Real-Time Architecture (2IN25)
Assignment 4

Dennis Peeten (0571361)
Oliver Schinagl (0580852)
Wilrik De Loose (0601583)
Tan Zhi Ming Joshua (0645373)

May 25, 2008

Contents

1 Introduction

Our world is getting more and more surrounded by electronic devices. Only a few years ago our surroundings were limited to radio's and televisions. Soon after we got our microwaves and washing machines, not to mention the boom in the mobile phone corner. All these devices require operating systems and nearly all of these OS require real-time computations.

We have done a literature study on Fixed Priority Scheduling with Deferred Preemption (FPDS) in which we looked at various design and implementation aspects of this type of scheduling. In chapter 2 we discuss development considerations while the architectural considerations are dealt with in chapter 3. The next chapter, chapter 4, is all about the application domains of FPDS. We conclude our report with some small results and notes on the discussion held after our presentation on this subject.

2 Motivation

Fixed priority scheduling with *arbitrary* preemptions (FPPS) is extensively documented in literature. This class of schedulers preempt a lower priority task as soon as a higher priority task becomes runnable. Upon preemption of a task a context switch is performed, in which the cpu postpones the execution of the current task and prepares itself to execute a different task. The operations associated with context switches take time, how much time usually depends on the system architecture and the state of the computation. The costs of context switches are generally neglected in FPPS literature because it is really hard to get a good estimate or bound on these costs.

As a compromise between *no* preemptions and *arbitrary* preemptions, FPDS was introduced. A scheduler that employs deferred preemptive scheduling, preempts tasks only at certain points that are defined by the system designer. By controlling the points at which a task may be preempted, a system designer has some knowledge about the state of the system and computation at which the task is preempted and at the same time may limit the number of preemptions that occur. This allows for more educated guesses of the costs of context switches and hence more accurate analysis of the (real world) schedulability of task sets.

An essential goal of OS resource managemnt for real-time and multimeda systems is to provide timely, guaranteed and protected access to system recources. Allthough extensive research has been commited on processor scheduling, or network scheduling alone, disk scheduling, co-processor scheduling et cetra have been studied to a smaller extend. Not to mention using those resources *simultaneously* wihtin a single node. Video applications may access high volume data from a disk, processes the data and transmit it accross the network. All these stages must be completed by a deadline. Obtaining simultaneous and timely access to multiple resources is known to be an NP-complete problem, making it a very complex to handle properly.

3 Development considerations

Arbitrary preemptions of tasks which are used in fixed priority preemptive scheduling, often result in cache state changes in between execution of the program. This causes difficulties in determining the effect of cache reload overheads on execution time. These preemptions make the problem of determining cache hit rates virtually intractable, making the system unpredictable. Although fixed priority non-preemptive scheduling can solve this problem, it comes at a cost of reduced schedulability.

Another solution to this problem is to use Fixed Priority Scheduling with Deferred Preemption (FPDS). FPDS allows for preemptions at pre-specified points within a task's execution. This has two main benefits. The first is that it allows for greater accuracy in determining the Worst Case Execution Time (WCET). By allowing preemptions at only specified points, we can more accurately predict the effect these preemptions will have on schedulability. More importantly, the second benefit which this provides is that it reduces the costs of preemption through selecting appropriate points of preemption. In selecting the preemption points, we can limit preemptions to take place at points where less amount of cache reloads are required, hence reducing the costs of preemption. Therefore, special care has to be taken in selecting these preemption points.

Let us first define two terms now. First, an active cache line is defined in [?] as a cache line that that contains a block of data that will be referenced in the future prior to its replacement. In other words, it is a cache line in which the next reference is a hit, had the task been allowed to run to completion. Secondly, a preferred preemption point for a given task interval t_i to t_j is defined in [?] as the instant within the interval having the minimum number of live cache lines.

Bearing these in mind, the task we have at hand is to search for the preferred preemption points. There are two main ways that these points can be determined, using a compiler or manually.

