Fitting an EDF based Scheduling Approach to Componentised Real(-Time) Systems

Stefan M. Petters[‡§] Martin Lawitzky[‡§¶] Ryan Heffernan[‡] Kevin Elphinstone[‡§]
[‡] NICTA* [§] University of New South Wales [¶] TU München
Sydney, Australia Sydney, Australia Munich, Germany
firstname.lastname@nicta.com.au

Abstract

Componentised systems, in particular those with fault confinement through address spaces, are currently emerging as a hot topic in systems research. This paper extends the unified rate-based scheduling framework RBED in several dimensions to fit the requirements of such systems. First, we have removed the requirement of the deadline of a task being equal to its period. Second, we introduce inter-process communication and end-to-end deadlines, reflecting the need to communicate and avoid fragmentation of the system through deadline partitioning. Additionally we also discuss server tasks, general I/O management, budget replenishment and low level details to deal with the physical reality of real systems work.

1 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 and the extension of real-time systems with non real-time functionality. Such 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 non real-time applications to be served in an efficient, best effort manner to ensure 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.

*NICTA is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs

While many scheduling variants have been proposed, the three most prominent areas have been time triggered, fixed priority based and deadline based scheduling. Time triggered scheduling is mostly deployed in safety critical applications. The reasons for this are focused mostly on ease of analysis and avoidance of preemption in the system. It has also been deployed to some degree in the automotive sector. OSEKTime [1] is a time triggered operating system, which provided the possiblity in one of its time slices to enable interrupts, to deal with interrupt driven hardware. However, anecdotal evidence has shown that this restriction on possible interrupts has been avoided for the sake of responsiveness. Fixed-priority based scheduling has a long history and is widely used in commercial and academic real-time operating systems. The main drawback is that a fixed-priority scheduled system may start missing deadlines with a lower utilisation than those scheduled under some deadline based scheduling approaches.

Various dynamic priority based approaches have been proposed. The most dominant is earliest-deadline first (EDF) [2] scheduling, which in general enables a higher utilisation while still meeting all deadlines. However, this has mostly been confined to academic work, which can be attributed to two reasons. Firstly, EDF deteriorates badly under overload situations and secondly, if multiple tasks are involved the partitioning of deadlines leads to pessimism in the analysis which ultimately raises the question whether it is really more efficient in terms of permissible load compared to fixed-priority scheduling. Additionally, the integration of applications of different criticality is non trivial.

The difficulty in managing overload in EDF has been addressed in the work of Brandt et al. [3]. Their rate-based earliest-deadline first (RBED) scheduler obtains this behaviour by implementing a form of proportional share scheduling by Albeni et al. [4]. In doing so it

realises temporal isolation between different tasks and enables the seamless integration of best effort, soft real-time and hard real-time tasks. It combines this with the ability to deal with the arrival, departure, and dynamic adjustment of parameters of tasks at runtime as well as dynamic slack management [5], making it very versatile.

However, there are still some shortcomings in RBED and the derived work [5], which have so far not been addressed. Fundamentally their work assumes independent tasks, which excludes the majority of communication in a real system, in particular shared resources and I/O. Additionally it assumes that all tasks are periodic and have a deadline equal to their period. We consider it essential to provide an integrated solution that supports multiple tasks participating in a system response. Within a componentised systems these shortcomings are particularly critical.

Contribution: Within this paper we extend RBED to solve the a number of systems issues. We detail the handling of I/O and server tasks implementing critical sections, as well as discussing the impact of these mechanisms have on the budget requirements of a task to maintain temporal isolation of different system responses. We also introduce a form of end-to-end deadlines transported via IPC to bring an implementation of a system closer to the deadlines imposed by the system. We have integrated the scheduler into an L4 microkernel and built an application system on top of our scheduler to demonstrate the real-world capabilities of our work.

