Using Fixed Priority Scheduling with Deferred Preemption to Exploit Fluctuating Network Bandwidth

Mike Holenderski, Reinder J. Bril and Johan J. Lukkien
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5600 AZ Eindhoven, The Netherlands
{*m.holenderski, r.j.bril, j.j.lukkien*}*@tue.nl*

***Abstract*—Fixed Priority Scheduling with Deferred Preemption (FPDS) offers a balance between Fixed Priority Non-preemptive Scheduling (FPNS) and Fixed Priority Preemptive Scheduling (FPPS), by allowing preemptions only at specified preemption points. It provides finer grained preemptions than FPNS, improving the schedulability of higher priority tasks, and a coarser grain preemptions than FPPS, reducing switching overhead incurred during arbitrary preemptions. In this paper we investigate the extent of improvement of FPDS with respect to FPPS and qualify the costs of switching multiple resources under FPPS and FPDS, and the cost of a preemption point. It forms a starting point for our research into employing FPDS in an industrial case study, to improve an existing multimedia processing system from the surveillance domain. We focus on extending FPDS with optional preemption points, to guarantee resource provisions to tasks in spite of fluctuating resource availability, in the context of reservation-based multi-resource sharing.**

I. Introduction

On the two sides of the Fixed Priority Scheduling spectrum we have Fixed Priority Non-preemptive Scheduling (FPNS) and Fixed Priority Preemptive Scheduling (FPPS) [8]. While FPNS favors the lower priority tasks, by postponing preemption by higher priority tasks until a lower priority running task completes, FPPS focuses on the schedulability of higher priority tasks. However, by allowing preemptions at arbitrary moments in time, FPPS ignores the cost of such preemptions. This overhead may become especially significant when tasks share multiple resources, e.g. cache, local or main memory.

Fixed Priority Scheduling with Deferred Preemption (FPDS) [4], [5], [7], [6], [3] finds a middle ground between FPNS and FPPS:

- It aims at reducing the cost of arbitrary preemptions in FPPS, by allowing them only at times convenient for the system (referred to as *preemption points*), e.g. at times where the context switch overhead due to preemption will be smallest. If FPDS is used as a guarding mechanism for critical sections, then there is also no need for access protocols to the shared resources (other than the processor), reducing the system overheads.
- It improves on FPNS by allowing shorter non-preemptive subjobs and thus improves the schedulability of higher priority tasks.

This work has been supported in part by the Information Technology for European Advancement (ITEA2), via the CANTATA project.

FPDS is a generalization of FPPS and FPNS, where FPPS can be modeled by FPDS with arbitrarily short subjobs (ignoring context switch and scheduling overheads), and FPNS by FPDS with tasks consisting of a single subjob.

We distinguish two kinds of resources: *preemptable* and *mutually exclusive*. When a preemptable resource (e.g. processor) is preempted, we can store and reload its state, incurring some bounded overhead. If we were to preempt a mutually exclusive resource (e.g. access to a shared memory location) then its integrity could be corrupted.

FPDS can be used to reduce the cost of context switches of preemptable resources, and provide simple access protocol to mutually exclusive resources. It promises a simple implementation of critical sections, compared to the intricate priority inheritance protocols used in FPPS, as [11], [14] reveal wrong implementation of these protocols in the existing real-time operating systems. In FPDS, the subjobs simply execute non-preemptively.

We are interested in employing FPDS in an industrial case, to improve an existing multimedia processing system from the surveillance domain. We focus on using FPDS to guarantee resource provisions to tasks in spite of fluctuating resource availability, in the context of reservation-based multi-resource sharing.

A. A surveillance system

There are two main tasks in the system: a video task τ_v and a network task τ_n. These tasks run on a platform containing two processors with cache and two local memories (LM and IRAM), communicating with the main memory M via DMA transfers over a shared system bus, as shown in Figure 1.

A camera monitoring a scene places the captured video frames in the main memory M. The video task τ_v loads the raw frames from the main memory M to local memory LM, does some video content analysis on them, encodes them and stores the result back to the main memory M. The main processor is responsible for scheduling and may offload some operations of τ_v and τ_n to the co-processor.

The network task τ_n loads the encoded frames from the main memory M to local memory IRAM, wraps them into packets and sends them over the network via the EMAC interface.

