max(0, Q_s^{\triangledown} - (X_{sr} - Q_s^{\mathrm{rem}}(t))))$;
6: **else**
7: $\quad$; {we already accounted the critical section upon depletion of X_{sr}}
8: **end if**
9: *SRPMutexUnlock*(r);

3) Budget depletion: We extend the budget-depletion event handler with the following rule: if any task within the component holds a resource, then the global resource ceiling is decreased according to SRP and the resource is marked busy. A component C_s may continue in its restored budget with a decreased system ceiling. The pseudo-code of the budget-depletion event handler is as follows:

Algorithm 3 on budget depletion:

1: **if** $C_s.lockedResource \neq 0$ **then**
2: $\quad r \leftarrow C_s.lockedResource$;
3: $\quad r.status \leftarrow busy$;
4: $\quad$ *SRPMutexUnlock*(r);
5: $\quad setComponentBudget(\max(0, Q_s^{\triangledown} - X_{sr}))$;
6: **else**
7: $\quad$; {apply default budget-depletion strategy}
8: **end if**

4) Budget replenishment: For each periodic server an event handler is periodically executed to recharge its budget. We extend the budget-replenishment handler with the following rule: if any task within the component holds a resource busy, then the global resource ceiling is increased according to SRP and the resource-access budget is replenished with X_{sr} of resource R_r. A component C_s may continue in its restored budget with an increased system ceiling for that duration, before the remainder of its normal budget Q_s becomes available. The pseudo-code of this event handler is shown in Algorithm 4.

VII. Experiments and results

This section evaluates the implementation costs of B-HSTP. First, we present a brief overview of our test platform. Next, we experimentally investigate the system overhead of the synchronization primitives and compare these to our earlier protocol implementations. Finally, we illustrate B-HSTP by means of an example system.

Algorithm 4 on budget replenishment:

1: $Q_s^{\triangledown} \leftarrow Q_s$;
2: **if** $C_s.lockedResource \neq 0$ **then** {C_s keeps a resource busy}
3: $\quad r \leftarrow C_s.lockedResource$;
4: $\quad setComponentBudget(X_{sr})$;
5: $\quad$ *SRPMutexLock*(r);
6: **else**
7: $\quad$; {Apply default replenishment strategy}
8: **end if**

A. Experimental setup

We recently created a port for μC/OS-II to the OpenRISC platform [13] to experiment with the accompanying cycle-accurate simulator. The OpenRISC simulator allows software-performance evaluation via a cycle-count register. This profiling method may result in either longer or shorter measurements between two matching calls due to the pipelined OpenRISC architecture. Some instructions in the profiling method interleave better with the profiled code than others. The measurement accuracy is approximately 5 instructions.

B. Synchronization overheads

In this section we investigate the overhead of the synchronization primitives of B-HSTP. By default, current analysis techniques do not account for overheads of the corresponding synchronization primitives, although these overheads become of relevance upon deployment of such a system. Using more advanced analysis methods, for example as proposed in [28], these measures can be included in the existing system analysis. The overheads introduced by the implementation of our protocol are summarized in Table I and compared to our earlier implementation of HSRP and SIRAP in [9], [10].

1) Time complexity: Since it is important to know whether a real-time operating system behaves in a time-wise predictable manner, we investigate the disabled interrupt regions caused by the execution of B-HSTP primitives. Our synchronization primitives are independent of the number of servers and tasks in a system, but introduce overheads that interfere at the system level due to their required timer manipulations.

Manipulation of timers makes our primitives more expensive than a straightforward two-level SRP implementation. However, this is the price for obtaining temporal isolation. The worst-case execution time of the lock operation increases with 380 instructions in every server period in which a component blocks, so that the total cost depends on the misbehaving critical-section length which causes the blocking. The budget replenishment handler only needs to change the amount to be replenished, so that B-HSTP itself does not contribute much to the total execution time of the handler. These execution times of the primitives must be included in the system analysis by adding these to the critical section execution times, X_{sl}. At the local scheduling level B-HSTP is more efficient, however, because we use a simple non-preemptive locking mechanism.

TABLE I
BEST-CASE (BC) AND WORST-CASE (WC) EXECUTION TIMES FOR SRP-BASED PROTOCOLS, INCLUDING B-HSTP, MEASURED ON THE OPENRISC PLATFORM IN NUMBER OF PROCESSOR INSTRUCTIONS.

Event	**single-level SRP** [10]		**HSRP** (see [9], [10])		**SIRAP** (see [9], [10])		**B-HSTP**	
	BC	WC	BC	WC	BC	WC	BC	WC
Resource lock	124	124	196	196	214	224	763	-
Resource unlock	106	106	196	725	192	192	688	697
Budget depletion	-	-	0	383	-	-	59	382
Budget replenishment	-	-	0	15	-	-	65	76

2) Memory complexity: The code sizes in bytes of B-HSTP's lock and unlock operations, i.e. 1228 and 612 bytes, is higher than the size of plain SRP, i.e. 196 and 192 bytes. This includes the transparently implemented self-blocking and overrun mechanisms and timer management. μC/OS-II's priority-inheritance protocol has similar sized lock and unlock primitives, i.e. 924 and 400 bytes.

Each SRP resource has a data structure at the global level, i.e. we have M shared resources. Each component only needs to keep track of a single globally shared resource, because local preemptions are disabled during global resource access. However, each component C_s needs to store its resource-access durations for all its resources $R_l \in \mathcal{R}_s$.

C. SIRAP and HSRP re-evaluated

From our first implementation of SIRAP and HSRP, we observed that SIRAP induces overhead locally within a component, i.e. the spin-lock, which checks for sufficient budget to complete the critical section, adds to the budget consumption of the particular task that locks the resource. SIRAP's overhead consists at least of a single test for sufficient budget in case the test is passed. The overhead is at most two of such tests in case the initial test fails, i.e. one additional test is done after budget replenishment before resource access is granted. All remaining tests during spinlocking are already included as self-blocking terms in the local analysis [2]. The number of processor instructions executed for a single test is approximately 10 instructions on our test platform.

HSRP introduces overhead that interferes at the global system level, i.e. the overrun mechanism requires to manipulate event timers to replenish an overrun budget when the normal budget of a component depletes. This resulted in a relatively large overhead for HSRP's unlock operation compared to SIRAP, see Table I. Since similar timer manipulations are required for B-HSTP, the difference in overhead for HSRP and SIRAP becomes negligible when these protocols are complemented with means for temporal isolation. Furthermore, the absolute overheads are in the same order of magnitude.

D. B-HSTP: an example

We have recently extended our development environment with a visualization tool, which makes it possible to plot a HSF's behaviour [29] by instrumenting the code, executed on the OpenRISC platform, of our μC/OS-II extensions. To demonstrate the behavior of B-HSTP, consider an example system comprised of three components (see Table II) each with two tasks (see Table III) sharing a single global resource R_1. We use the following conventions:

1) the component or task with the lowest number has the highest priority;
2) the computation time of a task is denoted by the consumed time units after locking and before unlocking a resource. For example, the scenario – $E_{s1,1}$; $Lock(R_1)$; $E_{s1,2}$; $Unlock(R_1)$; $E_{s1,3}$ – is denoted as $E_{s1,1} + E_{s1,2} + E_{s1,3}$ and the term $E_{s1,2}$ represents the critical-section length, c_{s1l};
3) the resource holding time is longest critical-section length within a component, see Equation 3.
4) the component ceilings of the shared resource, R_1, are equal to the highest local priority, as dictated by B-HSTP.

The example presented in Figure 3 complements HSRP's overrun mechanism with B-HSTP.

TABLE II
EXAMPLE SYSTEM: COMPONENT PARAMETERS

Server	**Period** (P_s)	**Budget** (Q_s)	**Res. holding time** (X_s)
IPS 1	110	12	4.0
IPS 2	55	8	0.0
IPS 3	50	23	7.4

TABLE III
EXAMPLE SYSTEM: TASK PARAMETERS

Server	**Task**	**Period**	**Computation time**
IPS 1	Task 11	220	6.5 +4.0+6.5
IPS 1	Task 12	610	0.0+0.17+0.0
IPS 2	Task 21	110	5.0
IPS 2	Task 22	300	7.0
IPS 3	Task 31	100	12+7.4+12
IPS 3	Task 32	260	0.0+0.095+0.0

At every time instant that a task locks a resource, the budget of the attached server is manipulated according to the rules of our B-HSTP protocol, e.g. see time 11 where task 31 locks the global resource and the budget of IPS 3 is changed. After task 31 has executed two resource accesses within its specified length, in the third access it gets stuck in an infinite loop, see time instant 211. Within IPS 3, the lower priority task is indefinitely blocked, since B-HSTP does not concern the local schedulability of components. IPS 1 blocks on the busy resource at time instant 227 and cannot continue further until the resource is released. However, the activations of the independent component, implemented by IPS 2, are unaffected, because IPS 3 can only execute 7.4

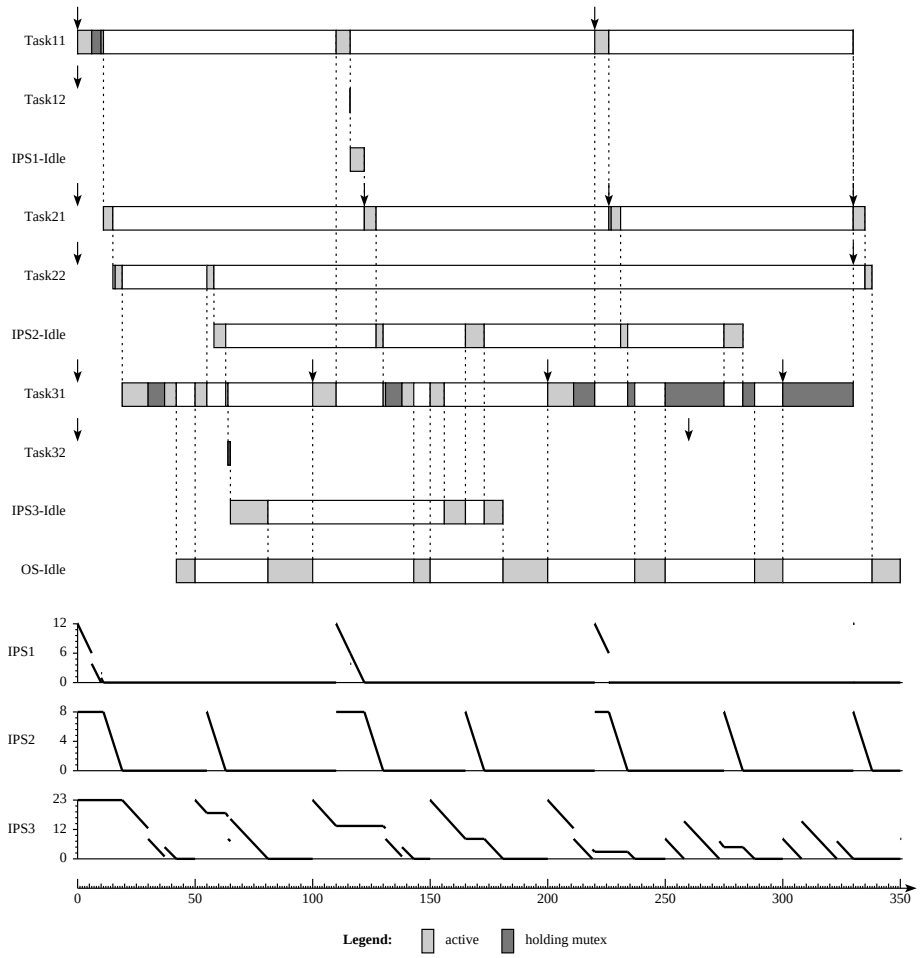

Fig. 3. Example trace, generated from instrumented code [29], combining HSRP and B-HSTP to arbitrate access between IPS 1 and IPS 3 to a single shared resource. IPS 2 is independent and continues its normal execution even when task 31 exceeds its specified critical-section length, i.e. starting at time 219. IPS 1 blocks on a busy resource and looses its remaining budget at time 227.

time units with a raised system ceiling, e.g. see time interval $[220, 235]$ where IPS 3 gets preempted after by IPS 1 that blocks on the busy resource and IPS 2 that continues its execution normally. Moreover, IPS 3 may even use its overrun budget to continue its critical section with a decreased system ceiling, see time interval $[273, 280]$, where IPS 3 is preempted by IPS 2. This is possible due to the inherent pessimism in the overrun analysis [1], [19] which allocates an overrun budget at the global level without taking into account that in normal situations the system ceiling is raised for that duration.

VIII. Discussion

A. Component ceilings revisited

We assume locally non-preemptive critical sections, which may reduce the component's schedulability. Suppose we allow preemptions of tasks that are not blocked by an SRP-based component ceiling, see (2). The blocking times of all tasks with a lower preemption level than the component ceiling do not change, providing no advantage compared to the case where critical sections are non-preemptive. Moreover, enforcement of critical-section lengths is a prerequisite to guarantee temporal isolation in HSFs, see Section IV.