Assumptions: We assume that tasks are described by an estimate of the worst-case execution time (WCET) of all code in the system, which involves a guaranteed delivery of some service within a given deadline. We also assume that there is some description of the worst-case interarrival of all events which satisfies the requirements of the real-time analysis developed by Albers and Slomka [6, 7]. In particular their analysis allows for bursts of events. 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. Relative deadlines may be chosen arbitrarily; i.e. they may be chosen longer or shorter than the period. The most significant assumption is that the cost of preemption in terms of (for example) reloading caches after the preemption is neglibile. However, this is not a fundamentally unsolvable problem, but rather avoids the presentation being too cluttered with unneccesary explanation. Finally, we assume a microkernel-based system which lends itself naturally to build componentised systems. The impact of this assumption is largely confined to the fact that critical sections are implemented as servers and that a system is assumed to have a fine-grained task structure.

Outline: In the next section we will briefly introduce the original RBED work. After this we will use Section 3 to successively introduce our extension to the work by Brandt et al. This covers in particular the removal of the assumption that dealine equals period, the addition of a message transported deadline model, budget replenishment options, sever tasks, and finally integration of I/O. In Section 4 we discuss our implementation of the proposed work, a case study showcasing the applicability of the scheduling framework and lessons learned. After this we will give an overview of related work before presenting our plans for future work.

2 RBED Summary

As our approach extends the work by Brandt et al. [3,5], we will briefly introduce the motivation and fundamental concepts of their approch. 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. At the very least, this last property implies that many systems containing real-time parts allow not a once and for all analysis of the schedulability of the system, but need to be able to isolate system parts of higher criticality against those with lower criticality. Additionally the system requires admission control.

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 multi-class 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 ensures that only time allocated by the resource allocator is used.

The resource allocation step is moved into a separate unit, which provides overall CPU share allocation called *budget* and adjustment 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. Obviously 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. The interested reader is directed to the original work [3]. Figure 1 provides an overview of the Nomenclature used.

U	utilisation of the entire taskset, $U = \sum_{\forall i} u_i$
u_i	utilisation of a given task τ_i, $u_i = E_i/T_i$
$r_{i,n}$	release time of a given job $J_{i,n}$
$d_{i,n}$	absolute deadline of a given job $J_{i,n}$
$x_{i,n}$	current service time $u_i(t-d_{i,n-1})$ of a given job $J_{i,n}$
C_i	WCET of a given task τ_i (later chain $\mathbf{S}_i$), it has to be noted that this needs to include the cost of system calls.
E_i	budget allocated to task τ_i (later chain $\mathbf{S}_i$)
D_i	relative deadline of task τ_i (later chain $\mathbf{S}_i$)
T_i	period/minimal interarrival time of task τ_i (later chain $\mathbf{S}_i$)
C_i^*	part of current job of task τ_i (later chain $\mathbf{S}_i$) completed

Figure 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 τ_i consists of multiple jobs $J_{i,n}$ released a time $r_{i,n}$ and has a minimum interarrival time of T_i. The releases can not 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:

Theorem 2.1 *The earliest deadline first (EDF) algorithm will determine a feasible schedule if $U \leq 1$ under the assumption $D_i = T_i$.*

Theorem 2.2 *Given a feasible EDF schedule, at any time a task τ_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 2.3 *Given a feasible EDF schedule, at any time a task τ_i may increase its period without causing any task to miss deadlines in the resulting EDF schedule.*

Theorem 2.4 *Given a feasible EDF schedule, if at time t task τ_i decreases its utilisation to $u_i' = u_i - \Delta$ such that $\Delta \leq x_{i,n}/(t-r_{i,n})$, the freed utilisation Δ is available to other tasks and the schedule remains feasible.*

Theorem 2.5 *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 [5] 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 budget borrowing is possible under the condition that the task may only use the budget with the deadline of the job it was taken from, thus maintaining the schedulability proof condition.

3 Our Extensions

3.1 Deadlines $\neq$ Period

As a first step we want to introduce the notion of a schedulability analysis as basis for allowing deadlines to be different to periods. The superposition analysis work by Albers and Slomka [6, 7] enables the integration of bursts of task activation as well as arbitrary relative deadlines. We will briefly outline the analysis and its rationale, but direct the interested reader to the original publications.

