Resource Management and Prioritization in an Embedded Linux System

Fredrik Johnsson

Olle Svensson

LUND UNIVERSITY

Department of Automatic Control

Msc Thesis
ISRN LUTFD2/TFRT--5952--SE
ISSN 0280-5316

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

Printed in Sweden by Media-Tryck
Lund 2014

Abstract

This master thesis tackles the problem of limited computing resources on a camera that is executing computing applications together with image acquisition and streaming. The thesis was carried out at Axis Communications in cooperation with the Department of Automatic Control at Lund University. The problem of limited resources on an Axis camera is handled by a two part solution where a resource manager (RM) distributes the available resources and services can adapt their service level (SL) in order to finish their jobs on time. The solution is based on game theory, where services are players, varying their service levels in order to get a good match between given resources and their computing requirements. This service level adaptation scheme is implemented for the streaming service on the camera and for some test services, performing mathematical operations. The resource manager is incorporated into `systemd`, and uses `cgroups` [16] to distribute the computing capacity. The experimental results show that the resource manager is fully operational and capable of managing and prioritizing resources as intended on the embedded system.

Acknowledgements

There are many people we would like to thank for helping us with this project. We would especially like to thank our supervisors, Umut Tezduyar-Lindskog, Axis and Martina Maggio, LTH Department of Automatic Control, the project would not have been possible without their feedback and knowledge. Would also like to thank engineering manager Pontus Bergendahl, Axis, who was responsible for actually making this master thesis happen.

Contents

1

Introduction

This master thesis treats the problem of assigning limited resources to embedded cameras. Axis cameras are used for the study. Axis is a company founded and based in Lund that manufactures network relayed surveillance cameras and video encoders.

1.1 Problem Formulation

To save energy and make better use of the available hardware, there is a trend to have multiple resource intensive applications running on Axis cameras. These applications are *services*, that should execute within a certain time and with variable precision requirements. At the same time, reliable and consistent video frame rate and quality is a necessary condition to be fulfilled, which calls for running video streaming services in isolation, without being subject to the interference of other services. The two conflicting requirements make different services compete for resources like CPU and RAM. This may result in poor performance of the camera, when the execution scenarios bring the camera under a load that is heavier than usual. For example, when answering calls from the network, the camera is subject to a heavier demand.

In these scenarios it would be advisable to have a technique to diminish the load produced by services that are not necessary, ideally without af-

fecting their timing properties. The quality of service reduction is often addressed via the introduction of service levels, where services can decrease the load generated on the hardware by lowering their service level, therefore producing results that have a lower quality. When the load conditions are back to optimal, the service can increase the service level, to provide the best available quality without harming the execution of the most important applications.

This work implements a game theoretic mechanism based on the Game Theoretic Resource Manager (*GTRM*), developed at Lund University [15] and a library that lets services implement the service level adaptation and the communication with a global resource manager. The goal is to demonstrate that it is possible to use *GTRM* on an Axis cameras running Linux and to use it to manage applications competing for resources. The evaluation features the streaming application, competing with load generators. The image quality is taken as the service level and the cameras are assumed to have a desired frame rate, therefore introducing for each frame a deadline of $1/desiredframerate$.

1.2 Related Work

The problem of allocating resources to running applications and at the same time varying the quality of the computation of these applications to avoid overload conditions has been addressed in many different ways, sometimes also using game theory.

For example, Wei et al. [25] have used game theory to assign resources to fully parallelizable tasks. Contrary to their approach, in our case applications are not fully parallelizable and could execute sequential sections. In some of these sections, assigning more resources would not speed up the application, while in others the benefits will be significant. The resource manager developed in this thesis, therefore, needs to act based on actual measurements.

Subrata et al. [23] solved the problem of balancing the load in grid com-

puting by applying game theory. Here the players are machines that wants to maximize their profit by finishing jobs that arrive according to a Poisson process. Grosu and Chronopoulos [13] made similar work with load balancing strategies. The load is distributed amongst different competing players which would hopefully reach a common state which would benefit all the players the most. However, there is no cooperation on the application side to reach a consensus.

Many resource managers are feedback oriented. The first resource managers that make explicit use of control theory and feedback loops was developed by Lu et al. [14], Steere et al. [22] and Eker et al. [12]. However they do not implement the concept of varying the computation quality, or service level.

The QoS-based Resource Allocation Model (Q-RAM) was proposed by Rajkumar et al. [19] for managing multidimensional resources. Here it is desired to minimize the QoS constraints while maximizing the total utility. The solution is centralized and every application receives a certain quality to be used for the computation and cooperates with the architecture by enforcing that quality. However, the amount of communication needed to achieve this goal is non-negligible and therefore it is not advisable for a video surveillance and streaming systems where the network bandwidth is used to stream the surveillance videos.

A solution that both manages the resources and the service level of an application is proposed in the ACTORS project [11], but just as the solutions proposed by [19, 21, 10] the solution is centralized. Separating the service-level adjustment and the resource management has been proposed in the context of network bandwidth allocation [20].

GTRM [15], that is used here as a reference point, decouples the resource assignment and the service level selection, but it is implemented with `SCHED_DEADLINE`, which is not included in the Linux kernel used for Axis cameras. Moreover, Axis cameras are already exploiting the resource allocation capabilities offered by `systemd`. In this work, a *GTRM*-like approach is implemented to be applicable to Axis cameras.

1.3 Outline

The remaining of this report is organized as follows.

- Chapter 2 gives a detailed description of the software and hardware used during this project.
- Chapter 3 details the implementation and design decisions, defining therefore how the resulting camera acts.
- Chapter 4 discusses the use cases that where taken as a reference for the project. These describe the product functionality and are relevant to test the resulting prototype.
- Chapter 5 outlines how the product was tested and shows the results obtained with the tests and discusses the findings of the thesis.
- Chapter 6 finally concludes the report and highlights future works.

2

Background

2.1 Game Theoretic Resource Manager

The aim of the resource manager is to make sure that the running applications have acceptable performance levels. The decision about how much resource to allocate to each application is based on its performance. The application's performance is measured in terms of a *matching function*, that tells how good the match is between the resource given to the application and the corresponding deadline. The assumption behind this is that the resource distribution determines the execution time to complete a job. The applications are supposed to be made of jobs.

The matching function is calculated as the difference between the applications deadline and the execution time of the jobs. Ideally the matching function should be zero. When zero, the application has just enough resources to meet its deadline running with some service level. When positive, the resources are abundant to execute the jobs timely, indicating that the job is done before deadline. A negative matching function means that too little resource is assigned, indicating that the application has missed or will miss its deadline.

The framework consists of two parts: the service level adaptation and the resource management. These two parts are independent and decoupled.

Service Level adaptation

The Service Level (SL) defines the quality of the service provided by the application. In the case of the streaming application, the service level is defined as the quality of the image to be streamed, but for a different application, the service level can mean something else. The main property of the SL is monotonicity. An increase in SL gives an increase in the required resource on the application's side. The idea is to change the SL to optimize the utilization of the amount of resources available.

When the performance is too low, the application is supposed to decrease its service level, while when the performance is too high, the application will increase the quality of the performed computation. This will make sure that the application is always presenting valid result in time but with varying quality, as a trade-off. This adaptation is done by the application itself, without the resource management policy interfering with it.

There are of course many possible ways of adjusting the service level, two of them were considered. In the first case, the application only considers information that are internally available, while in the second case, the application receives "hints" from *GTRM* and follows those hints.

The *independent adaptation* simply multiplies the current service level, the matching function and a constant scale factor ε. This adaptation decreases the service level if the performance is negative and increases it if the matching function is positive. The scaling factor ε slows down the adaption rate to avoid instability. The service level sl_i of an application i, is calculated from the matching function, f_i and from the previous service level as,

$$sl_i(t+1) = sl_i(t) + \varepsilon \cdot (f_i(t) \cdot sl_i(t)). \tag{2.1}$$

The *coordinated adaptation* follows a suggestion given by the resource manager, that includes also the variation of the resource allocation, that is the virtual platform vp. In fact, the resource manager sends to the application a *performance multiplier* PM_i that is used as an estimation of how much the service levels should change to match the current allocation, that

is unknown on the application side.

The performance multiplier PM_i is computed as,

$$PM_i = (1 + f_i) \cdot (vp_i(i+1)/vp(i)), \tag{2.2}$$

and the applications sets the new service level as

$$sl_i(t+1) = sl_i(t) + (\varepsilon \cdot sl_i(t) \cdot PM_i). \tag{2.3}$$

The test applications used in this thesis simply makes some random computations in an infinite loop to demand and make use of resources. In this case each iteration corresponds to one "job", but a job could for example be the processing of an image frame done by an image processing application. The quality of service for these applications is the amount of computations done each iteration, thus a higher service level means more computations done each iteration. One could also model the service level the other way around, meaning we have a fixed amount of computations done each iteration, thus increasing the service level would instead increase the amount of computational iterations.

The Test Application

The test application from [15] has been adapted to work with the implementation in this paper. Below follows an explanation of how it works to provide an example of how an application could implement SL adaptation and be managed by the *GTRM*. The applications considered in this paper typically have a periodic task to perform, here called job, with an associated soft deadline. The test application has a linear relationship between the SL and the time it takes for the CPU/CPUs to perform a job described by Equation 2.4,

$$C_{cpu} = a_{cpu} \cdot SL + b_{cpu}, \tag{2.4}$$

where C_{cpu} is the number of times a random number is generated which should be proportional to time it takes the CPU/CPUS to perform the job.

The SL adaptation is performed for each job. The average performance of the last ten jobs are then sent to the RM and the SL adaptation is made as described in the previous section. See the sequence diagram in Figure 3.2 for the application loop.

Resource Management

The Resource Manager (RM) measures the performance of the applications. It tries to distribute the resources in the best possible way to the running applications, for them to meet their performance requirements. The resources are modeled as "virtual platforms". A virtual platform represents a percentage of the total available resources; for example, the amount of time an application is allowed to use the CPU with respect to the other applications. Here "resources" could refer to something else than CPU, such as memory or network bandwidth, depending on what is allocated in the system. The *GTRM* is run in the main loop of `systemd`, (see section 2.2), which checks sockets for messages and dispatches jobs from the incoming messages. This means that the *GTRM* acts in irregular time intervals with the shortest being the time it takes to pass the main loop when there are no incoming messages and the longest depending on the amount of incoming messages and the execution time of the next task queued up in the prioritized queue that stores the jobs from the incoming messages.

Resource Allocation update

The resource update changes the virtual platforms allocated to each running application. It follows the algorithm described in "A Game-Theoretic Resource Manager for RT Applications" [15, page 4].

> [...] the RM assigns resources according to the rule:
>
> 1. it measures the performance[1] $f_i(t)$;

[1] As stated in its definition, f_i is a function of the service level s_i and the virtual platform v_i. However, here we intentionally hide this dependency and report only the dependency on time t, since the RM only measures a value over time.

2. it updates the virtual platform $\tilde{v}_i$ as follows:

$$\tilde{v}_i(t+1) = \tilde{v}_i(t) + \varepsilon_{RM}(t)\Big(-\lambda_i f_i(t) + \sum_{j=1}^{n} \lambda_j f_j(t)\tilde{v}_i(t)\Big), \tag{2.5}$$

   where $\varepsilon_{RM}(t)$ is a step-size sequence;

3. it computes the original value of bandwidth by

$$v_i(t+1) = m\tilde{v}_i(t+1),$$

4. it updates the time $t \leftarrow t+1$ and repeats.

Here $\tilde{v}$ is the normalized virtual platform and m is the number of computing elements. This means that the computed resource allocation v represents the percentage of the total resources allocated to the application and is converted into the actual value during the third step of the algorithm. CPUShares are used here in place of bandwidth, and specifies the minimum relative amount of CPU-time to assign to an application. For instance, if two applications are assigned 100 and 200 CPUShares each, the one assigned with 200 will get twice the amount of CPU-time compared to the other. CPUShares are also independent on the amount of CPU cores and this means that the resource manager does not have to compensate nor keep track of the amount of CPU-cores of the system, as opposed to the bandwidth implementation.

Decoupling

The theory behind the decoupling of the resource allocation and the service level assignment was developed at the Department of Automatic Control at the Lund University. The resulting resource manager is referred to as Game Theoretic Resource Manager (*GTRM*).

Decoupling the two adaptations makes it possible to obtain a linear time complexity for the resource adaptation algorithm. In fact, the amount of

operations that are necessary to perform the adaptation defined by Equation 2.5 depends only on the number of running applications. Also, one of the main benefits of this decoupling is that the task of adjusting the SL is given to the applications that have knowledge of how to tune their parameters in order to adjust them to their needs, to the amount of resource received and to the quality of the computation.

2.2 Systemd and Cgroups

Systemd [17] [18] is a daemon for Linux, that executes system management operations. It is the first process that starts during boot, and thus it is given the PID 1. `Systemd` implements a lot of features for increased performance and system management over previous start up processes, like initd. It also has different features for management of resources, using cgroups [16], which makes it interesting for a resource manager implementation.

Cgroups, abbreviated from control groups, can be used to set the amount of resources, such as CPU or memory, of a process or a group of processes via a virtual file system. This file system forms a tree where the resources of a parent folder are shared by its children. The division of the resources among the children is determined by the amount of "shares" the children has been given.

Each application can be run as a "service" by specifying a service file which defines many different parameters and options. In this file, it is possible to specify which application or applications should be associated with which service and for example how much CPU shall be given to this service. The service file can then be placed in a certain folder in the cgroup file hierarchy, see Figure 2.1.

Different folders are used to represent different cgroup controllers or a combination of controllers. Depending on which controllers are enabled, some features are available, such as limiting CPU and memory. Services can be grouped into different slices and share properties depending on which slice they belong to. One can for example set how much CPU-time

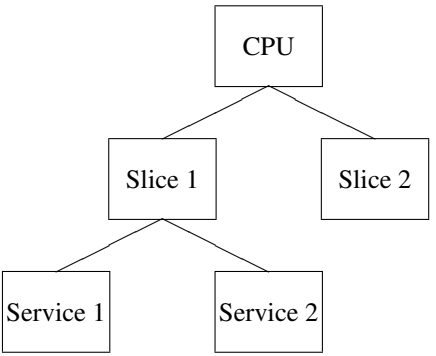


Figure 2.1 The slices and services in the cgroup tree.

Name	CPUShares	% of CPU
CPU	1	100
Slice 1	400	80
Slice 2	100	20
Service 1	200	$2/3 \cdot 80 = 53$
Service 2	100	$1/3 \cdot 80 = 27$

Figure 2.2 Table over assigned CPU for the different units.

shall be given to the applications in the slice and decide how the applications will divide it amongst themselves.

In this implementation of the *GTRM*, the resources are divided using the CPUShares property of the *slices* or *services*, both special cases of the base type *units*. Every slice is represented by a folder in the cgroup tree and has a CPUShares property that decides how much of the available resources

of the parent slice it will get. A slice contains services and services contains applications.

All the resources of a parent slice will be hierarchically divided to the units that are classified under this slice according to how much shares each of them has. A unit that has a third of the total shares of all units on the same level under its parent slice will receive a third of the resources available for the parent. In case of a multi-core system, the shares are distributed over all the CPU cores. For example if two services are given 100 and 300 CPU-shares respectively, running on a four core system, one service may use 100 % of one core and the other 100 % of each of the remaining three cores.

An example of this is shown in Figure 2.1 and Figure 2.2. Note that the top level folder in this example, the controller named CPU, is alone on its level meaning that the CPUShares for this slice do not matter since there is no competition. There are two slices dividing the CPU-controller, the first slice is given four times the amount of CPU-shares compared to the second slice. This means that the applications in slice 1, is assigned 80% of the CPU-time in total. The applications running in slice 1 are defined in two services, which also specify the amount of CPU-shares for each service. The first service is given a total of 200 shares and the second 100 shares. Service 1 will thus be given ⅔ of the shares in slice 1 and service 2 the remaining ⅓.

2.3 Video Streaming

The video streaming application is based upon the GStreamer multimedia framework [7] [8]. The framework is a modular system, where a chain is built by linking elements together in a pipeline to form a process chain. The data flows downstream from a source element, through filter elements and end up in a sink element, see Figure 2.3 for a graphical representation of the pipeline.

The data is contained into buffers. Buffers can contain one or more frames, flowing downstream. On Axis cameras, the source elements receive

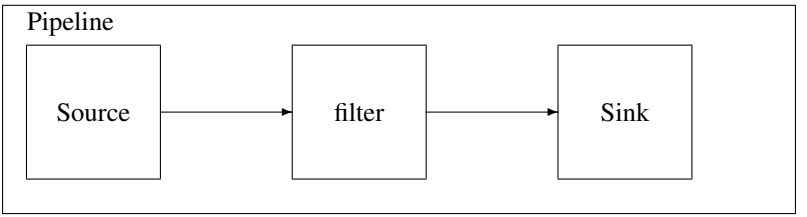


Figure 2.3 The GStreamer pipeline.

the images already compressed by external hardware through another application which reads a file descriptor in order to get the frames from the encoding device. The data is then sent through a number of filters and finally is released to the network by the sink element. This chain is dynamically created depending on different settings like which are beings codecs used (*mjpeg, h264 and more*) or network connection (*http or udp*).

