# Towards Real Multi-Criticality Scheduling

Stefan M. Petters
ISEP, IPP
Porto, Portugal
smp@isep.ipp.pt

Martin Lawitzky
TU München
Munich, Germany
ml@tum.de

Ryan Heffernan
Queens University
Kingston, Canada
heffernan.ryan@gmail.com

Kevin Elphinstone
NICTA/UNSW
Sydney, Australia
kelphinstone@nicta.com.au

***Abstract*—Componentised systems, in particular those with fault confinement through address spaces, are currently emerging as a hot topic in embedded systems research. This paper extends the unified rate-based scheduling framework RBED in several dimensions to fit the requirements of such systems: We have removed the requirement that the deadline of a task is equal to its period. The introduction of inter-process communication reflects the need to communicate. Additionally we also discuss server tasks, budget replenishment and the low level details needed to deal with the physical reality of systems. While a number of these issues have been studied in previous work in isolation, we focus on the problems discovered and lessons learned when integrating solutions. We report on our experiences implementing the proposed mechanisms in a commercial grade OKL4 microkernel as well as an application with soft real-time and best-effort tasks on top of it.**

***Keywords*-real-time, temporal isolation, microkernel, components, implementation**

## I. Introduction

The classification of embedded systems into hard real-time, soft real-time and non-real-time systems is being increasingly dissolved by the introduction of real-time aspects into every day devices. There is a trend towards systems which are no longer single purpose devices and extend real-time systems with non-real-time functionality [1]. An example of such a device would be a mobile phone. While not necessarily hard real-time, it certainly possesses soft real-time and best-effort properties. While the term *multi-criticality* has in the literature wider implications [2] we focus on the practical issues of temporal isolation of subsystems and implications in an open system environment.

In general embedded systems require the deadlines imposed by hard real-time applications to be met, the probability of missing a deadline for soft-real-time applications to be managed gracefully, and other applications to be served in an efficient best-effort manner to ensure that fair progress is made by all applications. Two other trends in the embedded systems area are the introduction of partitioning via memory protection and devices which have been developed using component frameworks. The latter has seen a resurgence of microkernel technology in consumer devices [3].

Various dynamic priority based approaches have been proposed. The most dominant is earliest-deadline first (EDF) [4] scheduling, which in general enables a higher utilisation while still meeting all deadlines when compared to fixed priority scheduling [4]. However, this has mostly been confined to academic work, which can be attributed to the facts that EDF deteriorates badly under overload situations and the scheduling overhead caused by more complex queue management. The latter may also decrease the advantage of EDF in terms of the utilisation bound.

The difficulty in managing overload in EDF has been addressed in the work of Brandt et al. [1]. Their rate-based earliest-deadline first (RBED) scheduler manages overload by implementing a form of proportional share scheduling [5]. In doing so it provides temporal isolation between different tasks and enables the seamless integration of best effort, soft and hard real-time tasks. It combines this with the ability to deal with the arrival, departure, and dynamic adjustment of task parameters at run time as well as dynamic slack management [6], making it very versatile. Other models that achieve temporal isolation have been proposed. For example, Shin and Lee [7] have proposed a periodic resource model in a hierarchical scheduling framework. Within the FRESCOR project, a capacity redistribution approach was developed [8]. The choice of RBED as starting point for this work is driven by its integrated solution of dynamic slack management capability.

However, there are still some shortcomings in RBED and the derived work [6], which we aim to address in this work. Fundamentally their work assumes independent tasks, which excludes the majority of communication in a real system and shared resources in particular. Additionally it assumes that all tasks are periodic and have a deadline equal to their period. Given that microkernels are very good matches for componentised systems [9] we will include in our discussion aspects pertaining to microkernels in general and OKL4 [10] in particular without loss of generality.

**Contribution:** Within this paper we extend RBED to solve a number of systems issues. We relax the RBED requirement of deadlines of tasks being equal to their respective periods by introducing a schedulability analysis, as well as discussing the impact of our proposed extensions on the analysis. We detail server tasks implementing critical sections and discuss the impact these mechanisms have on the budget requirements of a task to maintain temporal isolation of different system responses. We add preemption delay caused by the loss of working set of a task when preempted and consider the cost of various mechanisms in RBED. Finally, we discuss the implications of conservative assumptions in the analysis and report on experiences gained when building a system based on the above framework.

**Assumptions:** We assume that all real-time code in the system is respectively described by an estimate of the worst-case execution time (WCET). The term real-time code encompasses all code, which is involved in the guaranteed delivery of some service within a given deadline. We also assume some description of the worst-case inter-arrival of all events which satisfies the requirements of the real-time analysis developed by Albers and Slomka [11], [12]. In particular their analysis allows for bursts of events. It has to be noted that the worst-case description of non-real-time events is required to take the generated interrupt load caused by non-real-time events into account. Finally, we assume that critical sections are implemented as servers and that the overall system has a fine-grained task structure.

**Outline:** In the next section we will summarise the original RBED work. We will use Section III to successively introduce our extension to the work by Brandt et al. This covers in particular the removal of the assumption that deadline equals period, introducing interrupts, the cost of preemption, budget replenishment, server tasks, and finally the impact of budget enforcement. In Section IV we present a case study and lessons learned. After this we will discuss related work, as well as our plans for future work.

## II. RBED Summary

As our approach extends the work by Brandt et al. [1], [6], we will summarise the motivation and fundamental concepts of their approach. Traditional embedded systems were classified into dedicated hard real-time systems, soft real-time or general purpose systems. Today's systems have components from one or more of these domains and many systems are networked in some form and enable the installation of code post deployment. This last property implies that a once and for all schedulability analysis is insufficient for many systems containing real-time parts. Furthermore, such systems need to provide admission control and to be able to isolate system parts of real-time character against each other and those with best-effort character.

A central observation by Brandt et al. was that any system supporting applications of different criticality needs to do so natively instead of retrofitting best-effort scheduling into a RT scheduling framework or vice versa. They developed a scheduling framework based on EDF which provides temporal isolation of tasks and avoids the issue of EDF misbehaviour under overload. It seamlessly supports hard real-time, soft real-time and best-effort tasks, by separating the concepts of resource allocation and scheduling. The preemptive scheduler implements the EDF policy, but limits the time consumed to that allocated by the resource allocator.