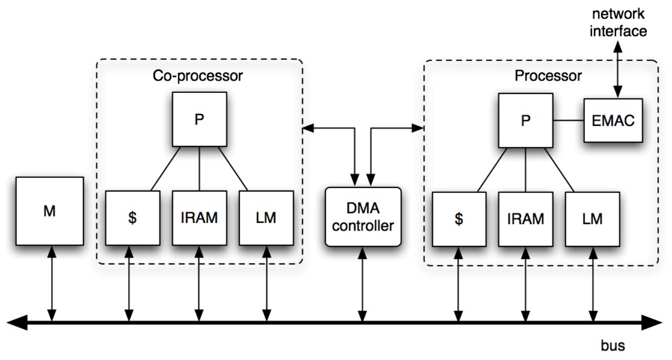

Fig. 1. Architecture of a surveillance system

Both tasks are periodic, with their period determined by the video stream rate.

Currently the processor is scheduled according to FPNS, with τ_n having higher priority than τ_v, because FPPS is not feasible due to its large context switch overheads. These are emphasized in the streaming multimedia domain, where tasks are computation and data intensive, and where tasks are dependent (producer consumer relation). However, due to fluctuating network bandwidth and a resource greedy video task, the network task cannot make optimal use of the processor: it may be scheduled when the network is congested, or the video task may hold the processor when the network is available. We would like to use FPDS in combination with reservations to guarantee resources to the video task, while optimizing the use of the available network, by providing finer grained preemption points.

B. Reservations and FPDS

We can guarantee the processor to the video task in spite of fluctuating network bandwidth in the current surveillance system by introducing reservations [9]. We implement the reservations with two servers: a deferrable server for the network task, allowing the exploitation of its preservation strategy [2], and a periodic server for the video task.

We use FPDS to schedule the reservations globally, with the network reservation having higher priority than the video reservation. We use FPDS to schedule each reservation locally. Selecting finer granularity of the preemption points in τ_v will allow τ_n to resume more frequently and transmit when the network is available. The deferrable server allows τ_n to postpone its transmission and release the processor to τ_v, in case the network is congested. Setting the preemption points in τ_n to packet boundaries can prevent aborting an ongoing transmission and avoid the cost of later retransmission.

C. Granularity of the preemption points

There is a trade off between the granularity of the preemption points and the overall schedulability of the system. On the one hand, if preemption points are spaced arbitrarily close, their cost of invoking the scheduler may lead to more frequent context switches and the need to execute expensive locking mechanisms on shared resources, and thus lower the useful utilization in the system [7]. On the other hand, a coarse grain of preemption points will lead to worse schedulability of higher priority tasks.

FPDS is also referred to as *co-operative scheduling* [4], because some control over preemption is moved from the scheduler towards the application. The real-time problem shifts from scheduling towards finding the right preemption points granularity.

D. Optional preemption points

It turns out that the additional cost due to preemption points may imply a too coarse granularity of preemption points, and thus discourage from using FPDS in certain applications. We would like to suggest the concept of *optional preemption points*, which are easy to implement and incur little overhead, compared to the traditional preemption points.

The optional preemption points aim at reducing the overhead associated with traditional preemption points through a tighter co-operation between the scheduler and the tasks. Traditionally, when a job arrives at a preemption point, it does a system call to the scheduler indicating it is ready to be preempted. Such a system call has two drawbacks:

1) The system call itself is expensive, especially when the preempted job is running in a different memory space than the scheduler.
2) The preempted job always has to assume that it will be preempted. If the job knew that it will *not* be preempted at the next preemption point it may choose a different execution path, e.g. initiate a long memory transfer from the main memory to the shared local memory, which it would not do if it was to be preempted.

[7] reduce the system call overhead in point 1, by having the kernel set a flag in a shared structure (we will refer to it as the *preemption flag*) when a task arrives with a higher priority than the running task. The running task checks during its next preemption point whether it needs to be preempted by reading the shared flag.

[13] presents a hardware supported solution to avoid a system call in point 1. When a new task arrives, the scheduler places it in the ready queue and loads dedicated registers with the address of the next preemption point. A special hardware component compares the address of the following executed instructions with the address of the next preemption point. When the address matches, then the corresponding switch routine is performed.

E. Outline of this paper

We would like to extend FPDS to multiple resources and apply it in reservation based scheduling, focusing on reducing preemption point granularity to exploit the available network bandwidth.

In this paper we would like to qualify the extent of improvement of FPDS on FPPS. In Section II we qualify the context switch overhead when multiple resources have to be switched, and see how this cost influences the granularity of preemption points. In section III we qualify the cost of a preemption point itself, as opposed to regular trap/interrupt mechanisms. We conclude with future directions in Section IV.

II. Cost qualification of multiple resources switching under FPPS and FPDS

The main idea behind FPDS is that the cost of context switches can be reduced relative to FPPS, if tasks are preempted at convenient times for the system, referred to as preemption points. For example, it is inefficient to preempt a task just after it's program code and data was loaded into the cache, because the program is likely to be flushed from the cache before the preempted task resumes.