For streaming using `h264` over `rtp`, two pipelines are used which look like this.

```
artpecsrc > cachesink
cachesrc > rtph264pay > rtpbin > udpsink
```

The first pipeline simply fetches images from the camera sensor, in the `artpecsrc`-element to be used by different video streams. *Artpec* is the system-on-a-chip on the camera, providing the images.

The second pipeline consists of different elements depending on what kind of stream is required. The first element will provide the pipeline with incoming buffers produced by the first pipeline. The `rtph264pay`-element [6] payload-encodes the H264 video frames into RTP [9] packets. `rtpbin` is an element which combines the features of many different RTP-elements, allowing multiple RTP-sessions. The final element, `udpsink`, sends data over the network using UDP.

2.4 Sockets and Epoll

This section introduces sockets and epoll for communication between processes on the system.

Sockets

The communication between applications and `systemd` is obtained through sockets [24]. The RM runs in the `systemd` main loop, therefore sockets are used to send data between the RM and the applications.

Sockets are the endpoints for *inter-process communication* (IPC) flows over a network, where the entities communicating may reside on the same physical device or are simply nodes on the same network. These sockets are provided by the `socket()` system routine in Linux, therefore their usage is regulated by the normal operating system APIs.

A call to the `socket()` API function creates a socket and returns an integer, a unique *file descriptor* representing the socket for future usage. A file descriptor in Linux is associated with an open file, where the file can be anything that can be written to or read from. Knowing the descriptor gives read and write access to the socket. A socket has a type, being it either *stream* or *datagram*, the latter is the one used by the *GTRM* socket.

Epoll

Epoll [5] is a Linux system call, allowing the user to listen to multiple sockets simultaneously. The API function `epoll_wait(...)` returns a queue of event-objects containing information about what file descriptors have received datagrams together with other information. By looking at this queue the user knows what file descriptor to read with to obtain the newly arrived data.

2.5 Equipment

The project involves experiments with two different cameras: the M1033 and the P3367, both manufactured by Axis. Both the cameras have

`systemd` installed, although the P3367 runs a more updated version of it.

Axis M1033

The Axis M1033 is a small camera as can be seen in figure 2.4, connected to the network either wired or wireless. It supports multiple H.264 streams and Motion JPEG running at a maximum resolution of 800x600 at 30 frames per second. It has two audio streaming channels, which means that it can both record and play audio clips. [3]

Figure 2.4 The Axis M1033 camera

It is very likely, due to its capabilities, that multiple applications run simultaneously on the hardware, especially due to the ability of recording and playing audio clips simultaneously. In fact, while the video is recorded and sent over the network, it is also possible to run other applications using the collected data, for example some motion detection application.

Axis P3367

The Axis P3367 [4] is a fixed dome network camera, see Figure 2.5 and 2.6, which is capable of multiple H.264 streams as well as Motion JPEG streams. It supports various frame rates and resolutions of up to 5 Megapixel at 12 frames per second. It also supports HDTV 1080p at 30 frames per second and has two way audio streaming capabilities.

Figure 2.5 The Axis P3367 camera, without its dome casing

The power is supplied using Power over Ethernet, therefore the camera does not need a separate power supply and is powered directly through the network cable. It features an *ARTPEC-4* [2] system-on-chip, developed by Axis, which contains a single-core CPU running at 400 MHz and a co-processor dedicated to video analytics.

It is clearly more advanced than the M1033 but it shares the same characteristics of being extremely flexible to execute multiple applications at the same time, especially due to the video analytics co-processor.

These cameras are very common among Axis' customers and also by developers, and will serve as realistic testing platforms for this project. Initially the M1033 was the only camera considered but during development a later version of `systemd` was brought to the P3367 and the project moved

Figure 2.6 The Axis P3367 camera, here in its final form

to that camera instead. Testing will thus only be carried out on the P3367 as the previous camera's `systemd` implementation differs to much to bring the software to both of them.

3

Implementation

This chapter describes the implementation of the *GTRM* framework for Axis cameras. The code was written in C and cross-compiled using Axis' compiler for the corresponding hardware platform. The resource allocation framework was tested in different use cases and with a variety of service files and slices. The resulting plots are generated with Octave.

3.1 Design choices

Introducing service levels in all the applications running on camera is not a realistic approach. This is because there are many different applications, some of which may not even be developed at Axis, and developers are not expected to modify all these applications to implement the performance measurements and service level features needed, despite the modification being trivial in many cases. Also, for some applications it is not possible to identify the concept of service level.

The service level is in this work implemented for the video streaming application and for some test application that generates mathematical load in the system. This thesis only focuses on the implementation of *GTRM* and assumes that a correct implementation of the framework implies also that the proof of convergence of the game theoretic strategy holds, as demonstrated for [15].

Also, the service level assignment is limited to the linear case, where resource requirements (in this case CPU requirements) are a linear function of the service level. It can be argued that the CPU requirement/SL relationship can be linearized around a point and hence still considered linear close to the operating point.

The matching function used to determine the performance of the application is time sensitive and reads as

$$f_i = \frac{D_i}{R_i} - 1 \tag{3.1}$$

where D_i is the soft deadline for the job and R_i is the job response time. This matching function is positive when the resource given is aboundant and negative if it is scarce. It is also zero in case there is a perfect match between the resource given and the service level set by the application. The aim of *GTRM* is to bring f_i to zero for all the running applications. It is assumed that f_i is measurable also for the non-service level aware applications. The matching function is updated as the average of the measured value for the last ten jobs. Resource management in this work is restricted to CPU; cgroups have a controller to handle memory management as well, the *memory* controller, but this one is not installed on the camera.

3.2 Inter-Process Communication

The resource manager is implemented within `systemd` and its code is therefore integrated into the `systemd` code. If an application has a poor matching function it should send information to `systemd` via UNIX-sockets. The Inter-Process Communication (IPC) consists of one socket declared within the `systemd` code and one for each application that the resource manager should monitor. The socket setup is inspired by the "Notify" feature of `systemd` which is used by services that want to notify `systemd` that they have started or about status changes [sysd-notify].

Retrieving new events

The system uses `epoll` to find out which sockets have received new messages. The events returned from `epoll_wait()` are then put in a prioritized queue. The first element in the prioritized queue is then dequeued, calling its callback function. This means that a message might not be read immediately after being retreived from the buffer. Most of the functions used to manage `epoll` are located in `sd-event.c`, which serves as a wrapper for many different events, such as those generated by `epoll`.

The `epoll_wait` prototype is

```
int epoll_wait(int epfd, struct epoll_event *events,
   int maxevents, int timeout);
```

and the function behaves as follows. When there are messages to be read on one of the file descriptors observed by the `epoll` instance identified with the file descriptor `epfd`, the function returns the number of file descriptors that have new messages to be read.

If this happens, `epoll_event *events` points to a buffer that the function fills with the incoming events. The buffer in the end contains a maximum of *maxevents*. The function waits for a maximum `timeout` value.

The `epoll_event` structure follows.

```
1  typedef union epoll_data {
2    void    *ptr;
3    int      fd;
4    uint32_t u32;
5    uint64_t u64;
6  } epoll_data_t;
7
8  struct epoll_event {
9    uint32_t     events; /* Epoll events */
10   epoll_data_t data;   /* User data variable */
11 };
```

The void pointer `epoll_event.data.ptr` will be set to point to a `sd_event_source` structure. This structure is the element that is enqueued

into the prioritized queue. The element contains information concerning the event priority and defines the event type. It also points to a structure containing the file descriptor for the event and its callback function.

The function `sd_event_run` is called once for every iteration of the main loop. The relevant code in this function is shown below.

```
1  _public_ int sd_event_run(sd_event *e,
2    uint64_t timeout) {
3    struct epoll_event *ev_queue;
4    unsigned ev_queue_max;
5    sd_event_source *p;
6    int r, i, m, timeout;
7    ev_queue = new(struct epoll_event, ev_queue_max);
8    m = epoll_wait(e->epoll_fd, ev_queue,
9    ev_queue_max, timeout);
10
11   for (i = 0; i < m; i++) {
12   if (ev_queue[i].data.ptr ==
13     INT_TO_PTR(SOURCE_MONOTONIC))
14     r = flush_timer(...);
15   else if (ev_queue[i].data.ptr ==
16     INT_TO_PTR(SOURCE_REALTIME))
17     r = flush_timer(...)
18   else if (ev_queue[i].data.ptr ==
19     INT_TO_PTR(SOURCE_SIGNAL))
20     r = process_signal(e, ev_queue[i].events);
21   else if (ev_queue[i].data.ptr ==
22     INT_TO_PTR(SOURCE_WATCHDOG))
23     r = flush_timer(e, e->watchdog_fd,
24     ev_queue[i].events, NULL);
25   else
26     r = process_io(e, ev_queue[i].data.ptr,
27     ev_queue[i].events);
28   }
29
30   p = event_next_pending(e);
31   r = source_dispatch(p);
32   return r;
33 }
```

The `epoll_wait` function is called and a for loop goes through all the

file descriptors and checks what their event type is to call the appropriate function to add them to the prioritized queue. The *GTRM* file descriptor is linked to a `IO_SOURCE` event which means that the function `process_io()` will be called in case there is an event on such file descriptor.

Subsequently, the next event is dequeued from the queue and dispatched by calling `event_next_pending()`. The code of this function is shown below.

```
1  static sd_event_source*
2    event_next_pending(sd_event *e) {
3      sd_event_source *p;
4      p = prioq_peek(e->pending);
5      if (!p)
6          return NULL;
7      if (p->enabled == SD_EVENT_OFF)
8          return NULL;
9      return p;
10 }
```

Finally, the `source_dispatch` function is called to activate the callback function linked to the top `sd_event_source*` element.

```
1  static int source_dispatch(sd_event_source *s) {
2    int r = 0;
3    switch (s->type) {
4      case SOURCE_IO:
5          r = s->io.callback(s, s->io.fd,
6        s->io.revents, s->userdata);
7          break;
8      case SOURCE_MONOTONIC:
9          r = s->time.callback(s, s->time.next,
10       s->userdata);
11         break;
12     case SOURCE_REALTIME:
13         r = s->time.callback(s, s->time.next,
14       s->userdata);
15         break;
16     /* More cases covered */
17     case SOURCE_WATCHDOG:
18         assert_not_reached("Wut? I shouldn't exist.");
19   }
```

```
20|     return 1;
21| }
```

The GTRM socket

The socket used by the applications to communicate with the resource manager and by the resource manager to retrieve information about the application health is setup as follows.

The socket is created and added to the `epoll` instance in the function `manager_setup_gtrm`, some information about it follows.

static int manager_setup_gtrm(Manager *m)
Used to setup the socket, event source and file descriptor for the resource manager. Called from `manager_startup` and `manager_reload`.

- m: Reference to the manager.
- Return value: Zero if function ran correctly, otherwise it is set to the corresponding error number.

This function contains the socket creation, obtained through `socket`:

```
fd = socket(AF_UNIX,
  SOCK_DGRAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0);
```

The socket uses the protocol family `AF_UNIX`, which provides efficient communication on the same machine. `SOCK_DGRAM` sets the socket type to be datagram since there is no need to resend missed obsolete data and no messages are expected to be lost when sent over the same machine. `SOCK_NONBLOCK` prevents the socket from blocking during a `read` call when there is no data to be read. A random address name is created and assigned to the socket by a call to `bind`. The file descriptor of the socket is then added to the `epoll` instance, and an `epoll_event` is associated to the file descriptor. The `epoll_event` contains a pointer to an `sd_event_source`. These functionalities are achieved through the call to `sd_event_add_io`.

```
r = sd_event_add_io(m->event, &m->gtrm_event_source,
  m->gtrm_fd, EPOLLIN, manager_dispatch_gtrm_fd, m);
```

The *GTRM* socket has been therefore added to the poll and a corresponding event is subsequently generated whenever a message is sent to that socket. Finally, in the function `manager_setup_gtrm` a priority is given to the `gtrm_event_source`. This affects how the prioritized queue sorts messages from this source type and the priority is set equally to the priority for notify messages.

Once the socket is setup and ready, an environment variable is set in Linux containing the address to the GTRM-socket. The environment variable is used by the applications to retrieve the destination of the messages to be sent about their performance.

GTRM socket's callback function: Once an event has been popped from the prioritized queue the callback function `manager_dispatch_gtrm_fd` is invoked. The callback function contains all the code that should be executed when a message is received on the *GTRM* socket. The function ex-

tracts the data from the message — the PID of the application sending the message, its performance, and its weight. The `hashmap`, which contains all the applications that are managed by GTRM, is then updated with this new data.

static int manager_dispatch_gtrm_fd
(sd_event_source *source, int fd, unit32_t revents, void *userdata)
Called upon receiving the performance of an application.

- source: source of the event.
- fd: file descriptor to the event source.
- revents: Set by the kernel to indicate what event on the file descriptor that triggered the call to this function. For the purposes of this work, it is always the incoming event.
- userdata: Contains a reference to the manager.
- Return value: Not used, always zero.

The function contains some declaration and initialization.

```
1  static int manager_dispatch_gtrm_fd
2    (sd_event_source *source, int fd,
3    uint32_t revents, void *userdata) {
4    Manager *m = userdata;
5    char buf[1024]; // read message
6    int n; // byte size of the read message
7    struct sockaddr_un *from; // sender's adress
8    socklen_t fromlen;
9    rm_app_t *app; // data about the applications
10   rm_app_t *app2;
11   fromlen = 1024;
12   ... // will be shown later
```

Within the function, the `rm_app_t` structure is used.

struct rm_app_t Represents an application being managed and consists of the following fields.

- tid: Applications PID.
- vp: Virtual platform.
- vp_old: Previous virtual platform.
- performance: Matching function of the application.
- weight: The current "weight" of the application.
- happy: Indicates if the application is happy with its current performance. This field was added to prevent application from sending their performance even when they are satisfied and nothing should be needed from the resource manager, eliminating unnecessary computations.
- sa: Socket Address, used to send back the performance multiplier.

A while loop reads the socket one message at a time, until there are no more unread messages.

```
1  // resuming from above
2  do {
3    memset(buf, '\0', 1023);
4    from = calloc(1, sizeof(struct sockaddr_un));
5    app = calloc(1, sizeof(rm_app_t));
6    n = recvfrom(fd, buf, 1024, 0,
7    (struct sockaddr *) from, &fromlen);
8  // with non-blocking sockets
```

```
 9       // if n is negative, there was no message
10       n = recvfrom(fd, buf, 1024, 0,
11       (struct sockaddr *) from, &fromlen);
12       if (n<0)
13         break;
14       if (n>1024) // reading 1024 characters at a time
15         log_error("manager_dispatch_gtrm_fd:
16       received too big message");
17
18       gtrm_char2gtrmstruct(buf, app);
19       pid_t pid = app->tid;
20       app->sa = from;
21       if(hashmap_get(m->gtrm_apps, pid) == NULL) {
22         hashmap_put(m->gtrm_apps, pid, app);
23       } else {
24         app2 = hashmap_get(m->gtrm_apps, pid);
25         gtrm_update_rm_struct(app,app2);
26       }
27     } while(n>0);
28     m->update_gtrm = true;
29     return 0;
30 }
```

The data contained in the message is used to create a `rm_app_t` structure. If the hashmap already contains a `rm_app_t` structure for the specific application, the `rm_app_t` in the hashmap is simply updated by calling the function `gtrm_update_rm_struct(app,app2)`. In the opposite case, the new application is added to the hashmap using the application PID as the key.

Above, a few library function contained in `gtrm_lib` were used. In particular, the `gtrm_char2gtrmstruct` function is used to convert the information between the received string and the GTRM compliant structure.

void gtrm_char2gtrmstruct(char* str, rm_app_t *re) Extracts data from a received string and stores it as a structure instead. The data

sent from an application consists of the PID, performance, weight and if the application is satisfied or not.

- str: String to extract data from.
- re: Struct to hold the extracted data.
- Return value: None, the result is stored in re.

Resource Manager Update

The resource manager is continuously recomputing the amount of resources that is to be given to the applications for as long as all the applications are not completely satisfied. The following code is contained inside the `manager_loop`-function of `manager.c`.

```
1 int manager_loop(Manager *m) {
```

Before the loop actually starts to run, a structure that contains the necessary data for the resource manager, `gtrm_t`, is created and initialized.

struct gtrm_t
Stores various parameters used by the GTRM.

- c1: Constant used for computing ε, determines how much the virtual platforms will be changed.
- c2: Another constant used for computing epsilon similar as c1.
- iterations: Keeps track of how many iterations the *GTRM* has run.
- all_happy: Used to indicate if we have to make any adjustments to the resource allocations.