The resource allocation step is moved into a separate unit, which provides overall CPU share allocation called a *budget*, and adjustments in the case of new arrival of tasks. If the requested budget for a new arrival task is not available, the allocator first reduces the budget reserved for best-effort tasks either until the requested allocation can be satisfied, or until a minimum budget set for the best-effort tasks is reached. The minimum budget ensures that a system still responds to some degree to non-real-time requests. In the case of budget reduction for the best-effort tasks being insufficient to satisfy the requested budget for the newly arrived task, the budget of soft real-time tasks is scaled. The budget of hard real-time tasks is never adjusted. This simple policy can be adjusted to the needs of a given system.

A major advantage of this is that it enables the choice of using less than the WCET as a budget request for soft real-time applications. This avoids excessive over-allocation of resources without impacting on the performance of hard real-time tasks [1]. Figure 1 provides an overview of the nomenclature used.

| | |
|---|---|
| $U$ | utilisation of the entire task set, $U = \sum_{\forall i} u_i$ |
| $u_i$ | utilisation of a given task $\tau_i$, $u_i = E_i/T_i$ |
| $r_{i,n}$ | release time of a given job $J_{i,n}$, where $n$ is the job identifier |
| $d_{i,n}$ | absolute deadline of a given job $J_{i,n}$ |
| $x_{i,n}$ | at time $t$ current service time $u_i(t - d_{i,n-1})$ of a given job $J_{i,n}$ |
| $C_i$ | WCET of a given task $\tau_i$, it has to be noted that this needs to include the cost of system calls. |
| $E_i$ | budget allocated to task $\tau_i$ |
| $D_i$ | relative deadline of task $\tau_i$ |
| $T_i$ | period/minimal inter-arrival time of task $\tau_i$ |
| $lag(t, x_{i,n})$ | to what degree job $J_{i,n}$ has received its nominal share at time $t$: $lag(t, x_{i,n}) = u_i(t - r_{i,n}) - x_{i,n}$ |

Fig. 1. Nomenclature Used

The original work has made a number of assumptions. All tasks are independent; i.e. there is no blocking communication between tasks and no runnability dependency. A task $\tau_i$ consists of multiple jobs $J_{i,n}$ released a time $r_{i,n}$ and has a minimum inter-arrival time of $T_i$. The releases cannot be overlapping i.e. $r_{i,n+1} \geq r_{i,n} + T_i$. Each job has a deadline $d_{i,n}$ relative to its release time. The deadline is assumed to be equal to the period $T_i = D_i$. The WCET $C_i$ of each task is estimated using well known techniques and is very likely larger than the real execution time required at runtime. The resource allocator provides a budget $E_i$, which is reserved to be used by each job $J_{i,n}$. In the case of hard real-time tasks the budget must equal the WCET $E_i = C_i$ to guarantee completion of the hard real-time task.

In the case of a task exceeding its budget the task is preempted, thus ensuring that the assumption of the schedulability argument holds. The schedulability argument is, under the above assumptions, an overall system utilisation $U \leq 1$. To enforce this, the resource allocator must coordinate and acknowledge all requested changes to the allocation, in particular changes to periods, budgets, or deadlines. The dynamic changes to these system parameters are supported by five theorems; c.f. [1] for corresponding proofs:

*Theorem II-A:* The earliest deadline first (EDF) algorithm will determine a feasible schedule if $U \leq 1$ under the assumption $D_i = T_i$.

*Theorem II-B:* Given a feasible EDF schedule, at any time a task $\tau_i$ may increase its utilisation $u_i$ by an amount up to $1-U$ without causing any task to miss deadlines in the resulting EDF schedule.

*Theorem II-C:* Given a feasible EDF schedule, at any time a task $\tau_i$ may increase its period without causing any task to miss deadlines in the resulting EDF schedule.

*Theorem II-D:* Given a feasible EDF schedule, if at time $t$ task $\tau_i$ decreases its utilisation to $u'_i = u_i - \Delta$ such that $\Delta \leq x_{i,n}/(t - r_{i,n})$, the freed utilisation $\Delta$ is available to other tasks and the schedule remains feasible.

*Theorem II-E:* Given a feasible EDF schedule, if a currently released job $J_{i,n}$ has negative $lag$ at time $t$ (the task is over-allocated), it may shorten its current deadline to at most $x_i/u_i$ and the resulting EDF schedule remains feasible.

The introduction of per job budgets enables easy tracking of available dynamic slack in the system, which may be due to the actual execution time being shorter than the budget allocated. The work by Lin and Brandt [6] provides several policies and respective correctness proofs on how such dynamic slack may be spent. Within our work we make use of two of the policies: the donation of dynamic slack to the earliest deadline task and the borrowing of budget from future jobs of the same task. The schedulability proof condition is maintained under the condition that the task may only use the budget with the deadline of the job it was borrowed from. This can be briefly demonstrated in a simple example. Assume a job $J_{i,n}$ of task $\tau_i$ with period/deadline 10 has used up its budget at time 5. It may borrow budget from job $J_{i,n+1}$, but is now scheduled with deadline of $J_{i,n+1}$ which is 20.

## III. Extensions to RBED

Our extensions aim is to relax some of the assumptions underpinning RBED with a focus on deployment in a commercial microkernel environment.

### *A. Deadlines $\neq$ Period*

Moving from a simple runnability requirement of the original work to more general deadline to period relations requires the replacement of the utilisation bound analysis in the resource allocator with more general schedulability analysis. The schedulability analysis by Albers and Slomka [11], [12] allows deadlines to be different to and in particular shorter than periods. Additionally their algorithm supports bursty behaviour of applications as well as task jitter.

We will briefly outline the analysis and its rationale, but direct the interested reader to the original publications for details of notation and proofs. We discuss the proofs in the context of issues in the online deployment of the analysis algorithm, as well as the modelling of operating systems behaviour in it.

The schedulability-analysis work of Albers and Slomka is based on representing a task as a step function. Starting with a critical instant, each job requires its WCET in processing time by its deadline. In the analysis model, this requested processing time is reflected as a computation request equal to the WCET at its deadline. The release times of all jobs throughout the analysis have to be chosen such that in any interval starting at the critical instant the model describes the worst-case number of jobs that may be released in such an interval. The *processor-demand function* (PDF) is the integration of all computation requests starting with the critical instant.