In the analysis the work of Albers and Slomka is based on a task that is represented as a step function. Starting with a critical instant, each job generates at its deadline a request equal to its WCET, which has to be completed by that deadline. The release times of all jobs throughout the analyisis have to be chosen such that it describes in any interval starting at the critical instant the worst-case number of jobs that may be released in such an interval. The resulting step function is called processor-demand functions (PDF). The PDF for all tasks are added to a system-wide PDF.

If this system wide PDF is for any interval starting with the critical instant larger than the interval, i.e. in an interval we have requested more computation time than is available in the interval, the test has failed. The dotted lines in Figure 2 indicate the maximum request possible in any interval.

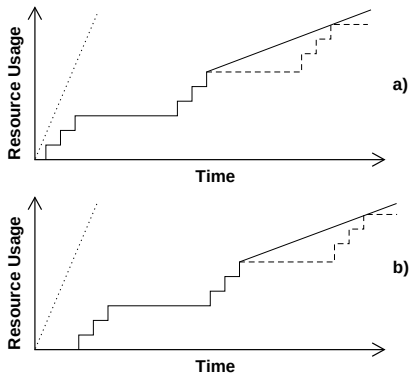

Figure 2: Sample Processor Demand Functions

The *exact* test described above may be costly if the task's interarrival time does not easily combine into a least common multiple. To alleviate this problem Albers and Slomka have chose 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 in tact for the first k jobs of a task and only afterward approximate with the line segments. This is lightweight enough to be deployed online. Figure 2 a) shows the PDF of a task, which has a deadline shorter than its minimum interarrival time. After an initial burst it is expected that there will be a time of quiescence before a reoccurance of the burst. The dashed line indicates the equivalent of the original analysis. Figure 2 b) has a similar distribution but has a deadline which is longer than its minimum interarrival time. In both cases the dotted line represents the available computation time for a given analysis interval.

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 best effort tasks WCETs may be unknown is immaterial. However, this also means that the cost of enforcing the budget in terms of interrupt handling and a full system call (ϵ) needs to be added to the analysis model on top of the budget E_i. The reasoning for this is that in the worst-case scenario the code has just entered a system call, which implies that the interrupt preempting the task may be delayed. The deadlines for best effort tasks are set to be equal to the period. The periods and budgets of best effort tasks are used to balance responsiveness and fair sharing of resources. Long periods may be used when responsiveness of the application is not a major issue, but may enable longer shares and thus enabling best effort tasks to work in longer stretches, reducing the management overhead of our scheduler. The budgets of best effort tasks relative to each other can be used to reflect different time complexity to achieve fairness or to improve the responsiveness of some best effort tasks over others.

3.2 End-to-End Deadlines

Within our work we combine the concepts of message based EDF (MEDF) scheduling [8] and RBED scheduling. The MEDF concept is driven by the observation that deadlines are usually system wide properties and thus are best reflected as such without introducing artificial partitioning of the physical deadline into individual task deadlines. Within MEDF a set of tasks responsible for a system response are scheduled using the same deadline. The deadline is propagated between the tasks of the task set using inter-process communication (IPC). By doing so partitioning of deadlines is avoided. The reason that this is relevant is that the release jitter of subsequent tasks caused by variation in the actual exe-

cution time of a preceeding tasks causes the analysis to be forced into a conservative assumption regarding task interarrival times of these subsequent tasks. The question may arise why one would not integrate the set of tasks into a single task. Possible motivations for such a move may be a componentised system, fault confinement, IP blocks, or simply license isolation.

We group one or more tasks τ_k, which are contributing to an overall system response with a physical end-to-end deadline into a set $\mathbf{S}_i$, called a task chain. All tasks in this task chain should be scheduled using the same deadline which will be transported by messages. Figure 3 illustrates such a task chain.