As a solution we could introduce an intermediate reservation level assigned and allocated to critical sections. In addition, we need to enforce that blocking times to other components are not exceeded due to local preemptions [12]. This requires an extension to our two-level HSF and therefore complicates an implementation. It also affects performance, because switching between multiple budgets for each component (or task) is costly [16] and breaks SRP's stack-sharing property.

B. Reliable resource sharing

To increase the reliability of the system, one may artificially increase the resource-access budgets, X_{sl}, to give more slack to an access of length c_{sil} to resource R_l. Although this alleviates small increases in critical-section lengths, it comes at the cost of a global schedulability penalty. Moreover, an increased execution time of a critical section of length c_{sil} up to X_{sl} should be compensated with additional budget to

guarantee that the other tasks within the same component make their deadlines. Without this additional global schedulability penalty, we may consume the entire overrun budget X_s when we choose HSRP to arbitrate resource access, see Figure 3, because the analysis in [1], [19] allocate an overrun budget to each server period at the server's priority level. In line with [1], [19], an overrun budget is merely used to complete a critical section. However, we leave budget allocations while maximizing the system reliability as a future work.

C. Watchdog timers revisited

If we reconsider Figure 1 and Figure 3 we observe that the time-instant at which a maximum critical-section length is exceeded can be easily detected using B-HSTP, i.e. when a resource-access budget depletes. We could choose to execute an error-handler to terminate the task and release the resource at that time instant, similar to the approach proposed in AUTOSAR. However, instead of using expensive timers, we can defer the execution of such an error handler until component C_s is allowed to continue its execution. This means that the error handler's execution is accounted to C_s' budget of length Q_s. A nice result is that an eventual user call-back function can no longer hamper temporal isolation of other components than those involved in resource sharing.

IX. Conclusion

This paper presented B-HSTP, an SRP-based synchronization protocol, which achieves temporal isolation between independent components, even when resource-sharing components misbehave. We showed that it generalizes and extends existing protocols in the context of HSFs [1], [2]. Prerequisites to dependable resource sharing in HSFs are mechanisms to enforce and monitor maximum critical-section lengths. We followed the choice in [1] to make critical sections non-preemptive for tasks within the same component, because this makes an implementation of our protocol efficient. The memory requirements of B-HSTP are lower than priority-inheritance-based protocols where tasks may pend in a waiting queue. Furthermore, B-HSTP primitives have bounded execution times and jitter. Both HSRP [1] and SIRAP [2], which each provide a run-time mechanism to prevent budget depletion during global resource access, have a negligible difference in implementation complexity when complemented with B-HSTP. Our protocol therefore promises a reliable solution to future safety-critical industrial applications that may share resources.

References

[1] R. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *Real-Time Systems Symp.*, 2006, pp. 257–267.

[2] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems," in *Conf. on Embedded Software*, Oct. 2007, pp. 279–288.

[3] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Real-Time Systems Symp.*, Dec. 2003, pp. 2–13.

[4] AUTOSAR GbR, "Technical overview," 2008. [Online]. Available: http://www.autosar.org/

[5] T. M. Ghazalie and T. P. Baker, "Aperiodic servers in a deadline scheduling environment," *Real-Time Syst.*, vol. 9, no. 1, pp. 31–67, 1995.

[6] T. Baker, "Stack-based scheduling of realtime processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, March 1991.

[7] Micrium, "RTOS and tools," March 2010. [Online]. Available: http://micrium.com/

[8] R. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *Real-Time Systems Symp.*, Dec. 2005, pp. 389–398.

[9] M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Protocol-transparent resource sharing in hierarchically scheduled real-time systems," in *Conf. Emerging Technologies and Factory Automation*, 2010.

[10] M. M. H. P. van den Heuvel, R. J. Bril, J. J. Lukkien, and M. Behnam, "Extending a HSF-enabled open-source real-time operating system with resource sharing," in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, July 2010, pp. 71–81.

[11] M. Åsberg, M. Behnam, T. Nolte, and R. J. Bril, "Implementation of overrun and skipping in VxWorks," in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, July 2010.

[12] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar, "Resource sharing in reservation-based systems," in *Real-Time Systems Symp.*, Dec. 2001, pp. 171–180.

[13] OpenCores. (2009) OpenRISC overview. [Online]. Available: http://www.opencores.org/project,or1k

[14] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronisation," *IEEE Trans. on Computers*, vol. 39, no. 9, pp. 1175–1185, Sept. 1990.

[15] G. Banga, P. Druschel, and J. C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *Symp. on Operating Systems Design and Implementation*, 1999, pp. 45–58.

[16] U. Steinberg, J. Wolter, and H. Härtig, "Fast component interaction for real-time systems," in *Euromicro Conf. on Real-Time Systems*, July 2005, pp. 89–97.

[17] G. Lipari, G. Lamastra, and L. Abeni, "Task synchronization in reservation-based real-time systems," *IEEE Trans. on Computers*, vol. 53, no. 12, pp. 1591–1601, Dec. 2004.

[18] Z. Deng and J.-S. Liu, "Scheduling real-time applications in open environment," in *Real-Time Systems Symp.*, Dec. 1997, pp. 308–319.

[19] M. Behnam, T. Nolte, M. Sjodin, and I. Shin, "Overrun methods and resource holding times for hierarchical scheduling of semi-independent real-time systems," *IEEE Trans. on Industrial Informatics*, vol. 6, no. 1, pp. 93 –104, Feb. 2010.

[20] L. Almeida and P. Peidreiras, "Scheduling with temporal partitions: response-time analysis and server design," in *Conf. on Embedded Software*, Sept. 2004, pp. 95–103.

[21] M. Behnam, T. Nolte, M. Åsberg, and R. J. Bril, "Overrun and skipping in hierarchically scheduled real-time systems," in *Conf. on Embedded and Real-Time Computing Systems and Applications*, 2009, pp. 519–526.

[22] G. Buttazzo and P. Gai, "Efficient implementation of an EDF scheduler for small embedded systems," in *Workshop on Operating System Platforms for Embedded Real-Time Applications*, July 2006.

[23] M. Bertogna, N. Fisher, and S. Baruah, "Static-priority scheduling and resource hold times," in *Parallel and Distrib. Processing Symp.*, 2007.

[24] R. Rajkumar, L. Sha, and J. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Real-Time Systems Symp.*, Dec. 1988, pp. 259–269.

[25] F. Ridouard, P. Richard, and F. Cottet, "Negative results for scheduling independent hard real-time tasks with self-suspensions," in *Real-Time Systems Symp.*, Dec. 2004, pp. 47–56.

[26] J. J. Labrosse, *Microc/OS-II*. R & D Books, 1998.

[27] M. M. H. P. van den Heuvel, M. Holenderski, R. J. Bril, and J. J. Lukkien, "Constant-bandwidth supply for priority processing," *IEEE Trans. on Consumer Electronics*, vol. 57, no. 2, May 2011.

[28] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau, "Evolving real-time systems using hierarchical scheduling and concurrency analysis," in *Real-Time Systems Symp.*, Dec. 2003, pp. 25–36.

[29] M. Holenderski, M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: Tracing, visualizing and measuring the behavior of real-time systems," in *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, July 2010, pp. 37–42.

Hard Real-time Support for Hierarchical Scheduling in FreeRTOS*

Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Moris Behnam
Mälardalen Real-Time Research Centre
Västerås, Sweden
Email: rafia.inam@mdh.se

***Abstract*—This paper presents extensions to the previous implementation of two-level Hierarchical Scheduling Framework (HSF) for FreeRTOS. The results presented here allow the use of HSF for FreeRTOS in hard-real time applications, with the possibility to include legacy applications and components not explicitly developed for hard real-time or the HSF.**

Specifically, we present the implementations of (i) global and local resource sharing using the Hierarchical Stack Resource Policy and Stack Resource Policy respectively, (ii) kernel support for the periodic task model, and (iii) mapping of original FreeRTOS API to the extended FreeRTOS HSF API. We also present evaluations of overheads and behavior for different alternative implementations of HSRP with overrun from experiments on the AVR 32-bit board EVK1100. In addition, real-time scheduling analysis with models of the overheads of our implementation is presented.

***Index Terms*—real-time systems; hierarchical scheduling framework; resource sharing, fixed-priority scheduling**

I. Introduction

In real-time embedded systems the components and components integration must satisfy both (1) functional correctness and (2) extra-functional correctness, such as satisfying timing properties. Hierarchical Scheduling Framework (HSF) [1] has emerged as a promising technique in satisfying timing properties while integrating complex real-time components on a single node. It supplies an effective mechanism to provide temporal partitioning among components and supports independent development and analysis of real-time systems [2]. In HSF, the CPU is partitioned into a number of subsystems (servers or applications); each real-time component is mapped to a subsystem that contains a local scheduler to schedule the internal tasks of the subsystem. Each subsystem performes its own task scheduling, and the subsystems are scheduled by a global (system-level) scheduler. Two different synchronization mechanisms *overrun* [3] and *skipping* [4] have been proposed and analyzed for inter-subsystem resource sharing, but not much work has been performed for their practical implementations.

We have chosen FreeRTOS [5], a portable open source real-time scheduler to implement hierarchical scheduling framework. The goal is to use the HSF-enabled FreeRTOS to implement the *virtual node* concept in the ProCom component-model [6], [7]. FreeRTOS has been chosen due to its main features, like it's open source nature, small size and scalability, and support of many different hardware architectures allowing it to be easily extended and maintained. Our HSF implementation [8] on FreeRTOS for idling periodic and deferrable servers uses fixed priority preemptive scheduling (FPPS) for both global and local-level scheduling. FPPS is flexible and simple to implement, plus is the de-facto industrial standard for task scheduling. In this paper we extend our implementation of HSF to support hard real-time components. We implement time-triggered periodic tasks within the FreeRTOS operating system. We improve the resource sharing policy of FreeRTOS, and implement support for inter-subsystem resource sharing for our HSF implementation. We also provide legacy support for existing systems or components to be executed within our HSF implementation as a subsystem.

* This work is supported by the Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS. Our HSF implementation code is available at http://www.idt.mdh.se/pride/releases/hsf.

A. Contributions

The main contributions of this paper are:

- We have supported *periodic task model* within the FreeRTOS operating system.
- We have provided *a legacy support* in our HSF implementation and have mapped the old FreeRTOS API to the new API so that the user can very easily use an old system into a server within a two-level HSF.
- We have provided an efficient implementation for *resource sharing* for our HSF implementation. This entails: support for *Stack Resource Policy* for local resource sharing, and *Hierarchical Stack Resource Policy* for global resource sharing with three diferent methods to handle *overrun*.
- We have included the *runtime overhead* for *local and global schedulability analysis* of our implementation.
- We describe the *detailed design* of all the above mentioned improvements in our HSF implementations with the consideration of minimal modifications in underlying FreeRTOS kernel.
- And finally, we have *tested and calculated the performance measures* for our implementations on an AVR-based 32-bit board EVK1100 [9].

B. Resource Sharing in Hierarchical Scheduling Framework

A two-level HSF [10] can be viewed as a tree with one parent node (global scheduler) and many leaf nodes (local schedulers) as illustrated in Figure 1. The leaf nodes contain

its own internal set of tasks that are scheduled by a local (subsystem-level) scheduler. The parent node is a global scheduler and is responsible for dispatching the subsystems according to their resource reservations. Using HSF, subsystems can be developed and analyzed in isolation from each other.

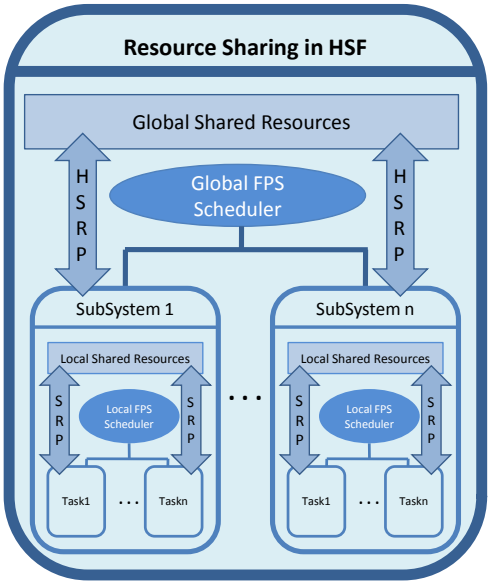

Fig. 1. Two-level Hierarchical Scheduling Framework

In a two-level HSF the resources can be shared among tasks of the same subsystem (or intra-subsystem), normally referred as *local shared resource*. The resources can also be shared among tasks of different subsystems (or inter-subsystem) called *global shared resources* as shown in Figure 1.

Different synchronization protocols are required to share resources at local and global levels, for example, *Stack Resource Policy (SRP)* [11] can be used at local level with FPPS, and to implement SRP-based overrun mechanism at global level, *Hierarchical Stack Resource Policy (HSRP)* [3] can be used.