Figure 2 depicts the PDF of a single task. After an initial burst the task specification prescribes a time of quiescence before a recurrence of the burst. The dotted line in Figure 2 indicates the interval/maximum computation request limit.

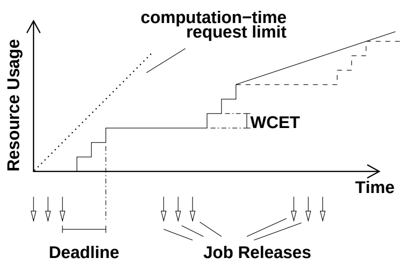

Fig. 2. Sample Processor Demand Functions

The PDFs for all tasks are added to a system-wide PDF. The test has failed if, for any interval starting with the critical instant, the system-wide PDF is larger than the interval, i.e. in an interval we have requested more computation time than is available in the interval.

The *exact* test described above needs to be evaluated at any deadline of any task up to the least-common multiple of inter-arrival times (i.e. a hyper period) from the point where the arrival pattern of all tasks becomes periodic. This is sufficient as all one-off effects are covered. In the literature other lengths for the required analysis intervals are described, however, for illustrative purposes we stay with this definition. For example, for two tasks with 9 and 11 units of respective inter-arrival time, this would mean 19 points in the graph that needs to be tested over a hyperperiod (i.e. least common multiple of periods) of 99 units. To alleviate this problem Albers and Slomka have chosen to approximate the PDF with line segments in such a way that the PDF is never larger than the approximating segments. However, as this is only an approximation, they have opted to keep the *exact* analysis intact for the first $k$ jobs of a task and only afterward approximate with the line segments. A major benefit of the approximation is the reduction of analysis time.

In order to understand the complexity of the analysis algorithm we need to formally discuss the number of tests required.

**Definition** We define the point in the PDF at which a task is changing from being modelled by its exact PDF to being modelled by its approximation as the *approximation-start* of this task.

*Theorem III-A:* The PDF of a system only needs to be checked at the approximation-start at the greatest interval starting with the critical instant, as well as within this interval at any reduction in gradient of the overall PDF. Beyond this interval it needs to be checked that the sum of the gradients of the approximations do not exceed a limit of 1.

*Proof:* Assume a time $t_0$ at which the PDF is just at the computation request limit; i.e. $PDF(t_0) = t_0$. In order to exceed the computation request limit at time $t_1 > t_0$ the gradient of a PDF approximated by a line segment starting at time $t_0$ needs to be greater than 1; i.e. the request created during an interval is greater than the interval itself.

$$PDF(t_1) - PDF(t_0) > t_1 - t_0 \tag{1}$$

Now assume at time $t_2 > t_1$ the PDF again at or below the computation request limit. In order to achieve this, the gradient of the approximation between $t_1$ and $t_2$ needs to be less than 1:

$$PDF(t_2) - PDF(t_1) < t_2 - t_1 \tag{2}$$

Since the gradient of the interval $[t_1, t_2]$ needs is less than one and the interval $[t_0, t_1]$ a test at $t_1$ is forced. In order to avoid *detection* the gradient would need to stay the same or still increase. This case is captured by the end of hyper period test. ■

To reconcile that not all tasks in the system have a known WCET bound we use budgets instead of WCETs for the analysis. Since the budgets are enforced, the fact that the WCET of best-effort tasks may be unknown is immaterial.

The next issue to discuss is the execution time of the analysis algorithm itself. The approximation with segments of slopes drastically reduces the amount of testing required. Albers and Slomka have reported [11] analysis times in the order of 10s to 100s of milliseconds. In our setting of a system with mixed criticality tasks we have the advantage of being able to adjust the settings of our best-effort tasks to match a release frequency of one of the real-time tasks and thus avoid extending the hyper period due to an incompatible period. Also the analysis effort is directly reduced as all best-effort tasks behave like a single task for analysis purposes.

However, even a substantial decrease of the analysis time by one to two orders of magnitudes with the above test definition would still be too large to be ignored. As such we propose to schedule the acceptance test as a separate task running with the best-effort budget.

A frequent problem in real systems is the occurrence of release jitter. This may be caused by partitioned deadlines and variation of execution times between the different partitions or arrival jitter of external triggers. The integration of this by reducing the minimum inter-arrival time of a sporadic task is unnecessarily pessimistic. The integration of jitter into the analysis has been discussed by Kolloch [13] and we will only briefly reiterate the rationale of the integration. Assume a task with a period of 50 units and 10 units of jitter. Any two jobs of the task have a minimal distance of 40 units, but the minimal interval in which 3 tasks may arrive is 90 units. More generally the minimum interval $t(n)$ for the arrival of $n > 1$ trigger events of a task with period $p$ and jitter $j$ can be described by $t(n) = (p - j) + (n - 2) * p$.

The deadlines for best-effort tasks are set to be equal to or longer than the period. There is a trade-off in the choice of longer vs. shorter deadlines. Longer relative deadlines result in a slight relaxation of the scheduling requirements during analysis and a reduction of the number enqueue operations into the ready queue at runtime, while still providing the same general throughput. Shorter deadlines allow for an increased responsiveness of short running tasks. Within this paper, the ready queue is the deadline sorted queue of schedulable tasks; i.e. tasks not blocked.

## *B. I/O and Interrupts*

In order to build systems we need to incorporate I/O. For our framework as presented in this paper, the discussion is restricted to the integration of interrupts into the analysis. Interrupt service routines (ISR) are not subject to EDF scheduling, but are notionally treated to be of higher priority than EDF scheduled tasks. Note that some implementations might defer ISR processing to a user-space task. As this user-space task would be subject to the scheduling scheme we expressly exclude that from our definition of an ISR. Within this discussion we assume no knowledge about the internal prioritisation of interrupts. In a first step we introduce a single interrupt source.

*Theorem III-B:* An interrupt service routine is introduced correctly into the analysis, by assuming the deadline of an ISR to be equal to its WCET.

