Adaptive Resource Management for Uncertain Execution Platforms

Lindberg, Mikael

2010

Document Version:
Publisher's PDF, also known as Version of record

Adaptive Resource Management for Uncertain Execution Platforms

Mikael Lindberg

Department of Automatic Control
Lund University
Lund, September 2010

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

ISSN 0280–5316
ISRN LUTFD2/TFRT--3249--SE

Printed in Sweden,
Lund University, Lund 2010

Abstract

Embedded systems are becoming increasingly complex. At the same time, the components that make up the system grow more uncertain in their properties. For example, current developments in CPU design focuses on optimizing for average performance rather than better worst case performance. This, combined with presence of 3rd party software components with unknown properties, makes resource management using prior knowledge less and less feasible.

This thesis presents results on how to model software components so that resource allocation decisions can be made on-line. Both the single and multiple resource case is considered as well as extending the models to include resource constraints based on hardware dynamics. Techniques for estimating component parameters on-line are presented.

Also presented is an algorithm for computing an optimal allocation based on a set of convex utility functions. The algorithm is designed to be computationally efficient and to use simple mathematical expressions that are suitable for fixed point arithmetics. An implementation of the algorithm and results from experiments is presented, showing that an adaptive strategy using both estimation and optimization can outperform a static approach in cases where uncertainty is high.

Acknowledgements

I would like to thank the Department of Control for giving me the opportunity to pursue a PhD degree. In particular, I want to thank my supervisor, Karl-Erik Årzén, for supporting me in my work and believing in my ideas. I would also like to thank Johan Eker and Anton Cervin for their discussions and encouragements.

I would also like to thank the following people:

Anders Robertsson, as he was instrumental in tricking me into accepting this position.

Eva Westin, for her genuine concern and help.

Karl-Johan Åström, for sharing his experience and enthusiasm for research.

Toivo Henningsson, for his insights and friendship.

Vanessa Romero Segovia, for being a good colleague and a forgiving roommate.

Leif Andersson, for his LaTeX-expertise and coffee time discussions.

Linus Pizunski and Marcus Skans at Axis, for supporting my research with equipment.

My beloved wife, Mirjam, for being there for me when work weighed heavily on my shoulders, and for supporting my decision to return to school.

My children, Mattis and Rebecka, for giving me ample opportunities to study systems under overload conditions.

This research has partially been funded by the VINNOVA/Ericsson project "Feedback Based Resource Management and Code Generation for Real-time System" , the EU ICT project CHAT (ICT-224428), the EU NoE ArtistDesign, the Linneaus Center LCCC, and the ELLIIT strategic research center.

Mikael

Contents

1

Introduction

1.1 Background and motivation

The central theme of this thesis is resource management for embedded systems. While embedded systems performance is typically evaluated through static worst case analysis methods, recent trends in hardware and software development have rendered these methods more and more difficult to apply. Increasing uncertainty in system properties, time varying resource demands and supply, and focus on unit cost make conservative methods unattractive for use in many situations.

The strength of real-time scheduling theory lies in the guarantees for system performance that it can provide. Conversely, the weakness comes from the reliance on prior information and the combinatorial nature of the problem formulation.

The approach taken in this thesis is to explore estimation- and feedback-based methods, thereby reducing the need for exact prior knowledge. By doing so, we consciously sacrifice the absolute guarantees provided by hard real-time theory. This is justified by the inherent robustness to transient performance loss in many systems, in particular media processing and non-critical control applications.

The methods presented draw upon several theoretical disciplines to provide a framework for resource management, including

- control theory
- system identification

- convex optimization
- reservation based scheduling

The target systems are embedded or cyber-physical systems which are such that resource constraints and uncertainty would make worst case methods infeasible.

1.2 Outline

This thesis is organized into chapters as follows. In Chapter 2, the formal definition of the problem is given. Chapter 3 then discusses relevant and related research. Some enabling technology is then presented in Chapter 4. Chapter 5 introduces modeling and estimation techniques suitable for cyber-physical systems with uncertain parameters. Chapter 6 presents an algorithm to solve the feedforward allocation problem and discusses its performance and qualities. Chapter 7 presents techniques for feedback disturbance rejection and regulation of structures with mixed physical and computational components. Experimental results are presented in Chapter 8 and Chapter 9 then concludes with discussion and remarks.

The thesis is based on the following publications:

Mikael Lindberg. A survey of reservation-based scheduling. Technical Report ISRN LUTFD2/TFRT--7618--SE, Department of Automatic Control, Lund University, Sweden, 2007.

Mikael Lindberg. Constrained online resource control using convex programming based allocation. In *Proceedings of the 4th International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID 2009)*, San Francisco, CA, USA, 2009.

Mikael Lindberg. Convex programming-based resource management for uncertain execution platforms. In *Proceedings of the Workshop on Adaptive Resource Management (WARM 2010)*, Stockholm, Sweden, 2010.

Mikael Lindberg. A convex optimization-based approach to control of uncertain execution platforms. In *Proceedings of 49th IEEE Conference on Decision and Control (CDC 2010)*, Atlanta, GA, USA, 2010.

Mikael Lindberg and K.E. Årzén. Feedback control of cyber-physical systems with multi resource dependencies and model uncertainties. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS 2010)*, San Diego, CA, USA, 2010.

1.3 Contributions

This thesis contains the following contributions

- a model for resource allocation for rate-based software components is proposed,
- an algorithm for solving convex allocation problems suitable for media platforms has been developed,
- a control scheme for software components with multi resource dependencies is proposed.

2

Problem formulation

This chapter introduces the problems treated in this thesis. The domain of portable multimedia devices serves as a basis for deriving the formal definition. An example from the control domain is added to provide additional motivation, in order to show that the resource management problem is not unique to multimedia applications.

Special attention is given to the point of view of platform providers, who would like to increase system robustness without posing overly prohibitive requirements on applications. In this context, a platform is a hardware and software design that supplies base functionality and resources. Complete products are then constructed by adding components built with a development kit which is distributed together with the platform. The Google Android platform is a recent example [7].

2.1 Example 1 — Portable media player

The first motivating example comes from the domain of portable media players, e.g. smart phones, mp3-players and portable video games. These devices often include additional functionality, though media processing tends to be the most resource demanding. The exact needs are hard to predict as the behavior of most algorithms in this domain are highly data dependent.

Figure 2.1 exemplifies the CPU utilization for decoding two versions of the same video stream with perfect playback. Note how the resource demands are significantly larger for the high resolution stream and

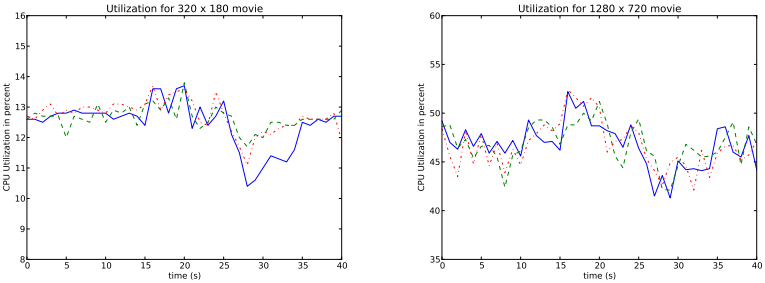

Figure 2.1 Resource requirements for decoding two versions of a H.264 movie on an Intel Core 2 Duo based MacBook. The left plot represents a movie encoded in SDTV-resolution while the right represents one encoded in HDTV resolution (720p). The experiment was run three times with varying results, as illustrated by the the three curves in each plot. The utilization measure represents the percentage of time the decoding process had exclusive access to the CPU measured with a sliding 1 second time window.

how the levels change over time. There is also a visible trend, with a slight increase in resource demand around 15 seconds and a dip at around 28 seconds. This is evident for both streams and is caused by the encoding standard, which uses different levels of compression depending on the level of motion in the source video. The experiment was run three times for each stream with different results each time, this despite the fact that decoding a specific movie stream is a deterministic sequence of operations. The reason for this is that modern hardware relies heavily on prediction and heuristics to minimize effects of memory latency and pipeline bubbles. Should these strategies fail, the system takes a performance hit. As the system doing the decoding in the example is executing a large number of background tasks in addition to the decoder, system state will vary from run to run.

The problem stated in traditional real-time terms would be to check the schedulability of a set of periodic tasks $\tau_0, ..., \tau_N$ with the corresponding periods $T_0, ..., T_N$ and worst case execution times $C_0, ..., C_N$. Assume for simplicity that each task has a relative deadline equal to its period. If the scheduling policy used is Earliest Deadline First (EDF)

and the system is a single core machine, the task set is schedulable if

$$\sum_{i=0}^{N} \frac{C_i}{T_i} \leq U_b \tag{2.1}$$

where U_b is the utilization bound. U_b depends on parameters such as the cost of context switches and is normally close to 1. For a media player task, the period would be equal to the frame rate at which the movie is encoded, which is easily accessible from the stream meta data.

If the test passes, deadlines will be met and the system performs as intended. If the test fails, the system is overloaded and tasks will miss their deadlines. Traditionally, a system should not admit tasks that will cause overload, but given the uncertainties mentioned above, it is not clear if enough information would be available to make such decisions. Specifically,

- the set of active tasks will change over time as the user enables different functions,
- the resource requirements of a task can vary greatly depending on input and
- the properties of 3rd party software might not be available during system design.

A system designer could choose to restrict the use cases supported by the device in order to counteract some of these points, but this could render the product unattractive to consumers. It is also probable that a user would rather have access to a function running with degraded performance than being denied this functionality completely. Therefore, the all-or-nothing property of hard real-time formulations are not suitable to this problem domain.

This thesis chooses to focus on the following aspects of the problem:

- **Uncertainty in hardware and software.** The reliance on prior information in traditional real-time systems is increasingly a bottle neck in designing feature rich embedded systems. Rather, the approach taken here will be to try to model the resource consumption of a system and estimate model parameters online.

This has the effect of reducing the work needed by both hardware designers and software developers, thereby reducing time to market for both product and 3rd party add-ons.

- **Allocation under overload conditions.** Portable devices will want to use as low power components (CPUs, batteries, radio transmitters etc) as possible. A cost efficient system will therefore often run near or in overload conditions in order to save power and unit cost.

  This thesis will strive to provide resource management strategies that effectively manage systems under both nominal and overload conditions.

- **Non-restrictive assumptions on software components.** One way to simplify the work of the system designer would be to shift some of the burden to the 3rd party developers, e.g. requiring them to supply worst case execution time estimates and other detailed resource demand information. As such figures requires technical expertise with the target platform, this could impede the supply of attractive 3rd party software.

  The application framework should put clear but lenient requirements on developers in order to make the platform simple and attractive to develop for.

2.2 Example 2 — Mobile robotics

Mobile robotics is another field where resource management is key to performance. Not only are computational resources scarce, there are usually many subsystems besides the computer that have to compete for power. The drive to increase autonomy, include more and more functions while simultaneously pricing them competitively is making the resource constraints more and more pronounced.

As the constraints grow increasingly severe, it becomes interesting to see if a holistic view on resource management can improve operational range or enable units to be built with cheaper components. By combining hardware and software models, a hybrid system description that is popularly referred to as a cyber-physical system (CPS) emerges.

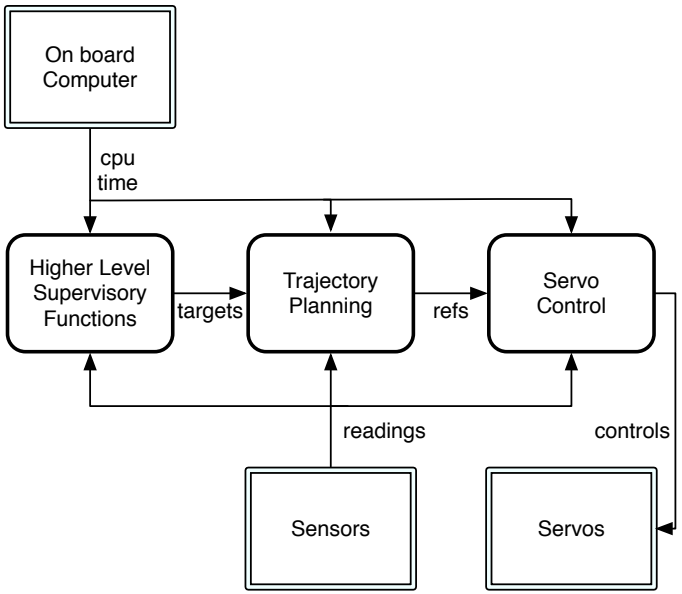

Figure 2.2 A schematic over a simple mobile robot system. The square blocks represent hardware functions that require power, the round-corner shapes are software functions that require cpu-time to run. The complex dependency situation makes it non-trivial to determine how to prioritize in a situation with insufficient resources to run the system at nominal performance.

The objective here is to study the interactions between hardware and software and through this learn how global system performance is affected by the dynamics of both.

Cyber-physical systems, particularly in the mobile robot case, share many properties with the media device class discussed previously. In particular, this includes a tolerance for degraded performance in subsystems. This makes it an interesting domain for studying resource allocation trade-offs.

Consider the schematic presented in Figure 2.2. The functions in a robot can be realized in hardware or software, making the resulting dependency graph include connections both from hardware to software

and vice versa. In order to handle a situation where some resource is scarce, a model that can express the total performance dynamics of the system would be useful. It is the aim of this thesis to present some initial thoughts on how this can be accomplished.

2.3 Overall goals

This section summarizes the problems derived from the examples. The objective of this work is to investigate how resource management can be done in situations where uncertainty in both demand and supply makes static methods infeasible. An effort is made to consider systems where components have dependencies, as this topic is less studied. While initially the work was focused on CPU resources inside a computer, the robot example makes it evident that if the availability of the CPU resource depends on the dynamics of other resources, a model encompassing both domains is desired. It is an explicit goal that the methods presented are realistically implementable on power constrained systems and as such cannot be too resource consuming in themselves.

3

Related Research

This chapter introduces the prior research that this thesis is based on. First a few central concepts are introduced. Then follows a historical survey of reservation based scheduling (RBS), one of the enabling technologies for this work. The theory is derived from both its real-time scheduling roots and its queueing theory counterpart.

Using the RBS formulation, a resource management policy can be posed as the solution to an optimization problem. Such methods are introduced in the section on optimal resource management, which has its roots in operations management. It is worth noting that these techniques are mostly feed-forward in style, i.e. the allocations are calculated based on models rather than on-line measurements. Some important works in the alternative branch, feedback and adaptation, are then discussed.

As this thesis will largely be about on-line strategies, finding ways to solve optimization problems reliably and efficiently is a central part. The domain of convex optimization lends itself to this type of formulations and some examples of this is introduced in Section 3.7.

Finally a brief survey of event based control of continuous systems is found in Section 3.8, as this topic is important to understanding the interaction between computers and physical phenomena.

This chapter is based on the technical report [50].

3.1 Important concepts

This section introduces important concepts that will be used in the presentation of relevant research.

Temporal Isolation

A highly desirable property of the RBS approach is that a task that has reserved a specific amount of a resource should have access to this regardless of what other tasks are running on the system. This is called temporal isolation and makes very good sense for the continuous media type of applications we have used as example so far. Video playback should continue unaffected if other applications are started (or if the OS spawns tasks in the background).

Components and composition

In order to handle the complexity of large systems, the ability to gather parts into component structures that are closed under composition is vital. Threads with priorities, the building blocks of traditional operating systems, do not compose [48]. By using hierarchical RBS techniques, it is possible to enforce temporal isolation and thereby create groups of threads with essentially the same outside properties as the atomic thread. This enables component wise testing and verification, but also removes the need to explicitly know the structure of 3rd party software.

Timing sensitive applications

In real-time situations, the timely completion of tasks is important. Normally, if a task has a real-time deadline, it is assumed to function nominally if the deadline is met and fail if the deadline is missed.

In soft real-time problems, deadlines are allowed to be missed occasionally and for applications in this domain it is interesting to discuss how the performance is affected by this. Applications where the performance depends on how well the deadlines are met are called *timing sensitive applications*.

Graceful QoS degradation

While it is possible to create an admission policy where tasks that would make the scheduler unable to sustain reservations would be de-

nied, this might not be desirable from a user perspective. For consumer applications, it can be preferable to have a slight (and predictable) degradation in QoS as opposed to being denied starting applications altogether. This becomes even more evident in embedded systems where resources are scarce. Consider a mobile phone user engaged in a video conference call when a SMS message comes in. Most would be content to have some slight degradation in video quality while still being able to accept the SMS message. If the playback application in question is designed to be aware of its resource allocation, it can be assigned lower QoS in an as graceful way as possible.

3.2 Reservation Based Scheduling

This form of scheduling is used together with a class of real-time applications whose quality of output depends on sufficient access to a resource over time. Such applications are difficult to handle in terms of traditional hard real-time theory. The typical situation involves some type of continuous media task (playback or encoding), and it was in fact the need for support for media software that ignited interest in the field. This was in the early '90s when computers started to make their way into mainstream media production and consumption. While this remains the favored use case also today, other forms of computing can also benefit from RBS. This includes classes of systems that have traditionally been considered hard real-time. Before discussing the different algorithms for RBS, the problem background will be presented in more detail.

Origins

The case for Reservation Based Scheduling (RBS) was perhaps most famously made by Mercer et al in [60]. The paper discusses processor reserves as a way to describe computational requirements for continuous media type applications and some challenges when this is implemented on a microkernel [78] architecture.

The basis of the analysis is periodic tasks, characterized by execution time C and period T. Mercer observes that C is likely difficult to compute and suggests that the programmer supplies an initial estimate and that the scheduler then measures and adjusts the estimate

(a feedback scheduling technique). The paper also introduces the concept of task CPU percentage requirement $\rho = C/T$ and the expected execution time of a task running at rate ρ as $D = C/\rho$. It is worth noting that these definitions are very close to what present day theory refer to as *bandwidth* and *virtual finish time* respectively.

Although [60] is frequently cited, many of the aspects of resource reservations and continuous media had already been discussed in earlier works.

Herrtwich presents a number of insights around the problem in the paper [41] from 1991. Like in [60], the use of conventional scheduling schemes is deemed as inefficient and perhaps not serving the user needs. Herrtwich also brings up the importance of preventing ill behaved applications from disturbing others (temporal isolation) and that the user might prefer graceful degradation of QoS to being prevented from starting new applications when the system is overloaded.

Herrtwich paper quotes heavily from the even earlier work [6] from 1989, which details how media type applications can be served by a resource reservation scheme based on preemptive deadline scheduling. The concept of resources is here extended to include not only CPU but also disk, networking and more. [6] presents more theory but lacks some of the softer insights in Herrtwich's work.

Taking a queue from telecommunication

In what seems like unrelated work, the telecommunications society was around this time researching queuing algorithms which, it would turn out, share properties with process scheduling problems. [27] discusses the sharing of a link gateway between clients using the Weighted Fair Queueing algorithm (WFQ) citing "protection from ill-behaved sources", essentially temporal isolation, as one of its main advantages.

The central idea of the algorithm is to schedule the jobs in the order they would have been completed by a weighted round-robin (WRR) scheduler. The job finishing times, though not named as such in this paper, are in subsequent works called *virtual finish times*. In other words, the scheduler decisions are based on how the task set would behave if each task was running on a private platform with a fraction of the actual system speed.

In this manner, the fairness property of the WRR scheduler and the finishing order of the jobs are preserved while the context switching

overhead is reduced. Apart from being one of the earliest examples of temporal isolation, it introduces the notion of basing the scheduling decisions on virtual time metrics. A similar scheme was presented in a thesis from 1989 by Zhang [85].

Virtual time

The WFQ scheme and how virtual time can be used is discussed in papers in the decade following [27]. One of the more comprehensive is [68], which further investigates how a generalized processor sharing scheme (GPS) can be approximated using virtual time techniques. Though initially a queue theoretical result, [68] is commonly cited in real-time scheduling papers as well. For example, the virtual time concept is used in the current Linux scheduler, which is described in detail in Section 3.2

Hierarchical Scheduling Structures

One desirable property in consumer grade systems is to be able to mix real-time applications with regular applications. Often this leads to a construction with a hierarchy of schedulers, typically with some hard-real time scheduler on top and soft real-time and regular best-effort schedulers underneath. In [83] Pawan Goyal et al suggests using a tree structure where each node is either a scheduler node or a leaf node. Parents schedule their children until leaf level, where the regular tasks sit, is reached. An example is provided in Figure 3.1. The paper also describes a variant of WFQ called Start-time Fair Queuing (SFQ) that provides better guarantee of fairness if the amount of available processing power fluctuates over time.