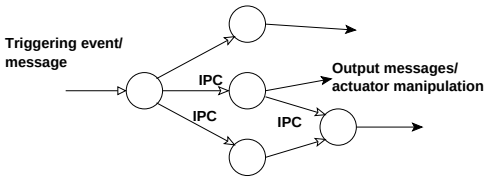

Figure 3: Taskchain

Definition The tasks τ_k in the task chain $\mathbf{S}_i$ have the following properties:

- The task chain is triggered by a single message or interrupt; in case of a required aggregation of data from multipe sources this aggregation needs to be implemented as server;
- T_i describes the minimum interarrival time of the triggering message of task chain $\mathbf{S}_i$;
- all tasks within the task chain have a direct communication relationship with at least one other task of the task chain $\mathbf{S}_i$;
- with the exception of a triggering message or interrupt the task chain is never blocked on a communication with a task outside the task chain which has a longer deadline than its own.

The last point may be enforced by not allowing synchronous communication with tasks outside the task chain. Beside the definiton above, which will be required throughout the paper, we introduce a second definition which states requirements to hold for this section only and will be relaxed in later sections.

Definition Each task is member of exactly one task chain:

$$\exists! \mathbf{S}_i : \tau_k \in \mathbf{S}_i \tag{1}$$

From a schedulability perspective the chained tasks adhering to the definitions above act as a single task with internal scheduling of subtasks. Special care needs to be taken in the case of partitioning of deadlines into more than one unit. Obviously this increases the constraints of the system, as now an additional deadline will have to be met. Additionally the partitioning leads to input message jitter for the second part of the partitioned deadline. Both these consequences at the very least complicate analysis, or in the worst case increase the pessimism of the schedulability analysis.

As such, a task chain $\mathbf{S}_i$ has the overall worst-case execution time C_i,

$$C_i = \sum_{\forall \tau_k \in \mathbf{S}_i} C_k \tag{2}$$

and is triggered with minimum interarrival time T_i.

Theorem 3.1 *A task chain $\mathbf{S}_i$ in a system is schedulable under the presented scheduling regime if:*

- *The scheduling proof [6] holds with E_i used as worst-case execution times within the analysis and*
- $E_i \geq C_i$

Proof As previously noted we can interprete the task chain as single task with internal scheduling. As such the case of case $E_i \geq C_i$ is the equivalent of a single task whose WCET may be pessimistic. The enforced budgets ensure that task chain is not adversely affected by another task chain exceeding its analysed E_i. ∎

3.3 Replenishment and Sporadic Tasks

We have integrated two of the algorithms presented by Lin and Brandt [5]: slack donation and the borrowing of budgets of future instances of the task chain. This is particularly driven by the notion that soft real-time task chains 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.

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 chain with a later deadline than the deadline of the current budget. A simple example of this is shown in Figure 4 a).

The deadline of the task chain *donating* the slack is noted alongside the slack amount. A task chain 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 chain has entirely consumed its budget, it can use the slack under the deadline of the slack. Whenever the idle task is running the budget is *consumed* by the idle task.

Theorem 3.2 *Slack s_i with associated deadline $d_{i,n}$ is preserved across an idle-task time δ_{id} at a rate* $\max(s_i - \delta_{idle}, 0)$.

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

In the case of an overrun of a task chain the budget is enforced and the task chain may receive the budget of a future release of the task chain with a respective deadline of that future budget, as shown in Figure 4 b). In this case there is obviously no donation of remaining budget once the task chain has completed, but instead the remaining budget needs to be preserved for the future release of the task chain.

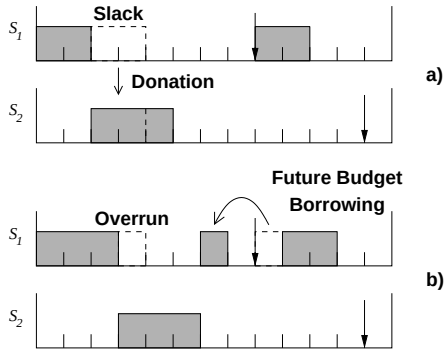

Figure 4: Slack donation and budget borrowing

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

3.4 Deadline and Budget Inheritance

In the case of communicating tasks beyond the limitations presented in the previous section, we need to consider server tasks which encapsule critical sections. The first consideration is where the time spent in the server needs to be attributed. Using dedicated budgets and deadlines for server tasks creates two problems:

- The partition running on the server budget and deadline blocks the client task. In the case where the server task is executed at the end or the beginning of a task chain this may not cause major problems, but if the server task is called by a client task which will then continue execution after the server returns, assumptions made in Section 3.2 will be violated.
- Secondly, in order to guarantee reasonable responsiveness, the server would need a short deadline compared to its minimum interarrival time, and thus put unnecessary constraints the system for its schedulability analysis.

