Universität Karlsruhe
Fakultät für Informatik
Institut für Betriebs– und Dialogsysteme (IBDS)
Lehrstuhl Systemarchitektur

Double-sorted Scheduling in Process Cruise Control Environments

Study Thesis

Sören Finster
November 22, 2005

Advisors: Prof. Dr. Frank Bellosa
Dipl.–Inf. Andreas Weißel

Declaration
Erklärung

I hereby declare that I did this thesis on my own, using no other sources than the ones cited.

Hiermit erkläre ich, dass ich diese Arbeit selbstständig und ohne Verwendung anderer als der angegebenen Quellen angefertigt habe.

Sören Finster
Karlsruhe, November 22, 2005

Abstract

Schedulers found in contemporary operating systems try to maximise the throughput and the responsiveness perceived by the user. In this thesis a scheduler policy is introduced which not only tries to fulfil those objectives but also takes the order of task execution into consideration. By keeping strict priority scheduling or by loosening priority restrictions this policy enables the system developer to define a criteria which then affects the order of the schedulers queue. This new scheduler policy is called *Double-sorted Scheduling* (DSS).

This work uses DSS to extend an existing policy called *Process Cruise Control* (PCC) which exploits information from embedded hardware monitors to adjust the CPUs core frequency at context switches to the optimal frequency of the next task. PCC allows energy savings of up to 22%. DSS additionally tries to eliminate unnecessary frequency adjustments by ordering tasks according to their optimal frequency.

DSS and PCC were implemented within the Linux kernel on the Intel XScale architecture. Evaluations show that the addition of DSS to the PCC environment results in less frequency adjustments. With this approach, not only the time, but also the power, lost due to unnecessary frequency adjustments is reduced.

Contents

Chapter 1

Introduction

> Scheduling is the process of assigning tasks to a set of resources.
>
> – Wikipedia

The function which is normally referenced as the operating systems scheduler manages the assignment of all tasks in the system to one or more CPUs. In this conventional setup, the scheduler takes care of one single resource – the computing time. It splits the CPU time in small slices and assigns periodically a certain amount of them to every task in the system. This behaviour is often implemented with several scheduler policies in mind. Examples include fairness and prevention of starvation. But most implementations are not aware of the second responsibility the scheduler has. By assigning the CPUs to the tasks, the scheduler also arranges the sequence of the tasks. The order of scheduler queues is affected by the priority assigned to certain tasks. This is called priority scheduling. It's policy requires that tasks with higher priority are executed before tasks with lower priority. Therefore tasks are only sorted with their priorities.

In an embedded environment with hard disk support, for example, accesses to the disk are very expensive concerning the power consumption. Also the time until the disk is in its working state (spin up of platters, disk arm movement out of parking position) has to be considered. Therefore several efforts were made to reduce disk accesses and to group necessary accesses. So, the hard disk can be shut down after the access. This behaviour is implemented by the operating system which delays the access from applications until there are enough disk operations or a timer runs off. The frequency of disk accesses would be a good (second) dimension for sorting tasks in the scheduler queue. Priority scheduling as sole sorting criteria is not sufficient. With priority scheduling the tasks within one priority level are not sorted.

Their order depends on many criteria and is implementation specific. If there are tasks with high disk access frequency and tasks with low disk access frequency in one priority level, the grouping of disk accesses is greatly easier if the queue is sorted with this criteria in mind. The disk accesses were issued sequential and the possibility, that two or more accesses can be grouped rises. This assumes that there are enough tasks in the system. Without sorting, the possibility of several short disk accesses is much higher since a task with no disk access can be scheduled between two tasks with disk access. Then, the timer of the operating system, which waits for more disk accesses, might run off. It then interrupts the working no-disk-access task, performs the disk access, unblocks the task which caused the disk access and then continues with the interrupted no-disk-access task. When this task used up its timeslice, the second disk-access task can state its wish for disk access. With a sorted queue, both disk accesses could have been grouped.

Another good application in which sorted scheduler queues are of great benefit is Process Cruise Control (PCC). This scheduler policy, introduced by Bellosa and Weißel in [2], uses performance counters of embedded systems to determine the appropriate clock frequency of every task running in a timesharing environment. The clock frequency of the CPU is then adapted to the appropriate clock frequency of the next task at every context switch. This is done to reduce the power consumption of the CPU. The prototype implementation in [2] shows energy savings of 22% for memory intensive applications at a maximum performance loss of less than 10%. This performance loss is caused by the function which determines the appropriate clock frequencies and by every frequency adjustment which is performed. The setup in [2] and the setup of this thesis use embedded environments which can perform very fast frequency adjustments. Therefore the ratio of the performance loss which is due to frequency adjustments is roughly 30%, which is relatively low. On systems without this capability the frequency adjustments will play a dominating role in performance loss. Therefore, the avoidance of unnecessary frequency adjustment is a crucial requirement for a good performance of PCC on non-embedded systems. Fig. 1.1 illustrates a worst case study of a scheduler queue. The CPU in this case is capable of operating at two different frequencies. This results in two classes of tasks. Those with optimal frequencies nearer at the lower frequency, and those with optimal frequencies nearer the high frequency. We speak also from *frequency domains* instead of classes. Because of two frequency domains there is only one threshold. As can be seen there are two tasks with optimal frequencies above this threshold and three tasks with optimal frequencies below. In this worst case, the CPU will perform four frequency adjustments.