3.1 Using a compiler

The compiler is used to analyze the worst-case execution path and preferred preemption points are determined along this path. The number of active cache lines at any point can be determined by analyzing the activity of the cache line with the help of the compiler. After the worst-case execution path is analyzed, the next worst is analyzed using the preemption points that have been found. If the execution time of this path is lower than the WCET, the analysis can stop. Otherwise, the analysis has to be repeated. The help of a compiler is needed to do the analysis because the analysis requires a lot of processing and monitoring. Also, the effects of caching become transparent to the user if a compiler is used.

3.2 Manually

The preferred preemption points can also be determined by the programmer with the help of insertion of traps in the source code to monitor when tasks of higher priority are ready to run. The programmer would already have an intuition as to where the number of live caches is minimized. Using this method, preferred preemption points can also be determined.

Considering the two methods, it is clear that using the compiler is a more systematic and transparent way of finding preemption points. In addition, preemption points found using a compiler will generally perform better because the analysis is more extensive. However, this analysis can also be performed manually as described above is a compiler cannot be used.

4 Architectural considerations

When using fixed priority scheduling with deferred preemption, one must take some extra architectural considerations into account. This chapter deals with a number of aspects such as the choice of processor, use of the memory and interrupt handling.

4.1 Processors

In this section, two types of processors are being compared. These two are pipelined processors and general purpose processors. Pipelining, a standard feature in RISC[1] processors, is much like an assembly line. Because the processor works on different steps of the instruction at the same time, more instructions can be executed in a shorter period of time. General purpose processors on the other hand are mainly used to achieve a decent average response time (although most architectures nowadays contain a pipeline feature).

The use of a pipelined processor can result in significant performance improvements. However, dependencies between instructions can cause pipeline hazards that may delay the completion of instructions. Christopher A. Healy et. al. posed a method to determine the worst case execution times of pipelined processors with cache in [**?**]. This approach reads all kinds of information about each type of instruction from a machine-dependent data file. This way the performance of a sequence of instructions in a pipeline can be analyzed.

In the paper of Healy et. al. they assume a system with a non-preemptive scheduling paradigm. In the case of deferred preemption scheduling, a task is split up in one or more sub-tasks which are all non-preemptive, thus making the approach applicable to FPDS. This means the worst case execution times can be calculated and thus the system can be analyzed if it's schedulable under FPDS.

It's clear that the use of pipelined processors have a huge advantage over general purpose processors when it comes down to performance. Yet, determining the worst case execution times is much more elaborate and complex. Both aspects need to be taken into account when choosing the right architecture for a specific real-time application.

4.2 Memory

4.2.1 Cache

When cache memory is to be used in real-time systems, special attention must be paid since cache memory introduces unpredictability to the system. This unpredictable behavior is due to cache reloads upon preemption of a task. When preemptions are frequent, the sum of such cache reloading delays takes a significant portion of task execution time. The cache reloading costs due to preemp-

[1]RISC, or Reduced Instruction Set Computer, is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions.

tions have largely been ignored in real-time scheduling.

Buttazzo states in [?] that it would be more efficient to have processors without cache or with cache disable. Although this would obviously get rid of the problem, it is not desirable because the use of cache can really improve system performance. Another approach is to allows preemptions only at predetermined execution points with small cache reloading costs. A scheduling scheme was given in [?] and is called Limited Preemptible Scheduling (LPS). The scheduling scheme only allows preemptions only at predetermined execution points with small cache reloading costs. This means that the method can be applied to FPDS.

In [?] a method to determine the worst case execution times was given for multiprocessors with cache. In that paper a cache simulation is used to categorize a cache operation. Using the outcome of such a simulation the blocking time can be determined very precisely. It was already stated that this approach is applicable to FPDS in paragraph 3.1.

4.2.2 Local and global memory

In any multiprocessing system cooperating processes share data via shared data objects. A typical abstraction of a shared memory multiprocessor real-time system configuration is depicted in figure 1. Each node of the system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network.

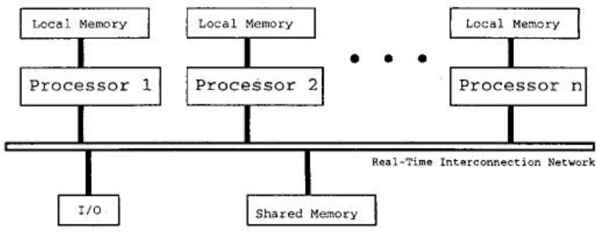

Figure 1. Shared memory multiprocessor system structure

However, the literature doesn't state much about the use of local and global memory when using FPDS. Therefore it is difficult to make a statement about this subject. Most likely the use of local memory shouldn't have much impact on the schedulability of the system under FPDS. Global memory on the other hand can indeed bring nondeterminism when using cache. FPDS is therefor more predictable because the cache misses can be determined more accurately.