- num_apps: Total amount of applications that we are managing.
- prev_apps: The amount of applications in the previous iteration.

```
1  gtrm_t *gtrm_t = calloc(1,sizeof(struct gtrm_t));
2  gtrm_t->num_apps = 0;
3  gtrm_t->prev_apps = 0;
4  gtrm_t->iterations = 0;
5  gtrm_t->all_happy = true;
6  gtrm_t->c1 = 0.1;
7  gtrm_t->c2 = 10;
8  while (m->exit_code == MANAGER_RUNNING) {
9  ...
```

Inside the loop, an if-statement makes sure that the resource manager is not run if not needed.

```
1  if ((!(gtrm_t->all_happy) || m->update_gtrm) &&
2    !hashmap_isempty(m->gtrm_apps)) {
```

The first step is to update the number of running applications.

```
1    gtrm_t->prev_apps = gtrm_t->num_apps;
2    gtrm_t->num_apps = hashmap_size(m->gtrm_apps);
```

int gtrm_compute_virtual_platforms
(Hashmap *apps, gtrm_t *gtrm_t)
Calculates the amount of resources (virtual platform) for an application.

- apps: The hash-map containing information about the applications being managed.

- gtrm_t: Struct with parameters used when calculating the virtual platforms.
- Return value: Not used, always zero.

```
1 gtrm_compute_virtual_platforms
2 (m->gtrm_apps, gtrm_t);
3 gtrm_t->iterations++;
```

The virtual platforms are then applied to the applications and a variable set by the dispatch is reset to false.

void gtrm_apply_virtual_platforms(Manager* m)
Computes and applies the amount of CPUShares that each application shall be given.

- m: Reference to manager, used to get the applications
- Return value: None.

```
1 gtrm_apply_virtual_platforms(m);
2 m->update_gtrm=false;
```

The final step inside the loop is to update the performance multiplier to be provided to the applications and log the relevant data.

void gtrm_update_performance_multipliers
(int gtrm_fd, Hashmap *gtrm_apps)

Calculates, updates and sends the performance multiplier to each application, using the performance, virtual platform and the previous virtual platform for each application.

- gtrm_fd: File descriptor used to send the performance multiplier.
- gtrm_apps: Hash-map containing rm_app_t structs for each application.
- Return value: None.

void gtrm_write_log
(Hashmap *gtrm_apps, unsigned int num_applications)
Writes information about the resource management to a log file, which can then be used to generate graphs about the applications behavior and the resource manager allocation.

- gtrm_apps: Hash-map containing rm_app_t structs for each application.
- num_applications: Used to make sure we do not try to print an empty hash-map.
- Return value: None.

```
1        gtrm_update_performance_multipliers
2        (m->gtrm_fd ,m->gtrm_apps);
3        gtrm_write_log(m->gtrm_apps, gtrm_t->num_apps);
4      }
5      }
```

If the loop exits it is necessary to deallocate the `gtrm_t` structure to avoid memory leaks.

```
1    free(gtrm_t);
2    return m->exit_code;
3  }
```

Applying the virtual platforms

The computed values for the virtual platforms need to be distributed to the applications. This is achieved via the following function.

```
1  void gtrm_apply_virtual_platforms(Manager* m) {
```

Three variables are defined for setting the shares, iterating through the hashmap and a structure for each application.

```
1    int shares;
2    Iterator i;
3    rm_app_t* a;
```

`HASHMAP_FOREACH` is a macro defined in `hashmap.h` and it is used to easily iterate through all the elements in the hashmap. The macro takes three parameters, a local variable to store the current element in the iteration, the hashmap to iterate through and an iterator.

```
1    HASHMAP_FOREACH(a, m->gtrm_apps, i) {
```

Since the virtual platform is given as a percentage of the total amount of available resource, each application's virtual platform is multiplied by a constant, `_TOTAL_SHARES`, to give the absolute amount of shares.

```
1      shares = a->vp * _TOTAL_SHARES;
2      manager_set_cpu_shares(m, a->tid, shares);
3      }
4  }
```

Finally the CPUShares of the application is set, end then the loop repeats until all the applications in the system have been updated.

int manager_set_cpu_shares(Manager *m, pid_t pid, int shares)
Sets the CPUShares of an application.

- m: Reference to manager, used to get the applications.
- pid: Process identifier of the application.
- shares: Amount of shares we want to set.
- Return value: Zero if successful, one otherwise.

Setting CPUShares

From the command line one can manually set the amount of CPUShare of an application by running the command, `systemctl set-property 'service name' CPUShares='shares'` followed by `systemctl daemon-reload`. These commands take a lot of time since they involve calls via `DBus` to `systemd`. That is the reason why in this work the resource manager itself is located within `systemd`, so that these calls can be circumvented, by setting the CPUShares directly.

```
1 int manager_set_cpu_shares(Manager *m, pid_t pid, int
      shares){
```

Two local variables are needed, a pointer to a `Unit`, which is a parent class to Service and Slice classes, describing how systemd should handle a process/application, and one to a `CGroupContext`. From the PID of an application, the corresponding Unit pointer can be obtained.

```
1   Unit* u;
2   CGroupContext* c;
3   u = manager_get_unit_by_pid(m, pid);
```

In case the application does not exist, because the application terminated, the Unit pointer produced previously will be a NULL-pointer. The applica-

tion is thus removed from the hashmap, which requires a reset of the virtual platforms, and the function call returns.

```
1   if(u == NULL) {
2   hashmap_remove(m->gtrm_apps, pid);
3   reset_virtual_platforms(m->gtrm_apps);
4   return 1;
5   }
```

If the Unit-pointer obtained was not NULL, a so called `CGroupContext` can be acquired from the Unit-pointer.

```
1   c = unit_get_cgroup_context(u);
```

In this context, the CPUShares can be set to what is desired and it is followed by a necessary call to apply the changes.

```
1   c->cpu_shares = shares;
2   cgroup_context_apply(c, CGROUP_CPU, u->cgroup_path);
3   return 0;
4 }
```

In Section 2.2 the concept of CPUShares was introduced. CPUShares assigns CPU-resources between the application proportionally to the amount of shares assigned to each application. This implies that assigning an application an amount of shares, will give different amount of resources depending on how the shares are assigned to the competing applications. The CPUShares also define a minimum amount of resources, an application can thus receive more resources than specified if there is free resources available. This is different to the approach in [15] where the resources where set exact via `SCHED_DEADLINE`. Some risks are introduced by using `SCHED_DEADLINE` which are not present in the CPUShares approach, e.g., assigning, in total, more resources than available, causing a kernel-panic, or not fully utilizing the system by assigning less resources, in total. `SCHED_DEADLINE` also has to take the amount of CPU cores into consideration, which CPUShares is independent of.

3.3 Service Level Update

The applications using the *GTRM* framework have all a similar structure, the main difference being how they implement the update in the service level to match the resource allocation. In fact, the service level changes are mapped into some parameter changes, that in turn affects the resource requirement. This is different on a per application basis and every application developer knows better what to change within the application to make it require less or more resource and provide a worse or better quality of service.

In general, however, a few elements can be identified. The service level adaptation should run periodically and a socket needs to be established and read during the adaptation phase. Below follows an example of a test application, in order to show how the service level adaptation can be performed and what functions that are provided by `gtrm_app_lib.c`. The test application and library are altered versions of the test application and library `jobsignaler.c` used within [15]. The modifications are mainly due to the presence of sockets, instead of the initial shared memory approach used by `jobsignaler`.

```
1  // Initial declarations, some not reported
2  // because irrelevant [...]
3  uint id;
4  _application_h* myself;
5  ...
6
7  int main (int argc, char* argv[]) {
8    // parsing information from char* argv[].
9    float service_level;
10   float a_cpu, b_cpu;
11   float a_mem, b_mem;
12   double epsilon, weight;
13   double deadline_seconds;
14
15   int jobs;
16   double performance;
```

The application creates the socket address and passes it as an argument to `gtrm_lib_setup_socket`. The file descriptor is then linked to a file

with the application name.

```
1  char* sock_path = "/root/temp/app";
2  unsigned int r_nbr = random_u64();
3  char* sock_name[100];
4  sprintf(sock_name,"%s/%u",sock_path,r_nbr);
5  gtrm_lib_setup_socket(sock_name);
```

int gtrm_lib_setup_socket(char* filename) Sets up a socket to communicate with the resource manager. Reads an environment variable set by `systemd` for the GTRM socket adress and creates a socket adress struct.

- filename: The socket needs a file to work, the parameter specifies its path.
- Return value: Zero, not used.

Subsequently the arguments for the `gtrm_lib_set` function are initialized and passed.

```
1
2  myself->weight = weight;
3  myself->application_id = getpid();
4  uint64_t deadline = (unsigned int)
5  ((double)1000000000 * deadline_seconds);
6  uint64_t ert[1] = {deadline};
7  gtrm_lib_set(myself, 1, ert);
```

Applications can have different job types corresponding to different deadlines within the same code. For example, this is the case of a video decoder/encoder, that could process different types of frames (I/B/P) with different requirements. Encoding an I frame requires to simply transfer the information from one place to the other, while encoding a B and P frame, in fact, requires that the difference between frames are calculated. The test

application has only one job type but the framework directly supports multiple job types. An application can have different job types corresponding to different deadlines. The test application only has one job type.

int gtrm_lib_set (_application_h* a, uint types, uint64_t* ert)
Initializes the application struct with job types and their expected response times. Initializes the application struct with job types and their expected response times.

- a: Struct representing an application.
- types: Number of different job types.
- ert: Array of expected response time for each job type.
- Return value: Exit status of function.

The main loop of the program follows.

```
1  for(;;) {
2    int64_t cpu_requirement = a_cpu * service_level
3    + b_cpu;
4    int64_t mem_requirement = a_mem * service_level
5    + b_mem;
6    int type = 0;
7    id = gtrm_lib_jobsignaler_signalstart
8    (myself, type);
9    // Do the required work
10   do_work(cpu_requirement, mem_requirement,
11   NOISE_PERCENTAGE);
12   gtrm_lib_jobsignaler_signalend(myself, id);
13   id = 0;
14   performance = gtrm_lib_get_performance_number
15   (myself, type);
16
17   // Adapt only if needed
```

```
18      if (performance < -0.01 || performance > 0.01) {
19        myself->happy = false;
20        // send performance to systemd
21        gtrm_lib_send_performance(myself, performance);
22        // if there is no multiplier to read use the
23      // simple service level adaption
24        if (gtrm_lib_update_performance_multiplier
25        (myself) == 0)
26          service_level += epsilon * service_level *
27        (myself->performance_multiplier-1);
28        else
29          service_level += epsilon *
30        (performance * service_level);
31        // saturation
32        if (service_level < MINIMUM_SERVICE_LEVEL)
33          service_level = MINIMUM_SERVICE_LEVEL;
34        if (service_level != service_level) // avoid NANs
35          service_level = 1.0;
36      } else if(myself->happy == false) {
37        myself->happy = true;
38        gtrm_lib_send_performance(myself, performance);
39      }
40    }
41  }
```

In the code, the following functions are used.

int gtrm_lib_signalstart (_application_h* a, uint type)
Indicates the start of a job.

- a: Struct for representing the application.
- type: Type of the job that is started.
- Return value: Identifier of the started job.

int gtrm_lib_signalend (_application_h* a, uint id)
Indicates the end of a job.

- a: Struct for representing the application.
- id: Identifier of the job that has completed.
- Return value: Exit status.

double gtrm_lib_get_performance_number
(_application_h* a, int job_type)
Calculates the performance (matching function) by averaging the performance of the last ten jobs of a specified type.

- a: Struct representing an application.
- type: Which job type for which we want to calculate the performance.
- Return value: Performance of the application.

int gtrm_send_performance_multiplier
(double pm, int fd, struct sockaddr *sa)
Sends a performance multiplier to an application via sockets.

- pm: Performance multiplier to send.

- fd: File descriptor used to send the performance multiplier.
- sa: Socket address.
- Return value: Always zero.

3.4 Video Streaming

This work features the modification of the video streaming application. The proposed solution and implementation is discussed here.

The streaming service, named `Monolith`, streams both video and audio. It handles image compression and filtering, manages several different streams, such as H.264 and MJPEG streams. This application has been modified to implement the service-level feature. The service level alters the image quality by modifying the pipeline to prepare the frame for the video streaming service.

In this pipeline an "identity" element is added. The identity element forwards the received frames to the next element without affecting them. This element is added just after the first (source) element. The identity element calls a callback function by sending a "handoff" signal each time it receives a new buffer, where a buffer can contain one or more frames.

The time between each incoming buffer to the identity element is the "job time" for the streaming application, which is used when calculating the matching function, see Equation 3.1. If a buffer contains only one frame, this job time should correspond to *fps* on the camera.

After the matching function is computed, it is possible to calculate a new service level for the application and to notify the resource manager.

In the streaming application, the relationship between execution time: R, SL, and VP is assumed to be: $R = \alpha \frac{s}{v}$, here the SL and the frame size are set to have a 1:1 relationship. The frame size is set to equal the SL by

updating the parameter *frame_size* in the source element which is read and updated each time an image is fetched from the camera. In the source element update of the frame size, a very simple P-controller is used, where the compression of the images is used as the control signal in order to get the desired frame size. Note that the compression is done on dedicated hardware and not by the CPU being controlled by the *GTRM*, hence the decrease in execution time when lowering the frame size comes purely from the handling of smaller frames in the Gstreamer elements.

void setup_sl_adapt
(float service_level, double epsilon, double weight, double deadline_seconds)
Sets up deadline for computing the matching function and sets up the Inter-Process Communication with the resource management.

- service_level: Initial service level.
- epsilon: Constant which specifies service level adaptation rate.
- weight: Determines how the much of the adaptation will be done by altering the service level or the changing the amount of resources.
- deadline_seconds: This deadline will correspond to the desired frame rate.
- Return value: None.

void sl_adapt() Performs the service level adaptation, notifies the resource manager and writes some logging.

- Return value: None.

The function which initiates the video pipeline is called `cache_video` and is located in `video.c`. The function code is quite complex but here we describe only what the features that are relevant for this thesis.

```
1 static GstElement *
2 cache_video (gpointer key, Props * props,
3   gpointer * user_data) {
4   // Initialization and variable definitions
```

First, all the different elements belonging to the pipeline are defined, including the newly introduced identity element, which is called `gtrm_sl_adapt`. A call to `setup_sl_adapt` sets the variables needed for the service level adaptation.

At first all the different elements for the pipeline are defined, including the identity element, which is called *gtrm_sl_adapt*. A call to *setup_sl_adapt* sets up the what is needed for the service level adaptation.

```
1   GstElement *gtrm_sl_adapt = NULL;
2   setup_sl_adapt (service_level, epsilon,
3   weight, deadline_seconds);
```

The identity element is created by the following calls and the `hand-off-signal` of the element is enabled and set up.

```
1   gtrm_sl_adapt = gst_element_factory_make
2   ("identity", "gtrm");
3   g_object_set (G_OBJECT (gtrm_sl_adapt),
4   "signal-handoffs", TRUE, NULL);
5   g_signal_connect (gtrm_sl_adapt, "handoff",
6   G_CALLBACK (gtrm_sl_cb), src);
```

Finally, the element must be inserted into the pipeline. This is obtained via the following calls.

```
1   gst_bin_add_many(GST_BIN (p), src, gtrm_sl_adapt,
2   convert, caps_filter, sink, NULL);
```

```
3    // More parameter definition here
4    gst_element_link_many (src, gtrm_sl_adapt, convert,
5    caps_filter, sink, NULL);
6  }
```

The callback function for the element consists of a single call to `sl_adapt()`, which is defined in `sl_adapt.c`. This adaptation looks more or less identical to that in the test application described in Section 3.3 and in Equations 2.1 and 2.3. The resulting service level is used to set the size of each frame, where a larger frame has higher image quality.

```
1  static void gtrm_sl_cb (GstElement * identity,
2    GstBuffer * buf, GstElement * src) {
3    int frame_size = sl_adaptation();
4    g_object_set (G_OBJECT (src), "frame_size",
         frame_size, NULL);
5  }
```

3.5 Resource Allocation

The resource allocation mechanism is realized using `cgroups`, `CPUShares` and *slices*. Using slices one can set a minimum amount of available resources for the applications that belong to the slice.

If the applications under a slice do not use all of the resources allocated to them, the unused share is free to be used by other slices. In a hierarchical manner, each slice can have sub-slices, therefore dividing the resource at a finer granularity level. The hierarchy is built within the `cgroups` folder.

The slices in the pie chart of Figure 3.1 represent two different sets of applications. The static yellow slice consists of applications that are not managed by *GTRM* and share the resources according to the predefined setting given by the slice. Applications that belong to this set usually do not vary their resource requirement or are hard real-time, which means they should be given enough resource so that their deadlines are met.

The red set shows two applications managed by the *GTRM* framework. These application might implement the service level adaptation paradigm.

