ISSN 0280-5316
ISRN LUTFD2/TFRT--5875--SE

Resource reservation and power management in Android

Martin Olsson
Samuel Skånberg

Department of Automatic Control
Lund University
March 2011

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

Document name
MASTER THESIS

Date of issue
March 2011

Document Number
ISRN LUTFD2/TFRT--5875--SE

Author(s)
Martin Olsson and Samuel Skånberg

Supervisor
Harald Gustafsson Ericsson Sweden AB Lund, Sweden
Karl-Eric Årzén Automatic Control Lund, Sweden (Examiner)

Sponsoring organization

Title and subtitle
Resource reservation and power management in Android (Resursreservationer och effekthantering för Android)

Abstract
Saving battery time for mobile devices has been a goal for the industry for many years. With the advent of smartphones, reduction of energy consumption is even more important since they consume a lot more energy than the generation of mobile phones before them. Consumers are demanding longer battery life and greener electronics. One way to meet these demands is to reduce energy consumption.
In order to make the mobile operating system utilize the Central Processing Unit (CPU) more efficiently, applications should have different reservations based on how much they need to use the CPU. A challenge the industry is facing is its lack of knowledge of the behavior of third party applications. Especially since they are an increasing portion of the applications run on smartphones. Without knowledge of how third party applications behave, it is hard to make good reservations for them. If there was a way to dynamically make reservations for the applications with adequate performance while they are running, the system could use this information to reduce battery consumption by e.g. clocking down the CPU when a high clock frequency is not needed. In this master thesis project, an open source resource manager called ACTORS Resource Manager (ACTORS RM) [5][6] for desktop Linux [57] is ported to the Android [37] operating system. The resource manager is also optimized for the applications being run there. A power management patch to the Linux kernel was also used to get greater control over the CPU's frequency changes.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
168

Recipient's notes

Security classification

Acknowledgment

We would like to start by thanking Karl-Erik Årzén and Harald Gustafsson for supervising our work and both them as well as Song Yuan for their valuable support and advice. We would also like to thank all of our other colleagues at Ericsson Research in Lund for making our time at the department a fun and incredibly educational experience.

Contents

Chapter 1

Introduction

Consumers have always demanded longer battery life of mobile phones. Some feature phones (the generation of phones before smartphones [58]) could have a standby time of up to 67 days [47]. With modern microprocessors that have decreased in size and cost and increased in computing power, more complex and resource intensive operating systems can be used on mobile phones.

Modern mobile operating systems can perform more advanced things. Applications being developed to these systems are of very different nature. They can be computing intensive, I/O intensive, idle and lightweight, graphic intensive, polling or event controlled. So the system must deal with both software such as a browser and software such as a lightweight calculator or E-book reader.

Modern mobile operating systems can offer a lot of new possibilities but it comes to a price: higher energy consumption. The operating system itself will use more resources than what it did before. And the applications running on the system are harder to control because their behaviour is harder to predict and make reservations for. Historically, mobile operating systems would be real time operating systems running applications with fixed priorities. This is no longer true with smartphones.

Because of these modern mobile operating systems and other kinds of dynamic systems, effort has been put into dynamically making reservations for applications. Ericsson is involved in a EU project called Adaptivity and Control of Resources in Embedded Systems (ACTORS) [4]. One part of ACTORS is the development of a resource manager (ACTORS RM) that handles dynamic reservations. The ACTORS RM is developed for a desktop Linux system.

A lot of effort has been put into making reservations for video decoding. The results are very promising and that is the reason how this master thesis came into being. A port of the ACTORS RM to the Android operating system has been made and its focus has not been on video decoding applications, but rather "normal" applications which are sporadic in nature and change their behaviour during their life cycle.

Chapter 2
Problem formulation

In this thesis project, a CPU resource reservation manager in Android will be introduced. The goal is to lower active power consumption by making the CPU run on a lower clock frequency.

A resource management framework will be developed which will make it easy for Android application developers to register with the resource manager.

This framework should have the following properties as stated in the job description for this thesis:

1. *Easily handle different application service levels. Each service level will require different levels of resources and generate different result qualities.*
2. *Easily giving application feedback to resource manager on how well the application can fulfill the service level.*
3. *Easily giving applications access to reserving processing time to meet their requirements.*
4. *Incorporating the usage of services used by the application in its service level handling. Examples of such services are the media server that handles all decoding and rendering of multimedia.*
5. *Develop default handling that is suitable for many applications.*
6. *Demonstrate the usage by adopting a few existing open source Android applications of different types to utilize the resource and power management.*

The challenges are the porting of the ACTORS RM, integrate it into the Android Java environment and to actually lower the energy consumption.

2.1 Division of work

The division of work has been listed in Table 2.1. It should be stressed that there has been a lot of collaboration on the report but for most chapters and sections there has been one person with a slightly bigger responsibility.