The hierarchical approach to scheduling is also proposed by other groups. The RTAI/Xenomai extensions to Linux runs a RT-scheduler as root and the Linux operating system as a thread. The structure is similar to the one proposed in [83]. The Bandwidth Server class of RBS algorithms, detailed in Section 3.2, also use a hierarchy of schedulers, typically with an Earliest Deadline First (EDF) scheduler [14] on top. Hard real-time tasks are scheduled directly by the EDF algorithm, while soft real-time tasks have dedicated "servers" that dynamically set their deadlines to achieve CPU reservations. Regular applications can be scheduled by a separate server. Lipari et al presented a hierarchical Constant Bandwidth Server construct called the H-CBS in [56] in 2001.

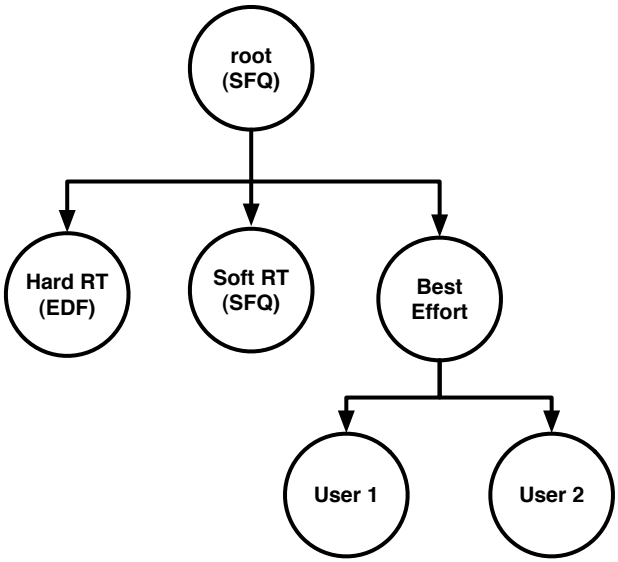

Figure 3.1 Hierarchical structure with schedulers. Note that SFQ is used on more than one level.

The choice of top level schedulers becomes more critical in the case of insufficient resources. Fixed priority servers favor high priority tasks while EDF schedulers will spread out the effects [20] [15].

Bandwidth Servers

The concept of bandwidth servers was derived from Dynamic Priority Servers (DPS) by Buttazzo [14]. DPS is a method to accommodate aperiodic or sporadic tasks in fixed priority systems, essentially through a hierarchy of schedulers. The Priority Server is a periodic task with a specified execution time. Arriving aperiodic tasks are placed in a queue and executed by the Priority Server when it is scheduled to run. In the original formulation, unused capacity is just lost.

Buttazzo brought the concept of a server presiding over a predetermined amount of CPU capacity to the dynamic scheduling algorithms. The Dynamic Priority Exchange Server (DPE) and the Total

Bandwidth Server (TBS) [75] were the first formulations using EDF as a root level scheduler. The objective was still handling aperiodic tasks and a lot of theory concerned handling of unused bandwidth. In 1998, Buttazzo and Abeni published [1], which introduces the Constant Bandwidth Server (CBS). By then, the Continuous Media (CM) problem had already been addressed using the Bandwidth Server metaphor by Kaneko et al in a paper from 1996 [45].

Constant Bandwidth Server

The CBS formulation is a popular construct for software reservations and is explained further in this section.

Consider a set of tasks τ_i where a task consists of a sequence of jobs $J_{i,j}$ with arrival time $r_{i,j}$. Let C_i denote the the worst case execution time (WCET) in the sequence and T_i the minimum arrival interval between jobs. For any job, a deadline $d_{i,j} = r_{i,j+T_i}$ is assigned.

A CBS for the task τ_i can then be defined as:

- A budget, c_s, and a pair (Q_s, T_s) where Q_s is the maximum budget and T_s is the period. The ratio $U_s = Q_s/T_s$ is called the server bandwidth. At each instant, a fixed deadline $d_{s,k}$ is assigned with the server with $d_{s,0} = 0$.
- The deadline $d_{i,j}$ of $J_{i,j}$ is set to the current server deadline $d_{s,k}$. If the server deadline is recalculated, then so is the job deadline.
- When a job associated with the server executes, c_s is decreased by the same amount.
- When $c_s = 0$ the budget is replenished to the value of Q_s and the deadline is recalculated as $d_{s,k+1} = d_{s,k} + T_s$. This happens immediately when the budget is depleted, the budget cannot be said to be 0 for any finite duration.
- Should $J_{i,j+1}$ arrive before $J_{i,j}$ is finished, it will be put in a FIFO queue.

CBS-hd One possible drawback with the CBS algorithm when dealing with things sensitive to deadline overrun is that although the server is completely replenished when budget c_s is exhausted, the new deadline might be too far into the future. The CBS^{hd} algorithm

changes the replenishment rule to better handle this. If $c^r_{i,j}$ is the remaining computational need for $J_{i,j}$ when the budget is exhausted, we apply the following replenishment rule:

$$\begin{aligned} &if(c^r_{i,j} \geq Q_s) \\ &\quad c_s = Q_s; \\ &\quad d_{s,k+1} = d_{s,k} + T_s; \\ &else \\ &\quad c_s = c^r_{i,j}; \\ &\quad d_{s,k+1} = d_{s,k} + c^r_{i,j}/U_s \end{aligned} \tag{3.1}$$

This means that if the overrun is less than the budget, the new deadline will be calculated less pessimistically. This is investigated in [17].

The Control Server (CS) Cervin and Eker presented in [19] a modification to the CBS scheme that would make it easier to handle the timing needs of a control application. Though control tasks are typically implemented using hard real-time scheduling, the inherent robustness in feedback control schemes can make hard real-time guarantees unnecessarily expensive. The CS makes the following change to the CBS setup:

- Each task τ_i is associated with a set of segments $S_{i,1}, ..., S_{i,n_i}$ of lengths $l_{i,1}, l_{i,2}, ..., l_{i,n_i}$ such that $\sum_{k=0}^{n_i} l_{i,k} = T_i$
- τ_i has a set of inputs I_i and outputs O_i
- Each set $S_{i,k}$ is associated with a code function $f_{i,k}$, a subset of the inputs $I_{i,k} \in I_i$ and a subset of the outputs $O_{i,k} \in O_i$
- The server has a segment counter m_s

The algorithm is also changed in the following way:

- The server is initiated with $c_s = m_s = 0$.
- When $c_s = 0$ then
  - $m_s := mod(m_s, n_i) + 1$
  - $d_s := d_s + l_{i,m_s}$ and

- $c_i = U_s l_{i,m_s}$

The result of this change is that the server budget c_s is spread out over a number of smaller segments, reducing the uncertainty as to when an input will be read, an output be set or a code function executed. A trade-of will have to be done between jitter and latency. A technique for splitting calculations is shown by which the latency can be reduced. The paper also demonstrates how by forming a cost function including both jitter and latency, an optimum may be calculated.

Rate Based Execution (RBE)

Rate Based Execution was originally proposed by Jeffay and Goddard in [44]. It is presented as a generalization of the sporadic task model by Baruah and Mok [11] and is essentially another scheme for setting the deadlines for the jobs $J_{i,j}$ released by a task τ_i, aiming in this case to limit the number of jobs the task can schedule during some interval. If the WCET of the jobs is known, it is also possible to compute a worst case bound for how much CPU the task will use within a specified time interval. This is slightly different from the CBS algorithm which limits the bandwidth directly.

The paper also provides feasibility conditions for non pre-emptive scheduling and for pre-emptive scheduling but with shared resources.

Proportional Share Scheduling (PS)

Perhaps the oldest time-sharing algorithms for any type of resource is the Round-Robin scheme. By adding weights to the participants in the ring so that they each round get to spend a time proportional to that weight with the resource, you get Weighted-Round-Robin (WRR). Such a division of a resource between participants is called Proportional Share resource management and constitutes a large class of scheduling algorithms. As mentioned before, WFQ is one way to achieve PS, but the class of algorithms derived from FQ is distinctive enough to have been given its own section.

There are several reasons as to why WRR is not as useful for RBS purposes as it might initially seem. While it might work for a general purpose CPU bound application (which is always ready to run and which never blocks), periodic tasks where computation time fluctuates

will not be served well by the WRR scheme. There are a few reasons for this:

- WRR does not handle overruns, it will preempt a task when its allocated time slice is spent. This means that you need to use WCET to determine the size of the reservation, which in turn will lead to a lot of waste.
- WRR does not have any scheme for slack reclaiming. Slack can be consumed by non-RT applications, but other participants in the RR scheme will not benefit.
- For tasks that block, the WRR scheme can result in unwanted levels of latency. The typical I/O-bound task will be blocking most of the time. When it wakes up, it will have to wait for its turn in the RR scheme and perhaps not have time to finish its work. It then has to wait another round before it may complete. These types of tasks are generally very latency sensitive, something which is difficult to handle in plain WRR.
- WRR does not handle sporadic real-time tasks without dynamically recalculating the weights (in which case it is a different algorithm).

There are many alterations to WRR to remedy these problems. The FQ family of algorithms explore one path but there are other well known examples, such as Deficit Round Robin (DRR) [74] or Group-Ratio Round Robin (GR3) [18].

Fair Queueing

Fairness was originally introduced by Nagle in [63] using an informal definition saying simply that a fair algorithm divides the resources between peers equally. The paper also includes what is essentially a prototype of the WFQ algorithm but with little formalism. [27] builds on [63], providing formal definitions and analysis. An algorithm for dividing a resource is defined as fair if

- no user receives more than its request,
- no other allocation scheme satisfying the first condition has a higher minimum allocation and

- the second condition remains recursively true as we remove the minimal user and reduce the total resource accordingly

For applications, the conditions can be expressed in another way. Assume the existence of a finite resource D and n users of that resource. Each user "deserves" a fair share equal to D/n of this resource, but is allowed to ask for less, in which case the difference can be allocated to a user who would like more. Let d_i denote the share a user requests and a_i the share he is given. The maximally fair allocation is then defined so that the share d_f is computed subject to the following two constraints:

$$\sum_{i=1}^{n} a_i = D \tag{3.2}$$

$$a_i = min(d_i, d_f) \tag{3.3}$$

To quantify the fairness of an allocation $\{a_1, a_2, ...\}$ we use a fairness function

$$Fairness = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2} \tag{3.4}$$

where $x_i = a_i / A_i^*$. The fairness will be between 0 and 1, where 1 represents a maximally fair allocation.

Using this metric, we can discuss how fair an algorithm is, how quickly it achieves it and how sensitive it is to fluctuating conditions. More notions of fairness does, however, exist. The formulation above is limited to calculating the overall fairness, but is difficult to apply to specified time intervals. For that, we need a more advanced formulation. In [36], Golestani introduces a notion of fairness based on the concept of normalized service. Let r_i be the service share allocated to a task τ_i and $W_i(t)$ the aggregate amount of service this task has received in the interval $[0, t)$. The normalized service is then $w_i(t) = \frac{1}{r_i} W_i(t)$. An algorithm is then considered fair in an interval $[t_1, t_2]$ if

$$w_i(t_2) - w_i(t_1) = w_j(t_2) - w_j(t_1) \tag{3.5}$$

or, in a more compact notation,

$$\Delta w_i(t_1, t_2) - \Delta w_j(t_1, t_2) = 0 \tag{3.6}$$

for any two tasks τ_i and τ_j that have enough work to execute during the entire interval and fair if this is true for any interval.

Unless work is infinitely divisible and all tasks can be serviced simultaneously, all scheduling algorithms relying on resource multiplexing will be unfair if $t_2 - t_1$ is chosen sufficiently small. The theoretical case that allows $t_2 - t_1$ to go towards 0 is called *fluid resource sharing* and is discussed in [68], that analyzes the Generalized Processor Sharing algorithm (GPS). Note that GPS would be completely fair given both definitions of fairness.

Variations of WFQ

While simple in concept, WFQ suffers from being computationally expensive and sensitivity to fluctuating resource availability. Several alterations to the original algorithm have been proposed to reduce these problems. For instance, the Start-time Fair Queueing approach mentioned earlier was introduced in [83] as one way of increasing robustness to resource fluctuations, while the Self-clocked Fair Queuing scheme [36] removes the need to explicitly calculate the ideal processor sharing solution.

Borrowed Virtual Time scheduling (BVT) Duda et al makes another change to the WFQ scheme in their article on the Borrowed-Virtual Time algorithm [29]. When using FQ for scheduling computer processes, interactivity becomes an issue and the BVT aims to handle just that.

The scheme distinguishes between virtual time, effective virtual time (EVT) and actual virtual time (AVT). The AVT A_i of a running task is increased by its running time divided by the weight r_i. It also accounts for context switch costs by introducing a switching allowance C and switches to task j when

$$A_j \leq A_i - \frac{C}{r_i} \tag{3.7}$$

The scheduler recalculates the AVT

$$A_i := max(A_i, SVT) \tag{3.8}$$

when a task wakes up, where SVT (the Server Virtual Time) is the AVT for any non-blocking task. This way, a daemon type task which spends most of its time sleeping cannot monopolize the CPU once it wakes up. Newly created tasks can be initiated in the same way.

Tasks which are interactive can be allowed to "warp" in time. They have the additional parameters $warp$, W_i, L_i and U_i, which here denotes if it is allowed to warp, the amount of time it can warp, the warp time limit and un-warp time requirement respectively. Warping temporarily reduces A_i, thereby making an warping task appear to have waited to execute longer than it actually has.

A multi CPU extension is presented where all CPUs have their own run-queues where all tasks are included. EVT for a task in a specific CPU run queue is here calculated as

$$E_i := \begin{cases} A_i + M & warp = 0 \\ A_i - W_i + M & warp \neq 0 \end{cases} \tag{3.9}$$

where M is a migration penalty added if τ_i ran on another CPU last.

The BVT algorithm have much in common with the recently introduced Linux scheduler, the completely fair scheduler (CFS).

The Completely Fair Scheduler The Completely Fair Scheduler (CFS) is a Linux scheduler that was introduced by Ingo Molnar in the 2.6.23 release of the kernel. The scheduler is called "The Completely Fair ", but the design document recognizes absolute fairness is impossible on actual hardware. The scheduling principle is simple, each task is given a `wait_runtime` value which represents how much time the task needs to run in order to catch up with its fair share of the CPU. The scheduler then picks the task with the largest `wait_runtime` value. On an imaginary completely fair system, `wait_runtime` would always be 0. This is essentially the BVT approach.

The implementation of this is slightly less simple. Each CPU maintains a `fair_clock` which tracks how much time a task would have fairly got had it been running that time. This is used to timestamp the tasks and then to sort them, using a red-black binary tree, by the key `fair_clock - wait_runtime`. As with BVT, penalties are given for migrating to other CPUs. Weights are also used, but as is common in

POSIX systems they are called nice levels and have the reverse meaning (a nice process would have a low weight). `wait_runtime` is also constrained so that heavy sleepers will not lag too far behind.

In subsequent patches, the group scheduling framework was introduced. In short, it is a hierarchical scheduling scheme where the run queue can be made up of both individual tasks and groups of tasks. The initial intent was to allow fair sharing of the CPU between users rather than tasks. However, the introduction of control groups (see below) made it possible to do arbitrary groupings, thereby making it a simple but flexible tool for CPU reservations.

At the time the CFS was being merged into the kernel mainline, there were several competing initiatives to bring reservations to the Linux scheduler. The winning patch-set introduced control groups, a general system for grouping tasks and annotating the groups with parameters. These parameters could then be used by various kernel subsystems without the need to change the POSIX task model. It is important to note that control groups in themselves do not alter the behavior of the system, they are just an organizational tool. It is up to the respective subsystems to then interpret the parameters. Some examples of control group aware subsystems are

- the CFS Group scheduler,
- the CPU affinity subsystem ("cpusets"),
- group freezer (suspends all tasks in the group) and
- resource accounting.

Adding new control group aware subsystems is at the time of writing the preferred way to introduce new user controllable functionality in the kernel, instead of adding new system calls.

Comparison between CBS and FQ

Having introduced both the bandwidth server and fair queuing approach to RBS, we can now compare the two methods and see how they differ. Such a comparison is presented in [3], which is summarized here.

First we take a look at the interface they provide for reserving bandwidth. The CBS dedicates an absolute share while FQ uses relative shares. FQ can emulate CBS but with the need to dynamically

recalculate the weights when a new task is admitted. FQ algorithms also typically provide bounds on delay, which can be seen as a bound on what deadline requirements a new task can pose. Both schemes have been extended with feedback to adjust weights or bandwidth to achieve some QoS set-point. On the other hand, FQ can more easily be used with mixed real-time and non-real-time tasks.

The run-time properties of the algorithms are also different. CBS does not use quantified time which makes its performance more consistent over varying hardware platforms. FQ is on the other hand simpler to implement. FQ enforces fairness at all times while CBS only guarantees bandwidth allocations between deadlines, making it less conservative. The paper makes the case that FQ is not suitable for media applications as it lacks the notion of task period or deadline, but one can argue that the maximum lag property of an FQ algorithm is a global deadline guarantee, shared by all tasks currently in the system. It is, however, true that the maximum lag often depends on the number of tasks in the system and the distribution of weights, making the temporal isolation property of FQ weaker. The paper also states that FQ would generate many context switches in order to enforce fairness. While CBS will have context switches as a function of the smallest period server, FQ uses a fixed scheduler time quanta for all tasks. However, as seen with e.g. the BVT algorithm, scheduling allowances can be worked in to reduce the number of context switches, at the cost of worse moment-by-moment fairness.

Latency-Rate Servers

In [77], a generalization of different FQ algorithms are proposed. The class of schedulers called Latency Rate servers (LR-servers) are defined as any scheduling algorithm which guarantees that an average rate of service offered to a busy task, over every interval starting at time Θ from the beginning of the busy period, is at least equal to its reserved rate. Θ is called the latency of the server. A large set of the FQ algorithms fit into this class, including WRR, WFQ and SCFQ. Even non-fair algorithms can qualify (one such example presented in the paper is the Virtual Clock algorithm).

[77] goes on to derive a number of results for this rather general class of schedulers, including delay guarantees and fairness bounds. One interesting result is that a net of LR-servers can be analyzed using

one equivalent single LR-server. This can be useful when considering a hierarchy of schedulers.

Alpha-Delta abstraction

Similar to the LR-server formulation is the Alpha-Delta abstraction proposed in [62]. Bounds for minimum service α and delay Δ, corresponding to rate and latency in the LR-server formulation respectively, is here derived from a real-time scheduling point of view.

3.3 Feedback allocation control

Adaptive Reservations

One problem when doing RB scheduling is that the execution time for a periodic task may vary over time. As it is undesirable to base our calculations on the worst case, it is likely some deadlines will be missed. While the CBS scheme can handle transient overruns, non transient changes will lead to eventually infinite deadlines (instability). One way to remedy this would be to dynamically set the budget for a server based on prior overrun statistics in a feedback control manner, though commonly referred to as adaptivity in computer science publications.

In [2], Abeni and Buttazzo introduce a metric called the scheduling error. If we have a periodic task τ_i with period T_i, then the scheduling error ϵ_s is defined as

$$\epsilon_s = d_s - (r_{i,j} + T_i), \tag{3.10}$$

i.e., the difference between the server deadline and the soft deadline of the task. Feedback using the server budget Q_s as the control signal would then be used to drive ϵ_s towards 0.

A few design techniques for such a controller are discussed in [58]. For the purpose of making the analysis simpler, a restriction is imposed so that even if there is extra unused bandwidth available, a task τ_i scheduled by a CBS will only receive the bandwidth Q_s, a so called *hard reservation*. Assume that τ_i is a periodic task being served with a CBS. This gives $r_{i,j+1} = r_{i,j} + T_i$, where T_i is the task period. Each job is associated with a soft deadline $d_{i,j} = r_{i,j} + T_i$, that is $d_{i,j} = r_{i,j+1}$. It makes sense to choose the server period T_s to be some multiple of

T_i. Let $f_{i,j}$ be the actual finish time for $J_{i,j}$ and $v_{i,j}$ be the finish time had τ_i been running alone on a CPU with the fraction $b_i = Q_s/T_s$ of the actual CPU speed, i.e. the virtual finish time. The article uses a modified definition of the scheduling error compared to 3.10

$$\epsilon_{i,j} = (f_{i,j-1} - d_{i,j-1})/T_i \tag{3.11}$$

which is the scheduling error experienced for $J_{i,j-1}$. Note that since hard reservations is being used, having both $\epsilon_j > 0$ and $\epsilon_j < 0$ are undesirable since the task would either be missing deadlines or wasting bandwidth. The relation

$$v_{i,j} - \delta \leq f_{i,j} \leq v_{i,j} + \delta \tag{3.12}$$

where $\delta = (1 - b_i)T_s$, tells us that we can make the CBS approximate the General Processor Sharing (GPS) algorithm by letting T_s go towards 0. Even with normal choices of T_s it is reasonable to use 3.11 and 3.12 to approximate the scheduling error with

$$e_{i,j} = (v_{i,j-1} - d_{i,j-1})/T_i \tag{3.13}$$

A difference equation for the evolution of the scheduler error is then presented in [4]. The paper also proposes a predictor based control structure and three examples of control design using invariant based design, stochastic dead bead design and optimal cost design respectively.