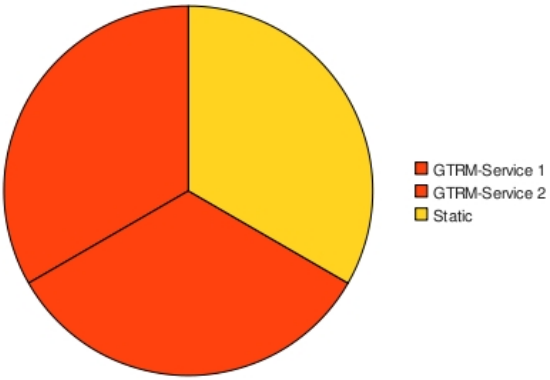


Figure 3.1 Pie chart for resource allocation.

Applications belonging to this slice usually do vary their resource requirements, or can change the quality of the computation to adapt to the available resources. In Figure 3.1 there are two applications (also called services) of this type.

All applications that are managed by the *GTRM* run as services under the GTRM-slice. A service can reserve a minimum percentage of allocated resources referring to its parent slice. In the Pie chart above the static slice would be guaranteed a minimum of ⅓ of the total resources while the two *GTRM* services are given half of the ⅔ reserved by its parent slice, therefore obtaining a minimum of ⅓ of the total available resources each.

In the *GTRM* slice, some of the applications are service level-aware and some are not. Ideally all applications running in the GTRM-slice should, if it makes sense to the application, have a concept of service level. Each application is assigned a weight, that sets how much responsibility for the adaptation is taken care of by the application that changes its service level and how much adaptation should be realized via resource allocation. All applications in the *GTRM* slice must implement the performance evaluation via a matching function and the communication with the resource manager

via a socket.

Resource allocation in practice

The resource split shown in Figure 3.1 is made by creating a `gtrm.slice` file in the folder `/etc/systemd/system`. Any application that wants to take advantage of the *GTRM* capabilities needs to create a service file in the same folder. When the service is started a folder named `gtrm.slice` will be created in the corresponding cgroup controller-folder to represent the slice. The controller is a combination of the CPU-and CPUAccounting, `cpu,cpuacct`. The applications run in this slice will be represented as a folder within the slice-folder. The resulting path to the service will look like this.

```
/sys/fs/cgroup/cpu,cpuacct/.slice/gtrm.service/
```

The unit file to represent the `gtrm.slice` is defined as follows. Here the only parameter specified is the amount of CPUShares. By setting this field the system is given the ability to change the CPUShares of this slice.

gtrm.slice

```
[Unit]
[Slice]
CPUShares=1024
```

Any service file in the `/etc/systemd/system` folder overrides a service file for the same application declared somewhere else. The following describes the service files used by our test applications.

gtrm-test.service and gtrm-test2.service

```
[Unit]
[Service]
ExecStart=/mnt/flash/test-gtrm-app 50 10 0 0 0 0.1 0.5 0.04
CPUShares=100
Slice=gtrm.slice
```

If no such file is created, by default, `systemd` defines a slice called `system.slice` located in `/usr/lib/systemd/system`. All services will belong to this slice unless another slice has been specified.

system.slice

```
[Unit]
Description=System Slice
Documentation=man:systemd.special(7)
DefaultDependencies=no
Before=slices.target
Wants=-.slice
After=-.slice
```

`DefaultDependencies` is set to *no* to disable some non-essential dependencies. The `Before` and `After` fields makes sure that the units are started in the correct order, if necessary delaying one unit to make sure the other starts first. The unit(s) specified in the `Wants-` field will start when this unit is started. In this case the `-.slice` will be started simultaneously as `system.slice`, but the `After` field ensures that `system.slice` will start-up first of them. This slice is given the default amount of CPUShares which is 1024, since nothing is specified in the unit file.

3.6 Sequence Diagram

The sequence diagram in Figure 3.2 describes the flow of execution of the system via pseudo-code. Some function calls that are not relevant have been left out.

Application

The application computes its performance (matching function), via a function called `calculate_performance()`. According to its value, and eventually to the performance multiplier received from the resource manager

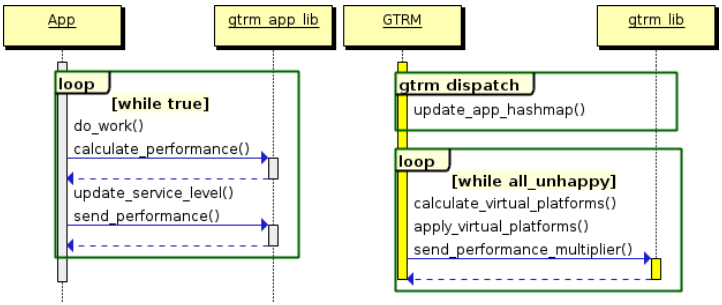


Figure 3.2 Sequence diagram

at the past step, the service level of the application is adjusted by calling `update_service_level()`. Finally, the resource manager is notified via the `send_performance()` call. Receiving the performance multiplier is done via a non-blocking socket which means that the application does not have to wait for the resource manager to send it. If there are messages containing performance multipliers on the socket, only the newest value is used. If the socket is empty the performance multiplier is set to one which corresponds to use of the update rule in Equation 2.1.

Resource manager, *GTRM*

The resource manager consists of parallel parts. Upon receiving the performance of an application, the handler (or dispatch function) corresponding to the reception event is executed. In the sequence diagram of Figure 3.2, this call is labeled as `gtrm_dispatch`. Its main responsibility is to read and parse the received message and then update the hashmap containing the relevant data about the application that sent the performance measurement. Meanwhile, the resource management runs as a part of the main resource manager loop. The resource management algorithm uses the information in the hashmap to calculate the new virtual platforms via the function `calculate_virtual_platforms` according to Equation 2.5. These

virtual platforms are then used to set the amount of CPUShares to be given to each application in the call to `apply_virtual_platforms`.

To ease the service level adaptation, the resource manager also sends a suggestion to the application, that depends on how much the virtual platform is changed in the current round via a call to `send_performance_number`.

4

Use cases

This chapter introduces relevant use cases that were the basis for the work and guided the design of the system. In all the use cases it is assumed that the resource available to the static slice is correctly sized, so that the *GTRM*-slice does not have any more available than the allocated resource and has to distribute it to the applications running within the slice.

4.1 Nominal Conditions

In nominal conditions, the system is well dimensioned. The applications running inside the *GTRM* slice can be run at maximum service level without overloading the hardware.

The service level for all the running applications should be at the highest possible value. The matching functions of the applications should be either positive or zero for the applications and the deadlines of all the running services should be met.

4.2 Overload Conditions (Streaming Dependent)

In the second case, the streaming application is overloading the system. The applications subsequently mentioned are all included in the *GTRM* slice,

that should allocate the resources available for them in order to match the deadlines of every applications, including the streaming service.

The detailed conditions addressed are the following:

4.3 Normal mode

1. The camera is filming a scene that causes a high load. This may happen because the camera itself is moving around or because the scenary is very dynamic.

2. Service level aware applications are running together with the streaming service. These applications (including the streaming one) adapt their service level (and therefore the quality of their computation).

3. The *GTRM* increases the resource given to the applications with the worst performance (matching function). The resource is taken from other, better performing, applications on the same slice. These better performing services lower their service level to accomodate the change in the resource allocation.

4. The scene becomes simpler or the camera stops, therefore inflicing a lighter load on the system. The service level of the applications can be increased since the streaming one is not so demanding in terms of resource.

5. The *GTRM* and SL adaptation will not drag down performance compared to the old system.

The matching function of all the applications should be close to zero, since they can all adjust their service level in a reasonable way during the execution. The frame rate for the video streaming should be constant during the execution, because the internal service adaptation of the streaming service should take care of adjusting the compression level of the image.

4.4 Overload Conditions (Non-Streaming Dependent)

At the beginning of this use case the static slice is not fully loaded, therefore some of the resource may be transfered from it to the *GTRM* managed slice.

1. The static slice is not consuming all the resource allocated to it, therefore the *GTRM* slice can collect some of the remaining resource. The applications belonging to the *GTRM* slice can raise their service level to a value that is higher than the one that would be the equilibrium point in the nominal conditions.
2. The resource demand of the static slice starts to grow.
3. The service level of the applications belonging to the *GTRM* slice adapt, lowering the quality of the computation, until a new equilibrium is reached.
4. The demand of the static slice becomes lower again and the resource is released to the *GTRM* slice. The service level increase and the resource is redistributed to the applicaions on the *GTRM* slice.

Again, the frame rate of the streaming application should be the same during the execution. Also, the applications on the static slice should be able to run without harm even when causing a higher (but still within the amount of resource statically allocated) load onto the system.

4.5 *GTRM* slice empty and filled

This is a corner case, where no applications are running within the *GTRM* slice. The static slice is allowed to use the entire amount of resource if necessary. During the test case, some applications of the *GTRM* slice are started and the virtual platforms should be adjusted accordingly, taking resource from the static slice until the size of the static slice meets the original

allocation. If sized correctly, the matching functions of the applications belonging to the *GTRM* slice should be zero or positive and their service level should settle to an equilibrium value.

4.6 Applications with different weights

During this test, some applications are run in the *GTRM* slice. However, these applications are highly heterogeneous — which means that they have different weights and they are willing to adjust their service level to various extents. Some applications are eager to help the infrastructure and will lower their requirement easily, while some other applications are more reluctant to reduce the quality of the performed computation.

1. As a starting point, all the applications running in the *GTRM* slice have a positive or zero matching function, meaning that they are satisfied with the amount of resource assigned to them.
2. The performance of one of the application decreases, for example due to an increase in its computational load.
3. The *GTRM* and the service level adaptation starts trying to compensate for that.

It is expected that the system reaches a stable point. The applications with the higher weights adapt their matching function mainly due to an increase in the amount of allocated resource. The allocation with lower weights cope with the reduced resource availability by decreasing their service level.

4.7 Nominal conditions in overload case

In this case, a number of applications are running with reasonable performance levels and no adaptation is needed. The system is subsequently loaded to the point where some of the applications can not be satisfied.

1. A number of applications are running with good performances and no service level adaptation or resource management is needed.

2. A new application is started with a default service level.

3. The resource manager takes care of allocating the resource to the new application, redistributing the available capacity. At the same time, the service level adjustment within the application tries to match the amount of resource given by the *GTRM*.

4. The amount of load introduced in the system is too much for the applications to be entirely satisfied. Some applications still have a negative matching function, despite having reduced their service level to the minimum value.

The system reaches a stable point where not all applications have a good performance. The applications that supports the service level adaptation lowered their quality as much as possible. The *GTRM* loop will continue to try to adjust to the current conditions.

5

Experimental Results

This chapter describes the tests that have been conducted on the entire architecture to validate the claims. The tests will resemble the envisioned use cases.

A first set of tests is obtained by starting and stopping different test-applications on the Axis P3367 camera. These applications run on the GTRM-slice and their resource is managed by the *GTRM* itself. The slices and services are started and stopped with the command line tool `systemctl` [1].

As previously discussed, the resource available in the camera is split into two different top slices. The first one is the system slice, that contains all the normal services. The second one is the *GTRM* slice, that contains applications that are possibly service level aware, the resource devoted to which are managed by the *GTRM*. The command line tool `systemd-cgls` shows the layout of the slice tree. Its output is shown in Figures 5.1 and 5.2. The system slice has been given $CPUShares = 1024$, while the *GTRM* slice got $CPUShares = 256$, meaning that the resource is split $\frac{4}{5}$ to $\frac{1}{5}$, between the two slices.

To test the system, several test applications are run together, specifying different parameters for each of them.When started, the test applications takes 10 parameters, described in the box below. These parameters are modified in different tests. By varying the `weight` parameter from 0

```
[root@axis-00408cc5a889 /etc/systemd/system]2605# systemd-cgls
├─1 /usr/lib/systemd/systemd
├─system.slice
│ ├─system-sshd.slice
│ │ └─sshd@0-192.168.0.55:22-192.168.0.1:45664.service
│ │   ├─2603 sshd: root@pts/0
│ │   ├─2605 -sh
│ │   └─3489 systemd-cgls
│ ├─policykit-system.service
│ │ └─2384 /usr/sbin/policykit_system -n
│ ├─policykit-parhand.service
│ │ └─2379 /usr/sbin/policykit_parhand -n
```

Figure 5.1 The first lines of the output from systemd-cgls.

```
│ ├─event-switch.service
│ │ └─983 /usr/bin/event_switch -n
│ ├─systemd-journald.service
│ │ └─424 /usr/lib/systemd/systemd-journald
│ ├─debug-shell.service
│ │ └─417 /bin/sh
│ └─systemd-udevd.service
│   └─416 /usr/lib/systemd/systemd-udevd
└─gtrm.slice
  ├─gtrm2.service
  │ └─2995 /mnt/flash/test-gtrm-app 50 10 0 0 0 0.1 0.5 0.04
  └─gtrm.service
    └─2720 /mnt/flash/test-gtrm-app 50 10 0 0 0 0.1 0.5 0.04
```

Figure 5.2 The last lines of the output from systemd-cgls.

to 1, the applications can be set somewhere between the extremes of only adapting through resource allocation and only adapt through service level modification. By varying the `a_cpu` and `b_cpu`, the emulated relationship between service level and resource requirement is changed. While `b_cpu` represents service level independent load, `a_cpu` denotes the relationship between the set service level and the service level dependent load that the application excerpt on the platform. The parameter `epsilon` makes the service level adaptation slower or faster. All services are given an initial $CPUShares = 100$, to be able to send their first message to `systemd` relatively fast.

test-gtrm-app.c(int argc, char* argv[])
The argv[] gets parsed to the following variables in the following orders:

- float service_level, Sets the starting value for the service level.
- float a_cpu, Used for calculating the cpu_req.
- float b_cpu, Used for calculating the cpu_req.
- float a_mem, Not used.
- float b_mem, Not used.
- double epsilon, Affects how quickly the SL adapts.
- double weight, Determines how much of the adaptation that will be made by the RM or by the SL adaptation. Is defined between 0 and 1.
- double deadline, The deadline for the job.

The CPU requirement is a linear function of the SL: $cpu\_req = a\_cpu * ls + b\_cpu$. Cpu_req is simply the number of calculations that will be performed each job.

In the tests, the applications are expected to converge to a matching function close to or equal to zero, with a service level that should stabilize over time. It is also expected that the resource manager initially assigns the same amount of resource to all the applications. As their demands and adaptation rates vary, the application's service level and virtual platforms do not have to converge identically, but if the system works as intended all the applications should end up with a matching function close to zero.

The tests are performed by starting the services one by one. Every time an application is added, the system is expected to reset the virtual platforms and assign to each application the same amount of resources. After that the system will then manage the resources and service level until all the applications are satisfied with their performance level. Notice that the resource manager is managing only 90% of the total shares assigned to it. This is a parameter of *GTRM* that could be changed upon request, and the 10% unused resource is intended as a slack, for example to run the resource manager itself.

Test 1: Four applications with the same weights, service level, and resource adaptation

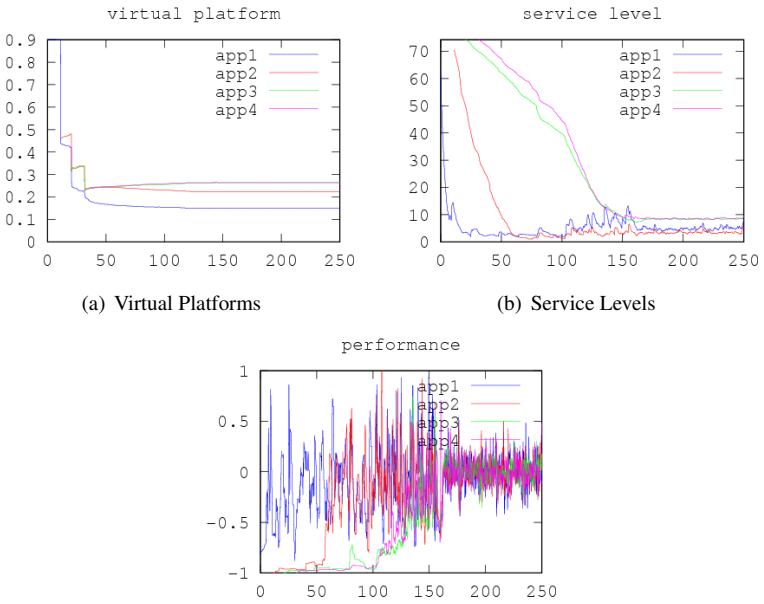
In this test, four applications are started, one every 10 seconds. The applications are started with the settings shown in Table 5.1. The result is shown in Figure 5.3.

app	$SL_{t=0}$	a_{cpu}	b_{cpu}	ε	*weight*	*Deadline*
app1	75	10	0	0.06	0.5	0.04
app2	75	20	0	0.06	0.5	0.04
app3	75	10	0	0.01	0.5	0.04
app4	75	10	0	0.01	0.5	0.04

Table 5.1 Settings for the test applications in Test 1.

It can be seen that the applications receive different amount of resources and settle to different service levels. Application 4, which is the last one to enter the pool, starts with a very negative matching function. The rearrangement of resources allows the application to obtain a larger virtual platform and therefore compute at a higher service level. It can also be observed that — despite the performance functions being quite noisy — the service levels settle to an equilibrium, as well as the virtual platforms.