Chapter/Section	Mainly responsible
1 Introduction	Samuel
2 Problem formulation	Samuel & Martin
2.1 Division of work	Martin
3 Background and related work	-
3.1 Linux	Martin
3.2 Android	Samuel
3.3 Resource management	Martin
3.4 Power management	Martin
3.5 Platforms	Martin
4 Design	Samuel & Martin
.1 Overview	Samuel
5 Implementation	-
5.1 Binder in C++ or in Java?	Samuel
5.2 MiniRM GUI (MiniRMServerController)	Samuel
5.3 Alterations to the resource manager	Samuel
5.4 Changes made to Android	Samuel
5.5 Changes made to the Linux kernel	Samuel
6 Adapted open source applications	Samuel
6.1 Missile Intercept	Samuel
6.2 SwiFTP	Martin
6.3 Android native Calculator	Martin
7 Result	Martin
7.1 Different application settings	Samuel
7.2 User scenario overview	Martin
7.3 The most interesting user scenarios	Martin
8 Conclusions	Samuel & Martin
8.1 Summary of achievements	Samuel & Martin
8.2 Follow-up of the problem formulation	Samuel
8.3 Future work	Samuel
8.4 Final words from the authors	Martin
Bibliography	Samuel & Martin
9 Acronyms	Samuel
A Areas	Samuel & Martin
B Graphs	Martin

Table 2.1: Division of work

Chapter 3
Background and related work

This chapter, background and related work, presents a number of concepts from Linux [57], Android [37] and the ACTORS project [4]. There is one section on Linux, 3.1, one on Android, 3.2, one discussing resource management in those two systems, 3.3, a fourth on power management, 3.4, and finally a section about the platforms used during the project, 3.5.

3.1 Linux

Wikipedia's definition of Linux [61]:

> Linux refers to the family of Unix-like computer operating systems using the Linux kernel. Linux can be installed on a wide variety of computer hardware, ranging from mobile phones, tablet computers and video game consoles, to mainframes and supercomputers. Linux is a leading server operating system, and runs the 10 fastest supercomputers in the world.

Below, a few fundamental concepts are explained that are indeed fundamental to understanding how the ACTORS RM works.

3.1.1 Processes

An essential part in the Linux system that the ACTORS RM uses is processes, see 3.3.3. The goal of the ACTORS RM is essentially to give processes the appropriate amount of resources with some help from e.g. cgroups. A process is in short a program being executed [32]. However, there is more to a process than just the executed program code. They also include at least one thread (thread of execution) and a data section with global variables. The kernel will need to manage all these different parts of the process, not just the program code, as efficiently as possible.

3.1.2 Threads

As mentioned above each process in a Linux system consists of threads among other things. In the same chapter as for the process above, threads are defined as *the objects of activity within the process* [32]. Every thread consists of a program counter, which is unique to each thread, a process stack and a number of process registers. The Linux kernel schedules individual threads rather than entire processes.

3.1.3 cgroups (control groups)

The concept of cgroups [19] was introduced with Linux kernel version 2.6.24. It is a feature with the purpose to *limit, account and isolate resource usage (CPU, memory, disk I/O etc) of process groups*. This means that from a process with just one thread up to an entire system can be placed in a cgroup which in turn can be given resource limits and a share among other things. The cgroup feature is an important part of the ACTORS RM, see 3.3.3 and 5.4.2, since it has a clear focus on managing groups of threads.

To assign resources to a cgroup, a share (of the total available resources) needs to be written to the `cpu.shares` file for that group. The cgroup system will take the sum of all the shares and then see how large quota the specific cgroup should have:

$$quota = \frac{cpu.shares}{sum\ of\ all\ shares} \tag{3.1}$$

All thread ID:s of an application are written to a file named `tasks` so that the scheduler will know what threads belong to that application's cgroup.

All cgroups are guaranteed to run as much as their quota says. But in a system with soft reservations, the part of the quota that is not used can be "stolen" by other cgroups. This can be observed in the result section, see e.g. Section 7.3.9. In a system with hard reservations however, cgroups are not allowed to overrun their quota.

Group on demand patch

There has been a patch created by Harald Gustafsson applied to the Linux kernel. The patch makes sure that it is possible for the ACTORS RM to manipulate the behaviour of the CPU frequency changes for the system. The dynamic changing of the CPU frequency is done by the cpufreq subsystem [29]. In the patch a new group on demand cpufreq governor is used.

A cpufreq governor decides what frequency should be used [31]. The patch is known as the group on demand patch[1].

There are a couple of terms related to this patched cgroup mechanism that should be explained:

[1] Also known as *grpondemand* in the system

- **cpufreq.usage_per_cpu** (or load) is the amount of CPU bandwidth the threads of a cgroup are currently consuming. This is a value to be read, it cannot be set. This value is normalized to the wall clock time due to the fact that more cycles can be run under the same amount of time if run on a higher clock frequency. For instance, this value would be e.g. 10 milliseconds (ms) on 600 MHz and 5 ms on 300 MHz.
- The **cpufreq.load_ceil** (or ceiling) for a cgroup is set with the purpose of defining the maximum load the threads of that cgroup are allowed to affect the decision making of frequency changes. This will prevent the threads of the cgroup to contribute to an increased clock frequency more than the amount specified by the ceil value.
- The **cpufreq.load_margin** for a cgroup is how much margin the scheduler should keep for that process. If there currently is not enough bandwidth to squeeze in both the load and the margin for a program the CPU frequency is increased, if possible. That is, if the load plus margin is lower than the ceiling.
- If the **cpufreq.fast** flag in a cgroup directory has been set to 1, it means that if the CPU frequency needs to be raised, it will be raised to the maximum frequency directly. Then the frequency will be re-evaluated and if possible it will be lowered to another level.