Organisation: Section II presents the related work on hierarchical scheduler implementations. Section III gives a background on FreeRTOS in III-A, a review of our HSF implementation in FreeRTOS in III-B, and resource sharing techniques in HSF in section III-C. In section IV we provide our system model. We explain the implementation details of periodic task model, legacy support, and resource sharing in section V. In section VI we provide scheduling analysis and in section VII we present the behavior of implementation and some performance measures. In section VIII we conclude the paper. The API for the local and the global resource sharing in HSF is given in Appendix.

II. Related Work

HSF has attained a substantial importance since introduced in 1990 by Deng and Liu [1]. Saewong and Rajkumar [12] implemented and analyzed HSF in CMU's Linux/RK with deferrable and sporadic servers using hierarchical deadline monotonic scheduling. Buttazzo and Gai [13] present an HSF implementation based on Implicit Circular Timer Overflow Handler (ICTOH) using EDF scheduling for an open source RTOS, ERIKA Enterprise kernel. A micro kernel called SPIRIT-μKernel is proposed by Kim *et al.* [10] based on two-level hierarchical scheduling methodology and demonstrate the concept, by porting two different application level RTOS, VxWorks and eCos, on top of the SPIRIT-μKernel. It uses an offline scheduler at global level and the fixed-priority scheduling at local level to schedule the partitions and tasks respectively. A detailed related work on HSF implementation without resource sharing is presented in [8].

A. Local and Global Synchronization Protocols

1) Local synchronization protocols: Priority inheritance protocol (PIP) [14] was developed to solve the priority inversion problem but it does not solve the chained blocking and deadlock problems. Sha *et al.* proposed the priority ceiling protocol (PCP) [14] to solve these problems. A slightly different alternative to PCP is the immediate inheritance protocol (IIP). Baker presented the stack resource policy (SRP) [11] that supports dynamic priority scheduling policies. For fixed-priority scheduling, SRP has the same behavior as IIP. SRP reduces the number of context-switches and the resource holding time as compared to PCP. Like most real-time operating systems, FreeRTOS only support an FPPS scheduler with PIP protocol for resource sharing. We provide support for SRP for local-level resource sharing in HSF.

2) Global synchronization protocols: For global resource sharing some additional protocols have been proposed. Fisher *et al.* proposed Bounded delay Resource Open Environment (BROE) [15] protocol for global resource sharing under EDF scheduling. Hierarchical Stack Resource Policy (HSRP) [3] uses the overrun mechanism to deal with the subsystem budget expiration within the critical section and uses two mechanisms (with pay back and without payback) to deal with the overrun. Subsystem Integration and Resource Allocation Policy (SIRAP) [4] uses the skipping mechanism to avoid the problem of subsystem budget expiration within the critical section. Both HSRP and SIRAP assume FPPS. The original HSRP [3] does not support the independent subsystem development for its analysis. Behnam *et al.* [16] not only extended the analysis for the independent subsystem development, but also proposed a third form of overrun mechanism called extended overrun. In this paper we use HSRP for global resource sharing and implement all the three forms of the overrun protocol.

B. Implementations of Resource Sharing in HSF

Behnam *et al.* [17] present an implementation of a two-level HSF in the commercial operating system VxWorks with the emphasis of not modifying the underlying kernel. The implementation supports both FPS and EDF at both global and local level of scheduling and a one-shot timer is used to trigger schedulers. In [18], they implemented overrun and skipping techniques at the top of their FPS HSF implementation and compared the two techniques.

Holenderski *et al.* [19] implemented a two-level fixed-priority HSF in μC/OS-II, a commercial real-time operating system. This implementation is based on Relative Timed Event Queues (RELTEQ) [20] and virtual timers [21] on the top of RELTEQ to trigger timed events. They incorporated RELTEQ queues and virtual timers within the operating system kernel and provided interfaces for it and HSF implementation uses these interfaces. More recently, they extended the HSF with resource sharing support [22] by implementing SIRAP and HSRP (with and without payback). They measured and compared the system overheads of both primitives.

The work presented in this paper is different from that of [18] in the sense that we implement resource sharing in a two-level HSF with the aim of simplified implementation while adopting the kernel with the consideration of being consistent with the FreeRTOS. The user should be able to choose the original FreeRTOS or HSF implementation to execute, and also able to run legacy code within HSF with doing minimal changes in it. The work of this paper is different from that of [22] in the sense that we only extend the functionality of the operating system by providing support for HSF, and not changing or modifying the internal data structures. It aims at simplified implementation while minimizing the modifications of the underlying operating system. Our implementation is simpler than both [18], [22] since we strictly follow the rules of HSRP [3]. We do not have local ceilings for the global shared resources (as in [18], [22]) which simplifies the implementation. We do not allow local preemptions while holding the global resources which reduces the resource holding times as compared to [18], [22]. Another difference is that both [18], [22] implemented SIRAP and HSRP (with and without payback) while we implement all the three forms of overrun (with payback, without payback, and enhanced overrun). We do not support SIRAP because it is more difficult to use; the application programmer needs to know the WCET of each critical section to use SIRAP. Further neither implementation does provide analysis for their implementations.

III. Background

A. FreeRTOS

FreeRTOS is a portable, open source (licensed under a modified GPL), mini real-time operating system developed by Real Time Engineers Ltd. It is ported to 23 hardware architectures ranging from 8-bit to 32-bit micro-controllers, and supports many development tools. Its main advantages are portability, scalability and simplicity. The core kernel is simple and small, consisting of three or four (depends on the usage of coroutines) C files with a few assembler functions, with a binary image between 4 to 9KB.

Since most of the source code is in C language, it is readable, portable, and easily expandable and maintainable. Features like ease of use and understandability makes it very popular. More than $77,500$ official downloads in 2009 [23], and the survey result performed by professional engineers in 2010 puts the FreeRTOS at the top for the question "which kernel are you considering using this year" [24] showing its increasing popularity.

FreeRTOS kernel supports preemptive, cooperative, and hybrid scheduling. In the fixed-priority preemptive scheduling, the tasks with the same priority are scheduled using the round-robin policy. It supports both tasks and subroutines; the tasks with maximum 256 different priorities, any number of tasks and very efficient context switch. FreeRTOS supports both static and dynamic (changed at run-time) priorities of the tasks. It has semaphores and mutexes for resource sharing and synchronization, and queues for message passing among tasks. Its scheduler runs at the rate of one tick per milli-second by default.

FreeRTOS Synchronization Protocol: FreeRTOS supports basic synchronization primitives like *binary, counting* and *recursive semaphore*, and *mutexes*. The mutexes employ *priority inheritance protocol*, that means that when a higher priority task attempts to obtain a mutex that is already blocked by a lower priority task, then the lower priority task temporarily inherits the priority of higher priority task. After returning the mutex, the task's priority is lowered back to its original priority. Priority inheritance mechanism minimizes the *priority inversion* but it cannot cure deadlock.

B. A Review of HSF Implementation in FreeRTOS

A brief overview of our two-level hierarchical scheduling framework implementation [8] in FreeRTOS is given here.

Both global and local schedulers support fixed-priority preemptive scheduling (FPPS). Each subsystem is executed by a server S_s, which is specified by a *timing interface* $S_s(P_s, Q_s)$, where P_s is the period for that server $(P_s > 0)$, and Q_s is the capacity allocated periodically to the server $(0 < Q_s \leq P_s)$. Each server has a unique priority p_s and a remaining budget during the runtime of subsystem B_s. Since the previous implementation not focus on real-time, we only characterize each task τ_i by its priority ρ_i.

The global scheduler maintains a pointer, running server, that points to the currently running server.

The system maintains two priority-based lists. First is the ready-server list that contains all the servers that are ready (their remaining budgets are greater than zero) and is arranged according to the server's priority, and second is the release-server list that contains all the inactive servers whose budget has depleted (their remaining budget is zero), and will be activated again at their next activation periods and is arranged according to the server's activation times.

Each server within the system also maintains two lists. First is the ready-task list that keeps track of all the ready tasks of that server, only the ready list of the currently running server will be active at any time, and second is the delayed-task list of FreeRTOS that is used to maintain the tasks when they are not ready and waiting for their activation.

The hierarchical scheduler starts by calling `vTaskStartScheduler()` API and the tasks of the highest priority ready server starts execution. At each tick interrupt,

- The system tick is incremented.

- Check for the server activation events. The newly activated server is replenished with its maximum budget and is moved to the ready-server list.
- The global scheduler is called to handle the server events.
- The local scheduler is called to handle the task events.

1) The functionality of the global scheduler: The global scheduler performs the following functionality:

- At each tick interrupt, the global scheduler decrements the remaining budget B_s of the running server by one and handles budget expiration event (i.e. at the budget depletion, the server is moved from the ready-server list to the release-server list).
- Selects the highest priority ready server to run and makes a server context-switch if required. Either `prvChooseNextIdlingServer()` or `prvChooseNextDeferrableServer()` is called to select idling or deferrable server, depending on the value of the `configGLOBAL_SERVER_MODE` macro in the `FreeRTOSConfig.h` file.
- `prvAdjustServerNextReadyTime(pxServer)` is called to set up the next activation time to activate the server periodically.

In idling server, the `prvChooseNextIdlingServer()` function selects the first node (with highest priority) from the ready-server list and makes it the current running server. While in case of deferrable server, the `prvChooseNextDeferrableServer()` function checks in the ready-server list for the next ready server that has any task ready to execute when the currently running server has no ready task even if it's budget is not exhausted. It also handles the situation when the server's remaining budget is greater than 0, but its period ends, in this case the server is replenished with its full capacity.

2) The functionality of the local scheduler: The local scheduler is called from within the tick interrupt using the adopted FreeRTOS kernel function `vTaskSwitchContext()`. The local scheduler is the original FreeRTOS scheduler with the following modifications:

- The round robin scheduling policy among equal priority tasks is changed to FIFO policy to reduce the number of task context-switches.
- Instead of a single ready-task or delayed-task list (as in original FreeRTOS), now the local scheduler accesses a separate ready-task and delayed-task list for each server.

C. Resource sharing in HSF

Stack Resource Policy at global and local levels: We have implemented the HSRP [3] which extends SRP to HSRP. The SRP terms are extended as follows:

- *Priority.* Each task has a priority ρ_i. Similarly, each subsystem has an associated priority p_s.
- *Resource ceiling.* Each globally shared resource R_j is associated with a resource ceiling for global scheduling. This global ceiling is the highest priority of any subsystem whose task is accessing the global resource. Similarly each locally shared resource also has a resource ceiling for local scheduling. This local ceiling is the highest priority of any task (within the subsystem) using the resource.
- *System/subsystem ceilings.* System/subsystem ceilings are dynamic parameters that change during runtime. The system/subsystem ceiling is equal to the currently locked highest global/local resource ceiling in the system/subsystem.

Following the rules of SRP, a task τ_i can preempt the currently executing task within a subsystem only if τ_i has a priority higher than that of running task and, at the same time, the priority of τ_i is greater than the current subsystem ceiling.

Following the rules of HSRP, a task τ_i of the subsystem S_i can preempt the currently executing task of another subsystem S_j only if S_i has a priority higher than that of S_j and, at the same time, the priority of S_i is greater than the current system ceiling. Moreover, whilst a task τ_i of the subsystem S_i is accessing a global resource, no other task of the same subsystem can preempt τ_i.

D. Overrun Mechanisms

This section explains three overrun mechanisms that can be used to handle budget expiry during a critical section in the HSF. Consider a global scheduler that schedules subsystems according to their periodic interfaces . The subsystem budget Q_s is said to expire at the point when one or more internal tasks have executed a total of Q_s time units within the subsystem period P_s. Once the budget is expired, no new task within the same subsystem can initiate its execution until the subsystems budget is replenished at the start of next subsystem period.

To prevent excessive priority inversion due to global resource lock its desirable to prevent subsystem rescheduling during critical sections of global resources. In this paper, we employ the overrun strategy to prevent such rescheduling. Using overrun, when the budget of subsystem expires and it has a task that is still locking a global shared resource, the task continues its execution until it releases the resource. The extra time needed to execute after the budget expiration is denoted as *overrun time* θ. We implement three different overrun mechanisms [16]:

1) The basic overrun mechanism without payback, denoted as BO: here no further actions will be taken after the event of an overrun.
2) The overrun mechanism with payback, denoted as PO: whenever overrun happens, the subsystem S_s pays back in its next execution instant, i.e., the subsystem budget Q_s will be decreased by θ_s i.e. $(Q_s - \theta_s)$ for the subsystems execution instant following the overrun (note that only the instant following the overrun is affected even if $\theta_s > Q_s$).
3) The enhanced overrun mechanism with payback, denoted as EO: It is based on imposing an offset (delaying the budget replenishment of subsystem) equal to the amount of the overrun θ_s to the execution instant

that follows a subsystem overrun, at this instant, the subsystem budget is replenished with $Q_s - \theta_s$.

IV. System Model

In this paper, we consider a two-level hierarchical scheduling framework, in which a global scheduler schedules a system S that consists of a set of independently developed and analyzed subsystems S_s, where each subsystem S_s consists of a local scheduler along with a set of tasks. A system have a set of globally shared resource (lockable by any task in the system), and each subsystem has a set of local shared resource (only lockable by tasks in that subsystem).