Comments on the results The plots in Figure 5.3 behave on the whole as expected from the settings in Table 5.1. The mean of the noisy perfor-



(a) Virtual Platforms

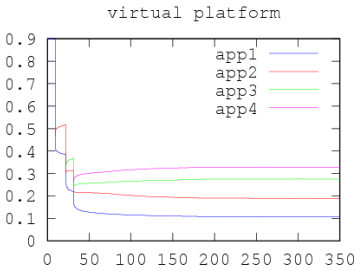
(b) Service Levels

(c) Matching Functions

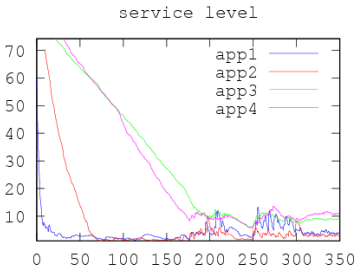
Figure 5.3 Results for Test 1.

mance converges to zero which corresponds to each application finding a pairing of SL and virtual platform (VP). In the plot a higher/lower SL gives a higher/lower Virtual platform. The SL/VP combination for app2 becomes different since the a_{cpu} is twice that of the other apps resulting in a half as large compared to what it would have been otherwise. The VP resets after a new app has been started as it should. The SL for app 3 and 4 converge slower than that of app 1 and 2 because of the lower ε in app 3 and 4.

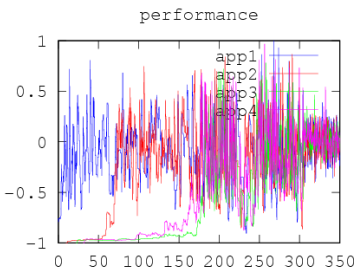
Test 2: Four applications with different weights, service level, and resource adaptation



(a) Virtual Platforms



(b) Service Levels



(c) Matching Functions

Figure 5.4 Results for Test 2.

In this second test, four applications are started as done for the previous one. The difference here is that these applications have different weights, which indirectly governs how much of the adaptation that should be made by the application or by the *GTRM*. For a summary of the relevant data for the experiment, see Table 5.2.

Figure 5.4 shows the results in terms of virtual platforms, service levels

app	$SL_{t=0}$	a_{cpu}	b_{cpu}	ε	*weight*	*Deadline*
app1	75	10	0	0.06	0.2	0.04
app2	75	20	0	0.06	0.4	0.04
app3	75	10	0	0.01	0.6	0.04
app4	75	10	0	0.01	0.8	0.04

Table 5.2 Settings for the test applications in Test 2.

and matching functions. As can be seen, the performance functions, despite being noisy, converge to signals with zero mean.

Comments on the result. This test behaves alot like the previous one but with one difference in how the SL/VL combinations converge. The main point of this test is to see how the weight affects the VPs convergence points. As can be seen in the plot of the virtual platforms in Figure 5.4, the applications with higher weights are assigned a higher VP which is in accordance with the theory.

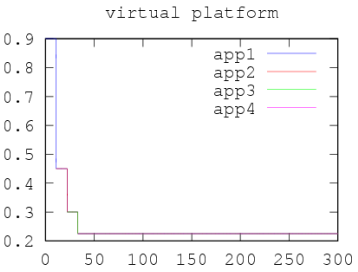
Test 3: Four applications with zero weights and service level adaptation

In this third test, four applications are started as done for the previous ones. However, the weights of the applications are set to zero. With all weights set to zero the virtual platforms are not expected to change. The result is shown in Figure 5.5.

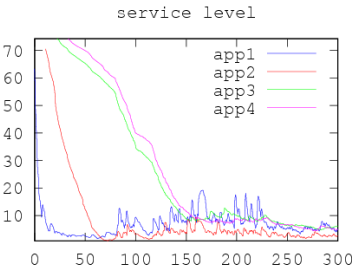
Comments on the results As expected there is no change in VP except for the even split when a new application is added. The performance looks a bit noisier than previously indicating an improvement when both SL and VP adaptation is being used.

Test 4: Four applications with weights equal to one and service level adaptation

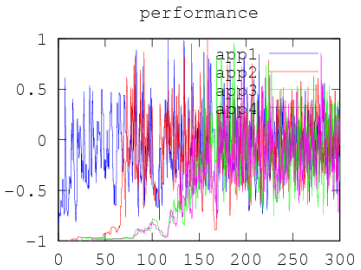
This test is the dual of the previous one but all the weights are set to 1. With all weights set to one the adaptation is expected to be done by varying the



(a) Virtual Platforms



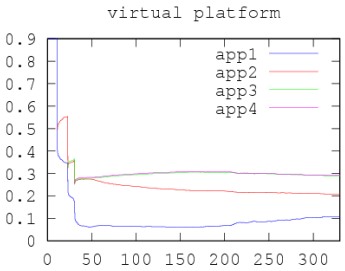
(b) Service Levels



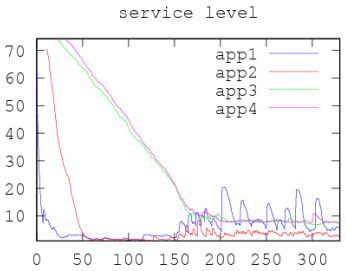
(c) Matching Functions

Figure 5.5 Results for Test 3.

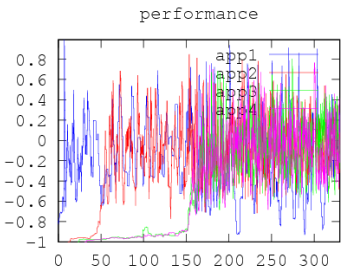
virtual platforms. The result is shown in Figure 5.6.



(a) Virtual Platforms



(b) Service Levels



(c) Matching Functions

Figure 5.6 Results for Test 4.

Comments on the results By comparing the two SL plots from Test1,in Figure 5.3,and Test4, in Figure 5.6, it seems like the SLs for app3 and app4 converge slower in the Test4 case. The VP plot also displays greater difference between the 4 applications than in the Test1 case. Although these changes are small in comparison to the changes in Test3 it can be argued that more of the adaptation is done by varying the VP in Test4 as opposed to in Test1. Once again the performance seems a bit noisier compared to the Test1 case indicating an improvement when both SL and VP adaptation is being used.

Test 5: Four applications with no service level adaptation

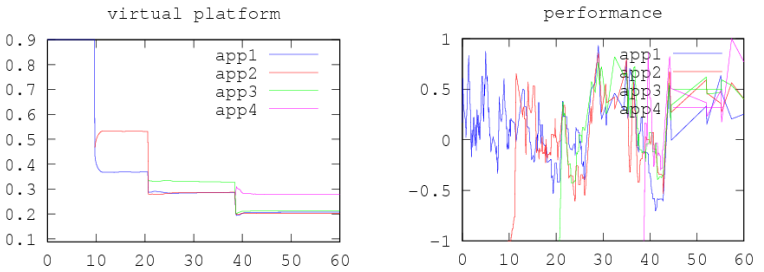
Four applications are started one by one with the settings in Table 5.3. As can be seen, the value of *a_cpu* is zero for all the applications, meaning that the service level is not at all affecting the amount of resources requested on the application side. According to proposition 3.2 in "A Game-Theoretic Resource Manager for RT Applications" [15], the normalized virtual platforms will tend to the values given by

$$\tilde{v}_i^* \rightarrow \frac{\lambda_i}{\sum_{j=1}^{n} \lambda_j}, \tag{5.1}$$

app	$SL_{t=0}$	a_{cpu}	b_{cpu}	ε	*weight*	*Deadline*
app1	75	0	10	0.06	0.2	0.04
app2	75	0	10	0.06	0.4	0.04
app3	75	0	10	0.01	0.6	0.04
app4	75	0	10	0.01	0.8	0.04

Table 5.3 Settings for the test applications in Test 5.

Figure 5.7 shows the convergence of the virtual platforms and the corresponding matching functions. As can be seen, the matching functions of all the applications become positive, meaning that the architecture is capa-



(a) Virtual Platforms (b) Matching Functions

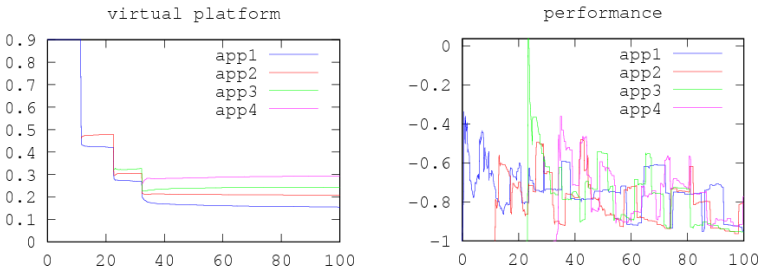
Figure 5.7 Results for Test 5.

ble of handling the load produced by the applications correctly, despite the reduction in the virtual platforms.

Also, as can be seen, all the virtual platforms converge to the same value, except for the one given to the last application, that is the last (since it started later) to reach a positive matching function. All the resources that are not distributed are therefore given to this application to help it recover faster. When the matching functions are positive, the allocation of resource stays unchanged, since there is no need for redistribution.

app	$SL_{t=0}$	a_{cpu}	b_{cpu}	ε	*weight*	*Deadline*
app1	75	0	120	0.06	0.2	0.04
app2	75	0	120	0.06	0.4	0.04
app3	75	0	120	0.01	0.6	0.04
app4	75	0	120	0.01	0.8	0.04

Table 5.4 Settings for the test applications in Test 5 with a higher *b_cpu* value.



(a) Virtual Platforms (b) Matching Functions

Figure 5.8 Results for Test 5 with a higher *b_cpu*.

Comments on the results

Test with a lower b_{cpu},

In this test only the VP and performance plots are of interest since $a_{cpu} = 0$ and hence changes in SL has no effect. The VP for the different applications does not seem to converge to the theoretical ones which probably is because of a too small static SL, b_{cpu}. The effects of this small b_{cpu} can be seen in the performance plot in Figure 5.7 when $25 < t < 35$ where the performance is stuck in the upper half plane indicating that the applications need a higher SL in order to make use of all of their available CPU. After time t = 45 the time between updates of the performance in the RM increases drastically, (the performance plots shows the performance that the RM has received). This is most likely due to a temporary problem with sending messages on DBus which in turn might have been caused by the flood of sent messages due to the high performance.

The test is repeated, increasing the value of b_{cpu}. Also, in this second case, a resource hungry application is started on the system slice in an attempt to limit the noise in performance. The settings for the test can be seen in Table 5.4 while Figure 5.8 shows the matching functions and the virtual platforms in this second case.

Test with a higher b_{cpu},

The values that the VP for the different applications converges to, compensated for the scale factor, together with the theoretical values and the weights are shown in Table 5.6. From the table it can be seen that the applications have not reached their theoretical values, however, they are not far from them. The discrepancy might be explained by all the noise that the performance still displays. This noise seems to have been reduced by the CPU load generator on the system.slice.

Test 6: Mixed load of applications with and without service level adaptation

In this last test, five applications are started one by one with the settings shown in Table 5.5. Figure 5.9 depicts the result of the run.

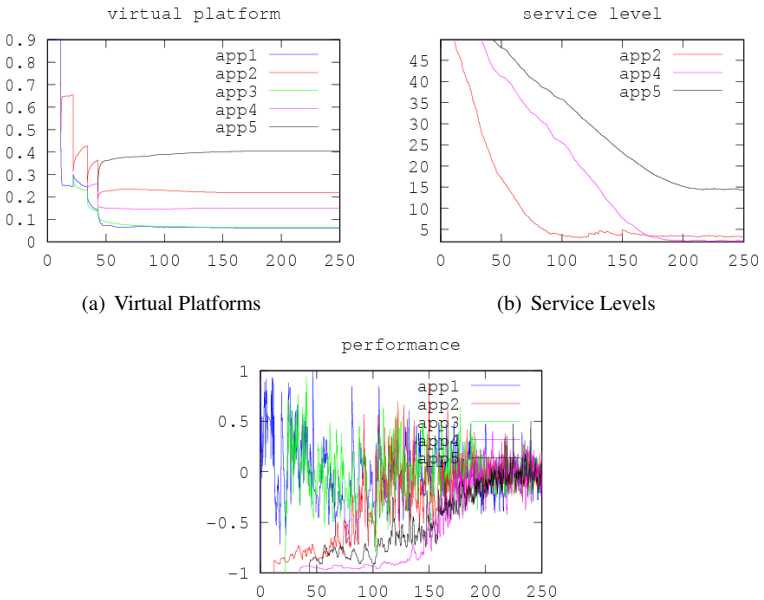
app	$SL_{t=0}$	a_{cpu}	b_{cpu}	ε	*weight*	*Deadline*
app1	50	0	30	-	0.5	0.04
app2	50	20	0	0.01	0.5	0.04
app3	50	0	30	-	0.2	0.04
app4	50	20	0	0.01	0.2	0.04
app5	50	10	0	0.002	0.8	0.04

Table 5.5 Settings for the test applications in Test 6

Comments on the results The plots looks as expected. That the weights influence how much VP an application gets can be seen in that $vp_{app5} > vp_{app2} > vp_{app4}$. The small ε for app 5 makes the SL adaptation very slow.

5.1 Result discussion

Overall the plots of the virtual platforms and the service level look good, with the exception of the extremely noisy matching functions. They do,



(a) Virtual Platforms

(b) Service Levels

(c) Matching Functions

Figure 5.9 Results for Test 6.

however, have zero mean which is why it is deemed as a good enough result. There are a couple of possible explanations for the noisy performance; one being the extra CPU that the *GTRM* slice receives when the system slice has spare resources to lend. This introduces a disturbance driving the performance up. However when loading the system slice with a CPU hungry application, no major changes in performance were shown. To reduce the amount of noise in the matching functions, one could average over a larger interval of samples, therefore smoothening the function's behavior.

app	*weight*	$VP_{theoretical}$	VP_{plot}
app1	0.2	0.1	0.17
app2	0.4	0.2	0.23
app3	0.6	0.3	0.28
app4	0.8	0.4	0.32

Table 5.6 Table showing the convergence of VP and weights in test 5b

6

Conclusion and Future Work

This thesis implemented a Game Theoretic Resource Manager (*GTRM*) on Axis cameras, to distribute the CPU among multiple running applications that can be adaptive in nature and vary their service level and the required computation, together with the quality of the offered service.

6.1 Result discussion

Overall the plots of the virtual platforms and the service level look good but the main problem is the extremely noisy performance plots. They do, however, have zero mean which is why it is deemed as a good enough result. There are a couple of possible explanations for the noisy performance; one being the extra CPU that the *GTRM* slice receives when the system.slice has spare resources to lend. This introduces a disturbance driving the performance up. However when loading the system.slice with a CPU load generator no major changes in performance were shown. Another explanation could be the delay in the socket communication since delays tend to make systems oscillate. A third reason is simply a bug somewhere in the code.

The time between actions from the *GTRM* is irregular, how this affects the system has not been taken into consideration.

6.2 Conclusion

The provided implementation demonstrated that the *GTRM* can be integrated into `systemd` and run in an Axis camera. To this end, many steps have been followed. First of all the Inter-Process Communication of the original *GTRM* was succesfully converted from shared memory to socket communication. The conversion was necessary to realize the integration with the camera framework. The implementation of the socket-based IPC resembled the code of the "Notify" feature, already included in the operating system. However, the resulting code was harder to understand and debug, compared to code completely written from scratch. The final result worked well and was well integrated into `systemd`.

The transition from shared memory to socket usage also required a new data structure for the applications. A hashmap was a natural choice. The applications PIDs were the natural choice to be used for the hashmap keys. The constant time-complexity of the data access provided by a hashmap does not negatively impact on the *GTRM* performance. The provided implementation also included macros for iterating through the hashmap, together with all the normal functions for adding and retrieving elements.

The resource management loop was inserted into the main loop of `systemd` and performed as expected. By changing to CPU-shares from `cpu.quota` and `cpu.period`, the risk of causing a kernel panic by assigning more resource than available was eliminated. Since the assignment of CPU-shares is done inside `systemd`, without the need of communication over DBus, it does not require a significant amount of time.

6.3 Future work

The performance of the system can be further improved when more applications are using the service level framework, therefore every application that can be ported to the idea of service levels and varying quality should be improved. Implementing the matching function calculation and the service level adaptation would be the ultimate solution to the resource management and prioritization problem. This would also eliminate the need for the static cgroup slice, since all the applications would be running in the same slice, managed by the *GTRM*.

The system itself is in need of further testing to track down bugs and memory leaks, if any. Future work would consist of taking this rough prototype and develop a more refined product. The consequences of the irregular update times of the VP in the *GTRM* should be further looked into, there are simple solutions to using the longest time between updates for all updates which would give uniform updating to the cost of a slower controller. The use of sockets should also be further looked into as this may be the root of the noisy performance plots. Different SL functions of multiple parameters could also be considered, which might result in faster convergence of the SL. This project is well integrated into Axis' version control and could easily be applied as a patch to future products. Also, it could be of interest to share this work with the `systemd` open source community.