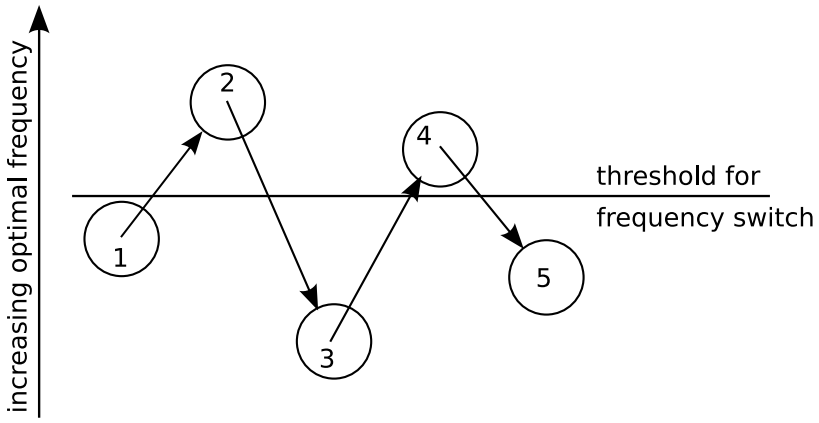

Figure 1.1: An unsorted queue

PCC is used in this thesis to implement and determine a new scheduler which considers a scheduling order based on a second criteria beside priorities. Therefore this scheduler policy is called Double-sorted Scheduling (DSS). In this thesis DSS concentrates on the avoidance of frequency adjustments in a PCC environment though its application can be much more general.

1.1 Why DSS is necessary

As tempting a frequency adjustment at every context switch is, it is also a time and power consuming process. Tasks are not only preempted by timer interrupts but also by I/O or they decide to give away the CPU by themselves. This can result in much more context switches than ten per second (which is because of the 100ms standard timeslice). And every context switch can cause the need for a frequency adjustment. If there are only processes with equal optimal frequency in the queue, than no adjustment is needed. But every additional process with different requirements increments the number of adjustments for the worst case by two. So there is an upper limit of adjustments for the period it takes to execute every process in a time slice epoch (assumed all processes use up their timeslice and are not interrupted). A time slice epoch is the time it takes from time slice distribution until all tasks have used up their timeslice. Because the maximum of adjustments is reached with 50% high and 50% low frequency processes this upper limit is the number of processes in the queue. However, the best case is always one adjustment, completely independent of the number of processes. The upper

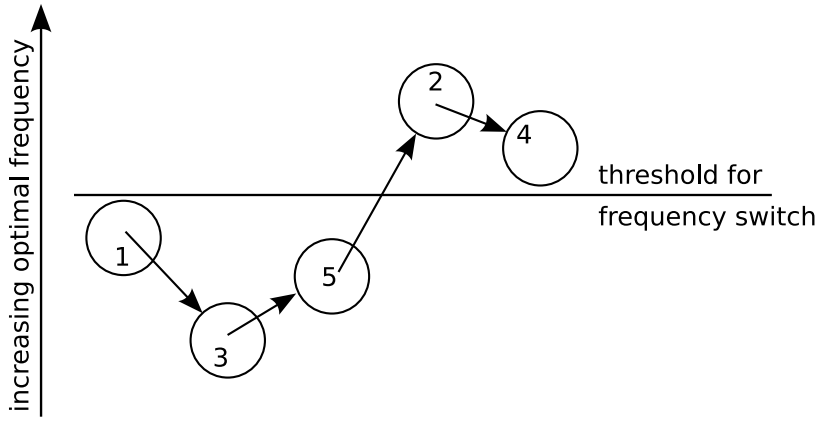

Figure 1.2: A sorted queue

limit is also valid for more than two frequency domains. A higher value would stand for more than one adjustment per task which is not possible in the examined time period. But the lower limit has to be increased to $\#frequencydomains - 1$. This is because after working all tasks of a certain frequency domain the scheduler has to adjust the frequency to work another domain. For the first domain there is no adjustment necessary because the scheduler can choose the domain according to the CPUs state. This lower limit is achieved by a sorted queue.

In Fig. 1.2 the best case of Fig. 1.1 can be seen. Before the queue was sorted every context switch caused a frequency adjustment. But after sorting, the new schedule causes only one single adjustment.

This leads to the question how much effort is put into sorting the queues and if it pays off. That is, if the saved time or power exceeds the time or power used for sorting. Whether time, power or a combination of both is crucial depends on used hardware and application. The developers of battery-operated devices without user interaction or time constraints possibly don't care about losing a few seconds if they can save some power and therefore keep their device running. On the other hand, the developers of workstation systems tend to be rather generous with power consumption if they can get a higher performance or save some time. In the following the term *cost* refers to a ratio of power to time which is appropriate for the specific application.

As a simple rule of thumb the cost for DSS per context switch should not exceed the cost of a frequency adjustment times the probability of a frequency adjustment per context switch. Since the calculation of this probability is not possible, only statistical evaluations could prove an indication. However

the probability increases with the number of different frequency domains and slightly decreases with more tasks in the system.

If the maximum cost of a frequency adjustment is lower than the cost of DSS per context switch, DSS uses more cycles or more power than it saves. In this case there is no need for DSS and it should be deactivated or even deactivate itself.

Chapter 2

Evaluation Platform

As testing platform an Intel PXA270 XScale Processor on a KARO Electronics Triton Starterkit 3 is used. Although the CPU is classified as micro controller and its main application area is in embedded devices like cell phones and PDAs it delivers enough features and performance to do quite well in desktop appliances. Further details can be found in [4] and about its predecessor PXA250 in [3].

The PXA270 provides sophisticated power management which made it first choice for this task. The processor is ARM V5 compatible and utilises a memory management unit for virtual to physical address translation. The performance monitoring feature which monitors events in the core and allows the developer to gather information about internal events (e.g. cache accesses, cache misses, executed instructions) is a prerequisite for the classification of processes. Additionally the CPU exhibits a feature called Wireless Intel Speedstep which offers detailed control over all frequencies of the internal clocks. This enables very fast frequency adjustments at context switches which take only 0.1 microseconds. This feature normally prevents the ap-