Slack Reclaiming

When scheduling soft real-time or non real-time tasks together with hard real-time tasks or other soft real-time tasks using RBS, inevitably there will be some unused bandwidth. The WCET values used for the hard real-time tasks will often be overestimations and soft real-time tasks can also underuse resources. Ideally, this "extra" bandwidth should be used to improve performance for other soft real-time tasks or non real-time tasks. As dynamic slack can only be detected at run-time it must also be allocated at run-time. The traditional way of handling slack was to schedule all real-time tasks first and then allow the slack consuming tasks to run on whatever remains.

The CBS algorithm itself has a manner of slack reclaiming in that if the current job $J_{i,j}$ terminates before c_s is spent, $J_{i,j+1}$ can begin to execute immediately. However, if the task has not finished before the remaining c_s has been spent, it will be given a new deadline based on the server period T_s, meaning that it can be forced to execute over an even longer period of time, effectively increasing computational latency.

Numerous methods have been proposed to solve this and other slack reclaiming problems. The inventors of the CBS algorithm has published several, including CASH [16], GRUB [55] and IRIS [59]. Lin and Brandt proposes a number of them in [49], the BACKSLASH algorithm being the most advanced. BACKSLASH uses the following four principles to determine who will get what from whom:

- Allocate slack as early as possible and with the priority of the donating task. This means that the scheduler should not wait until the completion of all real-time tasks before it allocates slack. By executing the slack at the same priority as the donator, there is no risk that it would disturb the execution of tasks that would not have been disturbed by that same donator.
- Allocate slack to the task with the highest original priority (earliest original deadline). Basically this means give the slack to the task in most dire need. See the principle below for a rationale for using the original priority/deadline.
- Allow tasks to borrow against their own future resource reservations (with the priority of the job from which resources are borrowed) to complete their current job. This is the standard deadline postponing from the CBS algorithm.
- Retroactively allocate slack to tasks that have borrowed from their current budget to complete a previous job.

Real-Rate Scheduling

One of the first examples of rate-based scheduling was proposed in [35]. The novel approach is to use some task output to measure the rate of progress and thereby eliminate the need for the software designer to assign deadlines or CPU share directly. Experiments presented in the paper are performed using a slightly modified Linux 2.0 series kernel augmented with an RMS based RBS scheme. A task with no known

period or CPU share requirements but a measurable progress is in [35] called a real-rate task.

The example studied is a video pipeline with a producer and a consumer that exchange data via a queue. Queue fill level is the metric used for progress. The scheduler samples the queue and decides if either of the two is falling behind or getting too far ahead. They use a half filled queue as the set-point and then design a PID controller to decide the CPU share needed. The period is decided using an heuristic based on the size of the share, lower share meaning longer period.

3.4 Direct scheduler control schemes

Both the CBS and FQ algorithms have been modified to dynamically set server quota and weights respectively to achieve adaptive RBS, but other schemes exist. One way is to provide reservations by directly controlling scheduler parameters without using an RBS layer.

Controlling Linux in a Nice Way

In [65] a PI-controller is used to dynamically adjust the nice value of a process in the Linux OS in order to achieve some predetermined bandwidth. At the time, Linux 2.6 was still using Ingo Molnar's $O(1)$-scheduler, which is examined in detail in [65]. The scheduler has some features that makes analysis tricky.

- It uses interactivity heuristics to determine which tasks are interactive (I/O-bound) or not. Based on this, a task can receive a priority bonus or penalty in the interval $[-5, 5]$.
- Nice values are inverted compared to priority levels
- Nice values are non-linearly mapped onto time slice sizes
- Tasks tagged as interactive are handled differently when they have used up their time slice.

A model for how to calculate the CPU share F_i for task τ_i from the nice value n_i is proposed as

$$F_i = \frac{t_i(n_i)}{\sum_{\forall j} t_j(n_j)} \tag{3.14}$$

where $t_i(n_i)$ is the time slice for τ_i given its nice value. If we have tasks $\tau_1, \tau_2, \tau_3, \tau_4$ with corresponding nice values $(n_1, n_2, n_3, n_4) = (0, 0, 0, -1)$, they would get the time slices $(t_1, t_2, t_3, t_4) = (100, 100, 100, 420)$ ms. From that we get that

$$F_4 = \frac{420}{100 + 100 + 100 + 420} \approx 58\% \tag{3.15}$$

Experiments using a standard Linux desktop shows that this gives a correct value within $\pm 1\%$. A controller is then implemented as a kernel module which samples the statistics of a task and then sets a new nice value by means of PI control. The reference value for a task is given through a `/proc` interface. The approach works well and is also extended to handle sleeping tasks. The transient behavior when controlling several tasks simultaneously is, however, not investigated much. Studying the results when controlling two tasks reveals some interference when one of the tasks changes its share. A possibly way to improve performance here would be to handle this as a multivariable control problem.

3.5 Allocation

With the establishment of a variety of RBS techniques, the next important question to discuss is how to calculate the reservations. Given a set of timing sensitive applications, individual application performance can be sacrificed to obtain better global performance. The theory of splitting a resource between consumers is often called resource allocation, though considering its mathematical properties, constrained control would be just as accurate.

Within the field of operations research, using optimization to solve logistics and resource allocation problems is common practice. Some of the iconic problems have been formulated here, including the knapsack and bin packing problems. Solving knapsack- and bin packing problems exactly is of NP-complete complexity [46] [23], making them unattractive for on-line use in limited computational capacity settings.

Constrained control theory

A popular tool for managing constrained dynamics in control is the Model Predictive Control (MPC) formulation. The default setup is postulating a convex cost function of the state trajectories, using an LTI-model as trajectory constraints [57]. Though mathematically feasible to use for allocation problems, few algorithms exist for limited precision computers. [34] presents an example where explicit MPC, which uses off-line pre-computed solutions, have been applied to embedded resource management. The approach is resource efficient and applicable on time scales appropriate to real-time systems, but the requirement of knowing the problem structure beforehand limits its uses for open systems.

Q-RAM

In 1997, R. Rajkumar presented his Quality-of-Service-based resource allocation model, Q-RAM [69]. In essence, this states the allocation as a single objective constrained optimization problem. The model as it was introduced allowed for multiple tasks using multiple resources. Tasks are given utility functions based on the allocated resources, but no technique for how to model a specific task is presented. Neither is the problem of solving constrained optimization problems online discussed. In a later paper [71], Rajkumar suggests one way of overcoming the NP-hard problem of general multi-resource allocation, but neither this paper discusses the algorithmic properties of the problem in detail.

ACTORS-model

In [73] the authors present another model, using a set of discrete resource consumptions and quality output levels. The resulting optimization problem is solved through mixed integer linear programming and allows for multi CPU resource models. The domain mentioned as the target in the paper is data-flow applications, but there is nothing explicitly in the model that ties it to this.

One drawback of this approach is the need to supply the resource levels. This is non-trivial and increases the complexity of developing applications. The approach taken in this paper requires only one defined level and utilizes a continuous quality measure, thereby making the optimization easier to solve.

3.6 Modeling of Cyber-physical systems

As we try to close the gap between hardware and software, the traditional models grow more and more cumbersome. While event based models are suitable for describing the state of a software system, their full semantics are difficult to couple with continuous dynamics without ending up with something difficult to analyze. This has been pointed out by e.g. [47].

One recent effort of modeling the interaction of physical dynamics and software resource allocation is presented in [33].

3.7 Convex Optimization

As more and more methods involve solving optimization problems online, the real-time performance of such solvers is an emerging field of research. In Model Predictive Control (MPC), the use of Explicit MPC techniques have brought the solution time for control sized problems down to milliseconds [84]. For more general problems, promising code generation techniques have been developed by Stephen Boyd [37].

The main drawback of both these approaches is that they assume that the problem structure is fixed, which is usually not the case for a multi-purpose software system.

3.8 Estimation

Uncertainty and time varying dynamics can be handled through online estimation techniques, as is common in feedback and adaptive control. For a system with varying time scales, both over time and between components, traditional periodic sampling will not fit well.

Recently, event based control and estimation has gained attention. Such theory is attractive because it can be more efficient than periodic sampling and also better account for situations where information is delivered in an event based fashion. This thesis has been inspired by such works as [40] and [72].

4

Implementation and Frameworks

This chapter presents the current state of RBS implementations and resource management frameworks with focus on the Linux platform. The chapter is based on [50].

4.1 OCERA

OCERA [64] stands for Open Components for Embedded Real-time Applications, and was a European project, based on Open Source, which provided an integrated execution environment for embedded real-time applications. From a RBS point of view, OCERA offers a number of interesting components. The OCERA code is based on the RTLinux extension. The patches are applicable to Linux kernels up to version 2.4.18.

Scheduler Patch

OCERA modifies the Linux kernel so that it provides "hooks" for modules implementing generic scheduling policies. The patch used for this is called the Generic Scheduler Patch (GSP). Our particular interest would be to use it to implement a resource reservation scheduling module.

Integration Patch

The Preemptive Kernel Patch is made to work with RTLinux using OCERA's Integration Patch. The Preemptive Kernel work was done by Robert Love with the aim of improving latency by making system calls possible to preempt.

Resource Reservations Scheduling module

A dynamically loadable kernel module that provides a resource reservation scheduler for soft real-time tasks in user space is distributed with the OCERA components. It uses a CBS-based algorithm, modified to handle some practical issues. It includes optional slack reclamation functionality using the GRUB algorithm. The module provides a new scheduling policy, `SCHED_CBS`.

Quality of Service Manager

OCERA also provides a QoS management services module. This is more or less a controller who changes the bandwidths according to the scheduling error. The approach is more or less that presented in Section 3.3.

Other work

OCERA includes a large body of work which is not directly related to RBS. This includes Application-defined Scheduling (ADS), enhanced memory allocation algorithms, and a wide range of improvements to the RTLinux kernel.

4.2 AQuoSA

AQuoSA [8] stands for Adaptive Quality of Service Architecture and is another initiative to bring QoS to the Linux kernel. It builds on the work provided by OCERA and is partially sponsored by the European FRESCOR project. Structure-wise AQuoSA retains the components used by OCERA.

The AQuoSA versions of the components are at the time of writing still being actively developed, with the latest release updating them for Linux kernels up to 2.6.30.

4.3 FRESCOR

FRESCOR [38] (Framework for Real-time Embedded Systems based on COntRacts) focuses on hard-realtime and contract based resource management. The project background is much like the basis for this project, dealing with the problem of WCET estimation, mixed requirements and maximizing resource utilization.

The idea is to automate much of the real-time analysis by exposing an API in the form of service contracts. Once the user has specified the requirements, the platform negotiates all current contracts to see if there is a valid solution.

A contract is a set of attributes describing the resource needs of the application as well as certain application properties. Some central concepts here are

- resource type (e.g. process, network or memory),
- minimum budget (WCET/T),
- maximum period and
- deadline.

These parameters are then used in for automated response time analysis to determine if the resource requirements can be met.

Spare capacity distribution

In order to address the goal of maximum resource utilization, FRESCOR suggests a form of slack reclamation here called Spare capacity distribution (SCD). As the admission policy works under worst case assumptions, it is likely the system will have spare capacity in most situations.

4.4 Xen

Virtualisation technologies makes it possible to partition a physical computer into several logical instances, each one running a separate operating system that thinks it is running alone on the hardware. The layer beneath the OS layer is sometimes called the hypervisor

layer as it uses a special mode enabled in modern CPUs called the hypervisor mode. Xen is an open source hypervisor created in 2003 at the University of Cambridge in a project headed by Ian Pratt [81].

Architecture

A Xen system has multiple layers, the lowest and most privileged of which is Xen itself. Xen can host multiple guest operating systems, each of which is executed within a virtual instance of the physical machine, a domain. Domains are scheduled by Xen and each guest OS manages its own applications. This makes up a hierachy of schedulers with the Xen scheduler on top.

Xen supports several top level schedulers, including EDF and BVT. Xen 2 also supported the Atropos scheduler, but this has been removed in Xen 3.

4.5 Xenomai

Xenomai [82] is a real-time development framework cooperating with the Linux kernel, in order to provide a pervasive, interface-agnostic, hard real-time support to user-space applications, seamlessly integrated into the GNU/Linux environment. It is an alternative to RTLinux and RTAI and is a possible platform for developing RBS schemes. It was launched in 2001 and in 2003 merged with the RTAI project. The projects split again in 2005, going after separate goals.

It achieves superior real-time performance compared to standard Linux while still allowing regular applications. The real-time kernel is a small, efficient run-time which executes the Linux kernel as a low priority task. Real-time tasks will then be run by the RT kernel, next to the Linux kernel. The Xenomai kernel also provides an API with extensive support for real-time primitives that the Linux kernel lack, including support for periodic task models. Xenomai uses a hypervisor simliar to the Xen hypervisor, but focuses on real-time performance. For example, it allows real-time tasks to receive interrupts even if a Linux process has requested interrupts to be turned off.

4.6 Class-based Kernel Resource Management (CKRM)

The CKRM project aims to create a framework for providing differentiated services to resources such as CPU, memory and I/O. This means that not only can a process reserve a certain CPU bandwidth, ity can also reserve I/O bandwidth, memory access etc. The class concept is used to group together tasks with similar goals and similar importance and in that way govern their right to resources. Each class is associated with a set of controllers responsible for managing access to the resources. The project changed named to Resource Groups lately, but have been somewhat coldly received on the Linux kernel mailing list (LKML), mainly motivated with it being a very large patchset. This is something which may be fixed with the Generic Process Container patchset.

4.7 Generic Process Containers

Generic Process Containers is a patchset which has seen a positive reaction on the LKML, aiming to do similar things as CKRM, but by extending cpusets, a construct already in the Linux kernel. The patchset extracts the process grouping code from cpusets into a generic container system, and makes the cpusets code a client of the container system. The intention is that the various resource management and virtualization efforts can also become container clients, with the result that

- the user space APIs are (somewhat) normalized
- the additional kernel footprint of any of the competing resource management systems is substantially reduced, since it does not need to provide process grouping/containment, hence improving their chances of getting into the kernel

4.8 Resource Kernels and Linux/RK

A more formal approach to what CKRM tries to accomplish has been suggested by Rajkumar et al in their work on Resource Kernels [70].

The authors try to solve a number of problems associated with multi-resource reservations, including the computational complexity of such a system, which has been shown to be an NP-complete problem.

Design Requirements The ambition of the project becomes evident from studying the goals set down for the design of the kernel. In short, they are as follows

- **Timeliness of resource usage.** An application which has made a reservation must be given access to it promptly when needed. An application must also be able to up and downgrade its resource usage (for adaptation and graceful degeneration purposes).
- **Efficient resource utilization.** By this, we mean that the OS must be able to satisfy the timeliness requirements while making as few restrictions as possible. E.g. by only allowing one process at a time, regardless of how small reservations it wants to make.
- **Enforcement and protection.** The enforcement of resources should be such that abuse by one application does not hurt other applications.
- **Access to multiple resources.** Access to multiple resources by the same process must be possible.
- **Portability and automation.** Applications should ideally be able to specify their resource requirements regardless of hardware (e.g. the CPU clock frequency). In addition, resource demands should be automatically tunable.
- **Upward compatibility with fielded operating systems.** The resource kernel should provide support for regular OS services such as regular scheduling algorithms, real-time structures with priority inheritance etc.

Reservation Model

The Resource Kernel uses the parameters C, T, D, S and L with the meanings of a computation time C every T time-units within a deadline D from the start time S over an allocation life-time L. The semantics allow for several types of reservations:

- *Hard reservations.* A hard reservation will not be replenished on depletion, even if possible.
- *Firm reservations.* Such a reservation will be replenished if all other reservations are depleted.
- *Soft reservation.* Will be replenished if possible, even if other non-depleted reservations exist.

Portable RK

A portable implementation of the Resource Kernel concepts is presented in [66] and shown how to work on the Linux kernel. It is considered portable since it does not require any changes to the OS code itself if given access to

- a fixed-priority scheduler
- an interface to change the priority of running tasks
- an interface for suspending and resuming jobs
- an interface for acquiring events within execution objects, needed for accounting reserves

The first three are available in most modern OSes, while the last required hooks to be inserted in e.g. the Linux kernel. The article states that investigations for how to use the native debugging interface could be exploited for these purposes and remove the need for modifying the OS. Arguably, the solution is not entirely portable yet.

4.9 ACTORS

ACTORS (Adaptivity and Control of Resources in Embedded Systems) [5] is a European Union funded research project with the goal to address design of resource-constrained software-intensive embedded systems. The strategies employed include

- data flow-based programming and code generation,
- virtualization and
- feedback resource management.

SCHED_EDF / SCHED_DEADLINE

A significant deliverable from the ACTORS project is an EDF-scheduler implementation for Linux kernels 2.6.30 and later. It is specifically designed for resource virtualization and introduces hard CBS type RBS as the top tier scheduler. Notably, it has support for multicore platforms.

5

Modeling and Estimation

This chapter discusses what is a suitable base for allocating resources, ways to model the resource consumption and the resulting utility of a software component, and techniques for estimating these quantities and model parameters online. It also presents a model for how resource availability is limited by CPU thermal dynamics. The chapter is based on the publications [51], [52], [53] and [54].

5.1 Portable media player *continued*

Returning to the example in Chapter 2, Section 2.1, a typical use case would involve core phone services, such as audio- and video telephony, the network stack and some 3rd party applications, such as GPS-based navigation software. These applications share a CPU through the OS scheduler, as per Figure 5.1. The system would also be running a number of largely dormant system services that will account for a comparatively small part of the total resource needs.

The arrival of Linux-based smart phones has enabled the use of RBS techniques to manage performance, rather than by fixed priority scheduling that was previously the norm. Managing the CPU resource so that software performance is satisfactory can therefore be seen as an allocation problem.

One complication is that the primary resource, the CPU, is limited in performance by power and heat constraints. Running software on the CPU produces heat and since most cell phone casings do not support

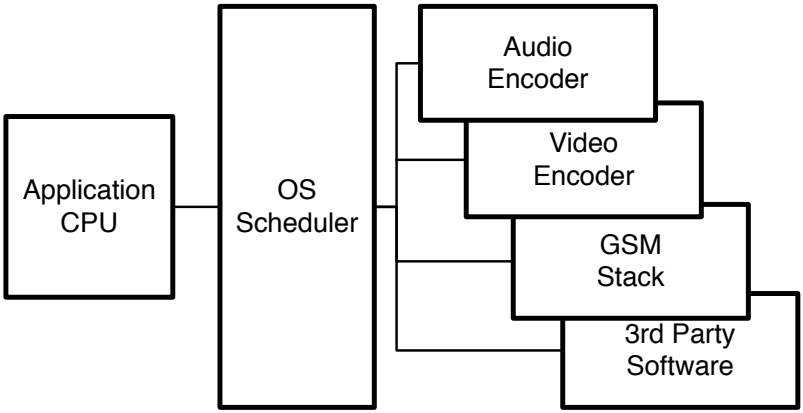

Figure 5.1 A typical use case for a smart phone with support for 3rd party software. The application CPU is commonly a low power chip with limited precision running a fully featured OS such as the Linux-based Android platform.

active cooling, the CPU temperature must be regulated by limiting usage.

Given the unknown properties of 3rd party components, the number of combinations of enabled software and the varying operational conditions, e.g. ambient temperature, allocation decisions will have to be made based on information collected at runtime.

This chapter will therefore introduce

- a resource consumption and performance model intended for the class of software components expected to be dominant in terms of resource demands,
- a model describing how available CPU resources depend on chip temperature and
- techniques for estimating model parameters online.

5.2 Allocation and utility

When allocating a limited resource, it is necessary to consider the *utility*, sometimes called return of investment (ROI) or reward, gained. The field of operations research (OR) is primarily about making such decisions, typically employing constrained optimization.

Assuming that the applications of interest are timing sensitive rather than hard real-time, one approach would be to model the utility of an allocation as a function of the resulting task performance. If hard realtime requirements are assumed, the notion of individual task performance must be dropped; as a task either performs nominally or fails. Deciding what tasks to admit and which to reject based on their nominal performance is essentially the knapsack problem, which unsuitability for use in these contexts has already been discussed in Section 3.5.

Deadline miss ratio

Using the number of task deadlines missed per time interval, deadline miss ratio (DMR), as a measure of utility has been suggested by several previous works, e.g. [39]. This has been primarily used for feedback resource management purposes but given a model of task execution times, a prediction of the DMR is feasible to use. The drawback of such a scheme is that it does not explicitly take into account that tasks can adapt to resource availability. This is commonly practiced in modern software, with examples in media processing [43], computer games [28] and control [9]. An adaptive task could therefore theoretically have the same utility regardless of resource allocation.