Bibliography

[1] systemctl man pages http://www.freedesktop.org/software/systemd/man/systemctl.html.

[2] Axis artpec-4 chip. http://www.axis.com/corporate/press/se/releases/viewstory.php?case_id=2374.

[3] Axis m1033 manual. http://www.axis.com/files/manuals/um_m1033w_47130_en_1206.pdf.

[4] Axis p3367 manual. http://www.axis.com/files/manuals/um_p3367v_49013_en_1211.pdf.

[5] Epoll. http://en.wikipedia.org/wiki/Epoll.

[6] Gst elements. http://www.freedesktop.org/software/gstreamer-sdk/data/docs/latest/gst-plugins-good-plugins-0.10/ch01.html.

[7] Gstreamer. http://gstreamer.freedesktop.org/.

[8] Gstreamer-tutorial. http://docs.gstreamer.com/display/GstSDK/Tutorials.

[9] Real-time transport protocol. `http://en.wikipedia.org/wiki/Real-time_Transport_Protocol`.

[10] Karl-Erik Årzén, Vanessa Romero Segovia, Stefan Schorr, and Gerhard Fohler. Adaptive resource management made real. In *Proc. 3rd Workshop on Adaptive and Reconfigurable Embedded Systems*, Chicago, IL, USA, April 2011.

[11] Enrico Bini, Giorgio C. Buttazzo, Johan Eker, Stefan Schorr, Raphael Guerra, Gerhard Fohler, Karl-Erik Årzén, Romero Vanessa, and Claudio Scordino. Resource management on multicore systems: The ACTORS approach. *IEEE Micro*, 31(3):72–81, 2011.

[12] Johan Eker, Per Hagander, and Karl-Erik Årzén. A feedback scheduler for real-time controller tasks. *Control Engineering Practice*, 8(12):1369–1378, January 2000.

[13] Daniel Grosu and Anthony T. Chronopoulos. Noncooperative load balancing in distributed systems. *Journal of Parallel and Distributed Computing*, 65(9):1022–1034, 2005.

[14] Chenyang Lu, John A. Stankovic, Gang Tao, and Sang H. Son. Design and evaluation of a feedback control EDF scheduling algorithm. In *Proceedings of the* 20^{th} *IEEE Real Time Systems Symposium*, pages 56–67, Phoenix (AZ), U.S.A., December 1999.

[15] Martina Maggio, Enrico Bini, Georgios Chasparis, and Karl-Erik Årzén. A game-theoretic resource manager for rt applications. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[16] Paul Menage. Sgroups. `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`.

[17] Lennart Poettering. Systemd. `http://www.freedesktop.org/wiki/Software/systemd`.

[18] Lennart Poettering. Systemd-blog. `http://0pointer.de/blog/projects/systemd.html`.

[19] Rauganathan Rajkumar, Chen Lee, John Lehoczky, and Dan Siewiorek. A resource allocation model for QoS management. In *Proceedings of the IEEE Real Time System Symposium*, 1997.

[20] Javier Silvestre-Blanes, Luís Almeida, Ricardo Marau, and Paulo Pedreiras. Online QoS management for multimedia real-time transmission in industrial networks. *IEEE Transactions on Industrial Electronics*, 58(3):1061–1071, March 2011.

[21] Michal Sojka, Pavel Píša, Dario Faggioli, Tommaso Cucinotta, Fabio Checconi, Zdeněk Hanzálek, and Giuseppe Lipari. Modular software architecture for flexible reservation mechanisms on heterogeneous resources. *Journal of Systems Architecture*, 57(4):366–382, 2011.

[22] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the* 3^{rd} *Symposium on Operating Systems Design and Implementation*, February 1999.

[23] Riky Subrata, Albert Y. Zomaya, and Björn Landfeldt. A cooperative game framework for QoS guided job allocation schemes in grids. *IEEE Transactions on Computers*, 57(10):1413–1422, October 2008.

[24] Stephen A. Rago W. Richard Stevens. *Advanced Programming in the UNIX Environment: Second Edition*. Addison Wesley Professional, 2005.

[25] Guiyi Wei, Athanasios V. Vasilakos, Yao Zheng, and Naixue Xiong. A game-theoretic method of fair resource allocation for cloud computing services. *The Journal of Supercomputing*, 54(2):252–269, November 2010.

A

Source Code Overview

In this section follows an overview of the header and source files used in the system and their attributes.

A.1 gtrm_lib.c/h

A library which contains various functions used by the resource manager to make various computations and inter process communication.

struct rm_app_t

Represents an application being managed and consists of the following fields.

- tid: Applications PID.
- vp: Virtual platform.
- vp_old: Previous virtual platform.
- performance: Performance, or matching function of the application.
- weight: The current "weight" of the application.

- happy: Indicates if the application is happy with its

  current performance. This field was added to prevent application from sending their performance even when they are satisfied and nothing should be needed from the resource manager, eliminating unnecessary computations.

- sa: Socket Address, used to send back the performance multiplier.

struct gtrm_t

Stores various parameters used by the GTRM.

- c1: Constant used for computing epsilon, determines how much we will change the virtual platform.
- c2: Another constant used for computing epsilon similar as c1.
- iterations: Keeps track of how many iterations the *GTRM* has run.
- all_happy: Used to indicate if we have to make any adjustments to the resource allocations.
- num_apps: Total amount of applications that we are managing.
- prev_apps: The amount of applications in the previous iteration.

void gtrm_char2gtrmstruct(char* str, rm_app_t *re)

Extracts data from a received string and stores it as a struct instead. The data sent from an application consists of the PID, performance, weight, and if the application is satisfied or not.

- str: String to extract data from.
- re: Struct to hold the extracted data.
- Return value: None, the result is stored in *re*.

int gtrm_send_performance_multiplier(double pm, int fd, struct sockaddr *sa)

Sends a performance multiplier to an application via sockets.

- pm: Performance multiplier to send.
- fd: File descriptor used to send the performance multiplier.
- sa: Socket address.
- Return value: Always zero.

double gtrm_get_epsilon(unsigned int iterations, unsigned int offset, double c1, double c2)

Calculates the constant epsilon used by the resource manager.

- iterations: Number of iterations run by the RM.
- offset: If one, resets the time used in the iteration. This restarts the calculation.
- c1: Constant, previously described.
- c2: Constant, also previously described.
- Return value: The calculated epsilon or one if this is the first iteration.

void gtrm_update_performance_multipliers(int gtrm_fd, Hashmap *gtrm_apps)

Calculates, updates and sends the performance multiplier to each application, using the performance, virtual platform and the previous virtual platform for each application.

- gtrm_fd: File descriptor used to send the performance multiplier.

- gtrm_apps: Hash-map containing rm_app_t structs for each application.
- Return value: None.

void gtrm_write_log(Hashmap *gtrm_apps, unsigned int num_applications)

Writes information about the resource management to a log file, which can then be used to generate some nice graphs.

- gtrm_apps: Hash-map containing rm_app_t structs for each application. «««< HEAD
- num_applications: Used to make sure we do not try to print an empty hash-map. =======
- num_applications: Used to avoid printing an empty hash-map. »»»> baa53d7d018201d6bd7f70a511b99ef4ec14b918
- Return value: None.

A.2 manager.c/h

One of the main files in `systemd` and is for example responsible for inter process communication. The following fields are added to the *Manager* struct in manager.h:

- gtrm_socket: String that represent the socket used by GTRM.
- gtrm_fd: A file descriptor, which is an integer, for communicating with GTRM.
- gtrm_event_source: Used to determine the source of an event and how this event is supposed to be handled.
- gtrm_apps: Hash-map containing all the applications being managed by the GTRM.

static int manager_dispatch_gtrm_fd(sd_event_source *source, int fd, unit32_t revents, void *userdata)

Called when we receive the performance of an application.

- source: source of the event.
- fd: file descriptor to the event source.
- revents: Set by the kernel to indicate what event on the file descriptor that triggered the call to this function. In our case it is always the incoming event.
- userdata: In this case it contains a reference to the manager.
- Return value: Not used, always zero.

static int manager_setup_gtrm(Manager *m)

Used to setup the socket, event source and file descriptor for the resource manager. Called from manager_startup and manager_reload.

- m: Reference to the manager.
- Return value: Zero if function ran correctly, otherwise it is set to the corresponding error number.

int gtrm_compute_virtual_platforms(Hashmap *apps, gtrm_t *gtrm_t)

Calculates the amount of resources (virtual platform) for an application.

- apps: The hash-map containing information about the applications being managed.
- gtrm_t: Struct with parameters used when calculating the virtual platforms.
- Return value: Not used, always zero.

void gtrm_apply_virtual_platforms(Manager* m)

Computes and applies the amount of CPUShares that each application shall be given.

- m: Reference to manager, used to get the applications
- Return value: None.

int manager_loop(Manager *m)

The main loop of the system which continuously calls gtrm_compute_virtual_platforms, gtrm_apply_virtual_platforms and gtrm_update_performance_multipliers if all applications are not satisfied with their performance.

- m: Reference to manager, used to get the applications.
- Return value: Exit code of systemd.

int manager_set_cpu_shares(Manager *m, pid_t pid, int shares)

Sets the CPUShares of an application.

- m: Reference to manager, used to get the applications.
- pid: Process identifier of the application.
- shares: Amount of shares we want to set.
- Return value: Zero if successful, one otherwise.

A.3 gtrm_app_lib.c/h

These files are used by the applications to get performance, send performance and setup IPC, for example. In the header file the following structs are defined:

struct _job_h

A struct used to represent a job. Applications can have different type of jobs with different deadlines. We only use one type however.

- id: Identifier to keep track of a job being executed.
- type: What type of job this is.
- start_timestamp: Time when the job was started.
- end_timestamp: Time when we were finished with job. The timestamps are used to calculate the total execution time of a job.

struct _application_h

Each application stores the relevant information in terms of resource management and SL-adaptation in this struct.

- application_id: Identifier for an application.
- jobs: Number of possible job types
- weight: Determines how to divide the adaptation between the resource manager and service level adaptation.
- performance_multiplier: Depends on how the virtual platform has changed and is used for better service level adaptation.
- total_jobs: How many jobs that has been launched in total.
- progress_jobs: Jobs in progress.
- completed_jobs: Total amount of completed jobs.
- expected_response_times: Array of the expected response time for each job type.
- happy: Indicates if the application is satisfied with its performance or not.

int gtrm_lib_setup_socket(char* filename)

Sets up a socket to communicate with the resource manager.

- filename: The socket needs a file to work, the parameter specifies its path.
- Return value: Zero, not used.

void gtrm_lib_send_performance(_application_h *h, double performance)

Sends the performance of an application to the GTRM.

- h: Struct representing an application.
- performance: Performance or matching function to send to the GTRM.
- Return value: None.

int gtrm_lib_set(_application_h* a, uint types, uint64_t* ert)

Initializes the application struct with job types and their expected response times.

- a: Struct representing an application.
- types: Number of different job types.
- ert: Array of expected response time for each job type.
- Return value: Exit status of function.

static int manager_setup_gtrm(Manager *m)

Used to setup the socket, event source and file descriptor for the resource manager. Called from manager_startup and manager_reload.

- m: Reference to the manager.
- Return value: Zero if function ran correctly, otherwise it is set to the corresponding error number.

double gtrm_lib_get_performance_number(_application_h* a, int job_type)

Calculates the peformance (matching function) by averaging the performance of the last ten jobs of a specified type.

- a: Struct representing an application.
- type: Which job type for which we want to calculate the performance.
- Return value: Applications performance.

int gtrm_lib_update_performance_multiplier(_application_h *a)

Receives the performance multiplier computed by the RM.

- a: Struct for representing the application, and storing the performance multiplier.
- Return value: Zero, never used.

int gtrm_lib_signalstart(_application_h* a, uint type)

Indicates the start of a job.

- a: Struct for representing the application.
- type: Type of the job that is started.
- Return value: Identifier of the started job.

int gtrm_lib_signalend(_application_h* a, uint id)

Indicates the end of a job.

- a: Struct for representing the application.
- Return value: Exit status.

A.4 sl_adapt.c

Used by the streaming application, *monolith*, to incorporate service level features.

void setup_sl_adapt(float service_level, double epsilon, double weight, double deadline_seconds)

Sets up deadline for computing the matching function and sets up the IPC with the resource management.

- service_level: Initial service level.
- epsilon: Constant which specifies service level adaptation rate.
- weight: Determines how the much of the adaptation will be done by altering the service level or the changing the amount of resources.
- deadline_seconds: This deadline will correspond to the desired frame rate.
- Return value: None.

void sl_adapt()

Performs the service level adaptation, notifies the resource manager and writes some logging.

- Return value: None.

test-gtrm-app.c(int argc, char* argv[])

The argv[] gets parsed to the following variables:

- float service_level, Sets the starting value for the service level.
- float a_cpu, Used for calculating the cpu_req.
- float b_cpu, Used for calculating the cpu_req.
- float a_mem, Not used.
- float b_mem, Not used.
- double epsilon, Affects how quickly the SL adapts.
- double weight, Determines how much of the adaptation that will be made by the RM or by the SL adaptation.
- double deadline, The deadline for the job.

The CPU requirement is a linear function of the SL: $cpu\_req = a\_cpu * ls + b\_cpu$. Cpu_req is simply the number of calculations that will be performed each job.

B

Source Code

This appendix presents all the source code written for this project.