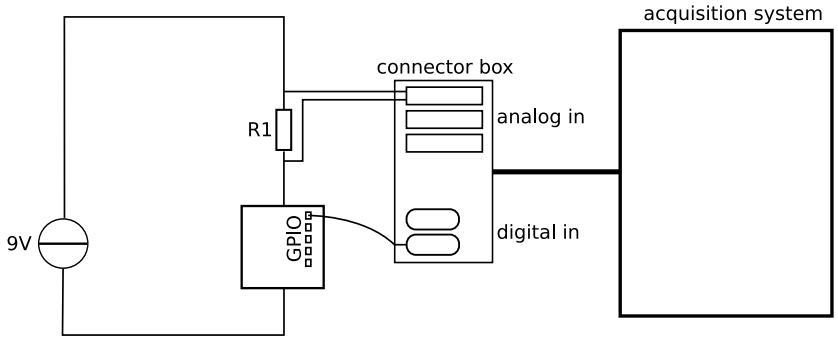

Figure 2.1: The measurement environment

plication of DSS since DSS becomes less useful the faster the CPU can do frequency adjustments. But for evaluating DSS concerning the saved adjustments, this setup is convenient. Furthermore, the setup allows to rise the cost of frequency adjustments artificially by inserting wait cycles in the adjusting function. Thus it can be analysed how well DSS performs at different adjustment costs.

The Triton Starterkit 3 includes, beside the standard components like Ethernet Controller and standard I/O, additional controllers for USB, CompactFlash, audio, touchscreen and more. The general purpose I/O pins proved as great help at automatically starting and stopping measurements. For further details see [5].

To measure the energy consumption, a sense resistor of 100 $m\Omega$ (R1 in Fig. 2.1) is placed in series with the power supply ($9V$). The voltage drop is measured with an acquisition system which samples at 10kHz. This gives detailed insight into energy consumption even during single timeslices. A single measurement results in hundreds of megabytes of data which are processed automatically by self-written scripts to obtain needed information. At this point the previously measured idle power of the board is subtracted to contemplate only the more important dynamic power consumption instead the power consumption of the whole board which is not affected by any power saving procedures of the processor. The measuring of the energy consumption was used for the implementation of PCC.

Software is mainly provided by KARO Electronics which includes a cross compiling toolchain and a modified Linux kernel (version 2.6.11). Supplementary to several self written programs some binaries of the Familiar Distribution of handhelds.org were used. Self written programs are mainly computational or highly memory based ones which were used to classify processes. As compiler the KARO supplied and for cross compiling configured gcc 3.3.2 was used. No optimisations were activated during compilation of self-written programs. The Linux kernel was compiled with standard parameters.

2.1 The Linux 2.6.8 Scheduler

The Linux 2.6.8 scheduler is highly optimised piece of work. It uses only operations of $O(1)$ complexity which makes it fast, scalable and hard to understand. This section is intended to give a rough summary about the internals of the scheduler. Only the parts relevant for this paper will be covered. For a complete introduction to the scheduler see [1] or [6]

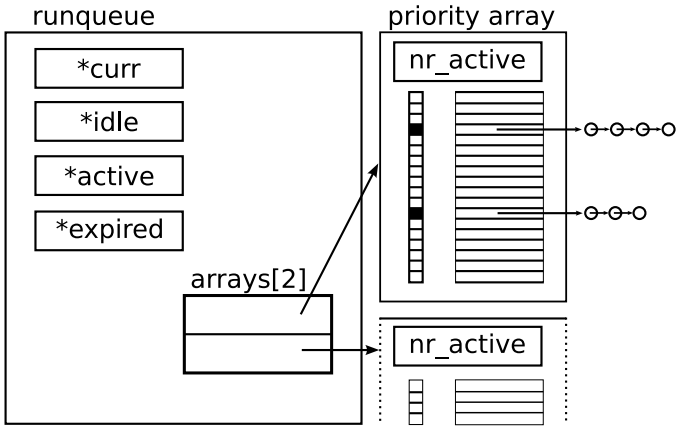

Figure 2.2: The schedulers datastructure (simplified)

2.1.1 Data structure

Runqueues

The runqueue data structure is the most basic structure in the scheduler. It keeps track of all runnable tasks assigned to a particular CPU. One runqueue is created and maintained for each CPU in a system. The runqueue is defined as a struct in `kernel/sched.c` and contains the following variables[1]:

- `unsigned long long nr_switches`

The number of context switches that have occurred on a runqueue since its creation. It is exposed in the proc filesystem as a statistic.

- `task_t *curr`

Pointer to the currently running task

- `task_t *idle`

Pointer to a CPUs idle task

- `prio_array_t *active`

Pointer to the active priority array. This priority array contains tasks that have time remaining from their timeslices.

- `prio_array_t *expired`

[1] only the variables with relation to DSS were listed

Pointer to the expired priority array. This priority array contains tasks that have used up their timeslices

- `prio_array_t arrays[2]`

The actual two priority arrays. Active and expired array pointers switch between these.

Priority Arrays

This data structure is the basis for most of the Linux 2.6.8 schedulers advantageous behaviour, in particular its $O(1)$ time performance. The scheduler always schedules the highest priority task in a system and if multiple tasks exist at the same priority level. they are scheduled round robin with each other. Priority arrays make finding the highest priority task in a system a constant time operation, and also make round robin behaviour within priority levels possible on constant time. Furthermore, using two priority arrays in unison makes transitions between timeslice epochs a constant time operation. The priority struct contains following variables:

- `unsigned int nr_active`

The number of active tasks in the priority array

- `unsigned long bitmap[BITMAP_SIZE]`

The bitmap representing the priorities for which active tasks exist in the priority array. For example - if there are five active tasks, three at priority 2 and two at priority 9, then bits 2 and 9 should be set in this bitmap. This makes searching for the highest priority level in the priority array with a runnable task a simple call to `__ffs()`, a highly optimised function for finding the highest order bit in a word.

- `struct list_head queue[MAX_PRIO]`

An array if linked lists. There is one list in the array for each priority level in the system. The list contains tasks and whenever a lists size becomes > 0, the bit for that priority is set in the bitmap. When a task is added to a priority array, it is added to the list of its priority.