*Proof:* Assume an interrupt is triggered at $t_1$ and executes for $C_{ISR}$ time. Further assume a task which would (ignoring the ISR) complete at $t_1 + \epsilon$ and has a deadline $d$ in the interval $(t_1 + \epsilon, t_1 + C_{ISR})$. The analysis would detect the violation at the deadline of the ISR at $t_1 + C_{ISR}$ as $PDF(d) = t_1 + \epsilon$ and $PDF(t_1 + C_{ISR}) = t_1 + \epsilon + C_{ISR}$ and thus violating the condition that the $PDF(t) \leq t$. ■

In the next step we need to consider several concurrent interrupts like in the critical instant. In order to reduce the number of tests necessary, we aim to combine these into a single entity.

**Definition** Several interrupt service routines are triggered *quasi simultaneously* when an ISR is triggered during the execution of another ISR and thus the ISRs are serviced back-to-back.

ISRs of such quasi simultaneous interrupts can be considered as a single big ISR without changing the analysis outcome or accuracy.

*Theorem III-C:* Quasi simultaneous interrupts service routines are analysed as a single ISR whose WCET is equal to the sum of the WCETs of all the ISR execution. This also covers multiple executions of the same ISR.

*Proof:* Follows the same lines as Theorem III-B. ■

## C. The Cost of Preemption

Preemption leads to two often ignored side effects: One is a delayed release of tasks due to non-preemptible sections which has an impact on the analysis, the other is cache-related preemption delay and has implications on budget assignment and system primitives.

In a first step we need to consider the impact of non-preemptible sections. Such sections consist of code which disables interrupts and thus prevents the delivery of those. Non-preemptible sections are often found in system calls; it may either span an entire system call in the case of a non-preemtible kernel, or some very brief time span, such as a context switch within the system call in the case of a preemptible kernel.

At this point it is also worthwhile to discuss possible preemption relations. We can make several observations:

1) During the execution of one budget unit a task may only preempt another task once. This is based on the assumption that a task may not block. Thus the only task switches that occur after a task has preempted another task are either other tasks preempting this one or at completion of the task or budget, the latter both leading to the completion of a job's budget.
2) In principle a task may be preempted several times during one job. This becomes obvious when considering a long running job of one task in the analysis.
3) A task may only be preempted by a task with a shorter relative deadline $D_j$ than its own $D_i$ [14].

Assuming a task $\tau_i$ calls several non-preemptible sections, the longest of which executes for $s_i$ units. Thus the longest delay suffered by task $\tau_j$ caused by non-preemptible sections indicated by $s^j$, is the maximum $s_k$ of all tasks with longer deadlines than $\tau_j$.

$$s^j = \max\left(\forall_{k:D_k > D_i} s_k\right) \tag{3}$$

This delay needs to be considered in the analysis at the deadline $d_{j,n}$ of task $\tau_j$ in excess of the actual budget $E_j$. However, now the budget associated with $s^j$ is double allocated, once for $\tau_i$ and once for $\tau_j$. This can be remedied by reducing the computation request in the analysis for $\tau_i$ by $s_i$, as the code may only be executed once. It has to be stressed that this only applies during the analysis. Similar to interrupts, the impact of the delayed triggering of $\tau_j$, needs to be considered by shortening the relative deadline $D_j$ accordingly. This needs to be in effect during analysis, as well as in the running system.

Besides the delay in terms of delayed triggering, the preempted task also suffers further delay. During execution of a task, the task creates a working set in the processor state. Depending on the hardware architecture, this might be as small as a few levels of pipeline content, or more substantial cache [15] and TLB entries. In case of a preemption this working set will be degraded or entirely destroyed and has to be rebuilt once the task continues executing. This reestablishment of the working set requires extra time to be spent. The approach of dealing with this extra time caused by a preempting task is developed in the previous work [16]. It operates by assigning a preempting task extra budget to account for the additional cost incurred by the preempted task. This extra budget is passed on to the preempted task at the time of preemption. This avoids the overhead of assigning the preemption overhead multiple times [14].

The same procedure can also be performed on interrupt service routines in the sense of adding the preemption cost to the WCETs in the analysis and having a mechanism to add budget to the preempted task accordingly. It has to be noted that for this operation we assume that the preemption cost $w_j$ is defined by the preempting task or ISR in terms of cache footprint and damage it can cause to the cache content of the preempted task, thus allowing interrupts which have a small footprint to be accounted for, without causing excessive overhead.

## D. Replenishment and Sporadic Tasks

In case of the replenishment outside the *normal* periodic allocation, two algorithms by Lin and Brandt [5] are used: slack donation and borrowing of budgets from future jobs of the task. This is driven by the notion that soft real-time tasks may be deliberately allocated budgets which are less than their WCET, in order to increase the utilisation at the expense of the occasional missed deadline. However, the introduction of the more formal schedulability analysis highlights issues which were not so evident in the original work. We will briefly introduce the two of the concepts which were used by Lin and Brandt [6] before discussion of the issues and our solution.

During normal operation the budgets are replenished at release time. In the case of unused budget, this dynamic slack is handed to the next task with a later deadline than the deadline of the current budget. A simple example of this is shown in Figure 3 a).

The deadline of the task *donating* the slack is noted alongside the slack amount. A task with a shorter deadline than the remaining slack cannot use this slack using its own deadline without violating the terms of the schedulability analysis. However, once the task entirely consumed its budget, it can use the slack with the deadline associated with the slack. Whenever the idle task is running the budget is *consumed* by the idle task.

*Theorem III-D:* Slack $l_i$ with associated deadline $d_{i,n}$ is preserved across an idle-task time $\delta_{idle}$ at a rate $\max\left(l_i - \delta_{idle}, 0\right)$.

*Proof:* The idle task can be considered as being part of the last task running. As such the idle task consumes budget for the time it is running. The new arrival of a task preempts the idle task and frees up the remaining slack. ■

In the case of an overrun of a task the budget is enforced and the task may receive the budget of a future release of the task with a respective deadline of that future budget, as shown in Figure 3 b). In this case there is obviously no donation of remaining budget once the task has completed, but instead the remaining budget needs to be preserved for the future release of the task.

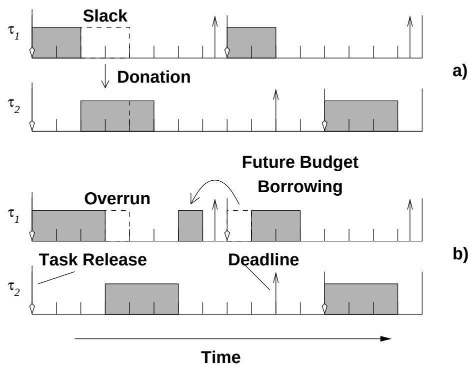

Fig. 3. Slack donation and budget borrowing

Obviously this future release now has a budget which is less than its *normal* budget. In the original work this shortfall may be covered by slack donations of other tasks and a potential variation in the actual execution time of a task.