Rate-based utility

For media applications, the quality of the output is strongly connected to the processing rate. This holds true for all parts of the media processing chain, from encoding to decoding. It is therefore natural to consider how resource allocation decisions impact the processing rate of the system and thereby indirectly the quality of the service. It is possible to view media stream processing as a special case of data flow or stream processing. In these programming models, data is often contained in packets or tokens which are then processed by a network of computational elements. The rate at which data tokens are processed

is a very tangible metric for the application performance. Data flow formulations exist for a large group of software relevant to embedded situations, ranging from automatic control to 3D-graphics. This supports making rate an important basis for resource allocation in heterogenous systems.

Compared with DMR, rate-based utility sets an absolute value to the resulting performance, though it cannot account for more subtle effects of adaptive tasks. As an example, a video decoder could decide to drop frames as a mechanism for rate adaptation. Deciding which frames to drop is important for playback quality, as discussed in [22]. For now, it will be assumed that if a task has such properties, it is able to make such decisions by itself. Explicitly including such information in the system global model is in this work considered infeasible for a general purpose allocation framework.

Rate is also an application centric metric which makes it easy to use from a developer's point of view. In many cases the desired execution rate is explicitly decided at design time, as with encoding frame rate for video, simplifying its use even further.

Given its central position in this work, the term *rate* deserves a clear definition. *Rate* signifies the number of occurrences of a pre-specified event during a counting time-period.

The pertinent choice of event and counting period is strongly situational. Consider for instance the difference between digital audio and video. The ear is much more sensitive to audio jitter than the eye is to frame jitter [76]. The audio stream is also sampled at a significantly higher rate than the typical video stream (16 kHz vs 25 Hz). While loosing one or several movie frames during a second might not even be noticeable for the viewer, loosing the same percentage of audio samples will make the audio sound very distorted. When considering a system with mixed time scales, selecting a suitable time period for allocation can be troublesome. This will be discussed further in Chapters 6 and 7.

Timing-based utility

Another alternative is to use the exact time that a task completes a job as a metric, as opposed to whether it always complete ahead of a deadline. This measure contains more information than the DMR and can be especially useful when co-ordinating dependent components. An

example of this is synchronization between audio and video data in a capture device, which is examined in more detail in Chapter 8.

5.3 Components with rate-based utility

This thesis will discuss mostly applications where utility is based on execution rates and cycle timing. The following section discusses how to model single- and multi-resource dependent components. From a resource consumption point of view, there will be no difference made between a single- or a multi-threaded process, or even something made up from a family of processes. The building blocks will be called components, where a component is an entity that

- consumes resources and outputs results opaquely,
- is responsible for distributing allocated resources to subcomponents,
- has a behavior determined by a dynamic mapping from a set of inputs u^i to a set of outputs y^i and
- where all significant dependencies affecting performance is modeled through this mapping.

There are two main reasons for this:

- **Composability.** In order to simplify building large systems including 3rd party components, a building block semantic where the set of parts is closed under composition is useful. This thesis will disregard the functional aspects of the component architecture, considering it a mostly solved problem, and focus on the resource-driven temporal qualities.
- **Extending into non software components.** It is the aim of this work to model systems built from a mix of software and hardware. Therefore, raising the level of abstraction is necessary to hide properties unique to software or hardware parts.

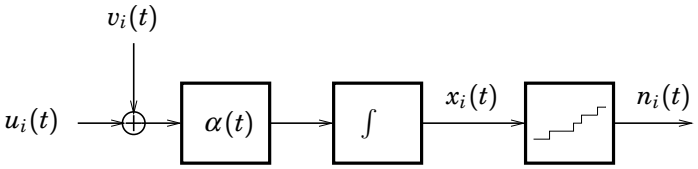

Figure 5.2 A cyclic computation software component. $u_i(t)$ is the CPU resource share assigned, $v_i(t)$ is the disturbance on the assigned share caused by the environment, $\alpha_i(t)$ is the execution speed of the CPU, $x_i(t)$ is the accumulated execution, the last block is a staircase function mapping $x_i(t)$ onto $n_i(t)$, which signifies the number of completed cycles.

The cyclic component model

The most basic case is a component that depends on a single resource. In a computational setting, this corresponds to the standard problem of a set of independent, CPU-bound tasks. In this work the notions of periods and deadlines are dropped and replaced by the notion of cycles. Furthermore, as the entities of interest can be made up from nested sets of tasks, the term component will be used.

Figure 5.2 shows the logical setup for the *cyclic component model*, used as the template further on. Generally speaking, the component accumulates the incoming resource, in this case executed CPU instructions, until some quantity has been achieved. At this point it outputs the result and commences a new cycle. The details of the model are as follows:

- $u_i(t)$ is the CPU resource share assigned to component i and is a dimensionless number.
- $v_i(t)$ is a zero-mean disturbance of this share caused by the scheduler as experienced by component i. The two major sources for this disturbance is the error in the approximation that the CPU is really fluidly divisible and certain system events that interrupt normal execution, such as hardware interrupts or virtual memory handling.
- $\alpha(t)$ denotes the speed at which the processor executes instructions and has the unit of completed instructions per time unit.

- $x_i(t)$ is the accumulated number of executed instructions for component i.
- $n_i(t)$ signifies the number of cycles completed by component i at time t and is calculated as

$$n_i(x) = \max k, s.t. \sum_{j=1}^{k} C_i(j) \leq x_i(t) \tag{5.1}$$

  for some sequence $< C_i(k) >$ that describes the amount of execution it takes component i to complete computation cycle k.

A cyclic component will block if and only if it is starved of CPU-time. The cycle execution time for cycle k is considered to be a weakly stationary stochastic process $C_i(k)$ and h denote a time interval such that $E\{C_i(k)\} < h$. It is assumed that $C(k)$ has a strictly positive lower bound.

From this it follows that if a component is started at t and executes until $t + h$, the expected number of completed cycles becomes

$$E\{\frac{h}{C_i}\} \approx \frac{h}{E\{C_i\}} \tag{5.2}$$

Assuming the share $u_i(t)$ is constant over h, equation (5.2) can then be used to approximate the dynamics of $n_i(t)$ as

$$n_i(t+h) - n_i(t) \approx \frac{1}{E\{C_i\}} h u_i(t) = h k_i u_i(t) \tag{5.3}$$

or in words, n_i evolves approximately like a discrete time integrator with the unknown gain k_i and is driven by the input $u_i(t)$. $(n_i(t) - n_i(t-h))/h$ is denoted $y_i(t)$ and is referred to as the *execution rate* of C_i. Figure 5.3 shows an example of $n_i(t)$ and the corresponding $y_i(t)$ for one of the components used in the simulations. In the figure $h = 1$. As a result, $y_i(t)$ will lag behind by as much.

Rate-error utility

While the rate $y_i(t)$ itself can be used as a utility metric, often an algorithm or component is designed with a specific rate in mind. Even

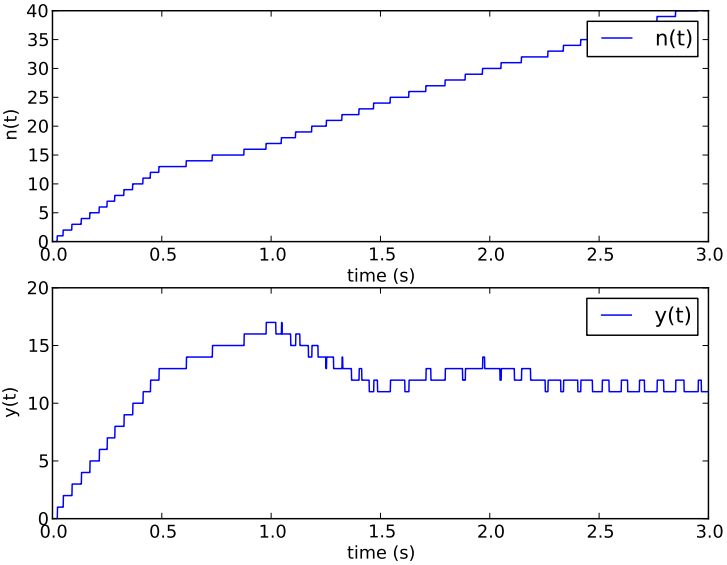

Figure 5.3 $n_i(t)$ and $y_i(t)$ for one of the components in Chapter 7. Note that these are not sampled curves. Despite their jaggedness, both signals are defined for all t. Rate estimation is here done by a one second time window, i.e. $h = 1$. Note that this will make $y_i(t)$ lag behind by as much.

if there are resources enough to increase execution rate beyond that, performance would not increase or in some cases even degrade. For this purpose, the component model is augmented by another parameter, the *rate set-point* r_i, denoting the optimal rate for this component. It is common that software components will limit their execution rate once $r_i(t)$ is achieved and the linear mapping between $u_i(t)$ and $y_i(t)$ in (5.3) must be altered to reflect this, resulting in

$$y_i(t) = \begin{cases} k_i u_i(t-h) & 0 \leq u_i(t-h) \leq \dfrac{r_i}{k_i}, \\ r_i & u_i(t-h) > \dfrac{r_i}{k_i} \end{cases} \tag{5.4}$$

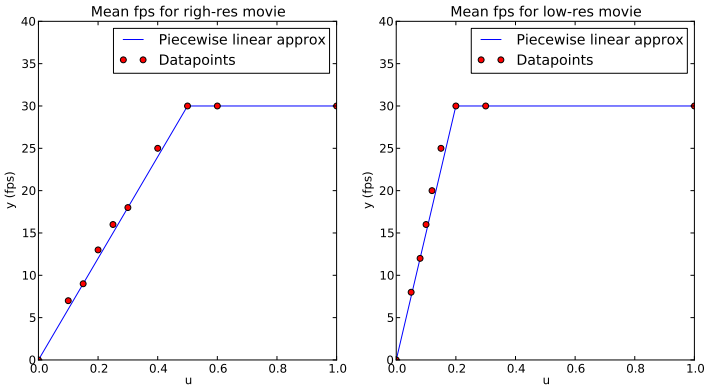

Figure 5.4 Experimental results of controlling CPU-share for the MPlayer decoder using Linux 2.6.27 and Control Groups. The diagrams show how the frame rate per second (fps) depends on the amount of CPU share allocated to the decoder. The rate increases linearly with share until the movie can be played back at encoded rate.

Figure 5.4 shows two cases which were produced using MPEG-4 video streams and the free MPlayer software. The videos are encoded at a fixed rate, in this case 30 frames per second (fps). When allocating a lower share of the CPU than required for full rate playback, MPlayer starts to skip frames, thereby adapting to the reduced playback rate.

Cycle completion timing

The other aspect of the cyclic component model is the event dynamics, i.e. exactly when the job cycles complete. In the cyclic component model, the completion time for cycle k is denoted $t_i(k)$ and modeled as

$$t_i(k+1) = t_i(k) + \frac{C_i(k)}{u_i(t(k))} \tag{5.5}$$

if $u_i(t(k))$ is assumed to be constant over the interval $[t_i(k), t_i(k+1))$, making it similar to the virtual finish time introduced in [27].

While this could be used for traditional deadline driven scheduling (e.g. keeping $t_i(k+1) < t_i(k) + D_i$)) it is also possible to control other

and sometimes more interesting metrics, such as the synchronization between two non-uniform sequences.

5.4 Multi-resource dependencies

It is desirable to model components that require multiple resources to execute. This thesis proposes to do this by extending the component blocking rule so that a component will block if and only if it is starved of one or more resources. It is assumed that a component has the capability of accumulating incoming flows, e.g. in FIFO queues for data or execution time deficit accounting in the scheduler for CPU-time. It follows that the component execution rate is limited by the rate at which resources are made available to it. Formally, if τ_i is dependent on the flows $u_0, ..., u_N$

$$y_i = \min(k_{i,0}u_0, ..., k_{i,N}u_N) \tag{5.6}$$

would describe its execution rate. From (5.6) it follows that an allocation strategy should try to keep the incoming flows equal to minimize over-provisioning and reduce the risk of buffer overflow. Furthermore, it points towards two important objectives for maximizing performance of these systems

1. calculating a steady state flow that maximizes the relevant performance metric
2. control transient effects that cause blocking

How point 2 is connected to the cycle completion dynamics is further discussed in Chapter 7.

5.5 CPU thermal dynamics

In order to take the thermal dynamics of the CPU into account when deciding on how much load the system can sustain, this thesis proposes a model for the thermal dynamics based on [32]. By limiting the load, or utilization, the temperature can be controlled even if the CPU is

only passively cooled. The model of the dynamics from CPU power P to CPU temperature T is on the form

$$\dot{T} = a(T_a - T) + bP + d \tag{5.7}$$

where a and b are constants depending on the thermal resistance and heat capacity of the processor and T_a the ambient temperature. d is a disturbance term which will be assumed have slow dynamics, such as heat generated by direct sunlight or by being placed on a heated surface. For off-the-shelfs CPUs, a and b are in the order of 10^{-4} and 10^{-3} respectively (see e.g. [33]), making the dynamics relatively slow. It is therefore assumed that it is possible to filter out measurement noise, which is therefore omitted from the model.

The relationship between CPU load U and P is then modeled as

$$P = P_{idle} + U(P_{max} - P_{idle}) \tag{5.8}$$

Sampling the combination of 5.7 and 5.8 can then be done under zero-order-hold assumptions.

A step response experiment was carried out on a Pioneer mobile robot [61] with an internal Intel Pentium III-based computer [80] in order to validate the model structure, giving the results shown in Figure 5.5. The on-chip temperature sensor has a sample period of 2 seconds.

5.6 Parameter estimation

A key assumption in this work is that the model parameters are not available beforehand and therefore have to be estimated online. This section discusses the aspects of this and some possible techniques. First the problem of estimating the execution rate from the sequence of observable cycle completion events is discussed. An estimator for the k-parameters based on the rate estimate is then suggested.

The central problem with computational components is that information comes in form of events instead of continuous signals. For example, there is only new information about the execution rate when a calculation cycle completes or when an expected event is missing.

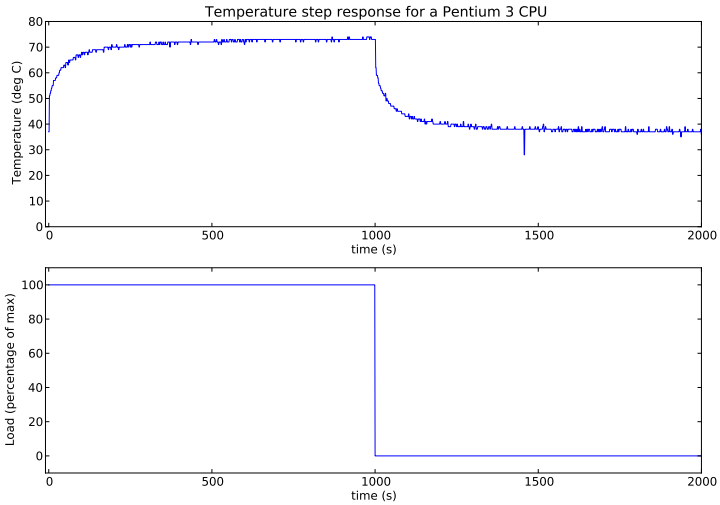

Figure 5.5 Results from a step response experiment performed on a Pioneer mobile robot. The experiments supports a simple first order model for the dynamics between load and chip temperature.

There are two main alternatives to estimate the execution rate from this, sliding time window event counting and event based filtering. It is assumed that

(a) while the variable CPU-speed $\alpha(t)$ cannot be directly measured, it is possible to measure $x_i(t)$ and thereby

$$\int_{t_0}^{t_1} \alpha(t)dt$$

(b) The completion of a cycle is observable through an event, defined as the tuple $(t_i(k), x_i(t_i(k)))$, where $t_i(k)$ is the time when component i completed cycle k.

The following properties are considered important for the resulting algorithm

- Time complexity
- Space requirements
- Sensitivity to noise
- How fast it can detect a change in rate

Sliding window event counting

Using the definition of rate (events/time period) it is natural to consider an approach where the number of events occurring over a predetermined time period is mechanically counted. Given a suitable window length, the method is straightforward in implementation, but needs an unknown amount of memory to keep the events, also the time complexity is proportional to the rate, i.e. unknown beforehand.

Event based filtering

An alternative approach is to view the estimation of y_i as a prediction problem, where the objective is to at any given time estimate the time between the last and the yet not arrived event. If $\Delta(k) = t(k) - t(k-1)$, using the assumed stationarity stochastic properties of n, a predictor can be written on the discrete time shift operator form

$$\hat{\Delta}(k+1) = \frac{B(q^{-1})}{A(q^{-1})}\Delta(k) \tag{5.9}$$

$$\hat{y}_i(k) = \frac{1}{\Delta(k+1)} \tag{5.10}$$

The selection of the polynomials B and A makes it possible to filter out specific noise components of the sequence and as long as the filter has unit stationary gain ($B(1)/A(1) = 1$) the proper mean will be obtained. There is one caveat however when dealing with a decreasing rate. If the prediction states that an event should occur but there is none, the estimate must be updated to reflect this. This can be done through noting that if t time has passed since the last event occurred and $t > \hat{\Delta}(n+1)$, then the highest possible current rate would be sustained if an event would arrive at the time $t + \epsilon$. A way to check for this is to tentatively update the prediction as if an event had occurred at

the time t and check if the estimated rate would be lower. If $\Delta_e(k)$ denotes the extended sequence $\{..., \Delta(k-1), \Delta(k), (t-t(k))\}$, the resulting estimator for $y(t)$ would then be

$$\begin{aligned} \hat{\Delta}(k+1) &= \frac{B(q^{-1})}{A(q^{-1})}\Delta(k) \\ \hat{\Delta}_e(k+1) &= \frac{B(q^{-1})}{A(q^{-1})}\Delta_e(k) \\ \hat{y}_i(t) &= \frac{1}{\max(\hat{\Delta}(k+1), \hat{\Delta}_e(k+1))} \end{aligned} \tag{5.11}$$

Advantages with this approach is that the filter is fixed in time and space complexity. There is also the added degree of freedom in selecting the filter polynomials, but the downside is that badly chosen polynomials can yield a noisy estimate. By choosing $\deg(A) = 1$ and $\deg(B) > 1$, the filter gets a finite impulse response (FIR) structure. As has finite memory, just like the time window filter, it is also called an event window filter. If instead $\deg(B) = 1$ and $\deg(A) = 1$, the structure is called autoregressive (AR) or infinite impulse response filter (IIR).

k-parameter estimation

Given an estimate of the current execution rate $\hat{y}_i(t)$, falling back on equation (5.4) results in the following estimate:

$$\hat{k}_i(t) = \frac{\hat{y}_i(t)}{u_i(t - h_i)} \tag{5.12}$$

Unfortunately, this estimate does not take the disturbance w_i into account. As it is possible to measure $x_i(t)$ directly, a better estimator would be

$$\hat{k}_i(t) = \frac{\hat{y}_i(t)}{x_i(t_1) - x_i(t_0)}, \tag{5.13}$$

if t and t_0 and t_1 are such that the events used to form $\hat{y}_i(t)$ occur in the interval (t_0, t_1).

5.7 Extension into mixed domain models

One important objective of this work is to model a system where there is noticeable interaction between the software components and the hardware components. The approach taken is to see it as a system which performance is governed by the flow of resources. The concept of *resource* is here extended to mean a quantity, bounded and non-negative, that through a system component is converted into another resource. The system performance is then expressed in terms of this transformation.

A *resource flow* is the exchange of a resource between two components of the system. In a CPS, a flow can be physical (e.g. heat or power) or virtual (computations or data). In this work physical flows are considered to be $\mathcal{L}_2$ functions while virtual flows are either $\mathcal{L}_2$ or a sequence of Dirac-spikes, with finite density. Formally, let $\mathcal{U}$ denote the set of generalized functions on $\mathbb{R}$ such that if $u(t) \in \mathcal{U}$ and, $t_1, t_2 \in \mathbb{R}$ then

$$\int_{t_1}^{t_2} u(t)dt \tag{5.14}$$

exists and has the sign of $t_2 - t_1$. The rationale behind resource flows as Dirac-spikes is the event-like manner in which many important virtual resources are generated. As an example, let $n_{CPU}(t)$ denote the number of completed instructions in a CPU and $n_{task}(t)$ the number of times a sporadic task running on that CPU has executed. Obviously, $n_{task}(t)$ depends on $n_{CPU}(t)$. In order to analyze the performance of the task, e.g. to check if it completes cycle n before a certain time, it is necessary to study how $n_{task}(t)$ and $n_{CPU}(t)$ evolve over time but due to their stair case nature, if t_0 denotes some specific time,

$$\lim_{h \to 0} \frac{n_{task}(t_0) - n_{task}(t_0 - h)}{h} \tag{5.15}$$

is either 0 or undefined. Therefore the dynamics of a system involving virtual flows must be expressed in discrete time, here called a *discrete flow dynamic*.