The literature on FPDS considers a single preemptable resource (the processor) and focuses on how FPDS can help reducing the cost of switching the processor [4], [7]. We would like to extend FPDS to multiple resources, taking into account the different switching costs of different resources. Depending on the implementation and the underlying architecture, some of the following factors may contribute to the context switch overhead:

1) Registers: During a context switch the register file is stored (in local or main memory), sometimes with hardware support. Traditionally all registers are stored and reloaded. [13] show how at runtime, the preemption is deferred to the next preemption point, where the kernel invokes a custom context switch routines (per preemption point), which save and restore *only the affected* registers of the preempting and preempted task. These routines are generated at compile time and execute at the tasks privilege level, to ensure memory protection.

2) Cache: If the program for the preempting task is not in the cache, it is first loaded from the main memory. Also any cached data (especially in data intensive multimedia applications) is likely to be flushed during the switch or the execution of the new task.

3) Local memory: Similarly, if the preempting and the preempted task are sharing the same locations in the local memory, the local memory is stored to and reloaded from to the main memory.

4) Main memory: Usually, memory performs best, when the addresses are accessed in a consecutive manner. If the main memory needs to be accessed by the scheduler to load its program code or data during a preemption point, then the sequence of memory accesses in the preempted task can be interrupted. In our surveillance application example in Section I-A, the memory access is a bottleneck.

5) DMA: In systems supporting virtual memory, memory locations are grouped into a block and blocks are grouped into pages, with virtual pages being mapped to physical pages. When DMA is also supported, there is a question of which addresses it should use: virtual or physical.

In case DMA transfers are setup using virtual addresses, the processor has to provide the mapping from virtual to physical addresses for the affected memory locations. In case of physical addresses, a DMA transfer which spans across pages will have to be chopped up into smaller pieces, because the physical pages may not be contiguous. In both cases, there is a setup overhead for a DMA transfer. Also, the operating system must not remap the pages involved in the transfer [10].

6) Deep pipelines: Switching a pipelined stream of operations on a VLIW like processor may incur large overhead if the instruction pipeline has to be flushed. In our surveillance example, the overhead can be particularly large, when a video-stream processing operations pipeline of τ_v offloaded to the co-processor, is interrupted by packet encoding operations of τ_n.

7) Network: There are two approaches to switching the network: wait for the pending packet(s) to be sent, or interrupt the current transmission. In the first case, the context switch overhead will include the time necessary to complete the transfer of the pending packets. In the second case, the switching overhead is moved to retransmitting the aborted packets the time the preempted task resumes.

The network resource behaves similarly to a hard disk, as it has a large latency relative to the processor. Just like the disk can be unavailable due to the seek time, the network availability can fluctuate, due to congestion in the network. However, since the network congestion is much less predictable, so is the availability of the network resource.

The surveillance system currently employs busy waiting approach, with the network task (once scheduled) waiting for the network to accept all the packets, before it releases the processor to the video task.

Under FPPS, depending on the architecture, all of these overheads may be incurred upon a context switch. Traditional FPDS can be used to reduce *one* particular overhead. For example, if saving and reloading the data in the local memory is the main bottleneck on a particular architecture, then FPDS can be used to prevent arbitrary preemptions within instruction sequences which operate on a shared location in the local memory.

In our surveillance example the two bottlenecks are the main memory and network access.

III. Cost qualification of a pre-emption point

A preemption point in itself incurs overhead, which can be modeled as shown in Figure 2.

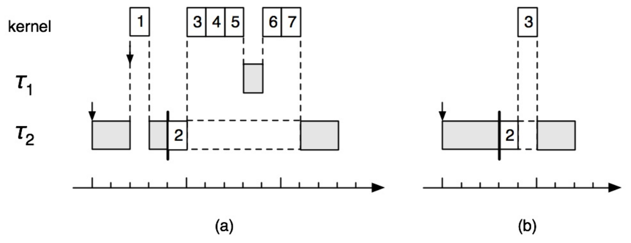

Fig. 2. Breakdown of a preemption point, when (a) a higher priority task is ready (b) no higher task is ready

1) While τ_2 is executing task τ_1 arrives. A corresponding interrupt is dispatched (timer or external, depending on task τ_1) and handled by the kernel. The scheduler in the kernel places τ_1 in the ready queue and stores in a register that a higher priority task has arrived. The kernel returns and τ_2 is allowed to continue.
2) When τ_2 reaches a preemption point, it makes a system call to the kernel. Here the task can execute a preemption point specific code, e.g. register specific context switching functions at the kernel for saving and reloading only

the affected registers [13], or make a system call to lock any mutually exclusive resources in case it will be preempted [7].