A. Subsystem Model

For each subsystem S_s is specified by a subsystem (a.k.a. server) timing interface $S_s = \langle P_s, Q_s, p_s, B_s, X_s \rangle$, where P_s is the period and Q_s is the capacity allocated periodically to the subsystem where $0 < Q_s \leq P_s$ and X_s is the maximum execution-time that any subsystem-internal task may lock a shared global resource. Each server S_s has a unique priority p_s and at each instant during run-time a remaining budget B_s.

It should be noted that X_s is used for schedulability analysis only and our HSRP-implementation does not depend on the availability of this attribute. In the rest of this paper, we use the term subsystem and server interchangeably.

B. Task Model

For hard real-time systems, we are considering a simple periodic task model represented by a set Γ of n number of tasks. Each task τ_i is represented as $\tau_i = \langle T_i, C_i, \rho_i, b_i \rangle$, where T_i denotes the period of task τ_i with worst-case execution time C_i, ρ_i as its priority, and b_i its worst case local blocking. b_i is the longest execution-time inside a critical section with a resource-ceiling equal to or higher than ρ amongst all lower priority task inside the server of τ_i. A task, τ_i has a higher priority than another task, τ_j, if $\rho_i > \rho_j$. For simplicity, the deadline for each task is equal to T_i.

C. Scheduling Policy

We are using a fixed-priority scheduling FPS at the both global and local level. FPS is the de-facto standard used in industry. For hard-real time analysis we assume unique priorities for each server and unique priorities for each task within a server. However, our implementation support shared priorities, which are then handled in FIFO order (both at global and local scheduling).

D. Design Considerations

Here we present the challenges and goals that our implementation should satisfy:

1) **The use of HSF with resource sharing and the overrun mechanism:** User should be able to make a choice for using the HSF with resource sharing or the simple HSF without using shared resources. Further, user should be able to make a choice for selecting one of the overrun mechanisms, BO, PO, or EO.
2) **Consistency with the FreeRTOS kernel and keeping its API intact:** To embed the legacy code easily within a server in a two-level HSF, and to get minimal changes of the legacy system, it will be good to match the design of implementation with the underlying FreeRTOS operating system. To increase the usability and understandability of HSF implementation for FreeRTOS users, major changes should not be made in the underlying kernel.
3) **Managing local/global system ceilings:** To ensure the correct access of shared resources at both local and global levels, the local and global system ceilings should be updated properly upon the locking and unlocking of those resources.
4) **Enforcement:** Enforcing server execution even at it's budget depletion while accessing a global shared resource; its currently executing task should not be preempted and the server should not be switched out by any other higher priority server (whose `priority` is not greater than the `systemceiling`) until the task releases the resource.
5) **Calculating and deducting overrun time of a server for PO and EO:** In case of payback (PO and EO), the overrun time of the server should be calculated and deducted from the budget at the next server activation.
6) **Protection of shared data structures:** The shared data structures that are used to lock and unlock both local and global shared resources should be accessed in a mutual exclusive manner with respect to the scheduler.

V. Implementation

A. Support for Time-triggered Periodic Tasks

Since we are following the periodic resource model [25], we need the periodic task behavior implemented within the operating system. Like many other real-time operating systems, FreeRTOS does not directly support the periodic task activation. We incorporated the periodic task activation as given in Figure 2. To do minimal changes in the underlying operating system and save memory, we add only one additional variable `readyTime` to the task TCB, that describes the time when the task will become ready. A user API `vTaskWaitforNextPeriod(period)` is implemented to activate the task periodically. The FreeRTOS delayed-task list used to maintain the periodic tasks when they are not ready and waiting for their next activation period to start. Since FreeRTOS uses ticks, period of the task is given in number of ticks.

```
// task function
while (TRUE) do {
    taskbody();
    vTaskWaitforNextPeriod(period);
end while
```

Fig. 2. Pseudo-code for periodic task model implementation

B. Support for Legacy System

To implement legacy applications support in HSF implementation for the FreeRTOS users, we need to map the original FreeRTOS API to the new API, so that the user can run its old code in a subsystem within the HSF. A macro `configHIERARCHICAL_LEGACY` must be set in the config file to utilize legacy support. The user should rename the old `main()` function, and remove the `vTaskStartScheduler()` API from legacy code.

The legacy code is created in a separate server, and in addition to the server parameters like period, budget, priority, user also provides a function pointer of the legacy code (the old main function that has been renamed). `xLegacyServerCreate(period, budget, priority, *serverHandle, *functionPointer)` API is provided for this purpose. The function first creates a server and then creates a task called `vLegacyTask(*functionPointer)` that runs only once and performs the initialization of the legacy code (executes the old main function which create the initial set of tasks for the legacy application), and destroys itself. When the legacy server is replenished first time, all the tasks of the legacy code are created dynamically within the currently running legacy server and start executing.

We have adopted the original FreeRTOS `xTaskGenericCreate` function to provide legacy support. If `configHIERARCHICAL_SCHEDULING` and `configHIERARCHICAL_LEGACY` macros are set then `xServerTaskGenericCreate` function is called that creates the task in the currently executing server instead of executing the original code of `xTaskGenericCreate` function.

This implementation is very simple and easy to use, user only needs to rename old `main()`, remove `vTaskStartScheduler()` from legacy code, and use a single API to create the legacy server. It should be noted that the HSF guarantees separation between servers; thus a legacy non/soft real-time server (which e.g. is not analyzed for schedulability or not use predictable resource locking) can co-exists with hard real-time servers.

C. Support for Resource sharing in HSF

Here we describe the implementation details of the resource sharing in two-level hierarchical scheduling framework. We implement the local and global resource sharing as defined by Davis and Burns [3]. For local resource sharing SRP is used and for global resource sharing HSRP is used. Further all the three forms of overrun as given by Behnam *et al.* [16] are implemented. The resource sharing is activated by setting the macro `configGLOBAL_SRP` in the configuration file.

1) Support for SRP: For local resource sharing we implement SRP to avoid problems like priority inversions and deadlocks.

The data structures for the local SRP: Each local resource is represented by the structure `localResource` that stores the resource ceiling and the task that currently holds the resource as shown in Figure 3. The locked resources are stacked onto the `localSRPList`; the FreeRTOS list structure is used to implement the SRP stack. The list is ordered according to the resource ceiling, and the first element of list has the highest resource ceiling, and represents the local `system ceiling`.

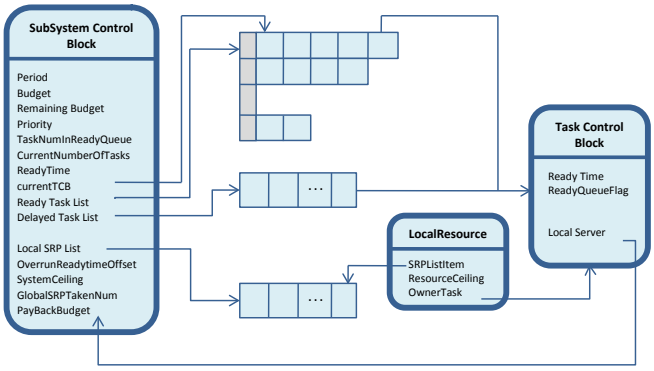

Fig. 3. Data structures to implement SRP

The extended functionality of the local scheduler with SRP: The only functionality extended is the searching for the next ready task to execute. Now the scheduler selects a task to execute if the task has the highest priority among all the ready tasks and its priority is greater than the current system ceiling, otherwise the task that has locked the highest (top) resource in the `localSRPList` is selected to execute. The API list for the local SRP is provided in the Appendix.

2) Support for HSRP: HSRP is implemented to provide global resource sharing among servers. The resource sharing among servers at the global level can be considered the same as sharing local resources among tasks at the local level. The details are as follows:

The data structures for the global HSRP: Each global resource is represented by the structure `globalResource` that stores the global-resource ceiling and the server that currently holds the resource as shown in Figure 4. The locked resources are stacked onto the `globalHSRPList`; the FreeRTOS list structure is used to implement the HSRP stack. The list is ordered according to the resource ceiling, the first element of the list has the highest resource ceiling and represents the `GlobalSystemCeiling`.

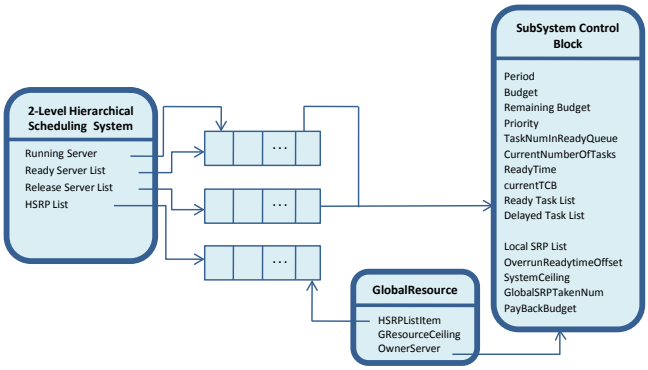

Fig. 4. Data structures to implement HSRP

The extended functionality of the global scheduler with HSRP: To incorporate HSRP into the global scheduler, `prvChooseNextIdlingServer()` and `prvChooseNextDeferrableServer()` macros are appended with the following functionality: The global scheduler selects a server if the server has the highest priority among all the ready servers and the server's priority is greater than the current `GlobalSystemCeiling`, otherwise the server that has locked the highest(top) resource in the `HSRPList` is selected to execute. The API list for the global HSRP is provided in Appendix.

3) Support for Overrun Protocol: We have implemented three types of overrun mechanisms; without payback (BO), with payback (PO), and enhanced overrun (EO). Implementation of BO is very simple, the server simply executes and overruns its budget, and no further action is required. For PO and EO we need to measure the overrun amount of time to pay back at the server's next activation.

The data for the PO and EO Overrun mechanisms: Two variables `PayBackBudget` and `OverrunReadytimeOffset` are added to the subsystem structure `subSCB` to keep a record of the overrun amount to be deducted from the next budget of the server as shown in Figure 4. The overrun time is measured and stored in `PayBackBudget`. `OverrunReadytimeOffset` is used in EO mechanism to impose an offset in the next activation of server.

The extended functionality of the global scheduler with Overrun: A new API `prvOverrunAdjustServerNextReadyTime(*pxServer)` is used to embed overrun functionality (PO and EO) into the global scheduler. For both PO and EO, the amount of overrun, i.e. `PayBackBudget` is deducted from the server `RemainingBudget` at the next activation period of the server, i.e. $B_s = Q_s - \theta_s$. For EO, in addition to this, an offset (O_s) is calculated that is equal to the amount of overrun, i.e. $O_s = \theta_s$. The server's next activation time (the budget replenishment of subsystem) is delayed by this offset. `OverrunReadytimeOffset` variable is used to store the offset for next activation of the server.

4) Safety Measure: We have modified `vTaskDelete` function in order to prevent the system from crashing when users delete a task which still holds a local SRP or a global HSRP resource. Now it also executes two private functions `prvRemoveLocalResourceFromList(*pxTaskToDelete)`, and `prvRemoveGlobalResourceFromList(*pxTaskToDelete)`, before the task is deleted.

D. Addressing Design Considerations

Here we address how we achieve the design requirements that are presented in Section IV-D.

1) **The use of HSF with resource sharing and the overrun mechanism:** The resource sharing is activated by setting the macro `configGLOBAL_SRP` in the configuration file. The type of overrun can be selected by setting the macro `configOVERRUN_PROTOCOL_MODE` to one of the three values: `OVERRUN_WITHOUT_PAYBACK`, `OVERRUN_PAYBACK`, or `OVERRUN_PAYBACK_ENHANCED`.
2) **Consistency with the FreeRTOS kernel and keeping its API intact:** We have kept consistence with the FreeRTOS from the naming conventions to API, data structures and the coding style used in our implementations; for example all the lists used in our implementation are maintained in a similar way as of FreeRTOS.
3) **Managing local/global system ceilings:** The correct access of the shared resources at both local and global levels is implemented within the functionality of the API used to lock and unlock those resources.
   When a task locks a local/global resource whose ceiling is higher than the subsystem/system ceiling, the resource mutex is inserted as the first element onto the `localSRPList`/`HSRPList`, the `systemceiling` /`GlobalSystemCeiling` is updated, and this task/server becomes the owner of this local/global resource respectively. Each time a global resource is locked, the `GlobalResourceTakenNum` is incremented.
   Similarly upon unlocking a local/global resource, that resource is simply removed from the top of the `localSRPList`/`HSRPList`, the `systemceiling` /`GlobalSystemCeiling` is updated, and the owner of this resource is set to `NULL`. For global resource, the `GlobalResourceTakenNum` is decremented.
4) **Enforcement:** `GlobalResourceTakenNum` is used as an overrun flag, and when its value is greater than zero (means a task of the currently executing server has locked a global resource), no other higher priority server (whose `priority` is not greater than the `systemceiling`) can preempt this server even if its budget depletes.
5) **Overrun time of a server for PO and EO:** `prvOverrunAdjustServerNextReadyTime` API is used to embed the overrun functionality into the global scheduler as explained in section V-C3.
6) **Protection of shared data structures:** All the functionality of the APIs (for locking and unlocking both local and global shared resources) is executed within the FreeRTOS macros `portENTER_CRITICAL()` and `portEXIT_CRITICAL()` to protect the shared data structures.