The case of server calls within the body of a client task (as opposed to the beginning or end of the task chain) is common in L4 based systems, as servers like the naming server are frequently called upon. The other problem with server tasks is the contention in the access of the server and the associated dynamic priority inversion. Priority inversion is caused when a high priority task is being blocked on a sever working for a low priority task, which in turn is preempted by a medium priority task. [9] The previously mentioned borrowing of future budgets ensures that a task is never out of budget, but may have a very long deadline.

This can be solved in several ways.

- 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.
- Providing a rollback and restart of the server task for the preempted client would require substantial spare budget for restarts. This would scale with the number of possible preempting threads trying to access the server.
- A multithreaded server would be the preferred solution, but this in turn requires substantial implementation by the server writer and ultimately only reduces the length of the critical section implemented by the server, which does not fully solve the problem. This argument is based on the observation that a server usually embodies a critical section and a set of common operations on it.
- 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.

Similar to work by Wang et al. [10] the budget associated with the deadline needs to be inherited alongside the deadline. Avoiding the budget inheritance is possible, but would require changing the schedulability analysis to take *blocking time* into account, which in our opinion would not lead to an improved schedulability bound.

3.5 I/O Management

The use of I/O deserves special consideration, as it requires somewhat counterintuitive solutions. In a microkernel setting drivers usually reside in user-level. In the investigated L4 kernel, interrupts raised by hardware devices are delivered as asynchronous IPCs to the driver tasks.

The first problem encountered is where to attribute in-kernel interrupt handling. As the interrupt is first received in the kernel, it can not be subjected to the scheduling rules for user-level applications. As such the in-kernel IRQ code needs to be considered in the analysis. For this, interrupt WCETs are attributed with deadlines such that in the model interrupts are *scheduled* immediately in the analysis, e.g. in case of two interrupts simultaniously the deadline would be the combination of the two WCETs for analysis purposes. The mechanisms in the real-time proof [6] assumed in this work are well equipped model bursts of interrupts. The deadlines of the respective tasks are obviously impacted by this and need to be adjusted accordingly.

The second problem is that devices may be associated with several applications and several deadlines. For example, a network device may receive packets from sensor nodes with readings, as well as requests for higher-level data from a remote user panel. In a microkernel environment the actual decoding of packets would occur in a user-level task and as such the user-level task would serve different task chains. As such the client deadline and budget is not known when the user-level driver starts decoding the package. Retroactively assigning the budget and deadline to the task chain in question may lead to a violation of the EDF policy in the case of the deadline of the client being further out than another runnable task. In this particular scenario, partitioning of budgets is the only viable option. Similar to interrupts, the resulting jitter needs to be considered for the interarrival time of the subsequent task chain.

4 Road Test

4.1 Implemementation

We have implemented the aforementioned algorithms in and on top of an OKL4 [11] kernel version 1.5.2 as proof of concept. All tests and measurements in this paper were taken on an XScale PXA255 based Gumstix [12] board. The actual scheduler was implemented in the kernel and 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 all tasks in the system and their reserved resources. It provides the kernel with the deadlines, budgets and minimum interarrival times of all tasks which then stores this information in the task control block.

The scheduler in itself is a purely EDF scheduler. The enforcement of budgets is realised by a timer, which is set up 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 chain is completed, or when a task chain changes its budget and deadline due to exhaustion of its current budget. Since deadlines and budgets may be passed on via messages in task chaining or deadline inheritance, not every release or completion forces a reprogramming of the budget watchdog.