2.1.2 How Scheduling Is Done

When a task is created a timeslice is calculated and it gets inserted in the active priority array. There, it is appended to the according linked list at the tasks priority level. Appending a task to the linked list is an $O(1)$ operation implemented in `list.h`. At the tasks priority level it is scheduled round robin with all other tasks at this level. After running and not having used all of its timeslice a task gets reinserted at the end of the linked list. When a task runs out of timeslice, it is removed from the active priority array and put into the expired priority array. During this move, a new timeslice is calculated. When there are no more runnable tasks in the active priority array, the pointers to the active and expired priority arrays are simply swapped. Because timeslices are recalculated when they run out, there is no point at which all tasks need new timeslices calculated for them. So, many small constant time operations are performed instead of iterating over however many tasks there happen to be.

2.2 Kernel Modifications

Before actually implementing the scheduler and the necessary sorting policy the kernel was modified to implement *Process Cruise Control* [2].

First of all the performance counter of the PXA270 is activated and configured at startup. The four performance registers are configured to count data cache accesses, data cache misses, executed instructions and instruction cache misses. Furthermore the task control block is modified to keep this values per process. These TCB values are accessible from userspace through the proc filesystem.

The scheduler adds the performance registers to the values in the TCB at every context switch. After that the registers are reinitialised to zero before the next task is activated. Therefore the TCB always represents the overall statistics of the process and enables the scheduler to compare the results of the last run of the task to the overall rating. Therefore the scheduler can detect changes in the behaviour of a task and react. To give an example it could weight the newest results lesser if they differ too much from the overall rating.

The already by Karo Electronics implemented cpufreq interface is revised and extended concerning the fast frequency switch by changing the turbo bit which was not implemented at all. The newly added features are integrated into the cpufreq sysfs interface and accessible from userspace for debugging purpose.

Chapter 3

Implementation

Implementing the scheduler can be done with different objectives in mind. Depending on the used hardware and the application, different behaviours can be desired. It is possible to keep the complexity restrictions of $O(1)$ but the schedulers datastructure gets enlarged which results in larger cache footprints and the requirement of more memory. The $O(1)$ restriction especially pays off when there are a lot of tasks in the queues and an $O(n)$ algorithm would render the system unusable. But a system which is designed to run with many tasks in the queues presumably already addresses a higher demand of memory, cache and CPU performance. Therefore the downsides are tolerable. This implementation is covered in Section 3.1.

On the other hand there is an implementation without enlarging datastructures (covered in Section 3.2). But it breaks with the $O(1)$ rule and will be in an inferior position if it comes to a high amount of concurrent tasks. Yet, the advantages of this implementation are a small cache footprint and low memory requirements. This is adequate for smaller systems or embedded systems whilst the $O(1)$ solution is capable of performing well on server applications.

Although these implementations differ in many details, there are some similarities. Both need an array which represents the costs of frequency adjustments as it can be seen in Table 3.1. In this hypothetical array a transition from 208 MHz to 416 MHz would take 1000 cycles. Since frequency adjustments to higher frequencies tend to take more time than to lower frequencies, the array is not symmetric. Hence the transition from 416 MHz to 208 MHz needs only 800 cycles. But these values depend highly on the setup and rule of thumbs are often invalid. That is because the cost of a transition depends on:

- Is there a need to adjust the core voltage?

MHz	104	208	416	520	620
104	0	1000	2500	3000	4000
208	800	0	1000	2800	3500
416	1200	800	0	2000	2500
520	2000	1500	1000	0	2000
620	3000	2500	2000	1500	0

Table 3.1: A hypothetical array of costs

- Can the new frequency be reached with only adjusting the frequency multiplier

This array is an important datastructure which the scheduler uses to make decisions. First of all the scheduler can calculate an order of frequency domains which proves to cause the least cost when traversing all domains. The array also indicates if DSS is useful or if the scheduler should deactivate it. This is if the minimum in the array is lower then the costs of DSS per context switch. Furthermore by analysis of the array certain frequencies can be sorted out. Sorting out can be necessary because the transition costs to or from the frequency domain are extremely high. This sorting out happens only if the setup tries to save time or prefers time for power saving.

3.1 Keeping Complexity Restrictions

To avoid breaking the $O(1)$ rule there are modifications to the scheduler datastructure necessary. These modifications cause re-implementations of some of the most important functions of the scheduler. The linked lists which contain the runnable tasks are no longer sufficient. Another layer of abstraction is needed. The finding of the next task is possible in $O(1)$ because of two reasons.

- The highest priority level with a runnable task is indicated by the bitfield in the priority array and therefore done in fixed time
- Finding the head of a linked list is an operation with a finite upper bound

Also the swapping of both priority arrays is an important operation for the desired $O(1)$ complexity. It avoids the $O(n)$ operation of moving n tasks from one array or queue to another. However the swapping does not influence the implementation of DSS.

3.1.1 Strict Priority Scheduling

If we want to retain strict priority scheduling we have to consider only a resorting within one priority level because the scheduler chooses one linked list with the candidates. In this case the duty of DSS is only to select the best fitting task (that means the one which implies the lowest overhead due to frequency adjustments). Because of the $O(1)$ constraint neither searching in the linked list nor sorting this list is acceptable. In fact the datastructure needs to be modified to provide a way the next task is found in $O(1)$. This is done by splitting up the linked in list in several linked lists which each represent a certain group of tasks. There are as many linked lists at one priority as the system provides steps in frequency adjustment. Therefore every linked list represents one frequency domain. In the easiest case (two frequency domains) there are only two linked lists needed. One, which holds the tasks identified to work at the low frequency, and one which holds those which work at high frequency. This case is illustrated in Fig. 3.1. The array of pointers to linked lists doubles in size and with every additional frequency domain one additional column is needed.