Forming such a model for physical systems can be done through sampling. Note however that while the sampled system description says nothing about the system behavior between sample points, signals like

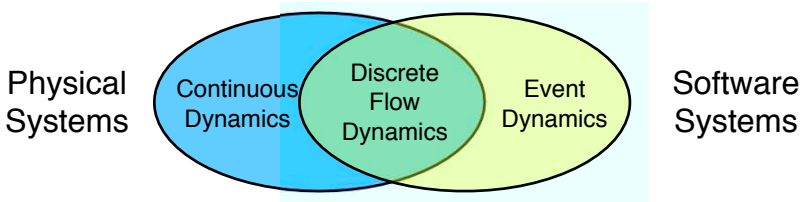

Figure 5.6 A diagram showing how the domains of physical and software systems have their own unique dynamics, but share the discrete flow dynamics.

$n_{CPU}(t)$ and $n_{task}(t)$ are defined for all t. Because of this, it is possible to talk about exactly when the changes in n occur.

To summarize, the dynamics of a physical system is often modeled through differential equations. These can be discretized or sampled to give a discrete time model. Software system are usually modeled through event-based dynamics, but these can be re-formulated as discrete flow dynamics, comparable with the discretized continuous model. This gives a common model domain with which to express the entire system dynamics. Figure 5.6 show how the two domains and the dynamic descriptions relate. How such a combined model can be used for resource management is explored further in Chapter 7.

6

Allocation

Chapter 5 presented the performance of the system expressed in two primary metrics, execution rate and event timestamps, both dependent on allocated resources. This chapter introduces allocation through convex optimization as a way to maximize system performance in the execution-rate sense. A specialized allocation algorithm targeted at embedded real-time systems is presented together with a discussion about its convergence properties and real-time behavior. This chapter is based on the papers [52] and [53].

6.1 Allocation under resource constraints

The purpose of resource management is to maximize system performance, a non-trivial task when there is not enough resources to run all components at nominal execution rates. In the terminology introduced in Chapter 5, it is to be expected that we cannot keep rate error of all components at zero, but have to compromise. To evaluate such a compromise, a global performance metric is needed. For the set of independent components, a natural choice would be an aggregate of the individual utility functions. The selection of such an aggregate is an important design parameter, which can be used to achieve a compromise suitable to the central system use cases. For example, it can make sense to prioritize system services over 3rd party components or minimizing the worst case rate error.

In this thesis, it will be assumed that the system performance ob-

jective is on the form

$$J(u) = \sum_{i=1}^{N} J_i(u_i) \tag{6.1}$$

where J_i is the contribution from component i. In the presence of a resource constraint, $\sum u_i \leq U$, and under the assumption that resources are positive quantities, the allocation problem can be posed as an optimization

$$\begin{aligned} \min J(u) &= \sum_{i=1}^{N} J_i(u_i) \\ \sum_{i=1}^{N} u_i &\leq U \\ u_i &\geq 0, \forall i \end{aligned} \tag{6.2}$$

Convex allocation problems

In order to find efficient and reliable ways to solve the allocation problem online, i.e. without human supervision, (6.2) needs to be restricted. By limiting the component contributions J_i to differentiable convex functions, the resulting form will qualify as a convex optimization problem [13]. As such it has several attractive properties:

- it will always be a feasible problem,
- it will have a unique solution,
- powerful theory exist for designing solvers for and
- it allows for general forms of system utility functions.

Recall the definition of the steady state rate error

$$e_i(u_i) = r_i - y_i = r_i - k_i u_i \tag{6.3}$$

Since Equation (6.3) constitutes an affine mapping, any J_i taken as a the composition of a convex function and $e_i(u_i)$ will also be convex [13, p. 79]. Conceptually, this means that we can now think of the problem in terms of distributing the rate error rather than the resources.

When designing the objective function, some attention should be given to its sensitivity to parameter changes. Since parameters will

be estimated online under noisy circumstances, adopting an LP-style objective can lead to sudden jumps in the solution. See [57, p. 151] for a discussion on the merits of LP and QP objectives.

Disabling components

For some components performance is only tolerable when y_i is close to r_i. For this purpose a lower rate bound y_i' can be introduced, which represents the lowest relevant execution rate. It would then be desirable to allocate resources so that all components (if possible) are within their respective regions of tolerable performance. If this is not possible, low priority components must be disabled until

$$\sum_{i=1}^{N} \frac{y_i'}{k_i} \leq U. \tag{6.4}$$

Solver design

Though parameters in this system might not change often, events such as the reconfiguration of an application, the arrival or deactivation of a new component or the change in CPU resource availability in response to risk of overheating might require that we quickly redistribute resources. Therefore, an algorithm suitable to solving (6.2) online is needed. More specifically, it is desirable that it

1. takes minimal system resources,
2. accounts for changing parameters as quickly as possible,
3. produces results in deterministic time and memory,
4. can improve upon a previous allocation even if aborted before optimum was computed, and
5. is suitable for implementation in fixed point arithmetics.

6.2 Incremental optimization

This section proposes an algorithm which can solve (6.2) efficiently and with desirable time and memory characteristics. The central idea of

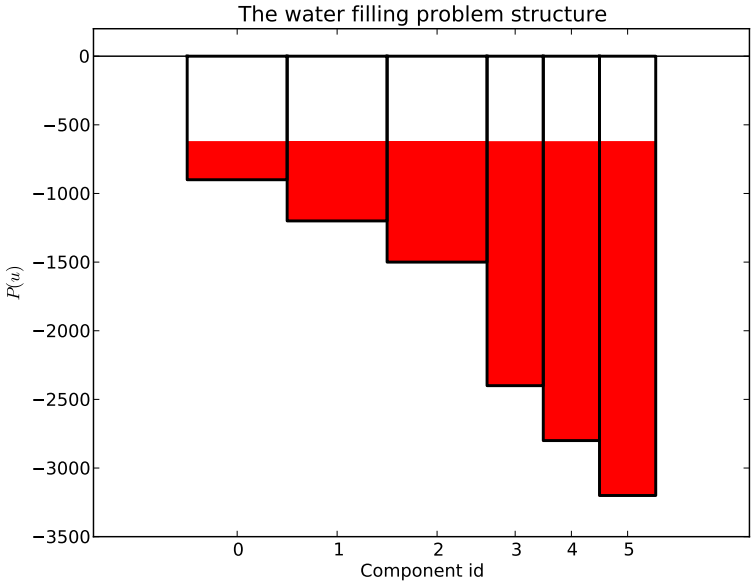

Figure 6.1 The structure of the water filling problem. The geometry of the "tubes" is defined by the cost function and the optimal solution is found as the water surface if the tubes are filled to a common level.

the algorithm is to see the solution as a sequence of resource transfers between two components, in effect solving the problem as a series of one-dimensional problems. The benefit of this approach is that each step is computationally inexpensive, predictable in execution time and has numerical properties well suited for fixed point implementations.

The problem structure is very similar to the traditional power distribution problem seen in the communications literature, see e.g. [13], as illustrated in Figure 6.1. Essentially, the components are represented by a set of connected "tubes", with dimensions so that if the tube is filled with a quantity of water u_i, the resulting surface level will correspond to the value of the marginal utility function $P_i(u_i) = \partial J_i / \partial u_i$. For the example in the figure, $J_i(u_i) = (r_i - k_i u_i)^2, \forall i$. In this case the tubes will be $-2k_i r_i$ high and $1/2k_i^2$ wide.

Algorithms used to solve this type of problem are often based on the water filling principle, i.e. the solution is calculated as if water in an amount equal to the allocatable resource had been poured into the connected tube set and allowed to settle. The resulting water surface will then define the optimal allocation. The method presented in [13] starts with empty tubes and then fills them until the resource is depleted. Mathematically this is done by solving a series of linear equations. Two drawbacks of this scheme when applied to the embedded system resource problem is that

- the solver starts from scratch, thereby loosing information about the previous solution, and that
- the linear equation method only works for a specific case of utility functions.

The algorithm presented below also relies on the water filling principle, but works by equalizing water level between two components at a time. This accounts for heterogenous sets of utility functions and is also easier to implement on limited precision systems as the expressions to be evaluated are simpler.

Assume that two components $\mathcal{C}_i, \mathcal{C}_j$ are picked from the set during the k:th step of the algorithm. Let J^k be the cost at the beginning of the step and $J^k_{i,j}$ denote the contribution by $\mathcal{C}_i, \mathcal{C}_j$ to J^k. Consider now what happens if an amount of resource δ is transferred from $\mathcal{C}_i$ to $\mathcal{C}_j$ so that their combined contribution to J^{k+1} is minimized, i.e. by solving

$$\begin{aligned} \min_{\delta} J^{k+1}_{i,j} = & J_i(u^k_i + \delta) + J_j(u^k_j - \delta) \\ & s.t. - u^k_i \leq \delta \leq u^k_j \end{aligned} \tag{6.5}$$

This ensures that

$$J^{k+1} \leq J^k \tag{6.6}$$

In other words, by in each step solving a subproblem, performance will improve incrementally. Solving this minimization subproblem for general convex functions $J_i(u_i)$ can be done by modifications to unconstrained methods such as Newton-Rhapson or even bisection.

One notable choice of $J_i(u_i)$ is $w_i e_i(t_u)^2$ in which case the allocation problem becomes a special case of quadratic programing (QP).

The quadratic form is attractive for several reasons, including that the subproblem can be solved in two simple steps and being less sensitive to small parameter changes than e.g. linear programming (LP). Let

$$\begin{aligned} J_i(u_i) &= w_i(r_i - k_i u_i)^2 \\ J_j(u_i) &= w_j(r_j - k_j u_j)^2 \\ \delta = \arg\min J_{i,j}(\delta) &= J_i(u_i + \delta) + J_j(u_j - \delta) \\ s.t. &-u_i \leq \delta \leq u_j \end{aligned} \tag{6.7}$$

As $J_{i,j}$ is a convex function, if it has an unconstrained minimum that violates the constraints on δ, the constrained minimum is found by picking δ on the constraint. The solution to (6.7) is then calculated as

$$\begin{aligned} \delta_{nc} &= \frac{w_i k_i r_i - w_i k_i^2 u_i + w_j k_j^2 u_j - w_j k_j r_j}{w_i k_i^2 + w_j k_j^2} \\ \delta &= sat(\delta_{nc}, -u_i, u_j) \end{aligned} \tag{6.8}$$

Selecting the pair C_i, C_j for each step is the other part of the algorithm. The proposed strategy is derived from the Karush-Kuhn-Tucker (KKT) conditions (see e.g. [13]). Posing (6.2) on standard form, the Lagrangian becomes

$$L(u, \lambda, \nu) = \sum_{i=1}^{N} J_i(u_i) + \sum_{i=1}^{N} -\lambda_i u_i - \nu(U - \sum_{i=1}^{N} u_i) \tag{6.9}$$

The KKT-conditions state that $\nabla L(u, \lambda, \nu) = 0$ in an optimal point. By studying the expression

$$\frac{\partial L(u_i, \lambda, \nu)}{\partial u_i} = \frac{\partial J_i(u_i)}{\partial u_i} - \lambda_i + \nu = 0 \tag{6.10}$$

it can be seen that in an optimal point, either $u_i = 0$ or $-\partial J_i(u_i)/\partial u_i = \nu$. Recall that $P_i(u_i) = \partial J_i(u_i)/\partial u_i$. If $u_i = 0$ and therefore $\lambda_i > 0$, then $P_i(u_i)$ must be less than ν. In other words, a point where $P_i(u_i) > P_j(u_j)$ and $u_j > 0$ does not minimize (6.5).

- If the algorithm tries to select C_i, C_j so that $P_i(u_i) > P_j(u_j)$ and $u_j > 0$, solving (6.5) results in $J^{k+1} < J^k$.

- If there is no such pair to select, then that point satisfies the KKT-conditions of (6.2) and the allocation is optimal.

It follows that such a strategy will make the algorithm converge to the optimum. The convergence speed will obviously depend on the specific transfer sequence. As the intended domain is real-time allocations, an efficient strategy is needed. It is desirable that each step reduces $J(k)$ as much as possible and from (6.5) it is evident that the size of the gain depends on

- the difference in $P(u)$ between the two components and
- the amount of resource available to redistribute.

The two criteria can be in conflict, particularly if there are large variations in k_i. However, it is here assumed that if a component requires much less resources then the others, it does not have to be part of the optimization. Rather, such a component will be seen as part of the background noise, as discussed in Section 5.3.

An intuitive strategy for picking the transfer pair is to sort the components according to $P_i(u_i)$ and select the two furthest apart, skipping those with highest $P_i(u_i)$ for which $u_i = 0$. The intuition behind this can be seen if $P_i(u_i)$ is interpreted as the water level, as the sought common surface must lie between the extremes. The proposed implementation uses a red-black tree that makes finding the pair an $O(1)$ operation and inserting them back after the transfer an $O(\log n)$ operation (see e.g. [25] for complexity analysis of red-black trees). As the algorithm uses an iterative loop and the persistent data allocated scales linearly with the problem, memory need for a system with a known size can easily be calculated.

To illustrate the workings of the algorithm, consider a case with three components $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$, and $J_i = e_i(u_i)^2, i = 1, 2, 3$. The components have the properties

i	r_i	k_i	u_i
1	55	50	1
2	25	60	0
3	20	60	0

Figures 6.2 and 6.3 illustrates how the algorithm then operates during the first iterations. Though this example uses the same form of cost

function for all components, it is worth noting that the algorithm allows for any mix of convex cost functions. This could be useful for a system designer when distinguishing between e.g. system services and 3rd party add-ons are allocated resources.

In the initial state, $\mathcal{C}_1$ is best off, seen by the high P_1 while P_2 is worst off. The first transfer then equalizes the potentials P_1 and P_2. As can be seen in Figure 6.3, the system cost decreases rapidly in the beginning. The performance resulting from a fair allocation, as defined by [27], is provided for comparison. The fair allocation will in general terms allocate an equal amount of resource to all components, which will be unfavorable for components with relatively low k and high r, such as C_1.

Figure 6.4 exemplifies a scenario where the problem parameters change over time, assuming allocation is recomputed every second. At 33 seconds, r_2 changes to 40, exemplifying something that could be caused e.g. by an internal mode switch or a user command. The algorithm will here reduce the performance of $\mathcal{C}_1$ and $\mathcal{C}_3$ slightly. At 66 seconds, the total resource level drops by 25%, which could happen for instance if the system was overheating and the CPU needed to be throttled to reduce heat generation. In this case, $\mathcal{C}_3$ is shut down, which may or may not be the intention of the system designer. Constrained allocation in this manner is not starvation free and care must be taken when choosing the component cost functions.

6.3 Experimental results

A series of experiments were run, using the algorithm to find an allocations for random component sets under overload conditions (i.e. $\sum_{i=1}^{N} r_i/k_i \geq U$). The aim with the simulations were to show the computational efficiency and get a feel for the convergence rate. The simulations were run on an 2.40 Ghz Intel Pentium(R) 4 based computer with 512Mb memory which was running Linux 2.6.27. The compiler used was gcc 4.3.2 using the -O3 compiler flag. The experiments use the $J = ||e(u)||_2$ cost function.

Figure 6.5 shows the iteration time as function of the number of components and in Figure 6.6 we see the termination time of the optimization. The variance come primarily from sorting artifacts and cache

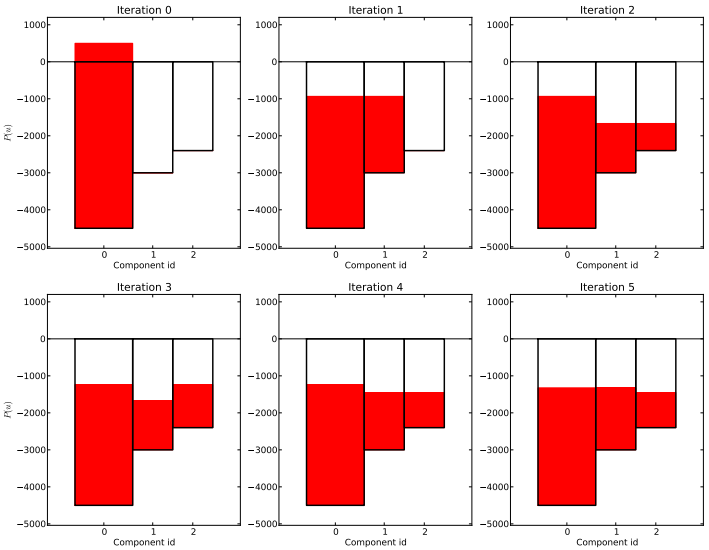

Figure 6.2 A sequence of allocation iterations, showing how the marginal utility level is equalized by pairwise transfers.

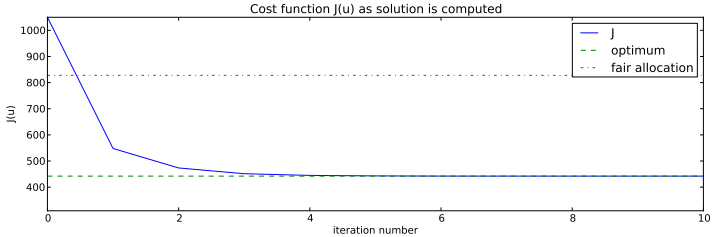

Figure 6.3 A plot of how the cost function decreases in value for each iteration in the case illustrated in Figure 6.2, with the performance resulting from a fair allocation provided for comparison.

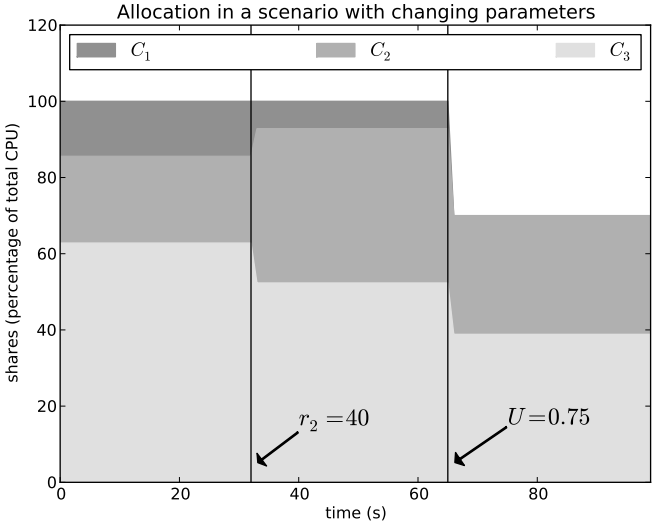

Figure 6.4 A scenario where the problem parameters change over time, assuming allocation is computed every second. At 33 seconds, r_2 changes to 40 and at 66 seconds, the total available resource level drops by 25%.

dynamics. Component sets were generated randomly and ran 10 times in succession. As a comparison number, a two variable QP problem with the structure of (6.2) took 500 ms to solve with a general QP solver written in C++ [21] on the same computer as used in the other experiments. This is most likely due to the larger overhead for initializing the algorithm, something that will make it resource expensive to use for a problem where parameters and problem structure change over time.

Studying Figure 6.5, iteration time appears to increase linearly with the problem size. This seems counterintuitive as the red-black tree is performs with $log(n)$-like complexity. However, as the number of components grow large, it becomes increasingly likely that some of them will be allocated zero resources. Finding a component with high $P_i(u_i)$ will then involve a linear search, which would explain the trend

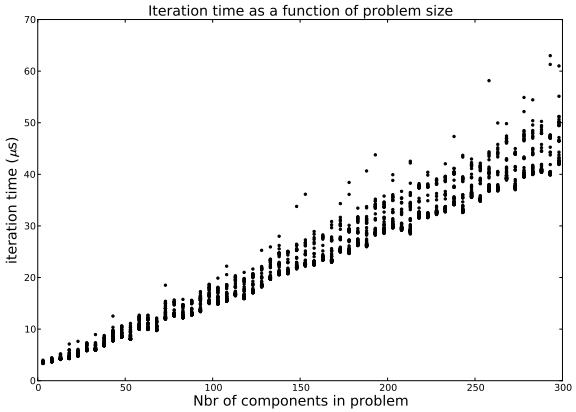

Figure 6.5 Measurements of iteration times. Optimization for each component set was run 10 times. The variance comes from a combination of sorting artifacts and cache dynamics.

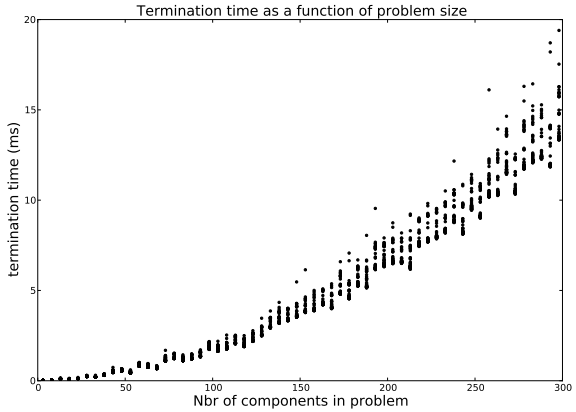

Figure 6.6 Measurements of optimization termination time. Optimization for each component set was run 10 times. Variance is due to sorting artifacts and cache dynamics.

in iteration time. As the component set grows larger, the variance in iteration time grows. This is because the sorting operations when re-inserting components into the red-black tree become more and more expensive. It is also to be expected that keeping the data structures in cache memory will be increasingly difficult for large problems.