The ready queue and send queues are deadline sorted. 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 primitive and have forced a reschedule after each IPC. *Call* IPCs are used by the kernel to identify servers. For all other synchonous send or asynchronous notify IPC the kernel checks whether the receiving thread has an individual deadline and budget and if not implements task chaining.

4.2 Case Study

We have developed a small, but non-trivial case study to identify issues with the proposed method and to demonstrate that the method may be deployed in a realistic scenario.

The system, shown in Figure 5, 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. Obviously the text messaging has best effort scheduling character while the audio transmission is soft real-time.

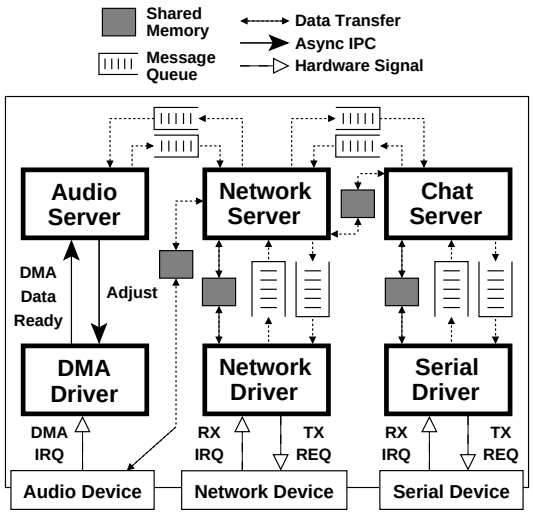

Figure 5: Case Study

Hardware drivers in this system are implemented as user-level threads. The audio driver has been omitted from Figure 5 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 initialize 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 [13].

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 (e.g. the network driver receives network packets and sends network packets) and requires knowledge from the sending to decode received packets and vice versa. Besides the threads depicted in Figure 5 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 not used past the initialisation phase. A naming server is responsible to dissolve 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 5. This monitoring task makes use of a virtual timer server and the network server.

4.3 Discussion

As efficient IPC system calls are the backbone of any microkernel, we have paid special attention to these. Given the proposed requirement that synchronous (blocking) IPC may only happen to servers or tasks with deadlines being passed on by the IPC, it becomes apparent that under these conditions the enqueue operation is always at the head of the queue. Making the enqueue operation $O(1)$ and neglegibily 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 6 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 fixed priority queues are roughly equivalent to our enqueue operation with enqueueing in the 1st position after the queue head.

The dequeue operation on the ready queue, once a task is blocked on a receive IPC is again $O(1)$ and with

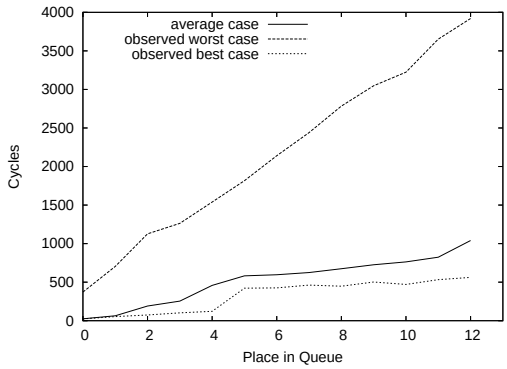

Figure 6: Enqueue Cost Measured

27 cycles 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 the message-based deadline passing and server tasks this might not be the preferable solution.

As indicated, the current implementation leaves room for optimisation. In particular direct process switch and lazy dequeuing should be sped up. As opposed to the case of fixed-priority scheduling [14], the EDF scheduling policy enables a much easier reasoning about which task of two communicating parties should run next.

Lazy dequeueing describes the effect of avoiding dequeue operations of tasks which are likely to be enqueued again shortly after dequeueuing. Again as opposed to fixed priority scheduling [14] 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.

5 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 [8]. 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 and temporal isolation, assuming instead all WCET estimates are conservative. Additionally the target application domain are 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 heavily dependent on message based deadline transportation, as individual "tasks" are very small, but a single input might trigger multiple state transitions.