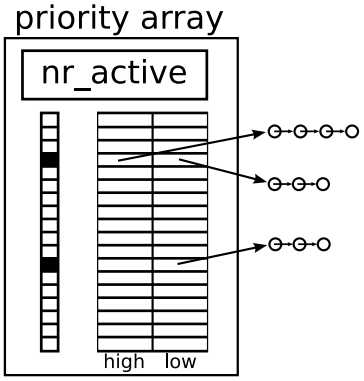

Figure 3.1: The modified priority array for two frequency domains

After such a modification the selection process is slightly more complex. The finding of the highest priority level with a runnable task is done as before. But then, the modified scheduler first checks the linked list which may contain a fitting task. This is the list which represents the frequency domain matching the actual processor state. If there is one, the new scheduler performed as well as the unmodified one, which also needed to access a linked list, although the linked list was not dynamically determined. But if there is no task in the best fitting group, the scheduler has to work its way through all remaining lists until it finds a task. By looking up the pre-defined array of costs from transitions between CPU states the scheduler can work the lists in

the order which finds better fitting tasks first. By first trying the best fitting linked list the working order across the priorities resembles a zigzag course, assumed all linked lists include at least one runnable task. This results in rising and falling clock frequencies with changes after each context switch. As can be seen in Fig. 3.2 there is some space for optimisation. This is covered in Section 3.1.2.

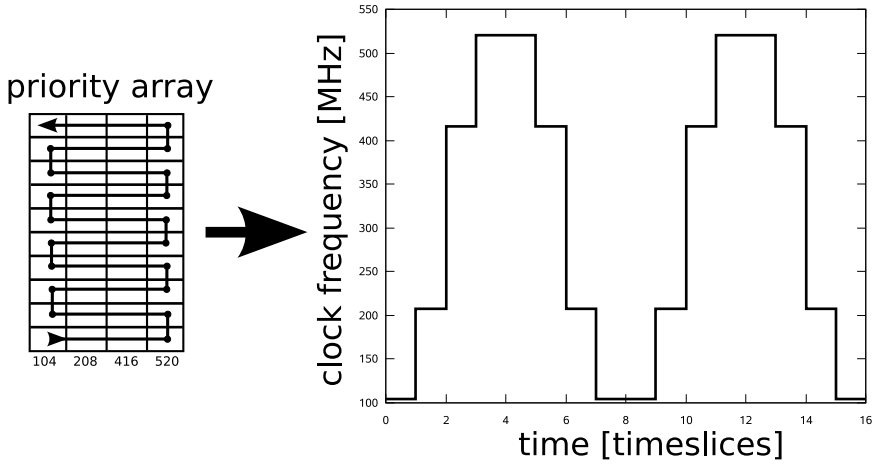

Figure 3.2: ZigZag

After all the scheduler still operates at $O(1)$ since:

- Finding the highest priority level with runnable tasks is done in constant time as before
- Choosing the best fitting linked list involves a lookup of the actual CPU state which is also done in constant time
- Worst case assumed, the scheduler has to traverse all linked lists at the particular priority level. Since the number of lists is bound to the capabilities of the CPU it is finite.
- Finding the next linked list to look in may involve a lookup of a precalculated array which is also done in constant time.

3.1.2 Loose Priority Scheduling

As the standard Linux scheduler implements 140 priority levels[1] there is enough space to arrange tasks that they can't get in connection as long as DSS keeps strict priority scheduling.

[1] From 0-99 reserved for real-time tasks

A first approach to this problem is to drastically reduce the supported priority levels. But this would also mean to change the public interface of the scheduler which results in further changes to the operating system. As impacts beyond the scheduler are unwanted this solution is not acceptable.

This has to be solved inside the scheduler by mapping the operating system priorities to less scheduler internal priorities. This technique has to be used very carefully especially in the real-time range of priorities. Mapping two on one can be implemented by using only as many linked list as there are frequency domains for two priority levels (see Fig. 3.3).

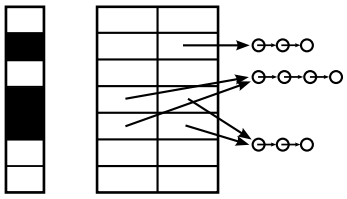

Figure 3.3: Two on one

3.2 Breaking Complexity Restrictions

As in Section 3.1, there are also two possibilities of implementing the scheduler with broken complexity restrictions. But both of them introduce much simpler changes to the scheduler and both result in an $O(n)$ algorithm. As mentioned before the datastructures stay intact.

3.2.1 Strict Priority Scheduling

To comply with the requirements, the implementation must not alter the schedulers datastructures. Therefore the only possibility is a sorting of the linked lists. But, instead of implementing a sorting function which could be called every time changes were made to the linked list, this implementation alters the `enqueue_task()` function which is called whenever a task is inserted into a linked list. The normal behaviour of this function is to insert the task at the end of the linked list. This can be done in $O(1)$ since the lists head has a pointer to the last entry in the list.

The modified function first checks if the list is empty that is, if no task is at this priority level. In this case it simply adds the task as before. In every other case, the function runs through the linked list in reverse order. For every task it checks if its frequency domain is equal or greater than the frequency domain of the task which is to be inserted. If a task matches, the

new task is inserted after the matching one. If none is found, the new task becomes the lists head. By this behaviour, the list is ordered by decreasing frequency domains. That is, first all the tasks of the highest frequency domain, than all of the second highest, and so on. The insertion process resembles bubble sorting since tasks of high frequency domains float up the list like bubbles. Besides the `enqueue_task()` function there are also the `requeue_task()` and `enqueue_task_head()` functions which manipulate the linked lists.

The `requeue_task()` function is responsible for putting a task at the end of the queue without the overhead of `dequeue_task()` followed by a call to `enqueue_task()`. Since this function is used to implement round robin scheduling inside one priority level there is some risk with altering it. If the last task with the highest frequency domain gets blocked and should be put at the end of the list, the sorting algorithm would put it at the lists head again. In this case the round robin scheduling would be broken and the task will get blocked and immediately unblocked as long as it has timeslice left. But also with more tasks of the highest frequency domain, the blocked task would only round robin with the task of his frequency domain. Since this function is mainly used in the realtime part of the scheduler, no modification is done.

The `enqueue_task_head()` function is solely used for the activation of idle tasks. The normal behaviour is to put the activated task at the head of the list (as the name of the function hints). Since this behaviour is only used in the SMP part of the scheduler, it can be ignored for the scope of this thesis. If DSS is implemented with SMP enabled, this function shouldn't be altered.