The trend in termination time is more according to intuition. As iteration time grows linearly and the number of iterations must grow at least as fast as problem size, the algorithm is at best quadratic in complexity.

With a solver that can determine an optimal allocation in milliseconds, periodic use of optimization in embedded system resource management is feasible.

7

Resource control

This chapter discusses the application of feedback control to a system where component are dependent on each other. The theory presented does not build on Chapter 6, but presents an alternative scenario, where the performance metrics are tied to the system rather than the individual component. The system model used is, however, based on Chapter 5. The chapter is based on [54].

7.1 Allocation vs feedback

One of the key simplifications made in the allocation algorithm proposed in Chapter 6 was to optimize the performance in a stationary sense, as introducing state dynamics into the optimization would require a more comprehensive solver. As such, the allocation strategy used in this thesis is largely a feedforward solution and therefore blind to performance metrics explicitly connected to state information.

Given perfect information about execution times and resource availability, these metrics could be controlled through the allocation strategy. However, as a central theme of this work is the presence of uncertainty and disturbances, it must be assumed that this is not possible. Specifically, it can be expected that transient phenomena will occur due to disturbances and structural changes in the system, such as hardware interrupts and the activation or reconfiguration of components. Assuming that these occurrences cannot be predicted, they must be addressed through feedback.

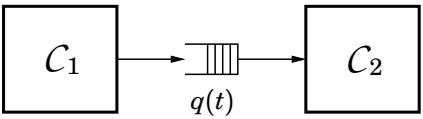

Figure 7.1 Two components connected through a FIFO-queue. $q(t)$ signifies the number of elements in the queue.

7.2 State related performance metrics

The two aspects of performance that will be discussed here are

- integrator dynamics and
- state transition events.

Queues and integrators

Asynchronous communication between components in software systems is commonly done through FIFO-queues. This is practiced both in consumer grade multimedia frameworks such as GStreamer [79] and more academically oriented actor-based languages such as CAL [12] [31]. A common problem that arises in such designs is the regulation of queue sizes, as this affects both end-to-end computational latency and storage requirements.

If a queue constitutes the connection from component $\mathcal{C}_1$ to $\mathcal{C}_2$, as illustrated in Figure 7.1, and $q(t)$ denotes the number of queue entries at time t, its resource dependent dynamics could be described by

$$\begin{aligned} y_1(t) &= k_1 u_1(t) \\ y_2(t) &= k_2 u_2(t) \\ q(t+h) &= q(t) + h(y_1(t) - y_2(t)) \end{aligned} \tag{7.1}$$

From an allocation point of view, an imbalance between y_1 and y_2 results in either starvation of $\mathcal{C}_2$ or unbounded queue length. Therefore, the feed forward strategy should strive for $y_1 = y_2$. Transient effects can then be attenuated through feedback.

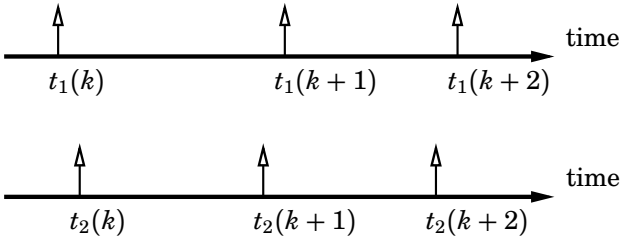

Figure 7.2 Synchronization between two event sequences is one possible objective that can be solved using feedback resource management. The synchronization error, $e_s(k) = t_1(k) - t_2(k)$, is difficult to address using the feedforward strategy as this relies on collecting information over a number of events.

State transitions

State models are commonly used for software systems, defining the behavior in terms of states and state transitions. A state transition is caused by some internal or external dynamic and the passing from one state to another represents an event that is significant to the software. This could for instance be the completion of a computation, e.g. the decoding of a video frame or the termination of an optimization. This thesis will specifically consider changes in the number of completed component cycles n_i, which are signified by the cycle completion events.

Given a number of state transitions over the time period h, as controlled through the allocation strategy, some applications will be sensitive to exactly when these occur. Keeping them uniformly distributed is a common goal, essentially the objective of classic scheduling techniques. Another possible scenario is to synchronize the state transitions between two components. This situation could arise when synchronizing sensors readings, such as audio and video capture, or as a strategy to reduce the complexity of a larger control problem. The synchronization problem could be seen as a generalization of the deadline problem, as the latter would arise if one sequence is set deterministically.

Let t_1 and t_2 denote two event sequences, as illustrated in Figure 7.2, generated as cycle completion events by the components we want

to synchronize. Using Equation 5.5 describing the cycle dynamics

$$t(k+1) = t(k) + \frac{C(k)}{u(t(k))}, \tag{7.2}$$

the synchronization error $e_s(k)$ is defined as

$$e_s(k) = t_1(k) - t_2(k) \tag{7.3}$$

As with $q(t)$, perfect allocation would result in no synchronization error, but given transient disturbances, a feedback approach could be employed to drive it to 0.

7.3 Hardware resources

Availability of the CPU resource is limited by hardware performance. Power and heat are two types of constraints in this setting. In this thesis the problem of thermal management is considered.

Normally a CPU can only operate properly if the temperature is kept below a certain level but if there no active cooling, as is the case in many embedded platforms, this must be respected through control of CPU power. The options to do this include voltage- and frequency scaling and idling (e.g. executing the HLT instruction [42]), the last of which will be used here. The main reason for this is that the speed of the CPU directly affects the component model parameters and any on-line estimates would then change due to control action. Controlling the power through U and then imposing the limit

$$\sum_{i=1}^{N} u_i \leq U \tag{7.4}$$

simplifies the estimation strategy. Control can be effectuated through limiting the available CPU-time using an RBS framework. The resource level U is then determined by the thermal control algorithm, which becomes a part of the resource controller.

Thermal control

Given that the temperature is modeled with first order dynamics with slow disturbances, a PI-controller [10] is a simple and effective choice, though anti-windup measures must be added to handle effects from control signal saturation.

The pure PI-controller is defined as

$$u(t) = K(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau) \tag{7.5}$$

By discretization of the dynamics using a forward Euler approximation, constraining the control signal U to the interval $[0, 1]$ and adding an anti-windup tracking term to the integral part, the resulting control algorithm, described as pseudo-code, is

```
e := r - y_T;
V := K*e + I;
U := sat(V, 0, 1);
if (Ti > 0) then
    I := I + K*h*e/Ti + K*h/Tr*(U - V);
else
    I := 0;
endif
```

where h is the sampling interval.

7.4 Case study — Encoding Pipeline

The prototypical system to be considered is a conversational video pipeline as displayed in Figure 7.3. The software part consists of three tasks, an audio encoder, a video decoder and a network stack. The encoders are assumed to have private access to capturing hardware and it is also assumed that they are capable of variable rate execution. The encoders are connected to the network through FIFO-queues, QA and QV. In order to send a network packet, the stack requires one frame of audio and video.

In order to evaluate the performance of this system, the following metrics are defined

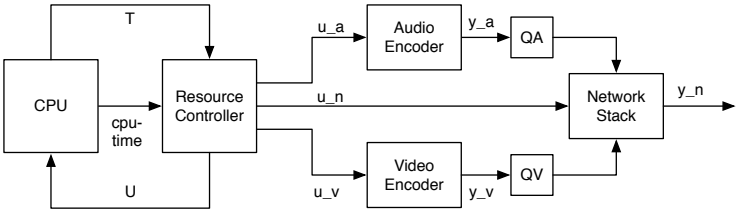

Figure 7.3 An overview of the conversational video pipeline

- **Sync error.** It is disturbing to the human eye when video and audio is out of sync and therefore it is natural to consider the difference in encoding timestamp between the corresponding audio and video frames. If both tasks are assumed to be cyclic and $t_a(k)$ and $t_v(k)$ denote the encoding timestamps for audio and video frames respectively, then

$$\begin{aligned} t_a(k+1) &= t_a(k) + \frac{C_a(k)}{u_a(t(k))} \\ t_v(k+1) &= t_v(k) + \frac{C_v(k)}{u_v(t(k))} \end{aligned} \tag{7.6}$$

  would be the corresponding dynamics. The sync error $e_s(k)$ is then defined as

$$e_s(k) = t_a(k) - t_v(k) \tag{7.7}$$

- **Latency.** Delay in the conversation is also an important quality metric and for this set up it will be the end to end latency, i.e. the delay from capture to network. If $q_a(t)$ and $q_v(t)$ are the number of elements at time t in QA and QV respectively, then let the average encoding latency $e_l(t)$ be defined as the sum of the encoding delay $D_{a,v}$ and the network delay D_n.

  The encoding delay is modeled as

$$D_{a,v} = \frac{\dfrac{C_a(k)}{u_a(t_a(k))} + \dfrac{C_v(k)}{u_v(t_v(k))}}{2} \tag{7.8}$$

which is the the average of the audio encoding time and video decoding time and the network delay as

$$D_n = (\frac{q_a(t) + q_v(t)}{2} + 1)\frac{C_n(k)}{u_n(t_n(k))} \tag{7.9}$$

i.e. the time it takes to process the queue backlog plus one cycle time for the packet itself. $e_l(t)$ is then defined as

$$e_l(t) = D_{a,v} + D_n \tag{7.10}$$

or in words, the computational delay combined with the network backlog in the queues.

- **Queue dynamics.** Using the cycle dynamics expressed in (5.3), the dynamics of q_a and q_v is modeled as

$$\begin{aligned} q_a(t+h) &= q_a(t) + h(y_a - y_n) \\ q_v(t+h) &= q_v(t) + h(y_v - y_n) \end{aligned} \tag{7.11}$$

Control design

Latency It follows from (7.10) and (5.6) that latency can be controlled through minimizing the queue lengths and then keeping a uniform steady state cycle time across all components. The approach taken in this work is to combine a feedforward control based on the total amount of resource with a feedback *re-allocation* to reduce queue length.

The nominal feed forward controls are computed by combining Equation (7.4) with

$$y_a = y_v = y_n \tag{7.12}$$

which gives

$$\begin{bmatrix} u_a^{ff} \\ u_v^{ff} \\ u_n^{ff} \end{bmatrix} = \begin{bmatrix} k_a^{-1} \\ k_v^{-1} \\ k_n^{-1} \end{bmatrix} \frac{k_a k_v k_n}{k_a + k_v + k_n} U \tag{7.13}$$

To control the queue lengths, the feedforward controls are then modified with a feedback term u_q. As the control system cannot violate

(7.4), u_q will be applied as

$$\begin{bmatrix} u_a \\ u_v \\ u_n \end{bmatrix} = \begin{bmatrix} u_a^{ff} \\ u_v^{ff} \\ u_n^{ff} \end{bmatrix} + \begin{bmatrix} \frac{k_v}{k_a + k_v} \\ \frac{k_a}{k_a + k_v} \\ -1 \end{bmatrix} u_q \tag{7.14}$$

subject to the constraints that the resulting controls $u_i \geq 0$.

To calculate $u_q(t)$, the closed loop dynamics of the queues are evaluated. It is assumed that the queues will be of equal length in steady state (see the section on sync error) so the feedback can be designed with any one in mind. Recall that

$$q_a(t + h) = q_a(t) + h(y_a - y_n) \tag{7.15}$$

Assume that the objective is to drive the queue length to some predetermined length r (e.g. zero). To achieve proportional control, let

$$h(y_a - y_n) = K_q(r - q_a(t)) \tag{7.16}$$

This converges to r for all $0 < K_q < -2$. Let u_q denote a feedback term, by which some of the resources allocated to C_n is transferred to C_a and C_v, thereby regulating the relative rates of queue item production and consumption. Substitute

$$\begin{aligned} y_a &= k_a(u_a^{ff} + u_q \frac{k_v}{k_a + k_v}) \\ y_n &= k_n(u_n^{ff} - u_q) \end{aligned}$$

and and solve for u_q to obtain the actual controls.

Sync error If C_a/u_a is approximated by $(k_a u_a)^{-1}$, it follows from the definition (7.7) that

$$e_s(k + 1) = e_s(k) + (k_a u_a)^{-1} - (k_v u_v)^{-1} \tag{7.17}$$

As there is a finite combined flow of CPU resource to the audio and video encoder, the approach taken here is to introduce u_s as a feedback term, modifying the feedforward allocation so that

$$e_s(k+1) = e_s(k) + (k_a(u_a + u_s))^{-1} - (k_v(u_v - u_s))^{-1} \tag{7.18}$$

is driven towards zero. While this seems to interfere with the queue control, the difference in time scale between the event-to-event dynamics makes its effects on the slower queue controller negligible. In fact, it is seen in Section 7.5 that controlling the sync error actually greatly simplifies the queue control. For proportional control, let

$$(k_a(u_a + u_s))^{-1} - (k_v(u_v - u_s))^{-1} = K_s e_s(k) \tag{7.19}$$

Under deterministic circumstances the sequence $e_s(k)$ will converge to zero for any $K_s \in (0, -2)$. Given the variations in the cycle execution times, some care should be taken when selecting K_s as the noise can drive the system unstable. As this work is done without a detailed noise model, K_s is chosen conservatively as -0.5. Then solve for u_s under the constraint that $u_a + u_s \geq 0$ and $u_v - u_s \geq 0$.

The resulting control structure is presented in Figure 7.4.

7.5 Simulation results

Simulation environment

In order to evaluate controls, a simulation environment has been developed in Python. The system dynamics is approximated by discretization with a time step of 1 ms, which incurs quantization on the cycle completion time stamps. This is, however, assumed to be of little effect as the variation due to noise is orders of magnitude greater for these simulations.

Cycle execution times have been generated as $D_i + X_i(k)$ where D_i is an a-priori unknown constant and $X_i(k) \in \exp(0.1D_i)$. The realizations used for the presented simulation results are shown in Figure 7.5. The randomness is meant to model both software execution time uncertainty and the stochastic properties of a modern CPU, including the

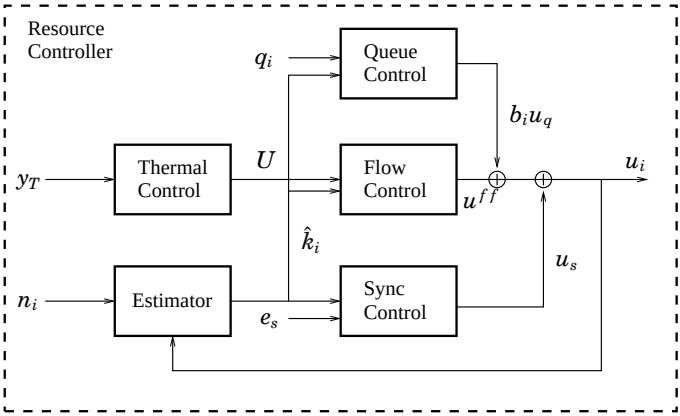

Figure 7.4 The resulting control structure including estimation, feed forward flow control based on the estimated model parameters and the available CPU resource and feedback based on queue state and sync error. b_i refers to the coefficients in Equation (7.14).

effects of caches, the memory bandwidth gap and deep pipelines. The D_i values were chosen randomly so that the resulting k-parameters would lie between 10 and 100. A real sequence of cycle times for a video decoder is provided for comparison in Figure 7.6. h is globally defined as 1.

Thermal control

The thermal model of the CPU is based on [33], but to make the effects of the thermal dynamics more visible in the simulations, the parameters have been scaled so the dynamics are faster. This makes the effects of control and disturbances more prominent. In the simulations, the parameters in Equation 5.7 is chosen as $a = 2$ and $b = 1.5$. The main purposes of the thermal model are

- to provide a scenario for the varying availability of CPU resource and
- to show how physical models can be combined with the software models,

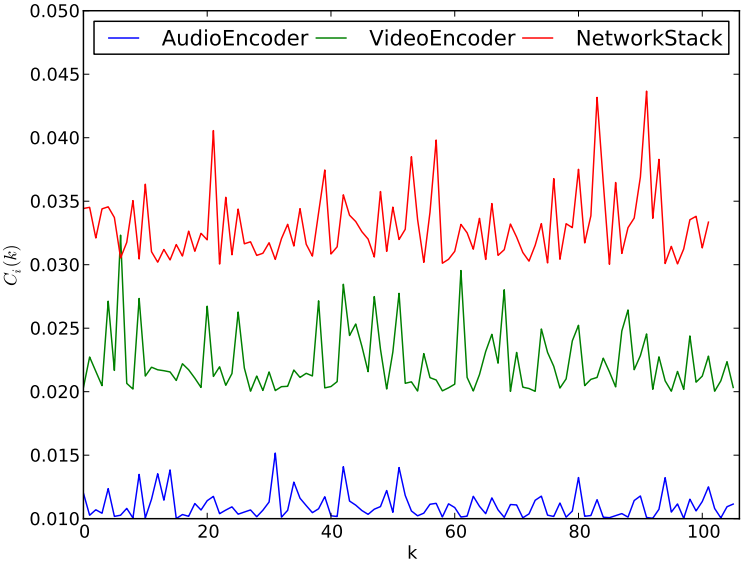

Figure 7.5 Generated cycle times used in the simulation examples below.

so switching for more realistic parameters would not change the design decisions significantly though the thermal controller must then be retuned. However, as the focus of this thesis is on the application performance as affected by both hardware and software, more advanced temperature control strategies are left for future research.

A scenario where a disturbance enters occurs at 5 seconds is shown in Figure 7.7. This could be a situation where the unit is left in direct sunlight that causes an insolation effect of 15 degrees C / s. The controller keeps the temperature by throttling the available CPU-time. As the temperature approaches the set point, the total utilization is lowered to about 80%. At 5 seconds, the disturbance causes the temperature to rise and the controller responds by lowering the utilization even further, settling at about 55%. If there are no active cooling measures or direct measurements of external disturbances, the set point

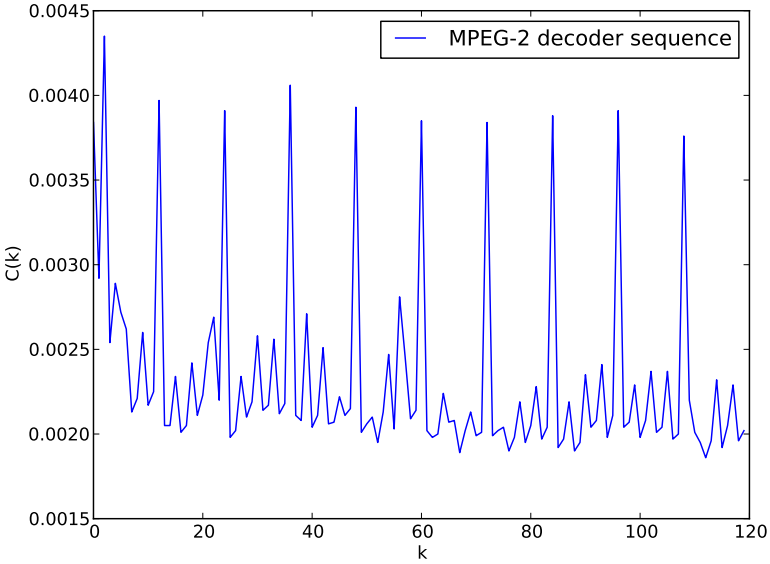

Figure 7.6 A sequence of execution times for an MPEG-2 decoder working on a video stream. This encoding standard interleaves complete image descriptions with delta descriptions. Decoding a complete image is significantly harder, causing the high peaks.

must be sufficiently below the critical level to keep the CPU from overheating.

Parameter estimation

Figure 7.8 shows the estimated execution rates and corresponding k_i-parameter estimates over the same simulation. The discontinuous nature of the virtual flows is evident in these plots. Note that it takes some time before the network stack starts to execute and this is because of the queue-controller. It throttles the network stack while the queues are filled and because of this, the k-parameter estimator needs more time to form $\hat{k}_n$. This causes the large overshoot in the queue length before it settles on the desired level.