*Theorem III-E:* The concept of slack donation and budget borrowing are valid with respect to the schedulability analysis introduced in Section III-A.

*Proof:* As described at the end of Section II, the budget borrowed is executed with the deadline associated with the expected job release corresponding to the budget and as such is consistent with the performed analysis. Looking at slack donation, existing budget which has been covered by the analysis is executed with the same or a relaxed deadline. Thus the schedulability criterion is either consistent with respect to the performed analysis or relaxed. The analysis has the assumption of non-blocking tasks. As such the idle task needs to consume slack, assuming the position of a virtual task without own budget. ■

The introduction of the schedulability analysis highlights an issue which was not discussed in the original work. The presence of sporadic tasks and conservative assumptions cause a dynamic slack (unmanaged slack) that is not covered by the introduced algorithms. This is caused by a reservation which is based on minimum inter-arrival intervals of sporadic tasks and a replenishment at job release with the same amount of budget even if a task is released later than the minimum inter-arrival interval.

Firstly we need to look at the effect of this unmanaged slack. To illustrate the case, we assume that hard and soft real-time tasks work within their assigned budgets and the above mentioned mechanisms; i.e. a task might borrow from a future, but this will usually be sufficient to complete a job. Added to this scenario we assume that jobs of best-effort tasks usually span several budget periods. Due to the unmanaged slack, the best-effort tasks are free to borrow from their future budgets within the previously laid out rules. In doing so the relative fractions in terms of budget of best-effort tasks are preserved in the case of continuous execution of these best-effort tasks.

This behaviour causes problems if a new task arrives and the real-time analysis algorithm is performed inside the resource allocator. The analysis algorithm would need to consider the backlog. This research has developed a different strategy. In case of the arrival of new task, all best-effort tasks are assigned zero budgets with immediate effect. The resource allocator uses budgets and deadlines of the best-effort tasks to maintain correctness of the schedule and thus assumes the role of the idle task. When it is scheduled the analysis can assume without loss of correctness that no backlog of work exists and thus it can assume the state of the critical instant. After recalculating all budgets, the resource allocator uses the priority and budget of the new task and reassigns the budgets of the best-effort tasks. As such the execution of the new task is delayed until the analysis and reassigning of budgets is complete. Alternatively the resource allocator could work off the budget of a best-effort task, with similar effects.

## E. Deadline and Budget Inheritance

Particularly in microkernel-based systems, device drivers and system services are implemented using server tasks. Besides fault containment, server tasks have the advantage to serialise access to a resource. Consequently server calls will be located more frequently in the task body rather than the start or the end of a task in a microkernel-based system.

The first consideration is how to represent the time spent executing the server in the analysis. When using dedicated budgets and deadlines for the server we need to look at responsiveness and correctness. In order to achieve responsiveness and correctness the server will need a short deadline compared to its minimum inter-arrival time. As a result the server will put unnecessary constraints on the system for its schedulability analysis. There is also the issue of the client being partitioned into a part before and a part after the server call. Overall the use of dedicated budget in combination with the more frequent calls to server tasks makes system construction and analysis more cumbersome.

The other problem with server tasks is the contention in the access of the server and the associated dynamic priority inversion [17]. Priority inversion is caused when a high priority task is blocked on a server working for a low priority task, which in turn is preempted by a medium priority task. The previously mentioned borrowing of future budgets ensures that a task is never out of budget, but may have a very long deadline. The priority inversion problem can be solved in several ways.

1) A server with its own deadline and budget would take care of this, however, as mentioned earlier this would potentially violate assumptions about the non-blocking of tasks in the analysis in Section III-A. This violation can be avoided by partitioning tasks into separate units with their own budget and deadlines.
2) Providing a rollback and restart of the server task for the preempted client would require substantial spare budget for restarts, which would scale with the number of possible preempting threads trying to access the server.
3) A multithreaded server would be the preferred solution, but this in turn requires substantial implementation by the server writer. Additionally, this only reduces the

length of the critical section implemented by the server, which does not fully solve the problem, unless the server is not stateful (e.g. implements non blocking data structures). This argument is based on the observation that a server usually embodies a critical section and a set of common operations preceding or succeeding it.

4) Deadline inheritance avoids the dynamic priority inversion problem. This raises the question where the budget for this operation would come from. Running on a budget with a longer deadline would obviously violate the assumption made in the schedulability analysis. The alternative is to combine deadline inheritance with budget inheritance. This implies that a client task using the server needs to have extra budget to execute the server on behalf of the client task which is already being served by the server task.

We have chosen the last option for our framework, as we considered it preferable to hide the solution of this problem from the application programmer. Similar to work by Lamastra et al. [18] or Wang et al. [19] the budget associated with the deadline needs to be inherited alongside the deadline.

A special case to be considered is when the inherited budget is insufficient to complete the service. The budget borrowing mechanism described in the previous section will ensure that there is always some budget available, although potentially with an unfavourable deadline. Assume several tasks have queued to enter the server, than the borrowed budget may no longer be the earliest deadline waiting to be served. In order to ensure temporal correctness, the task with the borrowed budget needs to be removed from the head of the server send queue and re-enqueued at the appropriate place, according to its new deadline and the server inherits budget and deadline from the new head of the send queue.

### F. Budget Enforcement

Finally we discuss the implications of budget enforcement. The budget enforcement is implemented using a timer interrupt. The ISR is used for adjusting scheduling parameters (i.e. borrowing of budget from future jobs), reinserting the task in the ready queue and send queues if appropriate and reschedule. There are two ways to delay interrupts:

1) Other interrupts which are handled prior to this interrupt
2) Non-preemptible sections will disable all interrupts (cf. Section III-C)

The delay caused by other interrupts is already covered in the schedulability analysis. In order to maintain correctness of the analysis, the budget enforcement needs to be triggered in a way such that all operations of budget enforcement are completed by the time the budget expires. As such the budget $E_i$ needs to be chosen so it contains the cost of budget enforcement $f$ and the non-preemptible section $s_i$ beyond the intended actual execution time and the previously mentioned preemption cost $w_i$. Note, that the preemption cost has already been passed on to the preempted task when the actual timer is programmed.