Shared variables in a critical section can also bring nondeterminism. When guarding a critical section with semaphores, a lower priority job can block a higher priority job. This can be overcome with various protocols like PIP, PCP and SRP (although the implementations of PIP and PCP recently have been proven flawed).

4.3 Interrupt handling

Real time systems usually have at least two groups of tasks. They are the application tasks and the interrupts. Both classes of task are repeatedly fired due to certain events. The difference between the two classes is that application tasks are usually periodic and they start executing due to events that are generated internally. In contrast, interrupts execute in response to external events.

Interrupts generally pre-empt a task in the same way a higher priority task would pre-empt a lower priority one. There are two main ways an interrupt can be handled. Interrupts can be handled using a Unified interrupt architecture where system services can be accessed from Interrupt Service Routines (ISR) or using a Segmented Interrupt Architecture wherein systems services may not be accessed from ISR.

In using the Unified interrupt architecture, interrupts are served immediately after they are invoked, and all interrupts must be disabled during the time the initial interrupt is served because the ISR can access the system services directly and there is no way to ascertain which ISRs make which kernel calls. Hence, there exists a possibility that interrupts may be disabled for too long and an interrupt may be missed.

In contrast, when using the Segmented Interrupt architecture, ISRs cannot call kernel services. Instead, the ISRs invoke a Link Service Routine (LSR), which are then scheduled by a LSR scheduler to run at a later time. LSRs can run only after all ISRs have been completed. They then call kernel services which schedule the LSR with respect to all other tasks. The kernel services schedules the LSR so that it only starts running if and when the appropriate resources are available. This means that it incurs a lower task switching overhead. Using this method of serving interrupts also helps to smooth peak interrupt overloads. When a burst of ISRs are invoked in rapid succession, the LSR scheduler helps to ensure that temporal integrity is maintained and will allow the interrupts to run in order of the way they were invoked. Additionally, LSRs run with interrupts fully enabled, which prevent missing of any interrupts during the execution of LSRs.

In conclusion, we find that using the segmented interrupt architectures have the benefit of lower task switching overhead, smoothing peak interrupts overloads and prevent the missing of interrupts that occur while LSRs are being served. Thus, Segmented Interrupt architecture is superior compared to the Unified interrupt architecture.

4.4 Multi-processor systems

Gai et. al. [**?**] Describe scheduling of tasks in asymmetric multiprocessor systems consisting of a general purpose CPU and DSP acting as a co-processor to the master CPU. The DSP is designed to execute algorithms on a set of data without interruption, hence the schedule for the DSP is non-preemptive. Gai et. al. treat the DSP scheduling as a special case of scheduling with shared resources in a multiprocessor distributed system, using a variant of the *coopera-*

tive scheduling method presented in [?] by Seawong and Rajkumar. Cooperative scheduling is appropriate in situations where a task can be decomposed into multiple phases, such that each phase requires a different resource. The basic idea of cooperative scheduling as described by Seawong and Rajkumar is to associate suitable deadlines to each phase of a job in order to meet the job deadline.
In order to apply this to the GPP + DSP multiprocessor architecture, Gai et. al. define their real-time model to consist of periodic and sporadic tasks subdivided into regular tasks and DSP tasks. The regular tasks (application tasks, interrupt service routines, ...) are executed entirely on the master CPU for C_i units of time. The DSP tasks execute for C_i units of time on the master CPU and an additional C_i^{DSP} units of time on the DSP. It is assumed that each DSP job performs at most one DSP request after C_i^{pre} units of time, and then executes for another C_i^{post} units of time, such that $C_i = C_i^{pre} + C_i^{post}$ as depicted in Figure 3.4.