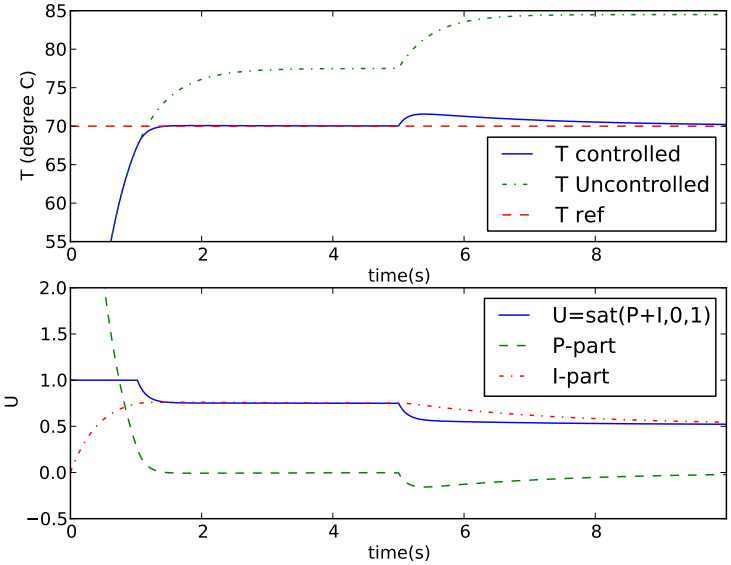

Figure 7.7 PI control of temperature with a constant disturbance of 15 degrees C / s entering at 5s. The uncontrolled dynamics are shown for comparison.

Even though the actual execution rate changes over time, the estimate mean remains stable while the variance increases as the execution rates drop in the later part of the simulation, as seen in the upper plot. This is because a single event being outside or inside the estimation window will affect the estimate more. The window-based estimation scheme will break down when the rate drops below 1 cycle per second.

Latency performance

As there is no information about future demands for the CPU resource, a reasonable strategy is to minimize latency at all times. This is done by utilizing all available CPU-time while respecting the temperature set-point, thereby reducing the execution time for all tasks and by con-

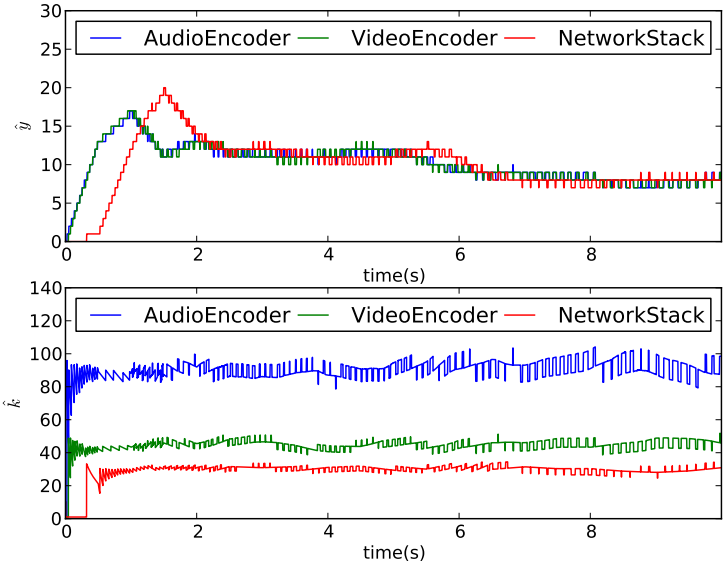

Figure 7.8 Rate and parameter estimation for all three tasks. Rates are estimated by counting events with a sliding time window with length 1 second. The network stack (red) lags behind in the beginning due to queue control.

trolling the queues. The reason why the queue controller is not trying to drive the queue lengths to zero is that this could cause blocking in the network stack, which in turn would reduce the accuracy of $\hat{k}_n$. This could be treated by designing a better estimator.

The latency control performance is displayed in Figure 7.9. A scenario without queue control is provided for comparison and the problem with this is evident. Even though the system reaches steady state, where the queue lengths no longer change significantly, effects of the initial transient remains and cause significantly higher latency.

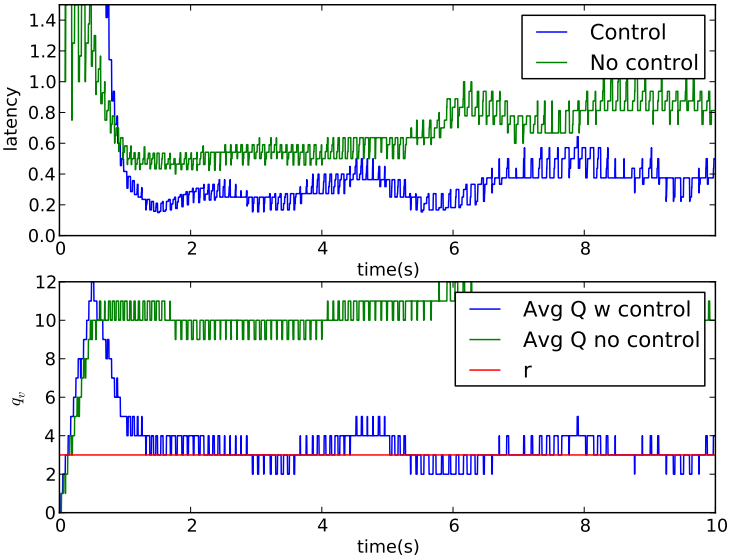

Figure 7.9 End-to-end latency and average queue length compared with and without control. The uncontrolled case (green) is about 2-3 times worse than what it obtained through control (blue).

Sync performance

The simulations reveal the importance of the sync controller. Figure 7.10 shows that the sync error while left uncontrolled will actually drive the system to a stall. The reason for this is that the queue controller uses the average queue length to do the re-allocation. Figure 7.11 shows that even before the k-parameter estimates converge, the sync controller keeps the audio and video stream tightly together. This means that the queue lengths are actually the same, an assumption which can then be safely used by the queue controller.

Figure 7.12 shows how the queue lengths diverge quickly without sync control and the resulting re-allocation by the queue controller actually starves both audio end video encoder. It would theoretically be

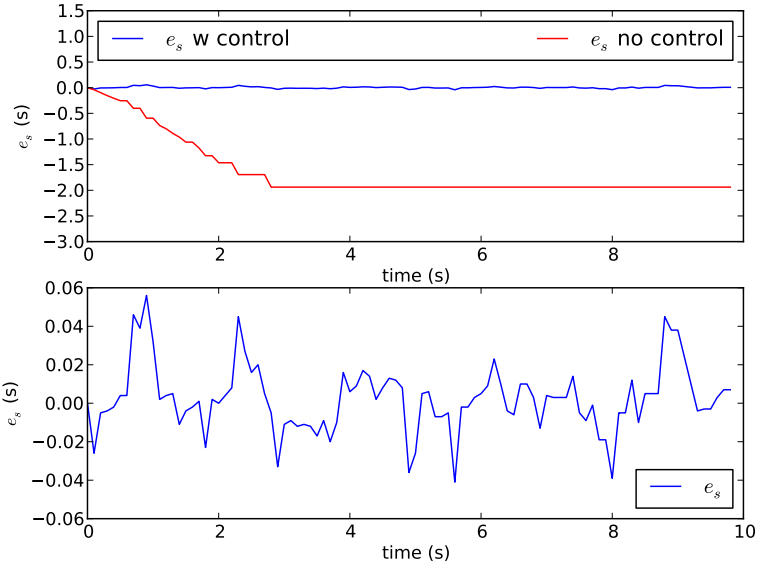

Figure 7.10 Sync error compared with (blue) and without (red) control. In the uncontrolled case, the encoding pipeline stalls which is why the encoding error seems to remain constant after 2.5 seconds. The lower plot shows the sync error from the upper plot in detail.

possible to form a MIMO-controller to handle both sync and queues in the same control law, but recall that the queue controller is a discrete time system that uses resource flow semantics and therefore cannot in a simple way utilize information about individual events. The sync controller on the other hand operates on the event sequence and thereby has access to the cycle completion time stamps.

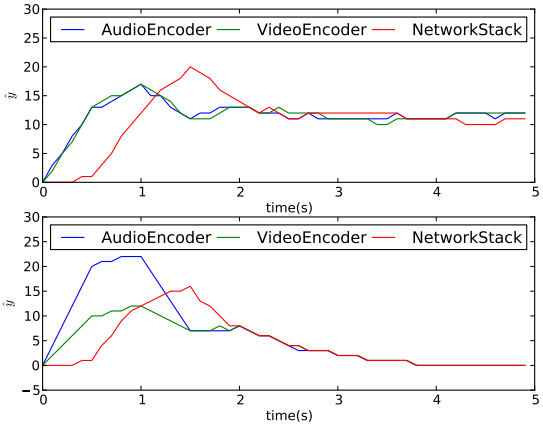

Figure 7.11 Execution rates compared with and without sync control. The curves have been averaged over a window of 0.1 s to provide better visibility.

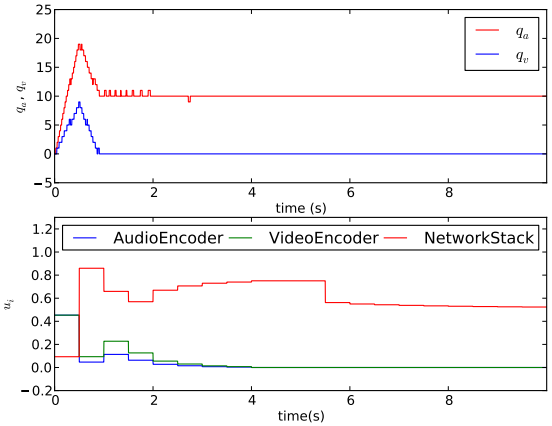

Figure 7.12 Queue lengths for a scenario with no sync control. The resulting allocation is based on the average queue length $(q_a + q_v)/2$.

8

Implementation and examples

This chapter describes a test system including synthetic tasks, estimator and allocator based on the Linux CFS scheduler used to perform the experiments presented in Chapter 6.

8.1 Motivation

During the process of research, a prototype resource management system was implemented to allow for experimentation and to provide insight into platform design problems. The framework was developed with special care not to make use of specialized kernel patches or extensive external libraries, as such dependencies could result in portability issues. It also serves as a demonstration that resource management can be employed using an off the shelf OS, which is especially important for consumer products where time-to-market is an important consideration.

8.2 Resource management architecture

Figure 8.1 shows a schematic of the system. The blue block signifies a standard Linux kernel with CFS group scheduling capabilities (v2.6.24

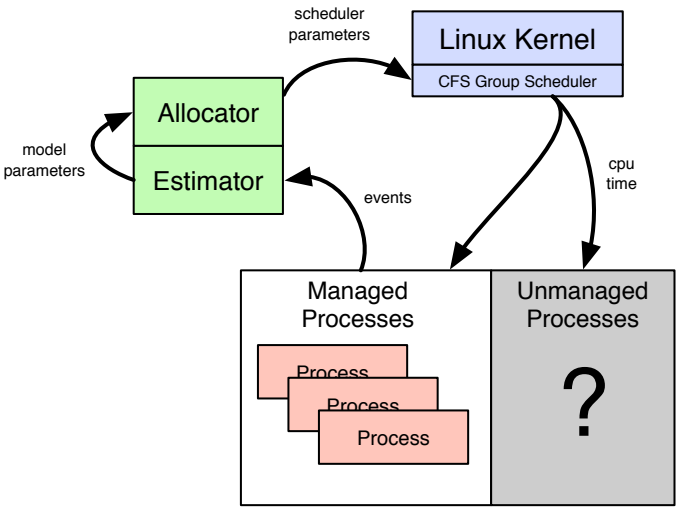

Figure 8.1 Resource management architecture.

or later up to at least v2.6.35).

The green block represents the resource management framework, which in turn consists of an allocator and estimator. These two are contained within the same process for data sharing purposes but as two separate POSIX threads.

The red blocks represent processes that are managed by the framework, meaning that they execute in a reservation and supply the estimator with data. It is assumed that these account for the majority of the resource consumption in the system.

MIPC - a minimalistic IPC protocol

Data is transmitted between components using a minimalistic IPC protocol developed for use with the resource manager. Data is sent as datagrams and MIPC supports both local UNIX sockets and IP sockets. As the estimation techniques support missing measurements, the potential overhead of using TCP-based connections can be eliminated.

MIPC itself only handles sending raw byte data so the client will

need to specify interfaces on top of that, typically by sharing C-struct definitions. The important MIPC-operations are

- `mipc_connect_server`
- `mipc_connect_client`
- `mipc_send`

and the full API is found in Appendix A. The protocol supports the bare minimum required for the framework and for comparison, the compiled binary is less than 10 kbytes in size and only have dependencies to the C standard library and libc, while DBus, a popular interprocess communications mechanism in desktop Linux distributions [26], is over 100 kbytes in size and have dependencies to over 1 Mbytes of additional libraries.

Unmanaged processes

It is here assumed that the majority of the resource requirements comes from a subset of the running processes. The remaining is seen as noise. Should these components require a noticeable amount of resources, this will affect the estimate of the relation between the allocated share and the resulting execution rate, thus effectively make the CPU seem slower.

Processes or threads

Keeping a component based abstraction level is an important goal, as this gives increased flexibility to the software designers. By using processes to track resource usage rather than threads, the anatomy of the components is hidden from the resource manager. Whether a process consists of one or several threads of execution should not matter in this framework.

Experiment setup tools

A range of Python based tools were developed in order to facilitate the scripting of test scenarios. A Python implementation of MIPC is used to signal processes. For operations that require root level privileges and access to non-standard system calls, a few C-based utilities were created and then wrapped in Python code.

8.3 Measuring time and resource consumption

The framework requires the managed components to push data to the estimation algorithm in the form of cycle completion events. These events contain

- a time stamp and
- a reading of the accumulated CPU-time for the component.

The resource measurement is performed as the time stamp is taken, as sending and processing introduce latencies. It also means that the resource cost of measuring is factored into the overall component resource requirements, thereby distributing the overhead rather than centralizing it on the estimator.

Measurements of resource consumption have been done using the `clock_gettime()` [67] system call rather than `getrusage` [24] as it provides better precision. An alternative would to be use the cpuacct-subsystem available with control groups, but the system call was chosen as it is both simpler to use and more efficient than parsing text files.

Reservations with CFS Group Scheduling

The group scheduling functionality (see Section 3.2) that was introduced with Linux v2.6.24 offers an easy to use way to do soft reservations. From an experimental point of view, this requires some extra effort as the allocation theory developed assumes hard reservations.

Shares are calculated by the algorithm as percentage of the total CPU capacity. This must then be translated into an integer number, as used by CFS, in such a way that the desired share equals the integer weight / the sum of all weights. To avoid quantization problems, the total available shares have been selected to be 10000. The CFS scheduler does not allow for an individual reservation period to be set on a process basis so apart from share, no additional parameters must be set.

Effectuating the allocation requires writing the shares to the virtual files in a control group type file system.

Synthetic components

In order to run experiments with relevant load profiles, a synthetic component was implemented and instrumented with logging and a MIPC-based reconfiguration interface. The components execute in accordance with the cyclic component model (see Section 5.3) and with parametrized behavior. Cycle time is drawn randomly from an interval specified at startup and changed periodically. The parameters that govern the behavior are

- **k_min, k_max** - controls the distribution of the cycle times.
- **change_interval** - determines how often cycle times are randomized.
- **rate** - the rate set-point.

The components also log all completed cycles together with a snapshot of the parameter set.

Estimator implementation

The estimation is a passive component in this framework and updates estimates only when needed, in this case triggered by the allocation thread. A MIPC-server is set up to collect all incoming data events from the managed components, which is then stored in a record for each individual client. The client submits a unique identifier, in this implementation the process id of the main component thread, which is then used as a primary key in a table containing all the managed components.

The implementation supports individual estimator functions for each component and supplies three default methods for rate estimation:

- time window (counting events over an interval)
- event window (FIR-structure, time for a fixed number of events)
- autoregressive filter of the event arrival intervals (IIR-structure)

As all events are stored, the estimation history can be replayed at the conclusion of the experiments and compared with the component log files.

Allocator implementation

The allocator is implemented as a periodic thread running at 1 Hz. The work order is

- pull parameters from the estimator
- iterate the incremental solver from Chapter 6 until the potentials are within the tolerance level
- effectuate the allocation by writing to the cgroup file system

The allocation algorithm utilizes a red-black b-tree [25] to sort the components by potential, which allows for robust performance. It is straight forward to both add and remove tasks from the tree, thereby allowing the structure to persist between iterations. This circumvents the often heavy set-up portions of off-the-shelf solvers.

A system parameter is the termination threshold. The algorithm checks the difference in potential between the highest and lowest level and if they are close enough, the solver terminates. In the simulations used for this thesis, the tolerance is set to 0.001.

8.4 Example runs

This section presents the results from a sequence of experiments run on a desktop Linux computer. The hardware was a 2.4 Ghz Pentium 4 with 512 Mb of main memory running a Linux 2.6.27-based Debian system. The background noise consists of the software that runs on a typical Debian desktop, including the X11/Gnome graphical environment, as well as an Apache web server and a MySQL database engine.

Estimation and control

The first experiment is to validate the estimation and control strategy. A single software component with constant but unknown cycle execution time is here controlled using feedforward based on an estimation of the k-parameter.

Figures 8.2 and 8.3 shows the scenario using one second time window estimation and a 15 long event window estimation respectively.

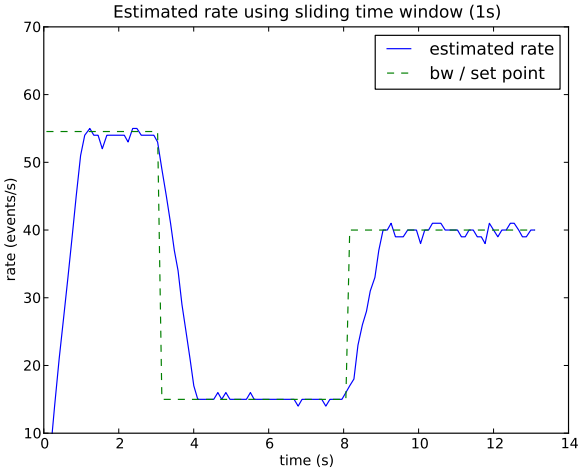

Figure 8.2 Estimation and control using time window

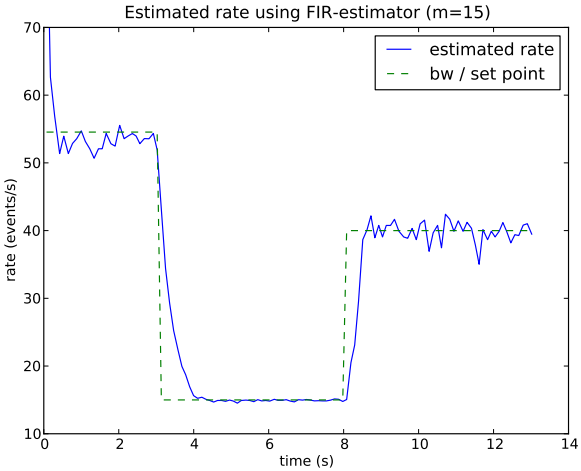

Figure 8.3 Estimation and control using event window

The time window performs well at higher rates, having more information to form the estimate, while the event window estimator is more responsive to change.

Constrained allocation

In this scenario, three components are running in a situation where the system is overloaded. The components change their cycle execution rates randomly every 3 seconds, with k_max/k_min equal to 2. Figure 8.4 shows the estimated rates over time and Figure 8.5 displays the cost with a comparison with static worst case allocation. Towards the end of the experiment, the component set is close to their worst case cycle times, and this causes the system cost to approach that of the worst case allocation.

To study how the performance depends on the level of uncertainty, a series of experiments were run with increasing k_max/k_min. In scenarios with low uncertainty, the dynamic strategy performs worse than the static alternative, as the estimated parameters will, due to the presence of noise, never be completely correct. However, as uncertainty grows, the dynamic strategy shows significant performance advantages, which is shown in Figure 8.6.

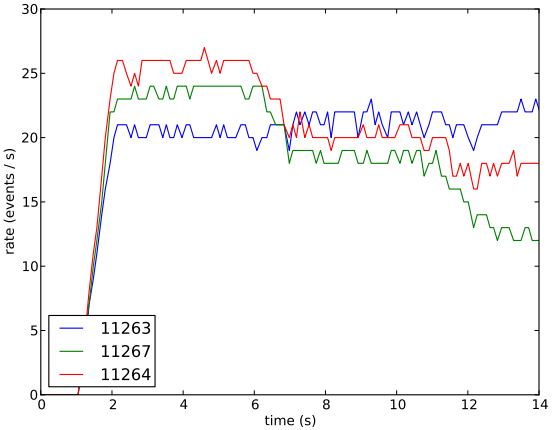

Figure 8.4 Allocation in an over-utilized system with rates estimated using a one second time window.

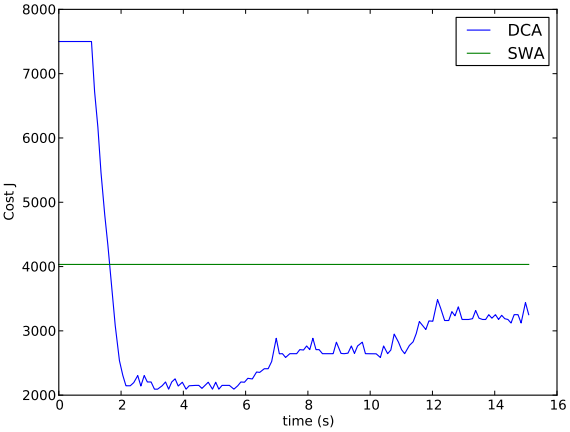

Figure 8.5 Cost for Dynamic Convex Allocation (DCA) vs Static Worst case Allocation (SWA) over time.