B.1 manager.c

Note that this file is a part of `systemd` and only what was added or changed to this file is included in the following code-snippets. For the rest of the code in this file, please refer to the open-source project [17].

```
1  #define GTRM_SOCKET "@/org/freedesktop/systemd1/gtrm"
2  ...
3  static int manager_dispatch_gtrm_fd(sd_event_source *
       source, int fd, uint32_t revents, void *userdata);
4  ...
5  int manager_new(SystemdRunningAs running_as, Manager **
       _m) {
6    ...
7    m->pin_cgroupfs_fd = m->gtrm_fd = m->notify_fd = m->
         signal_fd = m->time_change_fd = m->dev_autofs_fd
         = m->private_listen_fd = m->kdbus_fd = -1;
8    ...
9    m->gtrm_apps = hashmap_new(trivial_hash_func,
         trivial_compare_func);
10   ...
11 }
12 static int manager_setup_gtrm(Manager *m) {
```

```
13    union {
14        struct sockaddr sa;
15        struct sockaddr_un un;
16    } sa = {
17        .sa.sa_family = AF_UNIX,
18    };
19  int r;
20    if (m->gtrm_fd < 0) {
21      _cleanup_close_ int fd = -1;
22
23      /* First free all secondary fields */
24      free(m->gtrm_socket);
25      m->gtrm_socket = NULL;
26      m->gtrm_event_source = sd_event_source_unref(m->
            gtrm_event_source);
27
28
29      fd = socket(AF_UNIX, SOCK_DGRAM|SOCK_CLOEXEC|
            SOCK_NONBLOCK, 0);
30      if (fd < 0) {
31          log_error("Failed to allocate notification
                socket: %m");
32          return -errno;
33      }
34
35      if (getpid() != 1 || detect_container(NULL) > 0)
36          snprintf(sa.un.sun_path, sizeof(sa.un.sun_path)
                , GTRM_SOCKET "/%" PRIx64, random_u64());
37      else
38          strncpy(sa.un.sun_path, GTRM_SOCKET, sizeof(sa.
                un.sun_path));
39      sa.un.sun_path[0] = 0;
40
41      r = bind(fd, &sa.sa, offsetof(struct sockaddr_un,
            sun_path) + 1 + strlen(sa.un.sun_path+1));
42      if (r < 0) {
43          log_error("bind() failed: %m");
44          return -errno;
45      }
46
47      sa.un.sun_path[0] = '@';
```

```
48      m->gtrm_socket = strdup(sa.un.sun_path);
49      if (!m->gtrm_socket)
50          return log_oom();
51
52      m->gtrm_fd = fd;
53      fd = -1;
54    }
55    else
56
57    if (!m->gtrm_event_source) {
58        r = sd_event_add_io(m->event, &m->
              gtrm_event_source, m->gtrm_fd, EPOLLIN,
              manager_dispatch_gtrm_fd, m);
59        if (r < 0) {
60            log_error("Failed to allocate gtrm event
                  source: %s", strerror(-r));
61            return -errno;
62        }
63
64        /* Process signals a bit earlier than SIGCHLD, so
               that we can
65         * still identify to which service an exit
               message belongs */
66        r = sd_event_source_set_priority(m->
              gtrm_event_source, -7);
67        if (r < 0) {
68            log_error("Failed to set priority of gtrm
                  event source: %s", strerror(-r));
69            return r;
70        }
71    }
72    return 0;
73  }
74  ...
75  void manager_free(Manager *m) {
76    ..
77    hashmap_free_free_free(m->gtrm_apps);
78      sd_event_source_unref(m->gtrm_event_source);
79    ...
80      free(m->gtrm_socket);
81    ...
```

```
82 }
83 int manager_startup(Manager *m, FILE *serialization,
       FDSet *fds) {
84   ...
85     manager_setup_gtrm(m);
86   ...
87 }
88 int manager_set_cpu_shares(Manager *m, pid_t pid, int
       shares){
89     Unit* u;
90     CGroupContext* c;
91
92     u = manager_get_unit_by_pid(m, pid);
93     if(u == NULL) {
94       hashmap_remove(m->gtrm_apps, pid);
95       reset_virtual_platforms(m->gtrm_apps);
96       return 1;
97     }
98
99     c = unit_get_cgroup_context(u);
100    c->cpu_shares = shares;
101    cgroup_context_apply(c, CGROUP_CPU, u->cgroup_path)
           ;
102    return 0;
103 }
104 static int manager_dispatch_gtrm_fd(sd_event_source *
       source, int fd, uint32_t revents, void *userdata) {
105   Manager *m = userdata;
106   assert(m);
107   assert(m->gtrm_fd == fd);
108
109   if (revents != EPOLLIN) {
110     log_warning("Got unexpected poll event for gtrm");
111     return 0;
112   }
113
114   char buf[1024];
115   int n;
116   struct sockaddr_un *from;
117   socklen_t fromlen;
118   rm_app_t *app;
```

```
119   rm_app_t *app2;
120   char last_was_from[124];//keeps the adress (sun_path)
          from last sender
121
122   fromlen = 1024;
123   int counter = 0;
124   do{
125     counter++;
126     memset(buf,'\0',1023);
127     from = calloc(1,sizeof(struct sockaddr_un));
128     app = calloc(1,sizeof(rm_app_t));
129
130     //n is negative if there was no message if socket
            is non blocking
131     n = recvfrom(fd,buf,1024,0,(struct sockaddr *) from
            ,&fromlen);
132     if(n<0)
133       break;
134     if(n>1024){
135       log_error("manager_dispatch_gtrm_fd:received to
              big message");
136     }
137
138     gtrm_print_struct(app);
139     assert((size_t) n < sizeof(buf));
140
141     gtrm_char2gtrmstruct(buf, app);
142     pid_t pid = app->tid;
143     app->sa = from;
144
145     if(hashmap_get(m->gtrm_apps, pid) == NULL) {
146       hashmap_put(m->gtrm_apps, pid, app);
147     } else {
148       app2 = hashmap_get(m->gtrm_apps, pid);
149       gtrm_update_rm_struct(app,app2);
150     }
151
152   } while(n>0);
153
154   fprintf(stderr,"Read: %d messages ",counter);
155   m->update_gtrm = true;
```

```
156   return 0;
157 }
158 ...
159 int gtrm_compute_virtual_platforms(Hashmap *apps,
        gtrm_t *gtrm_t) {
160   unsigned int offset = gtrm_t->num_apps != gtrm_t->
          prev_apps;
161   if(offset)
162     reset_virtual_platforms(apps);
163   double c1 = gtrm_t->c1;
164   double c2 = gtrm_t->c2;
165   double epsilon = gtrm_get_epsilon(gtrm_t->iterations,
          offset, c1, c2);
166   float sumlambdafi = 0.0;
167   Iterator i;
168   _app_t* a;
169   bool all_happy = true;
170   HASHMAP_FOREACH(a, apps, i) {
171     sumlambdafi += (a->weight)* a->performance;
172   }
173
174   HASHMAP_FOREACH(a, apps, i) {
175     all_happy = a->happy && all_happy;
176     a->vp_old = a->vp;
177     float vp_old = a->vp_old / (_MAX_ASSIGNABLE);
178     float tmp = vp_old - epsilon * ( a->weight * a->
            performance - (sumlambdafi * vp_old)); //
            computed as sum to one
179
180     tmp = tmp * _MAX_ASSIGNABLE;// scaled
181     if(tmp < _MIN_SINGLE_ASSIGNABLE)
182       tmp = _MIN_SINGLE_ASSIGNABLE;
183     else if(tmp > _MAX_SINGLE_ASSIGNABLE)
184       tmp = _MAX_SINGLE_ASSIGNABLE;
185     a->vp = tmp;
186   }
187   gtrm_t->all_happy = all_happy;
188   return 0;
189 }
190
191 void gtrm_apply_virtual_platforms(Manager* m) {
```

```
192     int shares;
193     Iterator i;
194     rm_app_t* a;
195     HASHMAP_FOREACH(a, m->gtrm_apps, i) {
196       shares = a->vp * _TOTAL_SHARES;
197       manager_set_cpu_shares(m, a->tid, shares);
198     }
199 }
200 ...
201 int manager_loop(Manager *m) {
202     ...
203     gtrm_t *gtrm_t = calloc(1,sizeof(struct gtrm_t));
204     gtrm_t->num_apps = 0;
205     gtrm_t->prev_apps = 0;
206     gtrm_t->iterations = 0;
207     gtrm_t->all_happy = true;
208     gtrm_t->c1 = 0.1;
209     gtrm_t->c2 = 10;
210     ...
211       while (m->exit_code == MANAGER_RUNNING) {
212     ...
213       if ((!(gtrm_t->all_happy) || m->update_gtrm) && !
              hashmap_isempty(m->gtrm_apps)) {
214         gtrm_t->prev_apps = gtrm_t->num_apps;
215         gtrm_t->num_apps = hashmap_size(m->gtrm_apps);
216         gtrm_compute_virtual_platforms(m->gtrm_apps,
              gtrm_t);
217         gtrm_t->iterations++;
218         gtrm_apply_virtual_platforms(m); // set_shares
              according to platform
219        m->update_gtrm=false;
220         gtrm_update_performance_multipliers(m->gtrm_fd,m
              ->gtrm_apps);
221         gtrm_write_log(m->gtrm_apps, gtrm_t->num_apps);
222
223       }
224     }
225     free(gtrm_t);
226       return m->exit_code;
227 }
228
```

```
229 int manager_serialize(Manager *m, FILE *f, FDSet *fds,
        bool switching_root) {
230   ...
231   if (m->gtrm_fd >= 0) {
232     int copy;
233
234     copy = fdset_put_dup(fds, m->gtrm_fd);
235     if (copy < 0)
236         return copy;
237
238     fprintf(f, "gtrm-fd=%i\n", copy);
239     fprintf(f, "gtrm-socket=%s\n", m->gtrm_socket);
240   }
241   ...
242 }
243 int manager_deserialize(Manager *m, FILE *f, FDSet *fds
        ) {
244   ...
245   for (;;) {
246   ...
247     else if (startswith(l, "gtrm-fd=")) {
248       int fd;
249
250       if (safe_atoi(l + 10, &fd) < 0 || fd < 0 || !
            fdset_contains(fds, fd))
251         log_debug("Failed to parse gtrm fd: %s", l +
              10);
252       else {
253         if (m->gtrm_fd >= 0) {
254           m->gtrm_event_source = sd_event_source_unref(
                m->gtrm_event_source);
255           close_nointr_nofail(m->gtrm_fd);
256                 }
257
258         m->gtrm_fd = fdset_remove(fds, fd);
259             }
260
261     } else if (startswith(l, "gtrm-socket=")) {
262           char *n;
263
264       n = strdup(l+14);
```

```
265         if (!n) {
266           r = -ENOMEM;
267           goto finish;
268         }
269
270         free(m->gtrm_socket);
271         m->gtrm_socket = n;
272
273       }
274       ...
275       }
276     ...
277 }
278 int manager_reload(Manager *m) {
279   ...
280   /* Re-register gtrm_fd as event source */
281   q = manager_setup_gtrm(m);
282   if (q < 0)
283     r = q;
284   ...
285 }
```

B.2 manager.h

```
1 struct Manager {
2   ...
3   char *gtrm_socket;
4   int gtrm_fd;
5   sd_event_source *gtrm_event_source;
6   Hashmap *gtrm_apps;
7   bool update_gtrm;
8   ...
9 }
```

B.3 gtrm_lib.c

```
1 #include <stdbool.h>
2 #include "hashmap.h"
```

```
3  #include "gtrm_lib.h"
4
5
6  //used to calculate epsilon
7  int64_t time_since_start;
8
9  //data is stored in char as "pid-performance-happy-
       weight"
10 void gtrm_char2gtrmstruct(char* str, rm_app_t *re){
11
12   char *pch;
13
14   pch = strtok (str,"x");
15   re->tid = (uint)atoi(pch);
16
17   pch = strtok (NULL,"x");
18   re->performance = atof(pch);
19
20   pch = strtok (NULL,"x");
21   re->happy = (uint)atoi(pch);
22
23   pch = strtok (NULL,"x");
24   re->weight = atof(pch);
25 }
26
27 void gtrm_print_struct(rm_app_t *rm ){
28   printf("\ntid:%d\nvp:%f\nvp_old:%f\nperformance:%f\
         nweight:%f\nhappy:%d\n",
29   rm->tid,rm->vp,rm->vp_old,rm->performance,rm->weight,
         rm->happy);
30 }
31
32 int gtrm_send_performance_multiplier(double pm,int fd,
       struct sockaddr *sa){
33   int n;
34   socklen_t tolen;
35   tolen = (socklen_t) sizeof(struct sockaddr_un);
36   char buf[512];//2do no magic numbers
37   memset(buf,'\0',512);
38   sprintf(buf,"%f",pm);
39   n=sendto(fd,buf,512,0,(struct sockaddr *) sa,tolen);
```

```
40    return 0;
41  }
42
43  void gtrm_update_rm_struct(rm_app_t *src,rm_app_t *dest
        ){
44    dest->performance = src->performance;
45  }
46
47  double gtrm_get_epsilon(unsigned int iterations,
        unsigned int offset, double c1, double c2) {
48    double value = c2;
49    struct timespec time_info;
50    int64_t current_time;
51    clock_gettime(CLOCK_MONOTONIC, &time_info);
52    current_time = (int64_t) time_info.tv_sec*1000000000
53    + (int64_t) time_info.tv_nsec;
54
55    if(offset==1)
56      time_since_start = current_time;
57
58    if(iterations>0)
59      value = value * (double)(current_time -
            time_since_start)/1000000000.0;
60    else
61    {
62      time_since_start = current_time;
63      return 1;
64    }
65    return (c1) / (1.0 + value);
66  }
67
68  void gtrm_update_performance_multipliers(int gtrm_fd,
        Hashmap *gtrm_apps){
69    Iterator i;
70    rm_app_t *a;
71    double pm;
72    HASHMAP_FOREACH(a, gtrm_apps, i) {
73      if(a->vp_old != 0){
74        pm = (1+a->performance)*(a->vp/a->vp_old);
75        gtrm_send_performance_multiplier(pm,gtrm_fd,a->sa
              );
```

```
76      }
77    }
78  }
79
80  void gtrm_write_log(Hashmap *gtrm_apps, unsigned int
        num_applications) {
81    if (num_applications > 0) {
82      struct timespec time_info;
83      int64_t current_time;
84      clock_gettime(CLOCK_REALTIME, &time_info);
85      current_time = (int64_t) time_info.tv_sec
            *1000000000
86      + (int64_t) time_info.tv_nsec;
87      FILE* logfile = fopen("/root/gtrm.log", "a+");
88
89      Iterator i;
90      rm_app_t* a;
91
92      HASHMAP_FOREACH(a, gtrm_apps, i) {
93        fprintf(logfile, "%lld, %d, %f, %f, %f\n",
94        current_time, a->tid, a->vp,
95        a->performance, a->weight);
96      }
97    fclose(logfile);
98    }
99
100 }
```

B.4 gtrm_lib.h

```
1  #include <stdlib.h>
2  #include <inttypes.h>
3  #include <stdio.h>
4  #include "hashmap.h"
5  #include <sys/socket.h>
6  #include <sys/un.h>
7  #include <sys/types.h>
8  #include "hashmap.h"
9
10 #define _MIN_SINGLE_ASSIGNABLE 0.01
11 #define _MAX_SINGLE_ASSIGNABLE 0.90
```

```
12 #define _MAX_ASSIGNABLE (0.9000)
13 #define _TOTAL_SHARES 1024
14 #define _RM_DEADLINE 1000000 // nsec
15
16 typedef struct rm_app_t rm_app_t;
17 typedef struct gtrm_t gtrm_t;
18
19 struct rm_app_t {
20     pid_t tid;
21     float vp;
22     float vp_old;
23     float performance;
24     float weight;
25     uint happy:1;
26     struct sockaddr_un* sa;
27 };
28
29 struct gtrm_t {
30     double c1;
31     double c2;
32     unsigned int iterations;
33     bool all_happy;
34     unsigned int num_apps;
35     unsigned int prev_apps;
36
37 };
38
39 void gtrm_char2gtrmstruct(char* str, rm_app_t *re);
40 void gtrm_print_struct(rm_app_t *rm );
41 int gtrm_send_performance_multiplier(double pm,int fd,
       struct sockaddr *sa);
42 void gtrm_update_rm_struct(rm_app_t *src,rm_app_t *dest
       );
43 double gtrm_get_epsilon(unsigned int iterations,
       unsigned int offset, double c1, double c2);
44 void gtrm_update_performance_multipliers(int gtrm_fd,
       Hashmap *gtrm_apps);
45 void gtrm_write_log(Hashmap *gtrm_apps, unsigned int
       num_applications);
```