VI. Schedulability Analysis

This section presents the schedulability analysis of the HSF, starting with local schedulability analysis (i.e. checking the schedulability of each task within a server, given the servers timing interface), followed by global schedulability analysis (i.e., checking that each server will receive its capacity within its period given the set of all servers in a system).

A. The Local Schedulability Analysis

The local schedulability analysis can be evaluated as follows [25]:

$$\forall \tau_i \; \exists t : 0 < t \leq D_i, \;\; \texttt{rbf}(i,t) \leq \texttt{sbf}(t), \qquad (1)$$

where `sbf` is the supply bound function, based on the periodic resource model presented in [25], that computes the minimum

possible CPU supply to S_s for every time interval length t, and $\texttt{rbf}(i,t)$ denotes the request bound function of a task τ_i which computes the maximum cumulative execution requests that could be generated from the time that τ_i is released up to time t and is computed as follows:

$$\texttt{rbf}(i,t) = C_i + b_i + \sum_{\tau_k \in \texttt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil \cdot C_k, \tag{2}$$

where $\texttt{HP}(i)$ is the set of tasks with priorities higher than that of τ_i and b_i is the maximum local blocking.

The evaluation of sbf depends on the type of the overrun mechanism;

a) Overrun without payback:

$$\texttt{sbf}(t) = \begin{cases} t - (k+1)(P_s - Q_s) & \text{if } t \in W^{(k)} \\ (k-1)Q_s & \text{otherwise,} \end{cases} \tag{3}$$

where $k = \max\Big(\lceil \big(t-(P_s-Q_s)\big)/P_s \rceil, 1\Big)$ and $W^{(k)}$ denotes an interval $[(k+1)P_s - 2Q_s, (k+1)P_s - Q_s]$.

b) Overrun with payback [16]:

$$\texttt{sbf}(t) = \max\Big(\min\Big(f_1(t), f_2(t)\Big), 0\Big), \tag{4}$$

where $f_1(t)$ is

$$f_1(t) = \begin{cases} t - (k+1)(P_s - Q_s) - X_s & \text{if } t \in W^{(k)} \\ (k-1)Q_s & \text{otherwise,} \end{cases} \tag{5}$$

where $k = \max\Big(\lceil \big(t-(P_s-Q_s)-X_s\big)/P_s \rceil, 1\Big)$ and $W^{(k)}$ denotes an interval $[(k+1)P_s - 2Q_s + X_s, (k+1)P_s - Q_s + X_s]$, and $f_2(t)$ is

$$f_2(t) = \begin{cases} t - (2)(P_s - Q_s) & \text{if } t \in V^{(k)} \\ t - (k+1)(P_s - Q_s) - X_s & \text{if } t \in Z^{(k)} \\ (k-1)Q_s - X_s & \text{otherwise,} \end{cases} \tag{6}$$

where $k = \max\Big(\lceil \big(t-(P_s-Q_s)\big)/P_s \rceil, 1\Big)$, $V^{(k)}$ denotes an interval $[2P_s - 2Q_s, 2P_s - Q_s - X_s]$, and $Z^{(k)}$ denotes an interval $[(k+2)P_s - 2Q_s, (k+2)P_s - Q_s]$.

c) Enhanced overrun:

$$\texttt{sbf}(t) = \max\Big(f_2(t), 0\Big). \tag{7}$$

B. The Global Schedulability Analysis

A global schedulability condition is

$$\forall S_s \; \exists t : 0 < t \leq P_s, \;\; \texttt{RBF}_s(t) \leq \texttt{t}. \tag{8}$$

where $\texttt{RBF}_s(t)$ is the request bound function and it is evaluated depending on the type of server (deferrable or idling) and type of the overrun mechanism (see [16] for more details). First, we will assume the idling server and later we will generalize our analysis to include deferrable server.

d) Overrun without payback:

$$\texttt{RBF}_s(t) = (Q_s + X_s + Bl_s) + \sum_{S_k \in \texttt{HPS}(s)} \left\lceil \frac{t}{P_k} \right\rceil \cdot (Q_k + X_k). \tag{9}$$

where $\texttt{HPS}(s)$ is the set of subsystems with priority higher than that of S_s. Let Bl_s denote the maximum blocking imposed to a subsystem S_s, when it is blocked by lower-priority subsystems.

$$Bl_s = \max\{X_j |\; S_j \in \texttt{LPS}(S_s)\}, \tag{10}$$

where $\texttt{LPS}(S_s)$ is the set of subsystems with priority lower than that of S_s.

e) Overrun with payback:

$$\texttt{RBF}_s(t) = (Q_s + X_s + Bl_s) + \sum_{S_k \in \texttt{HPS}(s)} \left(\left\lceil \frac{t}{P_k} \right\rceil (Q_k) + X_k \right). \tag{11}$$

f) Enhanced overrun:

$$\texttt{RBF}_s(t) = (Q_s + X_s + Bl_s) + \sum_{S_k \in \texttt{HPS}(s)} \Big(\left\lceil \frac{t + J_k}{P_k} \right\rceil (Q_k) + X_k \Big). \tag{12}$$

Where $J_s = X_s$ and the schedulability analysis for this type is

$$\forall S_s, 0 < \exists t \leq P_s - X_s, \;\; \texttt{RBF}_s(t) \leq \texttt{t}, \tag{13}$$

For deferrable server, a higher priority server may execute at the end of its period and then at the beginning of the next period. To model such behavior a jitter (equal to $P_k - (O_k + X_k)$) is added to the ceiling in equations 9, 11 and 12.

C. Implementation Overhead

In this section we will explain how to include the implementation overheads in the global schedulability analysis.

Looking at the implementation we can distinguish two types of runtime overhead associated with the system tick: (1) a repeated overhead every system tick independently if it will release a new server or not, and (2) an overhead which occurs whenever a server is activated and it includes the overhead of scheduling, maybe context switch, budget depletion after consuming the budget then another context switch and scheduling and finally it includes the overrun overhead.

(1) Is called fixed overhead (fo) and it is the result of updating the system tick and perform some checking and its value is always fixed. This overhead can be added to equation 8. This equation assumes that the processor can provide all CPU time to the servers (t in the right side of the equation) now we assume that every system tick (st), a part will be consumed by the operating system (fo) and then instead of using t in the right side of equation 8, we can use $(1 - fo/st) \times t$ to include the fixed overhead. (st defaults to $1ms$ for our implementation.)

(2) Is called server overhead (so) and repeats periodically for every server, i.e. with a period P_i. Since the server

overhead is executed by the kernel its not enough to model it as extra execution demand from the server. Instead the overhead should be modeled as a separate server S_o (one server S_o corresponding to each real server S_i) executing at a priority higher that of any real server with parameters $P_o = P_i$, $Q_o = so$, and $X_o = 0$.

The overhead-parameters are dependent on the number of servers, tasks and priority levels, etc. and should be quantified with static WCET-analysis which is beyond the scope of this paper; however some small test cases reported in [8] the measured worst-case for idling servers are $fo = 32\mu s$ and $so = 74\mu s$, and for deferrable servers they are $fo = 32\mu s$ and $so = 85\mu s$ for three servers with total seven tasks.

VII. Experimental Evaluation

In this section, we report the evaluation of behavior and performance of the resource sharing in HSF implementation. All measurements are performed on the target platform EVK1100 [9]. The AVR32UC3A0512 micro-controller runs at the frequency of 12MHz and its tick interrupt handler at 1ms.

A. Behavior Testing

In this section we perform an experiment to test the behavior of overrun in case of global resource sharing in HSF implementation. The experiment is performed to check the overrun behavior in idling periodic server by means of a trace of the execution. Two servers S1, and S2 are used in the system, plus idle server is created. The servers used to test the system are given in Table I.

Server	S1	S2
Priority	2	1
Period	20	40
Budget	10	15

TABLE I
Servers used to test system behavior.

Note that higher number means higher priority. Task properties and their assignments to the servers is given in Table II. $T2$ and $T3$ share a global resource. The execution time of $T2$ is $(3+3)$ that means a normal execution for initial 3 time units and the critical section execution for the next 3 time units, similarly $T3$ $(10+8)$ executes for 10 time units before critical section and executes for 8 time units within critical section. The visualization of the executions of budget overrun without payback (BO) and with payback (PO) for idling periodic server are presented in Figure 5 and Figure 6 respectively.

Tasks	T1	T2	T3
Servers	$S1$	$S1$	$S2$
Priority	2	1	1
Period	15	20	60
Execution Time	3	$(3+3)$	$(10+9)$

TABLE II
Tasks in both servers.

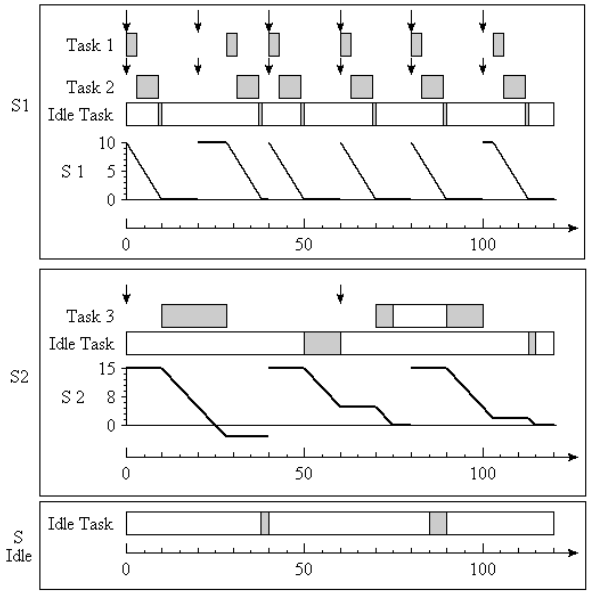

Fig. 5. Trace of budget overrun without payback (BO) for Idling server

In the visualization, the arrow represents task arrival, a gray rectangle means task execution. In Figure 5 at time 20, the high priority server $S1$ is replenished, but its priority is not higher than the global system ceiling, therefore, it cannot preempt server $S2$ which is in the critical section. $S2$ depletes its budget at time 25, but continues to executes in its critical section until it unlocks the global resource at time 29. The execution of S1 is delayed by 9 time units.

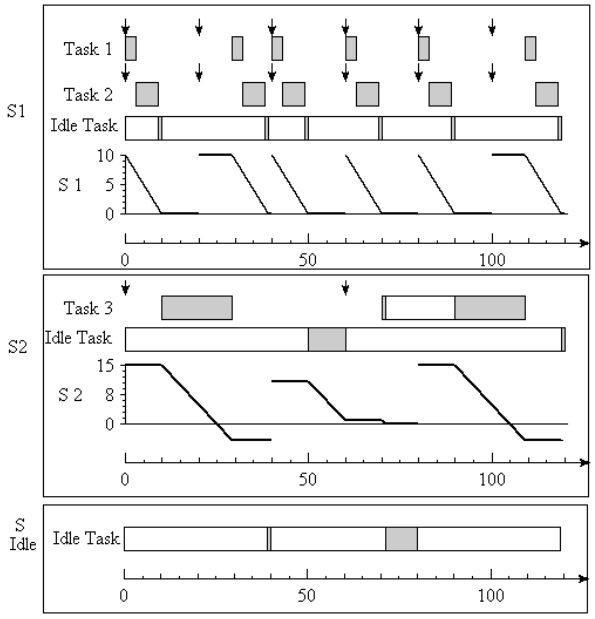

Fig. 6. Trace of budget overrun with payback (PO) for Idling server

In case of overrun with payback, the overrun time is deducted from the budget at the next server activation, as

shown in Figure 6. At time 40 the server $S2$ is replenished with a reduced budget, while in case of overrun without payback the server is always replenished with its full budget as obvious from Figure 5.

B. Performance Measures

Here we report the performance measures of lock and unlock functions for both global and local shared resources.

The execution time of functions to lock and unlock global and local resources is presented in Table III. For each measure, a total of 1000 values are computed. The minimum, maximum, average and standard deviation on these values are calculated and presented for both types of resource sharing.

Function	Min.	Max.	Average	St. Dev.
`vGlobalResourceLock`	21	21	21	0
`vGlobalResourceUnlock`	32	32	32	0
`vLocalResourceLock`	21	32	26.48	5.51
`vLocalResourceUnlock`	21	21	21	0

TABLE III
THE EXECUTION TIME (IN MICRO-SECONDS (μS)) OF GLOBAL AND LOCAL LOCK AND UNLOCK FUNCTION.