Besides this direct impact, there is also an indirect cost. The budget enforcement itself is a non-preemptible section and has to be considered in the analysis like any other non-preemptible section as explained in section Section III-C. However, opposed to other non-preemptible sections, this cost may only be incurred once per job.

## IV. Case Study

The case study served throughout the development of the approach as motivator for most of the solutions presented. Within the paper we use it to discuss the lessons learned during its development.

### A. Implementation

This work implemented the aforementioned algorithms in and on top of an OKL4 [10] kernel version 1.5.2 with an XScale PXA255-based Gumstix [20] hardware board. The budget accounting, enforcement, and inheritance, as well as the scheduler were implemented in the kernel. The resource allocator was realised in the root task, which is the first task launched in the system and initially holds ultimate control over access rights throughout the system. Thus it comes naturally to use the root task to keep track of the tasks in the system and their reserved resources. It provides the kernel with the deadlines, budgets and minimum inter-arrival times of all tasks which then stores this information in the task control block.

The scheduler in itself is a pure EDF scheduler. The enforcement of budgets is realised by a timer, which is set whenever a new task is scheduled with a new budget and deadline. The timer will be used to preempt the task if it runs out of budget. The new deadline may be necessary when either a task is newly released, a task is completed, or when a task changes its budget and deadline due to exhaustion of its current budget. Since deadlines and budgets may be passed on via messages in budget inheritance, not every release or completion forces a reprogramming of the budget watchdog.

The ready queue is deadline sorted, both in shortest deadline first order. The same applies to the send queues of tasks waiting on a server task. The priority and budget inheritance for servers is achieved by using the deadline and budget of the head of the send queue. This property is transitive in the sense that nested servers are equally affected by the deadline inheritance caused by the send queues of any outer nesting level servers.

In the proof of concept implementation we have removed the hand optimised fast path implementation of the IPC (inter process communication) primitive and have forced a reschedule after each IPC. *Call* IPCs are used by the kernel to identify servers.

Due to the non-dynamic nature of our application we have not implemented a complex schedulability analysis. This was driven by the realisation that many of the required numbers, like the WCET of ISRs or the preemption delay were not available.

### B. Test Application

We have developed a small, but non-trivial test application to identify issues with the proposed method and to demonstrate that the method may be deployed in a realistic scenario. The source code using a stock OKL4 kernel can be found at [21].

The system, shown in Figure 4, emulates an instant messaging device, allowing for two-way voice and text communication across a network. Audio transmission was chosen because it provides periodic deadlines which must be met; otherwise audible glitches can be heard at run time. Text transmission and receiving was added to add sporadic tasks to the system. In our model the text messaging has best-effort scheduling character while the audio transmission is soft real time.

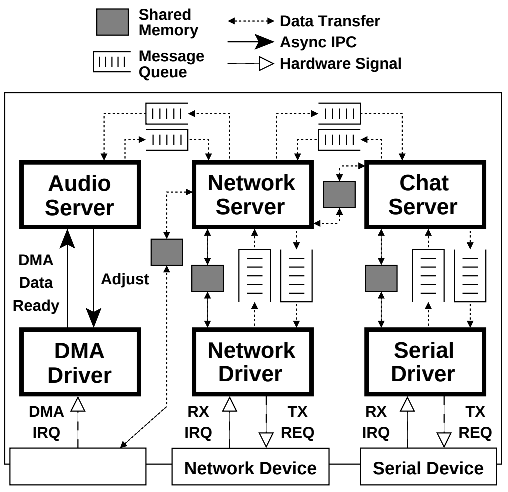

Fig. 4. Test Application

Hardware drivers in this system are implemented as user-level threads. The audio driver has been omitted from Figure 4 because DMA is used to perform the data transfers to and from the audio device. A DMA driver was developed for future scalability purposes, which allows multiple clients to initialise multiple DMA channels and also keeps track of DMA interrupts. It should be noted that the impact of DMA on the WCET is outside the scope of this paper [22].

Communication between threads is generally handled by a combination of shared memory and message queues. The shared memory is used to hold data payloads, while message queues hold pointers to the payloads in the shared memory. Every write to a message queue also implicitly involves an asynchronous IPC to the receiving thread to notify it of the valid data in the queue.

All tasks depicted have their own budget and deadline pair. The reason for this is that due to the low level nature of the case study. Most tasks work in two directions: For example, the network driver receives network packets and sends network packets. When network packets are received the scheduler requires knowledge about the receiver thread to schedule the network thread using the *right* budget and deadline.

In addition to the threads depicted in Figure 4 a number of servers in the system are active: The root task performs the resource allocator role for the scheduler and a core device server manages hardware devices. Both services are only used in the initialisation phase after booting. A naming server is responsible for resolving named object references in the flat name space and an event server is used to register notification callbacks for event notifications. The notifications are asynchronous messages whereas the registrations are synchronous IPCs. These two tasks are true servers implementing the deadline inheritance protocol. For evaluation purposes we also added a task that obtains and transmits information about the scheduling behaviour which is also not shown in Figure 4. This monitoring task makes use of a virtual timer server and the network server.

## C. Discussion

As efficient IPC system calls are the backbone of any microkernel, we have paid special attention to these. Initially we had also implemented the concept of passing deadlines and budgets not only in calls to servers, implementing budget inheritance, but also for synchronous (blocking) IPC. However, the latter turned out not to be used in the test application we built and hence was not discussed in depth in this paper, but appears to be useful for streaming applications or larger componentised systems [23]. When deadlines and budgets are passed alongside synchronous IPC, it becomes apparent that under these conditions the enqueue operation on the send queue is always at the head of the queue, making the enqueue operation $O(1)$ and negligibly small. In fact smaller than the equivalent fixed-priority implementation, showcasing another benefit of deadline passing.

However, asynchronous communication which must be used for any IPC with a deadline potentially longer than the deadline of the current task, may trigger a task and thus may force an arbitrary enqueue in the ready list. The current implementation of the ready list as linked list is thus of $O(N)$ with $N$ being the number of ready tasks. Figure 5 depicts measured data on the enqueue operation where the x-axis indicates the position a given task is enqueued to, with 0 being the head of the queue. The enqueue operation in the ready queues of a fixed-priority scheduler are roughly equivalent to our enqueue operation with enqueuing in the 1st position after the queue head.