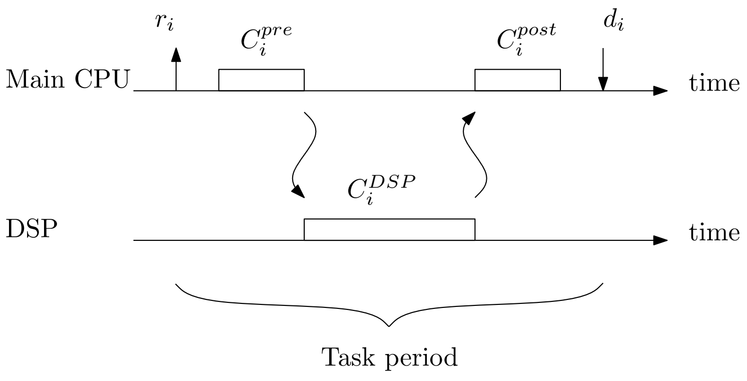

Figure 1: Structure of a DSP task

When executing a DSP task, a hole within each job is generated in the schedule of the master processor. Gai et. al. show that the earliest deadline first (EDF) and rate monotonic (RM) scheduling algorithms do not always yield a feasible schedule, while one exists. Figure 3.4 shows an infeasible task set when scheduled by RM or EDF. Figure 3.4 shows the same task set scheduled using a fixed priority assignment where $\tau_2 \prec \tau_1$, such that τ_2 executes in the holes of the master CPU's schedule.

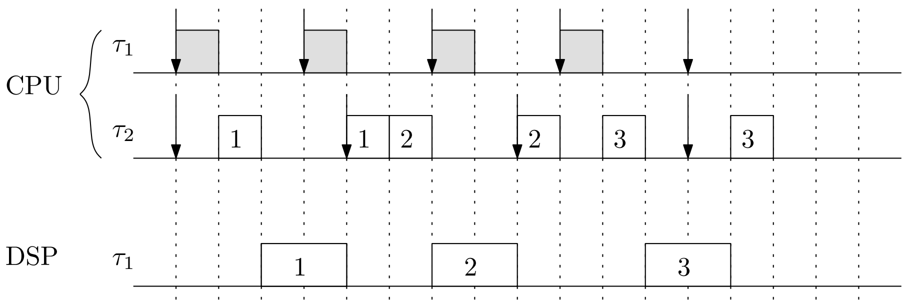

Figure 2: A task set cannot be scheduled by RM and EDF (τ_2 misses all deadlines).

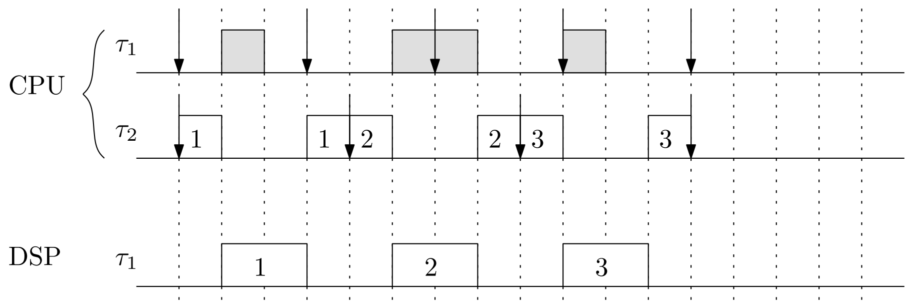

Figure 3: A feasible schedule with fixed priority assignment $\tau_2 \prec \tau_1$.

The main idea is to modify the scheduler to exploit these holes to schedule some other task on the master processor to improve the schedulability bound of the system. This is achieved by modeling the DSP request of a task τ_i as a remote procedure call that blocks τ_i for B_i units of time, waiting for its completion. The scheduling algorithm uses a fixed priority assignment. In order to determine the next task to be executed, the scheduler enqueues regular tasks and DSP tasks in two separate queues that are ordered by priority, as shown in Figure 3.4.