VIII. CONCLUSIONS

In this paper, we have provided a hard real-time support for a two-level HSF implementation in an open source real-time operating system FreeRTOS. We have implemented the periodic task model within the FreeRTOS kernel. We have provided a very simple and easy implementation to execute a legacy system in the HSF with the use of a single API. We have added the SRP to the FreeRTOS for efficient resource sharing by avoiding deadlocks. Further we implemented HSRP and overrun mechanisms (BO, PO, EO) to share global resources in a two-level HFS implementation. Under assumption of nested locking, the overrun is bounded and is equal to the longest resource-holding time. Hence, the temporal isolation of HSF is subject to the bounded resource-holding time.

We have focused on doing minimal modifications in the kernel to keep the implementation simple and keeping the original FreeRTOS API intact. We have presented the design and implementation details and have tested our implementations on the EVK1100 board. We have included the overheads for local-level and global-level resource sharing into the schedulability analysis. In future we plan to integrate the virtual node concept of ProCom model on-top of the presented HSF [6], [7].

REFERENCES

[1] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *IEEE Real-Time Systems Symposium (RTSS)*, 1997.

[2] G. Lipari, P. Gai, M. Trimarchi, G. Guidi, and P. Ancilotti. A hierarchical framework for component-based real-time systems. *Component-Based Software engineering*, LNCS-3054(2004):209–216, May 2005.

[3] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *IEEE Real-Time Systems Symposium (RTSS'06)*.

[4] Moris Behnam, Insik Shin, Thomas Nolte, and Mikael Sjödin. SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *Proc. EMSOFT*, pages 279–288, October 2007.

[5] FreeRTOS web-site. http://www.freertos.org/.

[6] J. Carlsson, J. Feljan, and M. Sjödin. Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems. In *36th (SEAA)*, 2010.

[7] Rafia Inam, Jukka Mäki-Turja, Jan Carlson, and Mikael Sjödin. Using temporal isolation to achieve predictable integration of real-time components. In *WiP Session of (ECRTS10)*, pages 17–20, 2010.

[8] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, S. M. H. Ashjaei, and Sara Afshar. Support for hierarchical scheduling in FreeRTOS. In *To appear in the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA11)*, September 2011.

[9] ATMEL EVK1100 product page. http://www.atmel.com/dyn/Products/.

[10] Daeyoung Kim, Yann-Hang Lee, and M. Younis. Spirit-ukernel for strongly partitione real-time systems. In *Proceedings (RTCSA00)*, 2000.

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

[12] S. Saewong and R. Rajkumar. Hierarchical reservation support in resource kernels. In *IEEE (RTSS01)*, 2001.

[13] G. Buttazzo and P. Gai. Efficient edf implementation for small embedded systems. In *International Workshop on (OSPERT06)*, 2006.

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

[15] N. Fisher, M. Bertogna, and S. Baruah. The design of an edf-scheduled resource-sharing open environment. In *IEEE (RTSS07)*.

[16] Moris Behnam, Thomas Nolte, Mikael Sjödin, and Insik Shin. Overrun Methods and Resource Holding Times for Hierarchical Scheduling of Semi-Independent Real-Time Systems. *IEEE TII*, 6(1), February 2010.

[17] Moris Behnam, Thomas Nolte, Insik Shin, Mikael Åsberg, and Reinder J. Bril. Towards hierarchical scheduling on top of vxworks. In *Proceedings of the Fourth International Workshop (OSPERT'08)*.

[18] Mikael Åsberg, Moris Behnam, Thomas Nolte, and Reinder J. Bril. Implementation of overrun and skipping in vxworks. In *Proceedings of the 6th International Workshop (OSPERT10)*, 2010.

[19] Mike Holenderski, Wim Cools, Reinder J. Bril, and J. J. Lukkien. Extending an Open-source Real-time Operating System with Hierarchical Scheduling. Technical Report, Eindhoven University, 2010.

[20] M. Holenderski, W. Cools, Reinder J. Bril, and J. J. Lukkien. Multiplexing Real-time Timed Events. In *Work in Progress session of (ETFA09)*.

[21] M.M.H.P. van den Heuvel, M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien. Virtual Timers in Hierarchical Real-time Systems. In *Work in Progress Session of (RTSS09)*, December 2009.

[22] M.M.H.P. van den Heuvel, M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien. Extending an HSF-enabled Open-Source Rel-Time Operating System with Resource sharing. In *(OSPERT10)*, 2010.

[23] Microchip web-site.

[24] EE TIMES web-site. http://www.eetimes.com/design/embedded/4008920/The-results-for-2010-are-in-.

[25] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *IEEE (RTSS03)*, pages 2–13, 2003.

APPENDIX

A synopsis of the application program interface to implement resource sharing in HSF implementation is presented below. The names of these API are self-explanatory.

1) `xLocalResourcehandle xLocalResourceCreate(uxCeiling)`
2) `void vLocalResourceDestroy(xLocalResourcehandle)`
3) `void vLocalResourceLock(xLocalResourcehandle)`
4) `void vLocalResourceUnLock(xLocalResourcehandle)`
5) `xGlobalResourcehandle xGlobalResourceCreate (uxCeiling)`
6) `void vGlobalResourceDestroy(xGlobalResourcehandle)`
7) `void vGlobalResourceLock(xGlobalResourcehandle)`
8) `void vGlobalResourceUnLock(xGlobalResourcehandle)`

RTOS-Based Embedded Software Development using Domain-Specific Language

Mohamed-El-Mehdi AICHOUCH* Jean-Christophe PRÉVOTET* Fabienne NOUVEL*
*Université européenne de Bretagne, INSA, IETR, UMR 6164, F-35708 Rennes, France
{Mohamed-El-Mehdi.Aichouch, Jean-Christophe.Prevotet, Fabienne.Nouvel}@insa-rennes.fr

***Abstract*—Using model-based approaches helps us to understand complex problems and their potential solutions through abstraction. Therefore, it is obvious that embedded systems, which are often among the most complex engineering systems, can benefit greatly from modeling techniques. However, the use of models in real projects often exhibits some limitations. They often cannot be used in existing embedded system due to their underlying modeling technology. In this paper, we propose a domain-specific language to create RTOS models and transform them into a specific source code that can be executed on existing embedded systems.**

I. INTRODUCTION

In this era of intense enthusiasm for automation in almost all time critical fields, real-time systems are more and more envisaged in industrial, commercial, medical, space and military applications. The design of real time embedded systems is a complex process that depends on the effective interplay of multiple disciplines, such as mechanics, electronics, and software engineering. Moreover, real-time systems have to process many concurrent activities within an accurate time-frame.

Real-time embedded systems have distinguishing requirements such as real-time performance and limited resources. The requirements are often satisfied by adopting a Real-Time Operating-System (RTOS). As a result, many embedded applications are designed and implemented to run directly on top of the RTOS. In some cases, the needs of the developed application can be fulfilled by a standard existing RTOS e.g. VxWorks [1], Windows-CE [2], QNX [3] etc. In other cases, it is often required to add new functionalities, modify or even develop a specific RTOS from scratch. In this article we focus on the second case, especially when we are facing the problem of adding, modifying or creating new functionalities. In the classic development process, adding or modifying new functionalities mean that the developer of the software take the existing RTOS source code and modify it by hand. This process has some limitations. First, modifying an existing code means that the designer has a good knowledge of the architecture. Second, it increases the time of the development phase and thus the financial costs of the project. To avoid these problems we propose to follow the Model-Driven Engineering (MDE) approach in order to help the designer in building a custom RTOS for embedded systems.

Model-driven software development is becoming a viable alternative to the increase of software complexity. A model-driven approach allows a developer to work at high level of abstraction without being concerned by the specificity of hardware and software. A developer only needs to focus on building an abstract model and automated tools may be used for the model-to-code transformation [4]. Consequently, the model-driven approach can provide significant advantages in terms of productivity and maintenance.

In this paper, we present a Domain-Specific Language to develop RTOS-based embedded software. Our approach begins by modeling the RTOS structure and aims to generate a concrete RTOS-specific implementation at the end. To reach this purpose, we have defined an RTOS-specific language to write the RTOS model. Then, we developed a transformation tool to transform the RTOS model into source code. Although targeting the specific µC/OS-II kernel as example for code generation, the proposed tool is also foreseen to generate other RTOS code.

The paper is organized as follows: section 2 reviews the basics of model-driven engineering; section 3 presents the related works. In section 4, the modeling approach of real-time operating systems using DSL is described. Finally, section 5 presents the obtained results and how the modeling approach may be used by a system designer.

II. BASICS OF MODEL-DRIVEN ENGINEERING

a. Model Driven Engineering

The Model-Driven Engineering (MDE) is an approach for software development, which is based on models as a first artifact in the development process. Then, a transformation is applied on these models to map the information from one model to another or to generate executable programs. A model is an abstraction, a sufficient simplification to understand the real system. In this context, a system may be defined using different sub-models connected to each other. The definition of a domain specific modeling language, called meta-modeling, is the key issue of the model driven engineering. It allows to define the rules, constraints that will be required to build a specific model. Once a model is completely defined, it is often necessary to apply models' transformation in order to generate custom code, documentation, test, validation, verification and execution.

After the adoption of the object oriented approach by the software industry, the model-driven approach may be seen

as the continuity of the initial approach. While the object oriented approach is founded on the notion of "*object that inherits from*" and "*object is an instance of*", in MDE the main concept is a "*model*" for which there exists no universal definition. According to the OMG [5], we retain this definition of a model:

Definition (Model): a model is an abstraction of a system, modeled upon a set of facts which was built for particular intent. A model should be used to answer the question about the modeled system.

A model should be a pertinent abstraction of the system that it models. The model must answer the questions that we have about it as precisely as the system itself can do. In other terms, it should be achievable to substitute the system by the corresponding model.

In MDE, the notion of model refers explicitly to the notion of well-formed language. Actually, an operational model is a model that can be manipulated by a computer. This language should be clearly defined, and the definition of a modeling language has been formalized using particular models, called "*meta-model*".

Definition (Meta-model): a meta-model is a specific model defining a language to describe other models.

The Object Management Group (OMG) uses these two notions to define the set of the Unified Modeling Language [6] standards.

b. Unified Modeling Language

UML is the most widely used standard for describing systems in terms of object concepts. UML is very popular in the specification and design of software, most often to be written using an object-oriented language. UML emphasizes the idea that complex systems are best described through a number of different views, as no single view can capture all aspects of such system completely. Moreover, it includes several different types of model diagrams to capture usage scenarios, class structures, behaviors, and implementations.

c. Model Driven Architecture

The extensive use of the UML has been a major point in the transition towards model-driven engineering. After the acceptance of the key concept of meta-model, many meta-models have emerged. In order to avoid the multiplicity of these meta-models within a domain and to circumvent the incompatibility between these models, the OMG has proposed a standard language to define meta-models. This language constitutes a model itself and corresponds to a "*meta-meta-model*" named Meta-Object Facility (MOF) [7].

Definition (meta-meta-model): a meta-meta-model is a model that defines a modeling language, i.e. the necessary modeling element to define a modeling language. It should have the ability to define itself.

According to these definitions of the different abstraction levels, the OMG has organized these notions of modeling hierarchically. The real world is represented at the lower level (M0). The models representing this reality are based at level (M1). The meta-model used to define these models are at level (M2). Finally, the meta-meta-model, unique and self-defined, is represented at the top level (M3).

Figure 1. OMG Hierarchical Modeling Levels

The Model-driven Architecture (MDA) [8] relies on the UML standard to describe the different phases of the development project cycle. In MDA, a Computational Independent Model (CIM) is elaborated in order to specify the solution to the requirement. Then, a Platform-Independent Model (PIM) of the system is developed and the model is transformed to obtain a Platform-Specific Model (PSM). This facilitates early validation and implementation on different platforms. The judicious and correct application of all these concepts increases productivity reduces software development time and provides high quality products.

d. Domain Specific Language

In the MDA approach, it may be noticed that the model-driven engineering is tightly associated to UML. However, an important point here is to separate the MDE approach from the UML formalism. The reason is that the model-driven engineering scope is wider than UML. Sometimes UML must be reduced or extended through mechanisms like Profiles. These mechanisms generally do not have the required precision and sometimes can lead to a wrong decision. In contrast, the model-driven approach encourages the creation of domain-specific language that the user can handle easily.

By definition, a Domain-Specific Language (DSL) is a language designed to be useful for a specific set of tasks, as opposed to a general purpose language. With DSL, architects are currently able to create models very rapidly and efficiently. They are also capable of generating executable code from the defined models in a very simple manner.

e. DSL vs. MDA

At the moment, DSL and MDA are the two major modeling options and are both well supported by the current modeling tools environment.

In our work, both techniques can be used, but we have made the choice to use DSL for three main reasons.

- DSL do not contain unnecessary aspects of what they are modeling. DSL tend to be much more focused on the details of the domain in question and use the terminology of that domain.
- The long-term cost of using a DSL can be much lower than using UML, because DSLs are created to fit a specific domain, In UML, the work that is required to apply a general-purpose UML to a specific purpose is often prohibitive.
- The platform-specific code generator is easy to write, it is just important to use the domain model that the DSL provides.

III. RELATED WORKS

There are a number of research projects that work on developing real-time and embedded systems, while applying the model driven development approach. Examples are OMEGA, HIDOORS, FLEXICON [10, 11, 12], etc.