3.2.2 Loose Priority Scheduling

Again the datastructures must not be altered. That is why the only possibility to loose the priorities is to migrate tasks between priority levels. Migration is achieved by inserting the task into a priority level different from the priority level the scheduler demands. Since the resulting priority level is calculated in the `enqueue_task()` function, a modification of this calculation denotes no changes beyond the scheduling function. In this part $O(n)$ algorithms can be used, though with great care, since the `enqueue_task()` function is a basic part of the scheduler and shouldn't get too bloated. But this offers the possibility to analyse the surrounding priority levels, if not the complete priority array, for possible optimisations. If such a great effort is justified depends on the costs of frequency adjustments and the incidence of occurrence of frequency adjustments. Former can help to make a decision

before DSS is started, but the latter can only be determined at runtime. This means, that there has to be a history of changes of the priority array and a function, which analyses this history. The results of this analysis provides a basis for the answer of the question how wide the `enqueue_task()` function should spread.

Different from the implementation of loose priority scheduling in Section 3.1.2 this case needs more attention. Although the priority scheduling restrictions are eased, the scheduler must not disregard priorities. Migrations between priorities have to be well founded and must not exceed at certain scope. How many priority levels a task can be boosted up depends on the application. It is even possible, that the application allows the downgrading of certain or all tasks. But in general, this behaviour is not desirable. In this thesis, only upgrading a tasks priority is allowed. The scope of the upgrading process involves only one priority level. This restriction is a tradeoff between the saving of frequency adjustments and the complexity of the scheduler. Additionally, the policy of priority scheduling remains mostly intact with this implementation since a tasks runs with a maximum priority of $priority_{task} + 1$.

A problem with greater scopes of migration is the increasing amount of wandering tasks. With each task that gets enqueued the structure of the priority array changes and initiates a process of analysis, followed by a possible reorganisation of multiple tasks. This process starts at least every time when a task used up its timeslice or when a new task is created. Possible blockings of tasks and tasks that give up the CPU by themselves introduce more processes of analysis and reorganisation.

Another problem with wandering tasks is the violation of priority scheduling. Even if the migration only takes place within the fixed scope, it is possible that a task can get a too high priority if no precautions were made. It is important, that no task gets two priority boosts or that the sum of the priority boosts does not exceed the limit. This can be solved by not touching the tasks priority attribute. Instead of changing it, the task only gets inserted into the appropriate queue. Then, it can easily be determined how much priority levels the task is already boosted by $|nr_{queue} - priority_{task}|$. Since there is no further checking of priority values, the task gets scheduled with this priority. Even if its real priority is different. This solution also hides the migration from the user. The tasks keep their given priorities and even changes to them are no problem.

Chapter 4

Evaluation

In this thesis, only the implementation which breaks complexity restrictions and keeps priority scheduling (covered in Section 3.2) is evaluated. For that purpose two simple programs were written. One called *alu*, which uses only arithmetic operations and fits completely in the cache. And *mem* which has only memory accesses and every access results in a cache miss. Therefore these programs are easy to classify by the scheduler. Furthermore, the programs can call the system call `nanosleep()` periodically with configurable intervals. It is called with the time value of one and a half timeslice. This enables the simulation of tasks with high and low preemption rates.

In all tests the operating system runs a minimum of additional tasks. As most important representative the sshd daemon which is used as the sole possibility of supervision. All additional tasks in the system are low frequency tasks. If the system is idle, no frequency adjustments were performed. Only the *alu* tasks can cause frequency adjustments.

4.1 Effective Operation

This test consists of a fixed number of tasks with an preemption rate of 0. This means, that all those task will use up their timeslice if they are not blocked for external reasons. The fraction of high frequency tasks is varied and the amount of frequency adjustments is measured as percentage of all context switches which occurred during runtime. The results can be seen in Fig. 4.1.

The amount of frequency adjustments the standard scheduler causes scales with the percentage of high frequency tasks. Reaching its maximum at 50% high frequency tasks with nearly 20% frequency adjustments. The optimised scheduler performs as expected. It also reaches its maximum at 50% but with

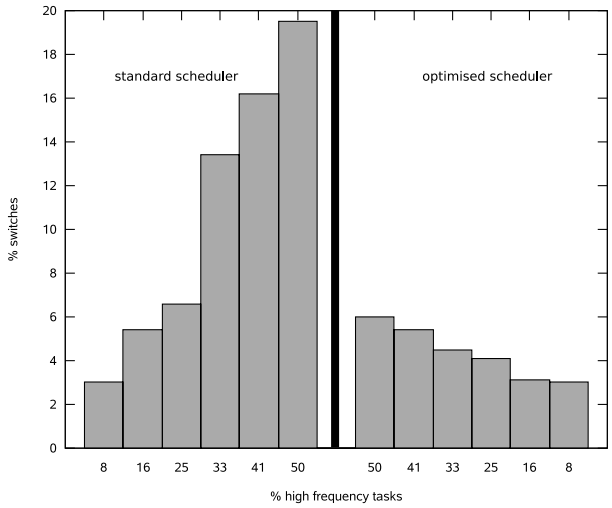

Figure 4.1: Proof of effective operation

only 6% frequency adjustments, which is less than a third of the standard scheduler. In theory, the optimised scheduler should operate at a constant rate, but this behaviour was not detectable in practise. This is because of the dynamic priorities the scheduler assigns to the tasks internally. Since this evaluation is based on Strict Priority Scheduling, only tasks within the same (dynamic) priority level are related to each other. By assigning dynamic priorities, the scheduler can spread the tasks over several priority levels. Another disturbance variable is given through the additional tasks running on the system. Since there are only additional tasks of low frequency, the impact on the results increases with the percentage of high frequency tasks. The more high frequency tasks are running, the higher is the probability that such an additional task runs before or after a high frequency task. The additional tasks have, due to their non interactive characteristics, different dynamic priorities and therefore can't be sorted with the significant tasks.