Jansen et al. [15] 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 (i.e. least common multiple of periods) their analysis will take substantial resources. Also the execution times are not enforced and thus uncontrollable behaviour in the case of a best effort or a soft real-time task misbehaving is not guaranteed.

The concept of budget inheritance during priority inheritance has been investigated by Lamastre et al. [16]. The motivation for their work was similar to ours in the sense of providing 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 interarrival times of the soft real-time components and thus the dynamic slack of a task may not be freely donated to another task. The concept has been extended to a bandwidth exchange server by Wang et al. [10] 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.

Resource sharing in a rate based environment has also been investigated by Liu and Goddard [17]. Instead of deadline inheritance they have implemented a deadline ceiling protocoll. Similar to the work by Brandt et al. they have implemented the approach inside the Linux kernel. While it supports servers it does not account for other communication or I/O.

6 Conclusions and Future Work

Within this paper we have presented an integrated scheduling approach detailing 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 integration of task chains which may be used to describe activation chains in a componentised system, the impact of system services on the budget planning, I/O, and server tasks.

While the work presented in this paper covers a large range of issues, there are still issues which may be addressed. The previously mentioned optimisation and subsequent 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 [18]. 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.

7 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.

References

[1] S. Poledna and et al., "OSEKTime: a dependable real-time, fault-tolerant operating system and communication layer as an enabling technology for by-wire applications," in *SAE 2000 World Congress*, (Detroit, MI, USA), pp. 51–70, Mar 2000.

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

[3] 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 *24th RTSS*, (Cancun, Mexico), Dec 2003.

[4] L. Abeni, G. Lipari, and G. Buttazzo, "Constant bandwidth vs. proportional share resource allocation," in *5th ICMCS*, vol. 2, (Florence, Italy), pp. 107–111, Comp. Soc. Press, 1999.

[5] C. Lin and S. A. Brandt, "Improving soft real-time performance through better slack management," in *26th RTSS*, (Miami, FL, USA), Dec 2005.

[6] K. Albers and F. Slomka, "An event stream driven approximation for the analysis of real-time systems," in *16th Euromicro Conf. Real-Time Syst.*, (Catania, Italy), Comp. Soc. Press, 2004.

[7] K. Albers and F. Slomka, "Efficient feasibility analysis for real-time systems with EDF scheduling," in *42rd DATE*, (Munich, Germany), 2005.

[8] T. Kolloch, *Scheduling with Message Deadlines for Hard Real-Time SDL Systems*. Dissertation, Institute for Real-Time Computer Systems, Technical University Munich, Germany, 2002.

[9] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *Trans. Computers*, vol. 39, pp. 1175–1185, Sep 1990.

[10] S. Wang, K.-J. Lin, and S. Peng, "Bwe: a resource sharing protocol for multimedia systems with bandwidth reservation," in *4th MSE*, (Newport Beach, CA, USA), Dec 11–13 2002.

[11] OK-Labs, *OKL4 Website*. URL http://www.ok-labs.com/products/okl4.

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

[13] 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 *9th RTCSA*, (Tainan, Taiwan), pp. 499–512, Springer–Verlag, Feb 2003.

[14] K. Elphinstone, D. Greenaway, and S. Ruocco, "Lazy scheduling and direct process switch — merit or myths?," in *3rd OSPERT*, (Pisa, Italy), Jul 2007.

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

[16] G. L. G. Lipari and L. Abeni, "A bandwidth inheritance algorith for real-time task synchronisation in open systems," in *22nd RTSS*, (London, UK), pp. 151–160, Comp. Soc. Press, Dec 2001.

[17] X. Liu and S. Goddard, "Resource sharing in an enhanced rate-based execution model," in *15th Euromicro Conf. Real-Time Syst.*, (Porto, Portugal), pp. 131–140, Jul 2003.

[18] M. P. Lawitzky, D. C. Snowdon, and S. M. Petters, "Integrating real time and power management in a real system," in *4th OSPERT*, (Prague, Czech Republic), Jul 2008. to appear.