We refer to [13] to indicate that some of the related projects use the development methodology based on UML for real-time and embedded application. They select a suitable subset of UML diagram types and concepts as required by the development methodology and then create a UML Profile that supports the modeling of the real-time features. However for validation of the UML specification, they use the UML Verification Environment into a specific commercial tool. For example, the Rhapsody CASE-tool [14] written in C++ is often utilized. However, this tool does not put emphasis on the implementation aspect in C. Moreover, the tool does not include custom RTOS services: it only manages existing RTOS after the transformation of the application models.

The HOPES [15] project proposes a generic RTOS APIs that can capture most of the typical RTOS services that can be used as a means for describing application's RTOS-behavior at an early design stage. Authors use their own transformation tool to generate fully functional code by transforming generic RTOS APIs to specific RTOS APIs. In HOPES, designers have to write the application model and then describe its behavior by inserting RTOS API calls. From this description, they are finally able to generate the RTOS source code.

Component-based Software Engineering has emerged as an alternative approach for the rapid assembly of flexible software systems, where the main benefits are reuse and separation of concerns. It has been applied in [16] to propose a componentization of the µC/OS-II. From the RTOS resources, reified as components, they present how component-based applications are designed on top of it. The componentization process requires a reengineering of the original implementation of the µC/OS-II. An RTOS implementation designed following the component paradigm and the aforementioned flexibility requirements involve an overhead in term of performance.

IV. RTOS-SPECIFIC LANGUAGE TOOL

The purpose of our approach consists in helping the system designer to build a custom RTOS very rapidly and efficiently. The main idea is to propose a tool in which the designer is able to integrate different RTOS services according to the application requirements and generate an optimized and light version of a RTOS.

This approach is composed of three main steps. The first is based on the design of a complete RTOS model; the second step consists in applying a transformation on the result model to produce an RTOS source code. Finally, the generated source code may be used by the developer through a RTOS Application Programming Interface (API).

a. Design the structure of the RTOS

Depending on the specifications of the application to be developed, the designer has to decide which services will be included in the RTOS. The purpose is to adapt the RTOS to the application. Many important decisions will be taken at this stage. For instance, the tasks' scheduling policy, tasks communication, tasks placement may be defined at this point. In our approach, we propose to facilitate these design decisions by using models. The developer builds a complete RTOS model from entity models corresponding to each service. The next step consists in transforming these models into source code using automatic code generation.

b. Transformation of the RTOS model

The most important benefit of DSL, as with modeling in general, is the boost in productivity that results from automatic code generation. The transformation engine will iterate over the RTOS model and will generate the associated code for each service model.

c. Use of the RTOS source code

After transforming models to source code, the developer has a ready to use RTOS in his possession. The next step is to include this generated source code into the project inside an Integrated Development Environment (IDE) and start writing the application code using the provided API of the RTOS. Three important benefits can be retained from this step: a clear view of the global architecture of the RTOS code, the advantage of getting an optimized RTOS source code and the development time that is saved with the automatic generation process.

In Section 2, it has been seen that model-driven engineering promotes the creation of domain-specific language. The first task in this area is to define the elements that compose the "*modeling language*" and the tool to create them. A modeling language is defined by five concepts: an abstract syntax, a concrete syntax, a mapping between the abstract and concrete syntax, the semantic domain and the mapping between the semantic domain and the abstract syntax.

d. RTOS Abstract syntax or Meta-Model

The abstract syntax defines the set of concepts and their relationships. The meta-modeling language OMG standard MOF offers the elementary concepts and relationships to define the abstract syntax. Some of these concepts are used by technologies like the Eclipse Modeling Project [17], and the Microsoft Visualization and Modeling SDK [18] which provide the basic framework for modeling. Both of them are based on the same basic concept inspired from the object

oriented approach. Figure 2 shows the architecture of these concepts. The concept of class is used to represent real objects. A class is characterized by properties which are named *reference* when they are typed by "TypedElement", and *attribute* when they are typed by a "DataType" (ex. Boolean, String, Int…).

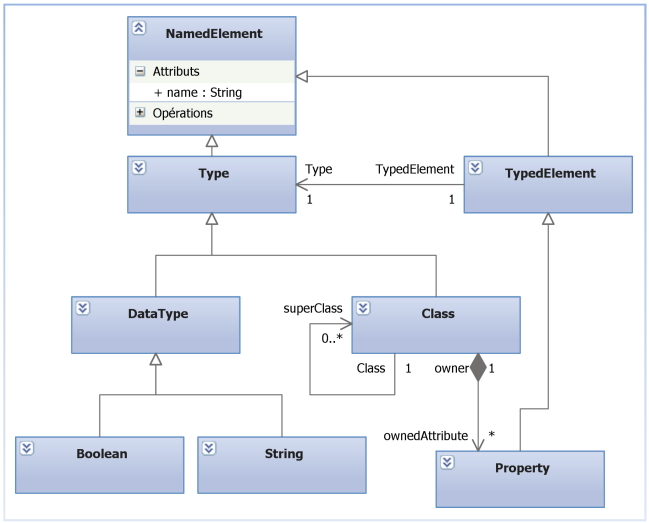

Figure 2. Main concept of meta-modeling (EMOF 2.0)

In reality, an RTOS is composed of services providing operations. Our RTOS meta-model is based on three main classes named "RTOSModel", "Service" and "Operation". Two containment relationships are then available. The "RTOSModel" class has the container for "Service", and the "Services" class contains a list of "Operation" (see figure 3).

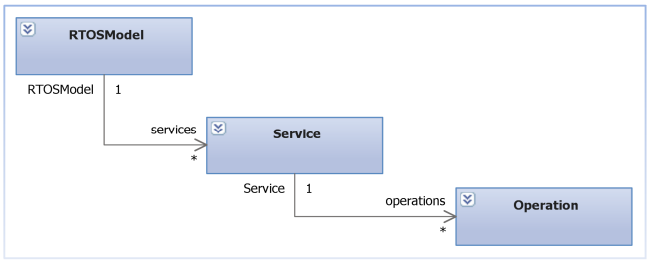

Figure 3. The main concept of the RTOS meta-model

This basic meta-model is extended to represent the real services of an RTOS. For each module in the RTOS, a meta-model entity is defined. For instance, the Task module is represented by the "*Task Manager*" class. Each "*Service Manager*" class inherits from the abstract "Service" class. Figure 4 shows some inheritance relationships. Because of this extensibility, we are able to integrate new "Service" that will be within a services' library.

According to this meta-model, it is possible to assist the software architect to make decision by providing semantic constraints and validation rules which can be applied on the model. These facilities reduce the probability of introducing errors at design time.

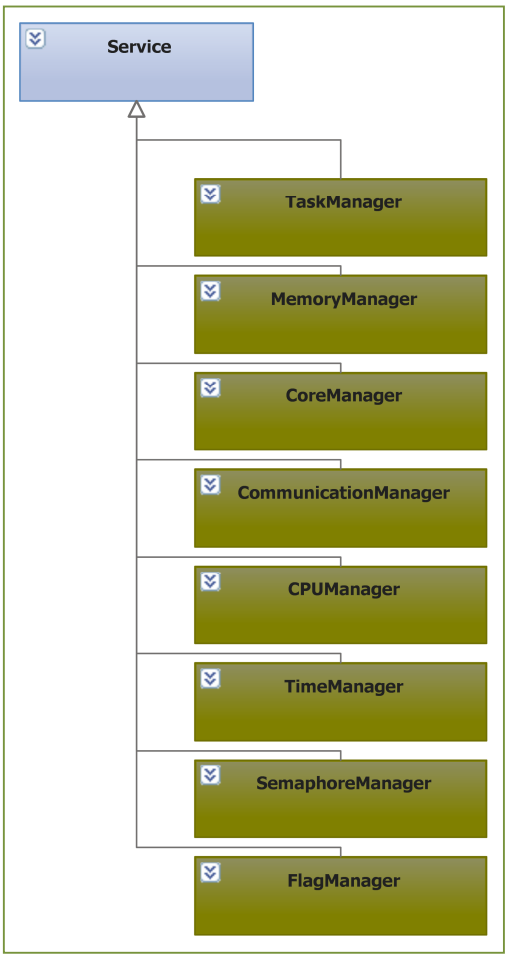

Figure 4. Real RTOS services represented in the abstract syntax

We have used the Microsoft Visualization and Modeling Standard Development Kit (SDK) to define an abstract syntax. It offers the possibility to create an abstract syntax by implementing an UML class diagram as a graphical meta-meta-modeling language. Using the DSL designer, we have defined the entire abstract syntax in terms of graphical notation.

e. RTOS Concrete Syntax

The concrete syntax describes how the information that is encoded in the underlying meta-model elements is presented through the User Interface of the designer tool. With this concrete syntax, the user can create model of RTOS. Our RTOS concrete syntax is based on the graphical notation and on the mapping between this notation and the abstract syntax. The graphical notation is created via the same DSL designer that is used to create the abstract syntax. We have decided that the "*RTOSModel*" class should appear as well as the diagram containing the "Service" models. We have associated a rectangular shape with compartment to the "*Service*" class in order to display its "*Operation*" list. In figure 5, we illustrate a model of the µC/OS-II RTOS implementing the concrete syntax.

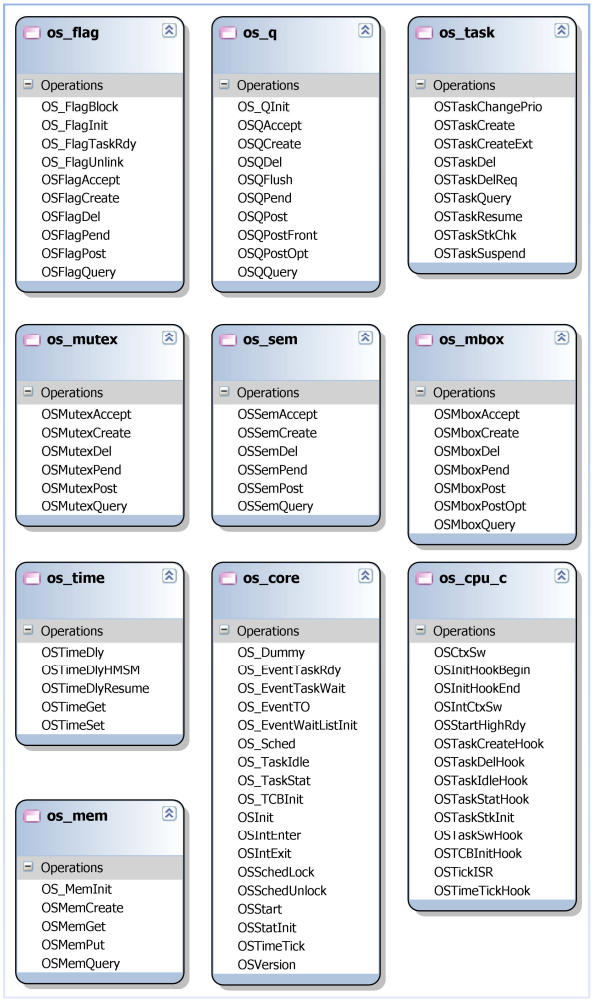

Figure 5. Design of the µC/OS using the RTOS-DSL tool

The µC/OS model illustrated in the figure 5 conforms to the RTOS meta-model. For instance, the "os_core" object is defined by the meta-model class "CoreManager", the "os_task" is defined by "TaskManager", etc. Each service model represents a real µC/OS-II module.

In order to write this RTOS model, we have used the toolbox showed in Figure 6. It allows the designer to choose a specific service and insert it within the diagram.

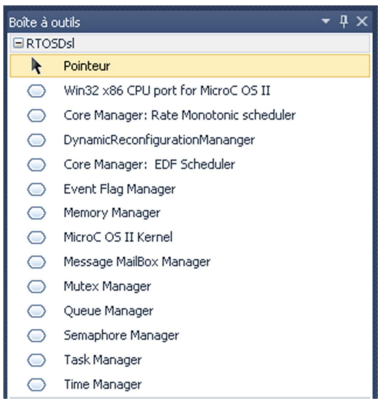

Figure 6. The toolbox of the RTOS DSL environment.

f. Models' transformation

As mentioned previously, the key task of domain-specific languages is to generate code and other artifacts. Historically, artifact generation has typically been spoken as simply as code generation. The analogy is often drawn between the move from assembly code to high-level third generation languages and the move from those languages to domain-specific languages. However, the current reality is a world where software-intensive systems are composed of a much more diverse group of artifacts than ever before, only some of which would traditionally be recognized as source code. Others might be configuration files, either for packaged applications or middleware and others might be the content of databases.

The benefits of using DSLs in these situations are also various. In some cases, raising the level of abstraction is an adequate benefit itself. In others, complex relationships must be maintained, and a graphical DSL brings order and visual understanding to data that would otherwise need more expert interpretation even if the abstraction level is only minimally affected.