3) The kernel checks whether a higher priority task is pending.
4) The kernel stores the state of τ_2. This can involve switching any of the resources listed in Section II.
5) The kernel loads the state of τ_1. This can involve switching any of the resources listed in Section II.
6) After task τ_1 executes, the kernel saves its state, in case τ_1 needs some of its state for its next invocation.
7) The kernel loads the state of task τ_2.

IV. Future directions

A. Quantify costs of preemption points and context switching

[7] quantifies the preemption point overhead for a processor. We would like to quantify the preemption point and switching costs in a multi-resource setting, based on our multimedia streaming surveillance application, to gain insight into the shortcomings of current solutions and serve as a reference point for our future work.

B. Extract preemption points from code

In order for FPDS to be accepted by system designers, it has to be easy to use. Rather than manually inserting preemption points inside the code, we would like the compiler to extract optimal preemption points, where the overheads are minimized.

[13] presents a solution where the compiler identifies preemption points (they refer to them as *switch points*) in the code with a minimal number of general purpose live registers. We would like to also include the switching overheads of other preemptive resources, besides the processor.

C. Reduce preemption point cost

We would like to investigate how optional preemption points can reduce the overhead of FPDS, compared to the traditional preemption points.

[13] showes how to reduce the switching overhead by saving and restoring only the affected registers. We would like to extend this approach to other resources.

D. Account for race conditions due to optional preemption points

In Section I-D point 2 we identified the potential of optional preemption points allowing a task to check the preemption flag *before* a preemption point and select an execution path depending on whether it will be preempted or not during its next preemption point. However, it gives rise to a race condition.

Let t_p be the next preemption point when the running task τ_r can be preempted. For τ_r to adapt its execution path, it may need to read the preemption flag at time $t_r < t_p$. The longer the difference between t_r and t_p, the longer the time when the flag cannot be changed by the scheduler, or the time when τ_r will not take it into account. We would like to investigate how to account for the race condition in the model of the computation time of subjobs and the response time analysis.

E. Avoid preemptions when tasks share multiple resources

In systems which exploit *multiple resources* the cost of preemptions is not straight forward, as it may depend on the resources currently used by the running task and those requested by the preempting task, leading to different costs at different preemption points. We would like to investigate finding appropriate preemption points, which take into account the switching overheads of multiple resources.

Acknowledgments

We would like to thank Rick Koeleman from VDG Security for the insightful discussions about their platform architecture.

References

[1] T. P. Baker, “Stack-based scheduling for realtime processes,” *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, 1991.

[2] R. J. Bril and P. J. Cuipers, “Towards exploiting the preservation strategy of deferrable servers,” in *To appear in: Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS 08)*, 2008.

[3] R. J. Bril, J. J. Lukkien, and W. F. J. Verhaegh, “Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited,” in *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS 07)*, pp. 269–279, 2007.

[4] A. Burns, “Preemptive priority based scheduling: An appropriate engineering approach,” in *Advances in Real-Time Systems*, S. Son, Ed. Prentice-Hall, pp. 225–248, 1994.

[5] A. Burns, M. Nicholson, K. Tindell, and N. Zhang, “Allocating and scheduling hard real-time tasks on a parallel processing platform,” University of York, UK, Tech. Rep. YCS-94-238, 1994.

[6] A. Burns, “Defining new non-preemptive dispatching and locking policies for ada,” *Reliable SoftwareTechnologies —Ada-Europe 2001*, pp. 328–336, 2001.

[7] R. Gopalakrishnan and G. M. Parulkar, “Bringing real-time scheduling theory and practice closer for multimedia computing,” *SIGMETRICS Perform. Eval. Rev.*, vol. 24, no. 1, pp. 1–12, 1996.

[8] 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.

[9] C. Mercer, R. Rajkumar, and J. Zelenka, “Temporal protection in real-time operating systems,” in *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software (RTOSS ’94)*, pp. 79–83, 1994.

[10] D. A. Patterson and J. Hennessy, *Computer Organization and Design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[11] D. Polock and D. Zobel, “Conformance testing of priority inheritance protocols,” in *Proceedings of the Seventh International Conference on Real-Time Systems and Applications (RTCSA’00)*. Washington, DC, USA: IEEE Computer Society, p. 404, 2000.

[12] L. Sha, R. Rajkumar, and J. P. Lehoczky, “Priority inheritance protocols: An approach to real-time synchronization,” *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, 1990.

[13] X. Zhou and P. Petrov, “Rapid and low-cost context-switch through embedded processor customization for real-time and control applications,” in *DAC ’06: Proceedings of the 43rd annual conference on Design automation*, pp. 352–357, 2006.

[14] D. Zöbel, D. Polock, and A. van Arkel, “Testing for the conformance of real-time protocols implemented by operating systems,” *Electronic Notes in Theoretical Computer Science*, vol. 133, pp. 315–332, 2005.