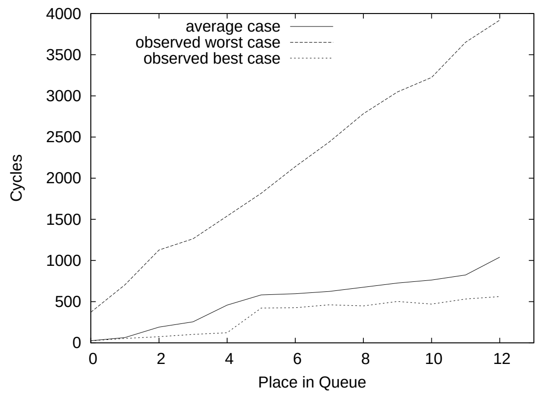

Fig. 5. Enqueue Cost Measured

The dequeue operation on the ready queue, once a task is blocked on a receive IPC is again $O(1)$. With 27 cycles we measured on the GUMSTIX the dequeue was faster than the fixed-priority implementation which reaches complexity up to the same level as the enqueue operation. On the given hardware platform the resetting of a timer for budget monitoring was only a matter of a few processor cycles. In order to speed up the enqueue operation, the ready queue may be implemented as priority heap making enqueue and dequeue $O(\log(N))$. In a system which makes heavy use of server tasks this might not be the preferable solution.

The structure of our application forcing two way communication and subsequently deadline partitioning and asynchronous IPC, made the task structure less elegant than the equivalent *straight forward* implementation using synchronous IPC.

As indicated, the current implementation leaves room for optimisation. In particular direct process switch and lazy dequeuing should lead to reduction of overheads. As opposed to the case of fixed-priority scheduling [24], the EDF scheduling policy enables a much easier reasoning about which task of two communicating parties should run next.

Lazy dequeuing describes the effect of avoiding dequeue operations of tasks which are likely to be enqueued again shortly after dequeuing. Again as opposed to fixed priority scheduling [24] the EDF-sorted ready queue provides scope to optimise without losing predictability. As defined, a task may block either when it has called a server or when it has completed the job. In the latter case, the task should be dequeued. In the former case we need to discern two distinct cases. If a task is blocked on a server and the server executes on behalf of that client task, the client task may stay enqueued in the ready queue, with the server task forming the new head of the queue. The reasoning is that the task will be ready once the server has returned and the server will be completed before the task is scheduled again. The second case is the blocking and deadline inheritance. In this case the scheduler can insert the task behind the server task it is blocked on, following the same reasoning as before.

From an application implementers point of view we experienced few problems, beyond the requirement of sticking to the rules of using non-blocking IPC for all communication bar server calls. The provision of WCET and inter-arrival estimates at task boot strapping was reasonably straight forward. However, in the case of larger systems when the dependencies caused by the asynchronous triggering of tasks turn out to be non-trivial, substantial knowledge of the application system structure is required by the resource manager. This is a substantial drawback of the approach, as this required system knowledge is directly opposed to the dynamic addition or removal of tasks in the system.

## V. Related work

A large body of work exists in the area of this paper. We aim to discuss the most relevant and representative subset within this section. The notion of deadlines transported by messages through a system has been developed by Kolloch [13]. He implemented the approach in RTEMS which is a small real-time executive without memory protection. While implementing deadline inheritance, he was not working with budgets, but assumed instead all WCET estimates are conservative. The target application domain is systems specified in SDL, whose state transition *tasks* are computationally light. This has two implications: Firstly, server task blocking is much more light weight and thus not overly affected by a somewhat conservative consideration in the schedulability analysis. Secondly, his algorithm is dependent on message based deadline transportation, as individual ”tasks” are very small, but a single input may trigger multiple state transitions.

Jansen et al. [25] have also worked on a EDF scheduling solution providing deadline inheritance. They presented a schedulability analysis for their algorithm, which has some similarities with the test used in our paper. However, in heavily loaded system with tasks having a long hyperperiod their analysis will take substantial resources. Also the execution times are not enforced and thus uncontrollable behaviour may result in the case of a best effort or a soft real-time task misbehaving.

The concept of budget inheritance during priority inheritance has been investigated by Lamastra et al. [18]. The aim of their work was similar to ours to provide support for real-time tasks of different criticality and best-effort tasks in a system making use of servers. However, their work assumes no knowledge of inter-arrival times of the soft real-time components and thus the dynamic slack of a task may not be donated to another task. This has been extended to a bandwidth-exchange server by Wang et al. [19] which returns inherited budget at a later point in time. Our work allows the change of parameters at runtime and enables deadlines to be different compared to the period of tasks.

A similar approach to ours has been presented by Steinberg et al. [23] utilising the L4 Fiasco kernel. Their approach works within a fixed priority framework and does not discuss the system level schedulability analysis impact of their solution.

A different angle in terms of isolation is taken by Nogueira and Pinho [26]. In order to satisfy demands for resources made by hard-real-time tasks they steal budgets from lower priority tasks. In their work the stealing is essential as little a-priori knowledge about execution-time requirements is assumed.

Resource sharing in a rate-based environment has also been investigated by Liu and Goddard [27]. Instead of deadline inheritance they have adopted a deadline-ceiling protocol. Similar to Brandt et al. they have implemented the approach inside the Linux kernel. While it supports servers it does not account for other communication.

Ju et al. [14] have developed an approach to consider cache-related preemption delay in dynamic priority systems. Their approach inflates the WCET estimates of a task by the potential cost of any possible preemption scenario. Ultimately this leads to a multiple accounting of the delay as one task may potentially preempt different tasks. We have avoided this with a budget passing from preempting to preempted task.

## VI. Conclusions and Future Work

Within this paper we have presented an integrated scheduling approach detailing and resolving many issues which have been abstracted in previous work, but are crucial to building real systems. We have taken a particular view in enabling componentised systems with strong fault isolation guarantees on top of a microkernel. The issues addressed in this paper are augmentation of a scheduling approach with schedulability analysis, the impact of system services on the budget planning and server tasks. The approach dealing with preemption delay avoids accounting for this delay more than once per preemption. Additionally we have implemented a non-trivial application on top the presented approach to explore real-world system integration issues.