B.5 gtrm_app_lib.c

```
1  #include <sys/types.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/socket.h>
5  #include <sys/un.h>
6  #include <string.h>
7  #include <netdb.h>
8  #include <stdio.h>
9  #include <stddef.h>
10 #include "gtrm_app_lib.h"
11
12 #define MAXMSG  512
13 #define MESSAGE "Message from client"
14 #define GTRM_SOCKET "@/org/freedesktop/systemd1/"
15 #define NO_OF_SENDS 5
16 #define WAIT_FOR_REPLY 0
17 #define DEBUG 0
18
19 int sock, addr_len;
20 struct sockaddr_un *gtrm_sa;
21
22 //creates a socket and binds it with sockaddr sa using
       filename as sun_path. returns size of sa.
23 void make_named_socket (char *filename) {
24   size_t size;
25
26   /* Create the socket. */
27   sock = socket (PF_UNIX, SOCK_DGRAM|SOCK_NONBLOCK, 0);
28   if (sock < 0) {
29     #ifdef DEBUG
30       fprintf(stderr,"socket() failed:\n");
31     #endif
32     perror ("socket");
33     exit (EXIT_FAILURE);
34   }
35   struct sockaddr_un sa;
36
37   /* Bind a name to the socket. */
38   sa.sun_family = AF_FILE;
```

```
39    strcpy (sa.sun_path, filename);
40
41    /* The size of the address is
42    the offset of the start of the filename,
43    plus its length,
44    plus one for the terminating null byte. */
45    size = (offsetof (struct sockaddr_un, sun_path) +
          strlen (sa.sun_path) + 1);
46
47    if (bind (sock, (struct sockaddr *) &sa,size) < 0) {
48      perror ("bind");
49      exit (EXIT_FAILURE);
50    }
51  }
52
53  int gtrm_lib_setup_socket(char *filename){
54    /* Make the socket. */
55    make_named_socket(filename);
56
57    /* Initialize the server socket address. */
58    //read gtrm address
59    char*  env;
60
61    env = getenv("GTRM_SOCKET");
62
63    //sometimes getenv won't give you the start of the
          address, so here we try and fix that.
64    if(env[0]!='@')
65    {
66      char* temp;
67      char *temp_two[100];
68      temp=(char*)temp_two;
69
70      strcpy(temp,GTRM_SOCKET);
71      while ( *temp!='\0')
72        temp++;
73
74      strcpy(temp,strstr(env,"gtrm"));
75      env=(char*)temp_two;
76    }
77    gtrm_sa = malloc(sizeof(struct sockaddr_un));
```

```
78
79    strcpy(gtrm_sa->sun_path, env);
80    addr_len = strlen(gtrm_sa->sun_path) + sizeof(gtrm_sa
          ->sun_family);
81
82    gtrm_sa->sun_path[0]=0;
83    gtrm_sa->sun_family = AF_UNIX;
84  }
85
86  void gtrm_lib_send_performance(_application_h *h,double
          performance){
87    char msg[100];
88    //data is stored in char as "pid-performance-happy-
          weight"
89    printf("pid = %d\n", h->application_id);
90    sprintf(msg,"%ux%fx%ux%f",h->application_id,
          performance,h->happy,h->weight);
91
92    int n;
93    n = sendto(sock,msg,100,0,(struct sockaddr*)gtrm_sa,
          addr_len);
94    if(n<0){
95        perror("sendto");
96        fprintf(stderr,"could not send performance");
97    }
98  }
99
100 int gtrm_lib_set(_application_h* a, uint types,
          uint64_t* ert) {
101   #ifdef _JOBSIGNALER_DEBUG
102     fprintf(stdout, "[set] started\n");
103   #endif
104   int retvalue = EXIT_NORMAL;
105
106   // Application dependant initialization
107   if (types > _H_MAX_JOBS) {
108     #ifdef _JOBSIGNALER_ERROR
109       fprintf(stderr, "[set] jobs number exceeding
              maximum; limited\n");
110     #endif
111     types = _H_MAX_JOBS;
```

```
112   }
113   a->jobs = types;
114   int c_act;
115   for (c_act = 0; c_act<types; ++c_act)
116     a->expected_response_times[c_act] = ert[c_act];
117
118   // Application independent initialization
119   a->total_jobs = 0;
120   a->progress_jobs = 0;
121   a->completed_jobs = 0;
122   a->performance_multiplier = 1;
123
124   #ifdef _JOBSIGNALER_DEBUG
125     fprintf(stdout, "[set] ended\n");
126   #endif
127   return retvalue;
128 }
129
130 double gtrm_lib_get_performance_number(_application_h*
      a, int job_type){
131   // Averaging the value of last jobs of specified type
        : the function
132   // uses all jobs of the given type that are in the
        latest
133   // records. If empty it will return zero. If called
        with -1 it will
134   // return the average over all type of jobs.
135   double sum_performances = 0.0;
136   double num_performances = 0.0;
137   int c_act;
138   int upvalue = a->completed_jobs;
139   if (a->completed_jobs > _H_MAX_RECORDS)
140     upvalue = _H_MAX_RECORDS;
141
142   for (c_act=0; c_act<upvalue; ++c_act) {
143     // this last condition in the if is used to avoid
          to report
144     // job of type 0 that are not real jobs but initial
          values of
145     // the vector: real jobs have a start timestamp
146
```

```
147     if ((a->jcompleted[c_act].type == job_type ||
148         job_type == -1 || job_type>=a->jobs) &&
149         a->jcompleted[c_act].start_timestamp != 0) {
150       int64_t deadline = a->expected_response_times[a->
              jcompleted[c_act].type];
151       int64_t response_time = a->jcompleted[c_act].
              end_timestamp
152         - a->jcompleted[c_act].start_timestamp;
153       double this_perf = ((double) deadline / (double)
              response_time) - 1.0;
154       if (this_perf<-1.0)
155         this_perf = -1.0; // thresholds
156       if (this_perf>+1.0)
157         this_perf = +1.0;
158       sum_performances += this_perf;
159       num_performances += 1.0;
160     }
161   }
162   if (num_performances == 0)
163     return -1.0;
164   else
165     // averaging
166     return sum_performances/num_performances;
167 }
168
169 int gtrm_lib_update_performance_multiplier(
      _application_h* h){
170   int n;
171   char buf[1024];
172   char final_buf[1024];
173   int got_something = 0;
174
175   do{
176     memset(buf,0,1024);
177     n = recvfrom(sock,buf,1024,0, NULL,0);
178
179     //if no msg left
180     if(n<0)
181       break;
182
183     if(n>1024)
```

```
184         printf("too big msg! >1024, n:%d\n",n);
185
186       assert(n<=1024);
187       strcpy(final_buf,buf);
188       got_something = 1;
189     } while(n>0);
190
191     if(got_something!=1)
192       return -1;
193     h->performance_multiplier = atof(final_buf);
194
195     #ifdef DEBUG
196       if(got_something!=0)
197         fprintf(stderr,"received pmulti:%f\n",atof(
              final_buf));
198     #endif
199     return 0;
200 }
201
202 int gtrm_lib_signalend(_application_h* a, uint id) {
203
204     // Get actual time
205     struct timespec time_info;
206     int64_t time;
207     clock_gettime(CLOCK_REALTIME, &time_info);
208     time = (int64_t) time_info.tv_sec*1000000000 + (
          int64_t) time_info.tv_nsec;
209     int retvalue = EXIT_FAILURE_JOBNOTFOUND;
210
211     #ifdef _JOBSIGNALER_MULTITHREADED
212       pthread_mutex_lock(&a->mutex);
213     #endif
214
215     // Looking for the job to be terminated
216     int c_act;
217     for (c_act=0; c_act<(a->progress_jobs %
          _H_MAX_RECORDS); ++c_act) {
218       if (a->jprogress[c_act].id == id) {
219
220         // Writing it
```

```
221         uint index_completed = a->completed_jobs %
                _H_MAX_RECORDS;
222         int type = a->jprogress[c_act].type;
223         int64_t start = a->jprogress[c_act].
                start_timestamp;
224         a->jcompleted[index_completed].id = a->total_jobs
                ;
225         a->jcompleted[index_completed].type = type;
226         a->jcompleted[index_completed].start_timestamp =
                start;
227         a->jcompleted[index_completed].end_timestamp =
                time;
228         a->completed_jobs++;
229
230         // Clearing up the progress log
231         a->jprogress[c_act].id = 0;
232         a->jprogress[c_act].type = 0;
233         a->jprogress[c_act].start_timestamp = 0;
234         a->jprogress[c_act].end_timestamp = 0;
235         a->progress_jobs--;
236
237         // Done
238         #ifdef _JOBSIGNALER_DEBUG
239           fprintf(stdout, "[stop] removed job %d into %d\
                  n", c_act, index_completed);
240         #endif
241         retvalue = EXIT_NORMAL;
242       }
243     }
244
245     #ifdef _JOBSIGNALER_MULTITHREADED
246       pthread_mutex_unlock(&a->mutex);
247     #endif
248     return retvalue;
249
250 }
251
252 int gtrm_lib_jobsignaler_signalstart(_application_h* a,
        uint type) {
253
254     // Get actual time
```

```
255   struct timespec time_info;
256   int64_t time;
257   clock_gettime(CLOCK_REALTIME, &time_info);
258   time = (int64_t) time_info.tv_sec*1000000000 + (
          int64_t) time_info.tv_nsec;
259   uint job_id = a->total_jobs;
260
261   #ifdef _JOBSIGNALER_MULTITHREADED
262     pthread_mutex_lock(&a->mutex);
263   #endif
264
265   // The number of jobs in progress should never exceed
          the max number
266   // Otherwise jobs will be overwritten and will never
          finish
267   uint index_in_progress = a->progress_jobs %
          _H_MAX_RECORDS;
268   a->jprogress[index_in_progress].id = a->total_jobs;
269   a->jprogress[index_in_progress].type = type;
270   a->jprogress[index_in_progress].start_timestamp =
          time;
271   a->jprogress[index_in_progress].end_timestamp = time;
272   a->total_jobs++;
273   a->progress_jobs++;
274
275   #ifdef _JOBSIGNALER_DEBUG
276     fprintf(stdout, "[start] added job %d\n",
            index_in_progress);
277   #endif
278
279   #ifdef _JOBSIGNALER_MULTITHREADED
280     pthread_mutex_unlock(&a->mutex);
281   #endif
282   return job_id;
283 }
284 int gtrm_lib_jobsignaler_terminate(_application_h* a) {
285   return EXIT_NORMAL;
286 }
287
288 int gtrm_lib_jobsignaler_signalend(_application_h* a,
          uint id) {
```

```
289
290   // Get actual time
291   struct timespec time_info;
292   int64_t time;
293   clock_gettime(CLOCK_REALTIME, &time_info);
294   time = (int64_t) time_info.tv_sec*1000000000 + (
          int64_t) time_info.tv_nsec;
295   int retvalue = EXIT_FAILURE_JOBNOTFOUND;
296
297   #ifdef _JOBSIGNALER_MULTITHREADED
298     pthread_mutex_lock(&a->mutex);
299   #endif
300
301   // Looking for the job to be terminated
302   int c_act;
303   for (c_act=0; c_act<(a->progress_jobs %
          _H_MAX_RECORDS); ++c_act) {
304     if (a->jprogress[c_act].id == id) {
305
306       // Writing it
307       uint index_completed = a->completed_jobs %
              _H_MAX_RECORDS;
308       int type = a->jprogress[c_act].type;
309       int64_t start = a->jprogress[c_act].
              start_timestamp;
310       a->jcompleted[index_completed].id = a->total_jobs
              ;
311       a->jcompleted[index_completed].type = type;
312       a->jcompleted[index_completed].start_timestamp =
              start;
313       a->jcompleted[index_completed].end_timestamp =
              time;
314       a->completed_jobs++;
315
316       // Clearing up the progress log
317       a->jprogress[c_act].id = 0;
318       a->jprogress[c_act].type = 0;
319       a->jprogress[c_act].start_timestamp = 0;
320       a->jprogress[c_act].end_timestamp = 0;
321       a->progress_jobs--;
322
```

```
323         // Done
324         #ifdef _JOBSIGNALER_DEBUG
325           fprintf(stdout, "[stop] removed job %d into %d\
                n", c_act, index_completed);
326         #endif
327         retvalue = EXIT_NORMAL;
328       }
329     }
330
331     #ifdef _JOBSIGNALER_MULTITHREADED
332       pthread_mutex_unlock(&a->mutex);
333     #endif
334     return retvalue;
335   }
```

B.6 gtrm_app_lib.h

```
1  #include <inttypes.h>
2  #include <sys/types.h>
3  #include <stdbool.h>
4  #include <assert.h>
5  #include <time.h>
6  #include <log.h>
7
8  #define _H_MAX_JOBS 10 // Number of different job types
9  #define _H_MAX_RECORDS 10 // Number of maximum records
10 #define _H_MAX_FILENAMELENGHT 1000
11
12 // Exit codes
13 #define EXIT_NORMAL 0
14 #define EXIT_FAILURE_UNDEFINEDAUTOSIGNALER -1
15 #define EXIT_FAILURE_SHAREDMEMORY -2
16 #define EXIT_FAILURE_JOBNOTFOUND -3
17
18 typedef struct {
19   uint id;
20   uint type;
21   int64_t start_timestamp;
22   int64_t end_timestamp;
23 } _job_h;
24
```

```
25 typedef struct {
26   unsigned int application_id;
27   int shared_memory_segment;
28   uint jobs; // Number of possible job types
29   double weight;
30   double performance_multiplier;
31   uint total_jobs;
32   uint progress_jobs;
33   uint completed_jobs;
34   uint64_t expected_response_times[_H_MAX_JOBS];
35   _job_h jprogress[_H_MAX_RECORDS];
36   _job_h jcompleted[_H_MAX_RECORDS];
37   pthread_mutex_t mutex;
38 uint happy:1;
39 } _application_h;
40
41 //public
42 int gtrm_lib_setup_socket(char* filename);
43 void gtrm_lib_send_performance(_application_h * h,
       double performance);
44 int gtrm_lib_set(_application_h* a, uint types,
       uint64_t* ert);
45 double gtrm_lib_get_performance_number(_application_h*
       a, int job_type);
46 int gtrm_lib_update_performance_multiplier(
       _application_h*);
47 int gtrm_lib_signalend(_application_h* a, uint id);
48 int gtrm_lib_signalstart(_application_h* a, uint type);
49
50 //internal
51 void make_named_socket(char *name);
```

B.7 video.c

This file is a part of the video streaming application and presented is what was added to it in this project.

```
1
2 static void
3 gtrm_sl_cb (GstElement * identity, GstBuffer * buf,
       GstElement * src)
4 {
```

```
5    int frame_size = sl_adaptation();
6    g_object_set (G_OBJECT (src), "frame_size",
         frame_size, NULL);
7  }
8
9  static GstElement *
10 cache_video (gpointer key, Props * props, gpointer *
       user_data)
11 {
12   ...
13   GstElement *gtrm_sl_adapt = NULL;
14   float sl_start = 100;
15   double epsilon = 0.8;
16   double weight = 0.5;
17   double deadline_seconds = 0.0333;
18   setup_sl_adapt (sl_start, epsilon, weight,
         deadline_seconds);
19   ...
20   gtrm_sl_adapt = gst_element_factory_make ("identity",
         "gtrm");
21   g_object_set (G_OBJECT (gtrm_sl_adapt), "signal-
         handoffs", TRUE, NULL);
22   g_signal_connect (gtrm_sl_adapt, "handoff",
         G_CALLBACK (gtrm_sl_cb), src);
23   ...
24   gst_bin_add_many (GST_BIN (p), src, gtrm_sl_adapt,
         convert, bmpenc,
25   caps_filter, sink, NULL);
26   if (!gst_element_link_many (src, gtrm_sl_adapt,
         convert, bmpenc,
27       caps_filter, sink, NULL)) {
28     error ("Could not link elements.");
29     goto error_link;
30   }
31   ...
```

B.8 sl_adapt.c

```
1  #include "sl_adapt.h"
```

```
2
3  void
4  setup_sl_adapt (float service_level_, double epsilon_,
       double weight_,
5      double deadline_seconds_)
6  {
7    // Parameter parsing
8    service_level = service_level_;
9    epsilon = epsilon_;
10   deadline_seconds = deadline_seconds_;
11   sock_name = (char *) calloc (1, 200); //2do add free
12   sock_path = (char *) calloc (1, 200); //2do add free
13   sprintf (sock_path, "/mnt/flash/fd");
14   int temp_pid = (int) getpid ();
15
16   myself = calloc (1, sizeof (_application_h));
17   sprintf (sock_name, "%s/%d", sock_path, temp_pid);
18   gtrm_lib_setup_socket ((char *) sock_name);
19   uint64_t deadline = (unsigned int) ((double)
         1000000000 * deadline_seconds);
20   uint64_t ert[1] = { deadline };
21   gtrm_lib_set (myself, 1, ert);
22   myself->weight = weight_;
23   myself->application_id = getpid ();
24 }
25
26
27
28 void
29 sl_adaptation ()
30 {
31   id = 0;
32   gtrm_lib_signalend (myself, id);
33   double performance;
34
35   int type = 0;
36   id = gtrm_lib_signalstart (myself, type);
37
38   performance = gtrm_lib_get_performance_number (myself
         , type);
39
```

```
40    // I want to adapt only if needed
41    if (performance < -0.01 || performance > 0.01) {
42      myself->happy = false;
43      // send performance to systemd
44      gtrm_lib_send_performance (myself, performance);
45    if (gtrm_lib_update_performance_multiplier (myself)
          == 0) {
46      service_level += epsilon * service_level * (myself
            ->performance_multiplier - 1);
47    } else {
48      service_level += epsilon * performance *
            service_level;
49    }
50
51    if (service_level < MINIMUM_SERVICE_LEVEL)
52      service_level = MINIMUM_SERVICE_LEVEL;
53    if (service_level != service_level) // avoid nans
54      service_level = 1.0;
55
56    } else if (myself->happy == false) {
57      printf ("happy appy\n");
58      myself->happy = true;
59      gtrm_lib_send_performance (myself, performance);
60    }
61    #ifdef LOGGING_APPLICATION
62      struct timespec time_info;
63      int64_t current_time;
64      clock_gettime (CLOCK_REALTIME, &time_info);
65      current_time = (int64_t) time_info.tv_sec *
            1000000000 + (int64_t) time_info.tv_nsec;
66
67      char name[200];
68      sprintf (name, "/mnt/flash/logs/%u.log", myself->
            application_id);
69      FILE *logfile = fopen (name, "a+");
70
71      if (logfile == NULL)
72        perror ("could not open file");
73      assert (logfile != NULL);
74      fprintf (logfile, "%lld, %f, %f, %lld,%u\n",
```

```
75      (long long int) current_time, performance,
            service_level,
76      (long long int) 0, id);
77      fclose (logfile);
78    #endif
79
80  }
```

B.9 sl_adapt.h

```
1  #include <limits.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <signal.h>
5  // Using the library:
6  #include "gtrm_app_lib.h"
7  #include <unistd.h>
8  #include <errno.h>
9
10 #define TOTAL_JOBS 100
11 #define NOISE_PERCENTAGE 0.0
12 #define MINIMUM_SERVICE_LEVEL 0.0001
13 #define ERROR_APPLICATION 0
14 #define LOGGING_APPLICATION 0
15 #define EXIT_APPLICATIONFAILURE -1
16 #define DEBUG 0
17
18 uint id;
19 _application_h *myself;
20 float service_level;
21 double epsilon;
22 uint64_t deadline_seconds;
23 char *sock_path;
24 char *sock_name;
25
26 void sl_adaptation();
27 void setup_sl_adapt (float service_level_, double
       epsilon_, double weight_,
28      double deadline_seconds_);
```

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

Document name
MASTER´S THESIS

Date of issue
July 2014

Document Number
ISRN LUTFD2/TFRT--5952--SE

Author(s)
Fredrik Johnsson
Olle Svensson

Supervisor
Umut Tezduyar-Lindskog, Axis
Martina Maggio, Dept. of Automatic Control, Lund University, Sweden
Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Sponsoring organization

Title and subtitle

Resource Management and Prioritization in an Embedded Linux System

Abstract

This master thesis tackles the problem of limited computing resources on a camera that is executing computing applications together with image acquisition and streaming. The thesis was carried out at Axis Communications in cooperation with the Department of Automatic Control at Lund University. The problem of limited resources on an Axis camera is handled by a two part solution where a resource manager (RM) distributes the available resources and services can adapt their service level (SL) in order to finish their jobs on time. The solution is based on game theory, where services are players, varying their service levels in order to get a good match between given resources and their computing requirements. This service level adaptation scheme is implemented for the streaming service on the camera and for some test services, performing mathematical operations. The resource manager is incorporated into systemd, and uses cgroups [16] to distribute the computing capacity. The experimental results show that the resource manager is fully operational and capable of managing and prioritizing resources as intended on the embedded system.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-121

Recipient's notes

Security classification