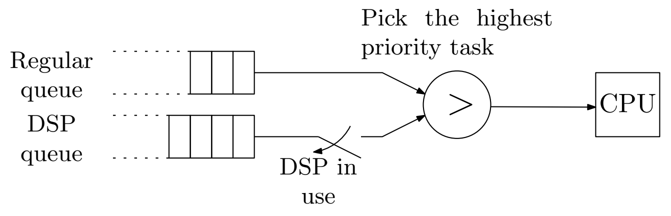

Figure 4: Structure of a DSP task

When the DSP is idle the scheduler selects the task with the highest priority between those at the head of the two queues. When the DSP is active, the scheduler select the highest priority task from the head of the regular queue only. In this way, a task using the DSP blocks all other tasks requiring the DSP, but not the regular tasks, which can freely execute on the master processor in the holes created by DSP activities.
Because the DSP tasks executing on the DSP cannot be preempted, it can happen that a lower priority DSP task τ_i is blocking a higher priority task that was released during the execution of τ_i on the DSP. The blocking of high priority DSP tasks by lower priority DSP task has to be accounted for in the schedulability test. This test is presented [**?**] using the hyperbolic bound.
The article furthermore present results of a simulation of the described algorithm and compares them to schedules generated by the distributed priority ceiling protocol (DPCP). However they do not mention the method of priority assignment used in their simulation.

5 Application domains

Having discussed the development and architectural considerations, we can now explore the application domains for FPDS. We have found that FPDS allows for better predictability of the WCET compared to scheduling that allows arbitrary preemption. This is because with FPDS the number of preemption points are limited. However, when deadline misses occur with FPDS, there is a higher chance that the task that misses its deadline is a higher priority task due to the limitation of preemption points as compared to arbitrary preemptions.

We will now divide the application domains into two main groups, control systems and HQ-video. For each of these groups, we will discuss whether using FPDS is suitable.

5.1 Control systems

We find that the area of control systems can once again be divided into two main groups. In the first group we have control applications where tasks that miss their deadline preferable have a lower priority. An example would be a fuel injection system in a car. We would like to miss tasks that control the amount of fuel injected into the engine rather than tasks that control whether fuel is injected into the engine at all. If the amount of fuel injected into the engine is wrong, the car can still function even though the ride may be bumpy. However, if the fuel is not provided to the engine, the car will not be able to function. In scenarios that pertain to the first group, schedules with arbitrary preemptions should be used in favor of FPDS because they would reduce the chance that a higher priority task misses its deadline.

In the second group, we have applications where missing any deadline would be equally disastrous. An example would be a space shuttle where any deadline miss may cause the shuttle to crash. In these cases, it is better to use FPDS with limited preemption points because FPDS allows us to predict the WCET more accurately.

5.2 HQ-Video

For high quality videos sound is of a higher priority than the image. A deadline miss for the audio would greatly decrease the perceived quality, but a deadline miss for images would not be as noticeable. In this case, we should choose scheduling with arbitrary preemption points so that the chance missing an audio deadline will be reduced.

6 Discussion and conclusion

The correct use of preferred preemption points in FPDS allows as system designer to put a bound on context switching costs, even when using features such as caches and instruction pipelining. These relatively cheap hardware acceleration techniques can benefit real-time systems as they gain performance and/or cut the hardware costs, while still allowing rigorous schedulability analysis for tasks sets consisting of periodic and sporadic hard real-time tasks. One drawback however, is that the response times of high priority tasks may be higher as opposed to arbitrary preemptive scheduling, because a higher priority tasks that is released during the execution of a lower priority task has to wait for lower priority task to reach a preemption point.

Note that tasks/algorithms used in a real-time system have to be suitable for the use of the mentioned acceleration techniques to be beneficial for the real-time system. In the case of memory cashes for example, an application that accesses the memory in random manner, such that the cache controller is not able to reliably predict the next memory access, won't benefit from cashes.

However, using preferred preemption points alone won't improve the predictability much when the real-time system receives (a lot of) interrupts. The segmented interrupt architecture for interrupt handling as described in this article, can help a great deal in improving the predictability. The interrupt service routine still preempts the running task, but because the ISR is implemented as a very simple function that only schedules a job (LSR) to handle the interrupt request, the overhead introduced can be bounded. These LSR's can be modeled as periodic or sporadic tasks and be taken into account in the schedulability analysis.

In conclusion we believe that fixed priority scheduling with deferred preemption, when used correctly, in combination with the segmented interrupt architecture, greatly improves the accuracy of the schedulability analysis for actual real-world real-time systems.