While the work presented in this paper covers a large range of issues, there are still issues which may be addressed. Firstly the previously mentioned optimisation of the in kernel mechanisms like direct process switch, lazy dequeuing and IPC fastpath implementation to name but a few, complemented by a detailed comparison with a fixed-priority-based version is an obvious avenue for future work.

We also have started to integrate RBED scheduling and power management [16]. The integration of the solutions presented in both papers and validation of this integrated approach will further improve the real-world appeal of the presented scheduling framework. Finally the provided bandwidth isolation mechanisms and resource reservation may open up the option of exploiting these in a multiprocessor setting.

## VII. Acknowledgements

We would like to thank Scott Brandt, Suresh Iyer and Jaeheon Yi for allowing us access to their RBED implementation and documentation which we used to guide our own implementation. Substantial part of the work presented in this paper was performed while the authors worked at NICTA. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. This work was also partially supported by FCT (CISTER Research Unit - FCT UI 608).

## References

[1] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes," in *Proceedings of the 24th IEEE Real-Time Systems Symposium*, Cancun, Mexico, Dec. 2003.

[2] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications," in *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, Prague, Czech Republic, Jul. 2008.

[3] G. Heiser, "Hypervisors for consumer electronics," in *Proceedings of the 6th IEEE Consumer Communications and Networking Conference*, Las Vegas, NV, USA, Jan. 2009.

[4] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[5] L. Abeni, G. Lipari, and G. Buttazzo, "Constant bandwidth vs. proportional share resource allocation," in *Proceedings of the 5th IEEE International Conference on Multimedia Computing and Systems*, vol. 2. Florence, Italy: IEEE Computer Society Press, 1999, pp. 107–111.

[6] C. Lin and S. A. Brandt, "Improving soft real-time performance through better slack management," in *Proceedings of the 26th IEEE Real-Time Systems Symposium*, Miami, FL, USA, Dec. 2005.

[7] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proceedings of the 24th IEEE Real-Time Systems Symposium*, Cancun, Mexico, Dec. 3–5 2003.

[8] A. Zabos, R. I. Davis, and A. Burns, "Utilization based spare capacity distribution," University of York, Department of Computer Science, York, YO10 5DD, United Kingdom, Technical Report YCS427 (2008), 2008.

[9] I. Kuz and Y. Liu, "Extending the capabilities of component models for embedded systems," in *Proceedings of the Third International Conference on the Quality of Software-Architectures (QoSA)*, Boston, MA, USA, Jul. 2007.

[10] Open Kernel Labs, "OKL4 community site," http://okl4.org.

[11] K. Albers and F. Slomka, "An event stream driven approximation for the analysis of real-time systems," in *Proceedings of the 16th Euromicro Conference on Real-Time Systems*. Catania, Italy: IEEE Computer Society Press, 2004.

[12] ——, "Efficient feasibility analysis for real-time systems with EDF scheduling," in *Proceedings of the 8th Conference on Design Automation and Test in Europe*, Munich, Germany, 2005.

[13] T. Kolloch, "Scheduling with message deadlines for hard real-time SDL systems," Dissertation, Institute for Real-Time Computer Systems, Technical University Munich, Germany, 2002.

[14] L. Ju, S. Chakraborty, and A. Roychoudhury, "Accounting for cache-related preemption delay in dynamic priority schedulability analysis," in *Proceedings of the 10th Conference on Design Automation and Test in Europe*, Nice, France, Apr. 2007.

[15] H. S. Negi, T. Mitra, and A. Roychoudhury, "Accurate estimation of cacherelated preemption delay," in *Proceedings of the 1st International Conference on Hardware/Software Codesign and System Synthesis*, Newport Beach, USA, Oct. 2003.

[16] M. P. Lawitzky, D. C. Snowdon, and S. M. Petters, "Integrating real time and power management in a real system," in *Proceedings of the 4th Workshop on Operating System Platforms for Embedded Real-Time Applications*, Prague, Czech Republic, Jul. 2008.

[17] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.

[18] G. Lamastra, G. Lipari, and L. Abeni, "A bandwidth inheritance algorithm for real-time task synchronisation in open systems," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium*. London, UK: IEEE Computer Society Press, Dec. 2001, pp. 151–160.

[19] S. Wang, K.-J. Lin, and S. Peng, "BWE: a resource sharing protocol for multimedia systems with bandwidth reservation," in *Proceedings of the 4th IEEE International Symposium on Multimedia Software Engineering*, Newport Beach, CA, USA, Dec. 11–13 2002.

[20] Gumstix, *Gumstix Website*, http://www.gumstix.com.

[21] NICTA, *VOIPDemo*, http://www.ertos.nicta.com.au/software/demonstrator/.

[22] T.-Y. Huang, C.-C. Chou, and P.-Y. Chen, "Bounding the execution times of DMA I/O tasks on hard-real-time embedded systems," in *Proceedings of the 9th IEEE Conference on Embedded and Real-Time Computing and Applications*. Tainan, Taiwan: Springer–Verlag, Feb. 2003, pp. 499–512.

[23] U. Steinberg, J. Wolter, and H. Härtig, "Fast component interaction in real-time systems," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems*. Palma, Spain: IEEE Computer Society Press, 2005.

[24] K. Elphinstone, D. Greenaway, and S. Ruocco, "Lazy scheduling and direct process switch — merit or myths?" in *Proceedings of the 3rd Workshop on Operating System Platforms for Embedded Real-Time Applications*, Pisa, Italy, Jul. 2007.

[25] P. G. Jansen, S. J. Mullender, P. J. Havinga, and H. Scholten, "Lightweight EDF scheduling with deadline inheritance," University of Twente, Centre for Telematics and Information Technology, Enschede, Netherlands, Technical Report TR-CTIT-03-23, 2003.

[26] L. M. Nogiera and L. M. Pinho, "Precedence constraints with capacity sharing and stealing," in *Proceedings of the 22nd IEEE Parallel and Distributed Processing Symposium*, Miami, USA, Apr. 2008.

[27] X. Liu and S. Goddard, "Resource sharing in an enhanced rate-based execution model," in *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, Porto, Portugal, Jul. 2003, pp. 131–140.