Overall the optimised scheduler performed well and the results can be seen as a proof of effective operation. Further evaluation will test the performance and the saving potential of the implementation under certain circumstances.

4.2 Savings At Higher Preemption Rates

For this test both tasks, *mem* and *alu*, were modified to periodically call the `nanosleep()` system function which puts the calling task to sleep for the specified amount of nanoseconds. In this case, the sleeptime is exactly

one and a half timeslice. One and a half timeslice is long enough to let the scheduler activate another task and short enough to possibly preempt a running task. The `nanosleep()` function operates with an accuracy of 10ms which is sufficient in this case.

Higher preemption rates are difficult to handle for the implemented sorting algorithm since they highly affect the internal dynamic priorities. This is because of the scheduler uses a heuristic to determine interactive tasks which is based on the time the tasks spend sleeping. Since interactive tasks get a priority boost and this boost depends on their interactivity rating, the set of tasks can get spread over several priorities.

The tested preemption rates cause sleeping times of 0.5%, 4% and 20% of the total runtime. The 0.5% sleeptime nearly doubles the context switches during execution compared with a preemption rate of 0. The 20% sleeptime causes ten times the context switches of the preemption rate of 0. For comparing the results of this tests, the efficiency of the sorting algorithm is measured in frequency adjustments in percentage of the overall context switches.

The first measurement is done with a fixed set of six tasks. Half *mem* tasks and half *alu* tasks. The results can be seen in Fig. 4.2.

The results of this test certify the optimised scheduler a lesser sensitivity to high preemption rates than the standard scheduler has. The optimised scheduler also has more frequency adjustments with rising preemption rate, but scales not as well with them as the standard one. The step from 0.5% sleeptime to 4% sleeptime has nearly no effect to the rate of frequency adjustments for the sorting scheduler. The standard scheduler reacts with nearly 5% more frequency adjustments on the higher preemption rate.

As mentioned before, the probability of a frequency adjustment decreases slightly with the number of concurrent tasks in the system. This decreasing has an asymptote of $\frac{1}{2}$. In the next test this fact has to be considered. This time, the same setup as in the last test applies, but there are 12 concurrent tasks now. The results can be seen in Fig. 4.3. Both schedulers perform better at no preemption than with six tasks and no preemption. This is the effect described above. But the higher the preemption rate gets, the lesser is the impact of this effect. At 5% sleeptime the unsorted scheduler with 12 tasks produces more frequency adjustments than with six tasks at the same sleeptime. This indicates, that the unsorted scheduler can not use the positive effect of more tasks with higher preemption rates. However, the sorted scheduler performs well. It causes less switches at no preemption and at higher preemption rates it uses the possibilities and produces less switches. The jump from 4% to 20% even does not cause any more switches. With six tasks, this jump was more difficult to handle for the scheduler.

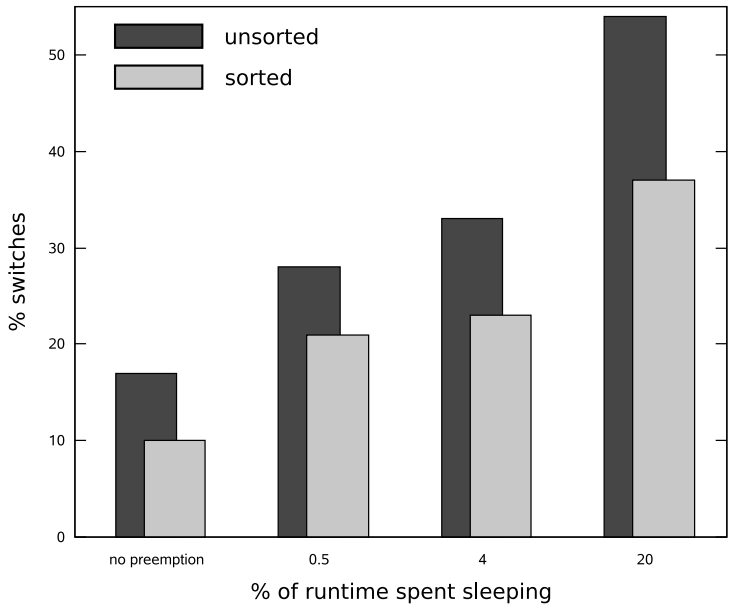

Figure 4.2: Six tasks with different sleeping times

4.3 Performance

The results of Section 4.2 state DSS effective operation and robust savings. But as important as savings are, they are useless if the saving algorithm renders the system useless. Therefore some measurements concerning the runtime of the tasks was made. The built-in `time` command of the bash gives details about the real overall runtime, the time the process was running and the time operating system functions were running. These values are based on the real time clock on the embedded board and therefore not affected by the frequency adjustments.

With six tasks the runtime for all tasks to complete shows a deviation of roughly one second which the sorted scheduler performs faster. This time saving is not related to the saving of frequency adjustments. The evaluation platform can perform frequency adjustments within 0.1ms which means, that for the saving of one second, 10,000 frequency adjustments would have to be saved. In the setup with six tasks, the maximum amount of saved frequency adjustments within one measurement does not exceed 6,000 frequency adjustments. This results in a maximum time saving of 600ms in the best case. This value is only valid for the highest preemption rate. With decreasing preemption rates this value also decreases until it reaches 30ms with no preemption. But the time savings seem to stay constant. This time advantage seems to have other reasons. One explanation is better usage of the