Nonetheless, it is still true that software systems are mainly developed in a world where the simple text file dominates. The ability for a set of text files to be managed together in single version control system means that this style of working is unlikely to disappear for some time. Moreover, the rise of Extensible Markup Language (XML) as the de facto standard language for data representation has probably ensured the feasibility of this approach in the near future.

Consequently, the most important transformation that can be applied to DSL consists in producing another artifact as a simple text file, whether that sources code or the persisted representation of a different DSL.

In order to generate the source code of the RTOS model, we use a customizable transformation "template" approach. The key to this technique is that the elements outside of the control markers (**<#** and **#>**) are provided directly to the output file, whereas code within the markers is evaluated and used to add structure and dynamic behavior. Figure 7 describes an example of the template source code. The example shows a small `if and else` branch test depending on the value of the flag "`GEN_FLAG_EN`" which is true when the developer put an "*Event Flag Manager*" service model in the design of the RTOS.

The Transformation Engine reads the "*RTOSModel*" as input, iterates over each "Service" to generate the output source code of the "Service" depending on the template source code and the parameters that are defined by the developer.

```
                                    /* ---------------------- EVENT FLAGS ---------------------- */
<#+
if(GEN_FLAG_EN == false)
{
#>
#define OS_FLAG_EN                  0u   /* Enable (1) or Disable (0) code generation for EVENT FLAGS    */
<#+
}
else
{
#>
#define OS_FLAG_EN                  1u   /* Enable (1) or Disable (0) code generation for EVENT FLAGS    */
<#+
}
#>
```

Figure 7. Template source code used by the Transformation Engine

The generated source code can be used easily by adding the source files generated into the IDE solution as showed in Figure 8.

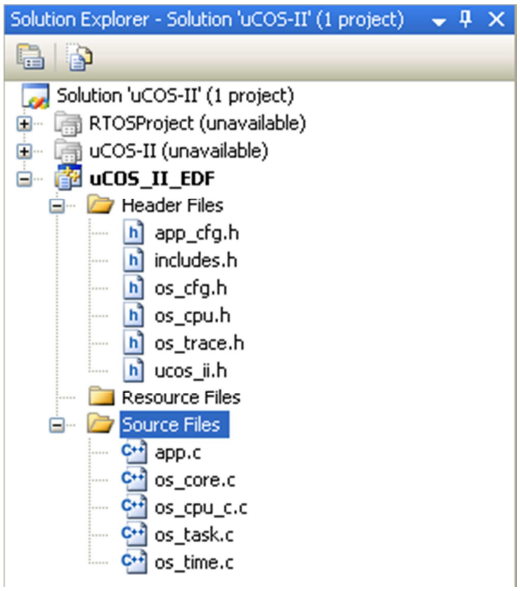

Figure 8. The generated source files have been added to the solution of the developed project

g. The µC/OS-II

As an example, the proposed template is based on the µC/OS source code. µC/OS-II is a preemptive, real-time multi-tasking kernel for microprocessors and microcontrollers. It is implemented in ANSI C and certified by the Federal Aviation Administration for use in software intended to be deployed in avionics equipment. It has been massively used in many embedded and safety critical systems products worldwide.

The main services provided by µC/OS-II are depicted in Figure.9, which describes the module structure of the kernel distribution. The main services are implemented by the Core, the Task, and Port modules.

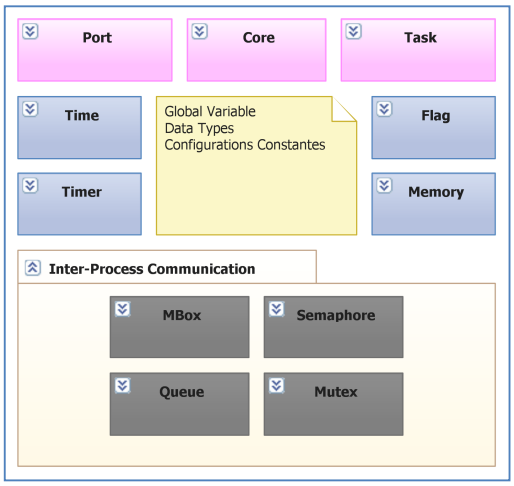

Figure 9. µC/OS II Modules

µC/OS-II is implemented as a monolithic kernel, i.e. it is built from a number of functions that share common *global variables* and *data types* (such as task control block, event control block, etc.).

It is a highly configurable kernel, whose configuration relies on more than 70 parameters. Since the kernel is provided with its source files, configurability is performed via conditional compilation at pre-compilation time, based on `#define` constants. µC/OS-II enables scaling down, the main objective being to reduce the memory footprint of the final executable (up to several Kbytes, depending on the processor). Thus, it is possible to avoid code generation of non-required services, or to configure essential properties of the kernel e.g. the *tick* frequency. The execution time for most of these services is both constant and deterministic, which is a compulsory requirement for real-time systems in order to avoid unpredictable *kernel jitter* [19].

The selection of µC/OS-II is based on two main reasons. First, the modularity of µC/OS-II allows the designer to add or remove services depending on the RTOS model. Second, the source code of µC/OS II has been ported onto multiple embedded platforms (DSPs, microcontrollers, soft cores in FPGAs, etc.). However, our RTOS meta-model, defined in Section IV -d, allows to model others existing operating system, and their automatic configuration, assuming that the existing OS supports OS configuration by `#ifdef` statement.

V. RESULTS

We have tested our prototype to generate a specific µC/OS-II RTOS which is compliant with the x86 architectures. First, we designed a minimal RTOS model containing the following services: Task, Time, and Core management with Rate-Monotonic scheduling policy. Figure 10 shows the minimal RTOS model.

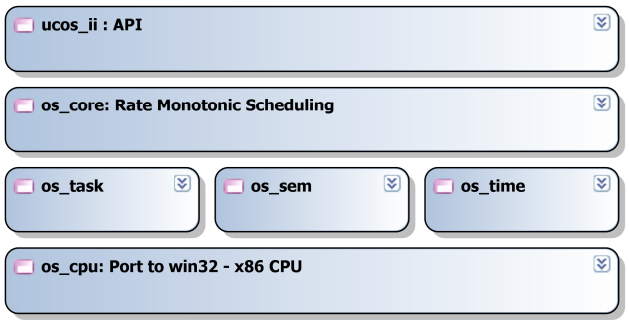

Figure 10 RTOS model with a Rate-Monotonic scheduling policy

Then, we have transformed the model into source code. This operation only requires a few seconds. As can be seen in Figure 10, we have not included all the services of the RTOS. Only the required services have been implemented, leading to a reduction of the memory footprint.

A typical real-time embedded application has been written according to the proposed API and a first test has been led with a rate-monotonic scheduler. The complete code has been implemented on a x86 architecture.

Note that, if the tests' results do not meet specific constraints that are required by the application (for example, deadline constraints, etc.); it is very simple to generate

another version of the OS with other services' attributes until a satisfactory solution is reached.

In a second example, we have implemented the same application but with a different scheduling policy. An Earliest Deadline First (EDF) has been used and a new simulation has been performed. Figure 11 depicts this new model. The model transformation engine modifies the existing source code of µC/OS-II since the algorithm of the scheduler function changed and a new member has been added to the "Task Control Block" `struct` to handle the deadline and the period of a task.

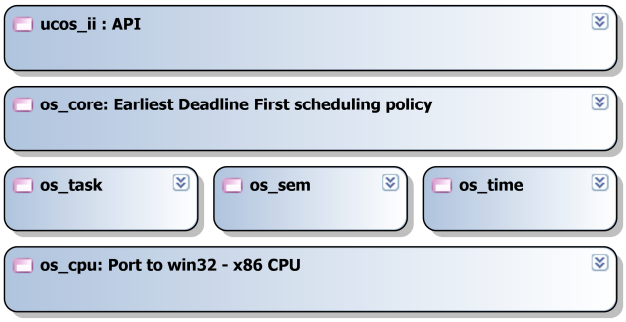

Figure 11 RTOS model with Earliest Deadline First scheduling policy

Flexibility is a particularly important factor to consider in model-driven systems. The useful capability to interchange RTOS services at high-level improves the reusability of new developed services. This facility is even more advantageous when it is employed in large scale industrial applications. This sometimes avoids hundreds of developers to work on hundreds of different but related parts of the models.

Note that compared with the traditional usage of the µC/OS-II by manually configuring its service (with `#ifdef` preprocessing), our method reduces the time of the design space exploration and the effort of the RTOS designer, by providing a high abstraction level and accurate language to automatically configure it.

With traditional approaches, verification of the application and OS for an architecture is a long task, performed by simulation and prototyping. As our methodology is library-oriented, the correctness of the generated OS is strongly dependent on the quality of the library. This approach allows a better structure for the code and makes its verification easier.

We plan now to work on the following improvements:

- RTOS DSL toolkit: Extending it with new service elements and others target RTOS source code transformation;
- Evaluation of the Generated OS Code: It is foreseen to improve its quality in terms of performances, code size, power, etc.
- Generic APIs: We strive to avoid the recode of all applications when the developer decides to change the target OS and architecture.

VI. CONCLUSION

In this paper we have presented a model-driven approach and a domain-specific tool for the development of RTOS-based embedded software. A modeling methodology is applied to build a model of the RTOS's architecture. As a case study, this model has been transformed to a target programming language based on µC/OS-II template source code. Our ambition is to provide assistance to developers of RTOS-based embedded application. The proposed tool [20] contributes to achieve this goal. It can be used to rapidly design a fully customized RTOS that satisfies the requirements and constraints of embedded applications.

VII. REFERENCES

[1] VxWorks from Wind River: http://www.windriver.com/

[2] Windows-CE from Microsoft: http://www.microsoft.com/windowsembedded/en-us/windows-embedded.aspx

[3] QNX Software System: http://www.qnx.com/

[4] Thomas O. Meservy and Kurt D. Fenstermacher, "Transforming Software Development: An MDA Road Map," IEEE Computer, Vol. 38(9) (2005) 52–58

[5] Ed SEIDEWITZ, "What models mean," IEEE Software, 20(5):26–32, 2003. (Cité pages 28 et 128.)

[6] OMG: UML 2.0 Superstructure Specification. Object Management Group, ptc/03-08-02 (August 2003)

[7] OMG: Meta Object Facility (MOF) 2.0 Core Specification. Object Management Group, ptc/03-10-04 (October 2003)

[8] OMG: Model Driven Architecture (MDA). Object Management Group, ormsc/2001-07-01 (July 2001)

[9] Len Fenster, Brook Hamilton, "UML or DSL: Which Bear Is Best? ," The Architecture Journal. Microsoft Corporation, Vol. 23(9) (2010) 32–37

[10] Jozef Hooman Towards, "Formal Support for UML-based Developement of Embedded Systems," In Proceedings PROGRESS 2002 Workshop, STW 2002

[11] João Ventura, Fridtjof Siebert, Andy Walter and James Hunt, "HIDOORS - A High Integrity Distributed Deterministic Java Environment", Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS), San Diego, 7.-9. January 2002.

[12] Khan, M.U.; Geihs, K.; Gutbrodt, F.; Gohner, P.; Trauter, R.; , "Model-driven development of real-time systems with UML 2.0 and C," Model-Based Development of Computer-Based Systems and Model-Based Methodologies for Pervasive and Embedded Software, 2006. MBD/MOMPES 2006. Fourth and Third International Workshop on , vol., no., pp.10 pp.-42, 30-30 March 2006

[13] M. Marcos, E. Estévez, U. Gangoiti, I. Sarachaga and Javier Barandiarán, "UML Modelling of Industrial Distributed Control Systems," Proceedings Sixth Portuguese Conference on Automatic Control, Controlo 2004, Faro, Portugal, June 7-9

[14] Modeling tool from I-Logix:
http://www.omg.org/mda/mda_files/ILogixMdatool.pdf

[15] Ji Chang Maeng, Dongjin Na, Yongsoon Lee, and Minsoo Ryu, "Model-Driven Development of RTOS-Based Embedded Software," A. Levi et al. (Eds.): ISCIS 2006, LNCS 4263, pp. 687–696, 2006. Springer-Verlag Berlin Heidelberg, 2006.

[16] F. Loiret, J. Navas, J.-P. Babau, and O. Lobry, "Component-Based Real-Time operating system for embedded applications," in Component-Based Software Engineering, ser. Lecture Notes in Computer Science, G. Lewis, I. Poernomo, and C. Hofmeister, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 5582, ch. 13, pp. 209-226.

[17] Richard C . Gronback, "Eclipse Modeling Project," Addison-Wesley (2009).

[18] Steve Cook, Gareth Jones, Stuart Kent and Alan Cameron Wills, “Domain-Specific Development with Visual Studio DSL Tools,” Addison-Wesley (2007).

[19] Angelov, C., Berthing, J., “A Jitter-Free Kernel for Hard Real-Time Systems,” In:Wu, Z., Chen, C., Guo, M., Bu, J. (eds.) ICESS 2004. LNCS, vol. 3605, pp. 388–394.Springer, Heidelberg (2005).

[20] RTOS-DSL source code available on: http://code.google.com/p/rtos-dsl