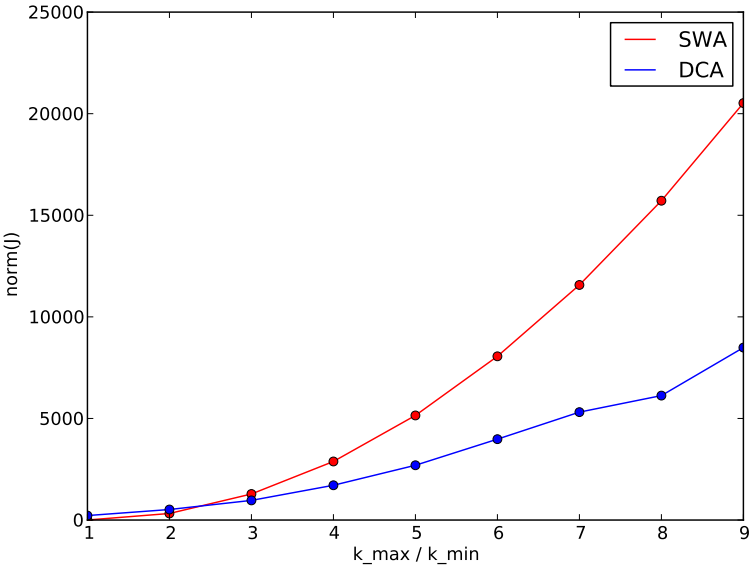

Figure 8.6 Average Cost for Dynamic Convex Allocation (DCA) vs Static Worst case Allocation (SWA) as uncertainty grows.

9

Conclusions

> It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience.
>
> –*Albert Einstein*[30]

9.1 Summary

The topic for this thesis is resource management under uncertain conditions, with focus on embedded systems. The contributions in this field are: a model for resource allocation for rate-based software components, an algorithm for solving convex allocation problems suitable for media platforms and a control scheme for software components with multi resource dependencies. The overall goal has been to find ways to overcome the limitations of worst case-based designs through adaptivity and feedback control. A short discussion on the contributions is given below.

Models for resource allocation

The currently dominant methods for resource management in embedded systems are based in real-time scheduling theory. While these methods are at this point very precise, they are also constrained by the need for prior information. The models used in real-time contexts

are meant to be applicable to many types of software, but this expressiveness also makes them complex. An important principle in control design is to find simple models that capture the essence of the problem, an idea that inspired the cyclic component model.

The cyclic component model is designed to mimic the input-output models used traditionally for control in order to simplify the application of control theoretic results. It is through these similarities that cyclic components can be integrated into dynamic models containing both software and hardware.

In order to estimate the dynamics of cyclic components on-line, it is necessary to find alternatives to sample-based approaches. This is partially due to that information is only available from the components as they output completed results, but also from the fact that a system can be constructed from components working on many different time scales. This will make it hard to find sample intervals. By instead formulating the states in continuous terms, this problem is removed and also makes it easier to integrate the estimations into event-based dynamics.

Convex allocation algorithm

Deciding on how to spend constrained resources is about making compromises, which is the domain of optimization. The algorithm presented in this thesis provides flexibility when designing policies for sharing computational resources, while still being lightweight enough to be used for adaptivity on limited hardware. By allowing the mixing of cost functions, a system designer can control the trade-off made during overload, even if neither software nor hardware specifications are known at design time. Specifying performance in application centric metrics, proposed in this thesis to be the desired execution rate, makes it easy for 3rd party developers to express the needs of their applications.

Control of multi-resource dependent components

Components that have dependencies to more than one resource are important building blocks for cyber physical systems. Finding system wide policies for resource management requires modeling of the entire supply chain, where components that influence more than one flow will likely have a high impact on performance. The scheme presented

in this thesis uses control-like structures as unforeseeable disturbances can only be attenuated through feedback.

The case study of a conversational video pipeline exemplifies several problems that must be dealt with in such systems, including synchronization and supply/demand imbalances. It also demonstrates some effects resulting from a resource where supply depends on utilization through some dynamic, in this case the thermal properties of the CPU.

9.2 Future work

Below is given some future directions of research based on the work presented in this thesis.

Event-based control and sensing

Resource management is to a large extent about answering the questions "how much" and "when". This thesis has been about systems where the number of events are large and the problem formulations concerned with the quantity of events generated. In situations where the events are few, an event-based control approach could be more effective. For mobile systems, the trade-off between sensing and acting is one interesting direction.

Stability and convergence of adaptive reservations

If adaptive resource management is to be employed in safety critical or otherwise high risk situations, theory to prove convergence and stability in dynamic situations is necessary. While the optimization algorithm presented will converge to the optimum, it is not clear what will happen to components that are started or resumed from suspension in overload conditions. This is essentially a reinforced learning or dual control situation, but the simplicity of the component models used could make it possible to find efficient strategies.

Mixed models for cyber physical systems

The conversational video example presented only contains resource dependencies in one direction. In more complex situations, it is likely that the software will be controlling hardware, which in turn can require

power and other resources to function. The modeling of such a system would require further development of the resource flow concepts presented in this thesis.

A

Listings

A.1 MIPC

```
#define MIPC_FAILED -1
#define MIPC_SUCCESS 1

#define MIPC_MODE_LOCAL 1
#define MIPC_MODE_INET 2

#define MIPC_MAX_MSG_SIZE 1024
#define MIPC_PORT_BASE 40000

#define MIPC_PATH "/tmp/mipc/"

struct mipc_msg {
        int size;
        void *data;
};

typedef struct mipc_msg mipc_message;
typedef int mipc_connection;

int mipc_create_socket(char *path);
int mipc_exists(char *path);
```

```
void* mipc_loop(void *arg);
int mipc_connect_server(int addr,
                        int (*recieve)(mipc_message *msg),
                        int mode);
mipc_connection mipc_connect_client(int addr);
mipc_connection mipc_connect_inet_client(char *hostname,
                                         int port);
int mipc_send(mipc_connection conn, mipc_message *msg);
mipc_message* mipc_new_message(void);
void mipc_free_message(mipc_message *msg);
```

B

Bibliography

[1] Luca Abeni and Giorgio C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS '98)*, pages 4 – 13, Madrid, Spain, 1998.

[2] Luca Abeni and Giorgio C. Buttazzo. Adaptive bandwidth reservation for multimedia computing. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, Washington, DC, USA, 1999. IEEE Computer Society.

[3] Luca Abeni, Giuseppe Lipari, and Giorgio C. Buttazzo. Constant bandwidth vs proportional share resource allocation. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems (ICMCS '99)*, volume 2, pages 107 – 111, Florence, Italy, 1999.

[4] Luca Abeni, Luigi Palopoli, Scuola Superiore, and Jonathan Walpole. Analysis of a reservation-based feedback scheduler. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS 2002)*, pages 71–80, Austin, Texas, USA, 2002.

[5] ACTORS: Adaptivity and control of resources in embedded systems. `http://www.actors-project.eu`, April 2008.

[6] David P. Anderson, Shin Tzou, Robert Wahbe, Ramesh Govindan, and Martin Andrews. Support for continuous media in the DASH system. In *Proceedings of the 10 International Conference on Distributed Computing Systems (ICDC '90)*, pages 54 –61, Berkeley, CA, USA, 1990.

[7] Android.com. `http://www.android.com`, 2010.

[8] Aquosa. `http://aquosa.sourceforge.net/`, 2010.

[9] K.E. Årzén. A simple event-based PID controller. In *Proceedings of the 14th IFAC World Congress*, 1999.

[10] Karl Johan Åström and Richard M. Murray. *Feedback systems: an introduction for scientists and engineers*. Princeton University Press, 41 William Street, Princeton, New Jersey, 2008.

[11] Sanjoy K. Baruah, Aloysius K. Mok., and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium (RTSS '90)*, pages 182–190, Lake Buena Vista, Florida, USA, 1990. IEEE Computer Society.

[12] Shuvra S. Bhattacharyya, Gordon Brebner, Jörn W. Janneck, Johan Eker, Carl von Platen, Marco Mattavelli, and Mickaël Raulet. OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems. *ACM SIGARCH Computer Architecture News*, 36(5):29–35, 2009.

[13] Stephen P. Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge University Press, Cambridge, 2004.

[14] Giorgio C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Kluwer Academic Publishers, Dordrecht, Netherlands, 1997.

[15] Giorgio C. Buttazzo, Marco Spuri, and Fabrizio Sensini. Value vs. deadline scheduling in overload conditions. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, Pisa, Italy, 1995.

[16] Marco Caccamo, Giorgio C. Buttazzo, and Lui Sha. Capacity sharing for overrun control. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000)*, pages 295 – 304, Orlando, FL, USA, 2000.

[17] Marco Caccamo, Giorgio C. Buttazzo, and Lui Sha. Elastic feedback control. In *12th Euromicro Conference on Real-Time Systems (ECRTS 2000)*, pages 121–128, Stockholm, Sweden, 2000.

[18] Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein, and Haoqiang Zheng. Group ratio round-robin: O (1) proportional share scheduling for uniprocessor and multiprocessor systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 36–36, 2005.

[19] Anton Cervin and Johan Eker. The control server: A computational model for real-time control tasks. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS 2003)*, pages 113 – 120, Porto, Portugal, 2003.

[20] Anton Cervin, Johan Eker, Bo Bernhardsson, and K.E. Årzén. Feedback–feedforward scheduling of control tasks. *Real-Time Systems*, 23:23–53, 2002.

[21] Computational geometry algorithms library. `http://www.cgal.org`, 2010.

[22] Hojung Cha, Jaehak Oh, and Rhan Ha. Dynamic frame dropping for bandwidth control in MPEG streaming system. *Multimedia Tools and Applications*, 19(2):155–178, 2003.

[23] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. *Approximation algorithms for bin packing: A survey*, pages 46 – 93. PWS Publishing Co, Boston, MA, USA, 1997.

[24] IEEE Computer Society. Portable Applications Standards Committee. *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard Base Specifications*. IEEE, 3 Park Avenue, New York, NY 10016-5997, U.S.A., 2004.

[25] Thomas H. Cormen. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[26] Software/dbus. `http://www.freedesktop.org/wiki/Software/dbus`, 2010.

[27] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 1–12, New York, NY, USA, 1989. ACM.

[28] Tech analysis: Crackdown 2 demo. `http://www.eurogamer.net/articles/digitalfoundry-crackdown2-demo-blog-entry`, June 2010.

[29] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. *SIGOPS Oper. Syst. Rev.*, 33(5):261–276, 1999.

[30] Albert Einstein. On the method of theoretical physics. *Philosophy of Science*, 1(2):163–169, 1934.

[31] Johan Eker and Jörn W. Janneck. CAL language report. Technical Report UCB/ERL M03/48, University of California at Berkeley, 2003.

[32] Alexandre P. Ferreira, Daniel Mosse, and Jae C. Oh. Thermal faults modeling using a rc model with an application to web farms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS 2007)*, pages 113–124, Pisa, Italy, 2007.

[33] Yong Fu, Nicholas Kottenstette, Yingming Chen, Chenyang Lu, Xenofon D. Koutsoukos, and Hongan Wang. Feedback thermal control for real-time systems. In *Proceedings of the 16th Real-Time and Embedded Technology and Applications Symposium (RTAS 2010)*, pages 111–120, Stockholm, Sweden, 2010.

[34] Tobias Geyer. *Low Complexity Model Predictive Control in Power Electronics and Power Systems*. Cuvillier Verlag, Göttingen, 2005.

[35] Ashvin Goel, Jonathan Walpole, and Molly Shor. Real-rate scheduling. In *Proceedings of the 10 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)*, pages 434 – 441, Toronto, Canada, 2004.

[36] S. Jamaloddin Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of the 13th IEEE Conference on Networking for Global Communications (INFOCOM '94)*, 1994.

[37] Michael Grant and Stephen P. Boyd. CVX: Matlab software for disciplined convex programming, version 1.21. `http://cvxr.com/cvx`, 2010.

[38] Michael González Harbour. FRESCOR: Framework for real-time embedded systems based on contracts. `http://www.frescor.org`, April 2008.

[39] Tian He, John A. Stankovic, Michael Marley, Chenyang Lu, Yin Lu, and Tarek Abdelzaher. Feedback control-based dynamic resource management in distributed real-time systems. *Journal of Systems and Software*, 80(7):997–1004, 2007.

[40] Toivo Henningsson and Anton Cervin. Comparison of LTI and event-based control for a moving cart with quantized position measurements. In *Proceedings of the European Control Conference*, 2009.

[41] Ralf Guido Herrtwich. The role of performance, scheduling and resource reservation in multimedia systems. In *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, pages 279–284, London, UK, 1991. Springer-Verlag.

[42] Intel Corporation, Santa Clara, CA, USA. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M*, June 2010.

[43] Damir Isovic and Gerhard Fohler. Quality aware MPEG-2 stream adaptation in resource constrained systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS 2004)*, pages 23–32, Catania, Italy, 2004.

[44] Kevin Jeffay and Steve Goddard. A theory of rate-based execution. In *Proceedings the 20th IEEE Real-Time Systems Symposium, (RTSS '99)*, pages 304 – 314, Phoenix, AZ, USA, 1999.

[45] Hiroyuki Kaneko, John A. Stankovic, Subhabrata Sen, and Krithi Ramamritham. Integrated scheduling of multimedia and hard real-time tasks. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, Washington, DC, USA, 1996.

[46] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer-Verlag Berlin, Heidelberg, Germany, 2004.

[47] Edward A. Lee. Cyber-physical systems-are computing foundations adequate. In *Proceedings of the NSF Workshop On Cyber-Physical Systems*, Austin, TX, USA, 2006.

[48] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33 – 42, 2006.

[49] Caixue Lin and Scott A. Brandt. Improving soft real-time performance through better slack reclaiming. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium, 2005 (RTSS 2005)*, Miami, FL, USA, 2005.

[50] Mikael Lindberg. A survey of reservation-based scheduling. Technical Report ISRN LUTFD2/TFRT--7618--SE, Department of Automatic Control, Lund University, Sweden, 2007.

[51] Mikael Lindberg. Constrained online resource control using convex programming based allocation. In *Proceedings of the 4th International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID 2009)*, San Francisco, CA, USA, 2009.

[52] Mikael Lindberg. A convex optimization-based approach to control of uncertain execution platforms. In *Proceedings of 49th IEEE Conference on Decision and Control (CDC 2010)*, Atlanta, GA, USA, 2010.

[53] Mikael Lindberg. Convex programming-based resource management for uncertain execution platforms. In *Proceedings of the Workshop on Adaptive Resource Management (WARM 2010)*, Stockholm, Sweden, 2010.

[54] Mikael Lindberg and K.E. Årzén. Feedback control of cyber-physical systems with multi resource dependencies and model uncertainties. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS 2010)*, San Diego, CA, USA, 2010.

[55] Giuseppe Lipari and Sanjoy K. Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS 2000)*, pages 193 – 200, Stockholm, Sweden, 2000.

[56] Giuseppe Lipari and Sanjoy K. Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*, Taipei, Taiwan, 2001.

[57] Jan Marian Maciejowski. *Predictive control: with constraints*. Pearson Education Limited, Edinburgh Gate, Harlow, 2002.

[58] Cucinotta Palopoli Marzario, Tommaso Cucinotta, Luigi Palopoli, Luca Marzario, and Giuseppe Lipari. Adaptive reservations in a Linux environment. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 238 – 245, 2004.

[59] Luca Marzario, Giuseppe Lipari, Patricia Balbastre, and Alfons Crespo. IRIS: a new reclaiming algorithm for server-based real-time systems. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)*, pages 211 – 218, Toronto, Canada, 2004.

[60] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the International Conference on Multimedia Computing and Systems*, pages 90 – 99, 1994.

[61] MobileRobots Inc, Amherst, NH, US. *Pioneer 3 Operations Manual*, 2006.

[62] A Mok, X Feng, and D Chen. Resource partition for real-time systems. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTAS 2001)*, pages 75 –84, Taipei, Taiwan, 2001.

[63] John B. Nagle. On packet switches with infinite storage. *IEEE/ACM Transactions on Networking (ToN)*, 35(4):435 – 438, 1987.

[64] Ocera. `http://www.ocera.org/`, 2010.

[65] Martin Ohlin. Feedback Linux scheduling and a simulation tool for wireless control. Licentiate Thesis ISRN LUTFD2/TFRT--3240--SE, Department of Automatic Control, Lund University, Sweden, 2006.

[66] Shuichi Oikawa and Ragunathan Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the 5th IEEE Real-Time Technology and*

Applications Symposium (RTAS'99), Vancouver, British Columbia, Canada, 1999.

[67] The single UNIX ® specification, version 2: clock_gettime. `http://opengroup.org/onlinepubs/007908775/xsh/clock_gettime.html`, Jan 1997.

[68] Abhay K. Parekh and Robert G. Gallager. *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single Node Case*, pages 533 – 546. IEEE, 1 edition, 2007.

[69] Ragunathan Rajkumar, Chen Lee, and Dan Siewiorek. A resource allocation model for QoS management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 298 – 307, San Francisco, CA, USA, 1997.

[70] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. *Resource kernels: a resource-centric approach to real-time and multimedia systems*, pages 476–490. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[71] Saowanee Saewong and Ragunathan Rajkumar. Cooperative scheduling of multiple resources. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS '99)*, page 90, Phoenix, AZ, USA, 1999.

[72] J. Sandee, W. Heemels, and P. van den Bosch. *Case Studies in Event-Driven Control*, volume 4416 of *Lecture Notes in Computer Science*, pages 762–765. Springer Berlin / Heidelberg, 2007.

[73] Vanessa Segovia and K.E. Årzén. Towards adaptive resource management of dataflow applications on multi-core platforms. In *Proceedings of Work-in-Progress Session at ECRTS 2010*, Brussels, Belgium, 2010.

[74] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round-robin. *IEEE/ACM Transactions on Networking (TON)*, 4(3):375–385, 1996.

[75] Marco Spuri and Giorgio Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, 1996.

[76] Ralf Steinmetz. Human perception of jitter and media synchronization. *Selected Areas in Communications*, 14(1):61 – 72, 1996.

[77] Dimitrios Stiliadis and Anujan Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking (ToN)*, 6(5):611–624, 1998.

[78] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.

[79] Wim Taymans, Steve Baker, Andy Wingo, Ronald S. Bultje, and Stefan Kost. Gstreamer application development manual. `http://gstreamer.freedesktop.org/data/doc/gstreamer/0.10.30/manual/manual.ps`, 2010.

[80] Versalogic Corporation, Eugene, OR, US. *Model VSBC-8 Reference manual*, 2007.

[81] Xen. `http://www.xen.org`, 2010.

[82] Xenomai. `http://www.xenomai.org`, 2010.

[83] P Xingang, P Goyal, X Guo, and H Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the USENIX 2nd Symposium on OS Design and Implementation (OSDI '96)*, pages 107–122, 1996.

[84] Melanie N. Zeilinger, Colin N. Jones, Davide M. Raimondo, and Manfred Morari. Real-time MPC–stability through robust MPC design. In *Proceedings of the Joint 48th IEEE Conference on Decision and Control and 28th Chinese Control Conferenc*, 2009.

[85] Lixia Zhang. *A new architecture for packet switching network protocols*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1989.

Department of Automatic Control
Lund University
Box 118
SE-221 00 Lund Sweden

Document name
LICENCIATE THESIS

Date of issue
September 2010

Document Number
ISRN LUTFD2/TFRT--3249--SE

Author(s)
Mikael Lindberg

Supervisor
Karl-Erik Årzén

Sponsoring organisation

Title and subtitle
Adaptive Resource Management for Uncertain Execution Platforms

Abstract

Embedded systems are becoming increasingly complex. At the same time, the components that make up the system grow more uncertain in their properties. For example, current developments in CPU design focuses on optimizing for average performance rather than better worst case performance. This, combined with presence of 3rd party software components with unknown properties, makes resource management using prior knowledge less and less feasible.

This thesis presents results on how to model software components so that resource allocation decisions can be made on-line. Both the single and multiple resource case is considered as well as extending the models to include resource constraints based on hardware dynamics. Techniques for estimating component parameters on-line are presented.

Also presented is an algorithm for computing an optimal allocation based on a set of convex utility functions. The algorithm is designed to be computationally efficient and to use simple mathematical expressions that are suitable for fixed point arithmetics. An implementation of the algorithm and results from experiments is presented, showing that an adaptive strategy using both estimation and optimization can outperform a static approach in cases where uncertainty is high.

Key words
adaptive, resource management, real-time systems

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
120

Recipient's notes

Security classification