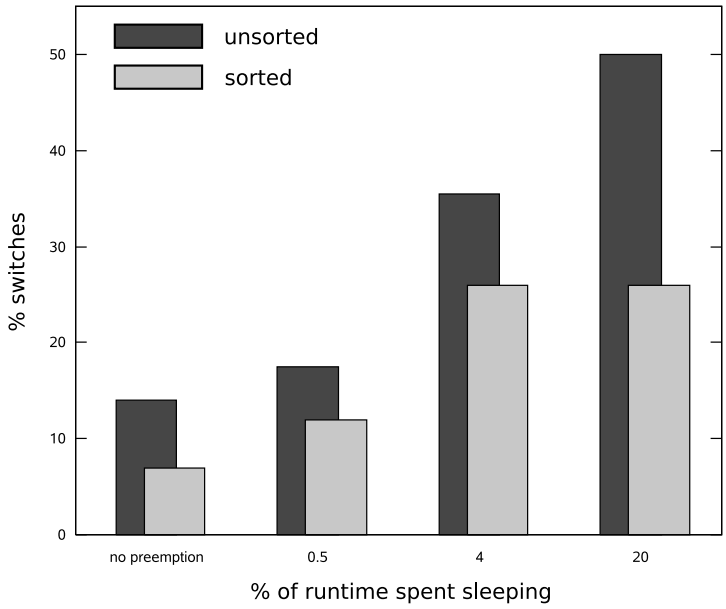

Figure 4.3: 12 tasks with different sleeping times

instruction cache. Since the test is ran with identical tasks, they share the same machine code. If the instruction cache has not enough cache lines to hold both tasks in it, an alternating scheduler queue causes instruction cache misses every context switch. Memory accesses at every context switch can make a significant difference in time performance.

At fist sight, the situation with 12 tasks seems to be similar. There is also a time advantage for the sorting scheduler at all preemption levels. The saved time is again much too high to be traced back to saved frequency adjustments. But at closer inspection the time value is not constant at all preemption levels. There is a small fraction of the time savings which scales with the saved frequency adjustments. This proves, that with 12 tasks there is already a measurable time advantage for the sorting scheduler which is due to saved frequency adjustments. Considering the short time a frequency adjustment takes on the evaluation platform (0.1ms), this is a remarkable result.

These results get more impressive with wait cycles introduced in the frequency adjustment routine. With frequency adjustments of ten times the original value the runtime of the schedulers drifted apart. The slower frequency adjustments, which now took a complete millisecond to perform, boosted the time savings. With 12 tasks the sorting scheduler now performed 22% faster at the highest preemption rate than the standard scheduler. Even with no preemption the sorting scheduler performed 8% faster.

To determine the overhead introduced by DSS, a special test setup was built. A set of tasks was prepared in which no task shares the machine code of another. This eliminates the positive effects of better instruction cache usage. By assigning different preemption rates to the tasks, a wide spreading of the tasks over the schedulers queue is achieved. Therefore the implementation of DSS was unable to sort the tasks. This is because nearly every task had a different dynamic priority and therefore was in a different queue. The test showed, that DSS was not able to save any frequency adjustments. The runtime of the sorted scheduler was consequentially longer than the one of the standard scheduler. The results show, that the introduced overhead does not exceed 3% in time.

Chapter 5

Future Work

The implemented prototype of DSS performed well and showed effective operation. But there is room for improvement.

An implementation of the scheduler described ins Section 3.1 with loosened priority scheduling is very promising. It wouldn't suffer from the scheduler dynamic priorities. A second approach to this would be to eliminate the capability of dynamic priorities. But then, the scheduler wouldn't be aware of interactive and non-interactive tasks. This may or may not be desired in the application.

There are also possibilities to make the scheduler more autonomous. It could determine by itself which frequency domains are promising and which are not. This evaluation could even happen at runtime. For example, the scheduler could keep track how often frequency domains were used. If it encounters a frequency domain which is very seldom used or only used by few tasks, it could decide that this domain should not be used any more. In the implementation of Section 3.1 the shut down of a complete frequency domain could be done in very short time by simply altering some pointers. The switched off frequency domain could even be reactivated if there are again enough tasks.

The scheduler could be more generalised and an API could be defined to access the schedulers facilities. Support from application level is not needed, but depending on the application it could help the scheduler a lot to make decisions. For example, it could be useful to define the thresholds between frequency domains out of userspace or to switch DSS off for a certain time. It may also be useful to enforce the assignment of one or more tasks to a certain frequency domain.

Chapter 6

Conclusion

Double-sorted Scheduling is a good approach to encounter the situation of Process Cruise Control environments. PCC has a high potential in power saving but its broader acceptance is prevented by the lack of hardware support. Most platforms already support power saving facilities. But regardless of the way those power saving facilities are implemented, their appliance costs time and power. In most cases these costs are too high because PCC makes heavy use of frequency adjustments. This problem can be solved in many cases by the usage of Double-sorted Scheduling.

DSS proved to reduce the performed frequency adjustments significantly. In PCC environments it will reduce the power consumption caused by frequency adjustments. Also the time overhead which is caused by them will be reduced. The small overhead which is introduced by DSS is bearable even in worst case scenarios. By the techniques mentioned in Chapter 5 this overhead even can be reduced.

The saving potential of a PCC system rises with the number of context switches. Especially interactive tasks tend to cause more context switches because of a higher preemption rate. Therefore DSS is particularly suited for systems with higher preemption rates. But even without preemption, DSS shows good performance and is therefore also usable with more static systems.

Even systems without special high speed frequency adjusting facilities can utilise PCC combined with DSS to achieve a lower power consumption and keep the CPU cooler without performance loss greater than a certain percentage (e.g. with 10% loss compared to the execution at the highest clock speed).

Bibliography

[1] Josh Aas. Understanding the linux 2.6.8.1 cpu scheduler. Technical report, Silicon Graphics Inc. (SGI), 2005. `http://josh.trancesoftware.com/linux/`.

[2] Frank Bellosa and Andreas Weißel. Process cruise control. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, Grenoble, France*, 2002.

[3] Uwe Brinkschulte and Theo Ungerer. *Microcontroller und Mikroprozessoren*. Springer Verlag, 2002.

[4] Intel Corporation. *Intel XScale Core Developer's Manual*, 2004. `http://download.intel.com/design/intelxscale/27347302.pdf`.

[5] Karo Electronics. *Product Homepage for Starterkit 3*. `http://www.karo-electronics.de/starter-kit-3-pxa270.html`.

[6] Robert Love. *Linux Kernel Development*. Sams, 2004.