3.2 Android

Android is an open source mobile operating system (or software stack) developed by Google together with other members of the Open Handset Alliance [9]. It is based on a modified Linux kernel and the language for writing applications is Java. Applications are developed using the Android Software Development Kit (SDK) and Google has developed a plugin for the popular open source Integrated Development Environment (IDE) Eclipse so that developers easily can start to use the Android SDK. An emulator to test the applications on are shipped with the SDK. Figure 3.1 is a visualization of the Android software stack. The work done in this thesis has primarily been in the Applications, Libraries and Linux kernel layers.

3.2.1 Why Android?

The reasons for choosing Android as the testing platform are several. The primary one is that Ericsson Research is already doing a lot of work on Android, they have expertise on that and want to be able to reuse the findings of this thesis project if successful. Another reason is that Android is a Linux system and the ACTORS RM should not be too hard to port to Android since it already runs on a desktop Linux system. A third reason is that Android is open source; The Linux kernel is licensed under the GNU General Public License (GPL) v2. The Android system (the rest of the system minus the kernel) is to a large extent released under

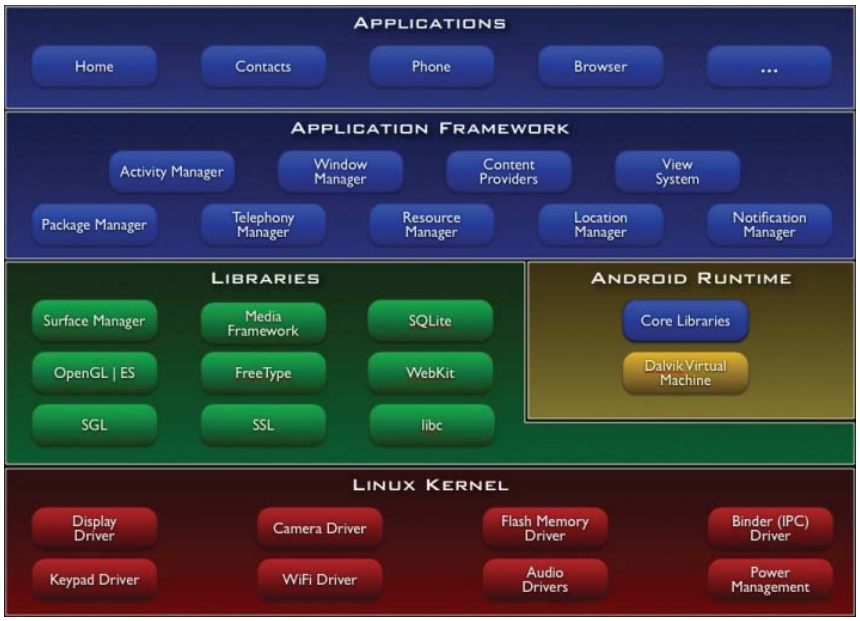

Figure 3.1: Android software stack [17]

the Apache 2.0 license. In other words, there is full access to source code which allows for doing modifications to the system.

3.2.2 Android fundamentals

Android applications are written in Java and bundled into Android packages with the suffix `.apk`. Each application is assigned a unique user ID it will be run under. There are exceptions where multiple applications are run under the same user ID [10]. The system user e.g. runs a lot of applications under the same user ID (system). Each user is assigned a certain level of permissions. This means that an application will not get access to another application's data unless that application explicitly has given permission for the first one to do so.

One thing that makes Android applications different from applications in other systems is that an Android application does not control its own life cycle. An Android application has no `main` function, instead it has a number of callback life cycle functions that the Android system will call when it seems appropriate. It may seem confusing but this is a good way to relieve the developer from tasks he/she does not want to do anyway, e.g. sleeping when idle. It is much like programming an Graphical User Interface (GUI) application with a toolbox such as Swing, GIMP Toolkit (GTK) or Qt.

An application can consist of different components that do different things. The focus in this project has been on the two most common components, Activity

and Service. The different components of the application can share the same thread. It is the system that will call the `onCreate` method when a service or an activity is first created [38][39].

Activity

An Activity presents the user with a user interface. It can be a window with a menu or just a dialog window. When an activity is being started its `onCreate` method will be called, followed by its `onStart` method and lastly its `onResume` method as can be seen in Figure 3.3. When a new activity is started, the previous activity that was in focus will have its `onPause` methods called before it looses focus. It will still be held in memory but if the system needs to free some memory, then all activities that have been paused can be destroyed by calling their `onDestroy` methods.

To create a new activity one will have to extend the Activity base class in Android as shown in Figure 3.2.

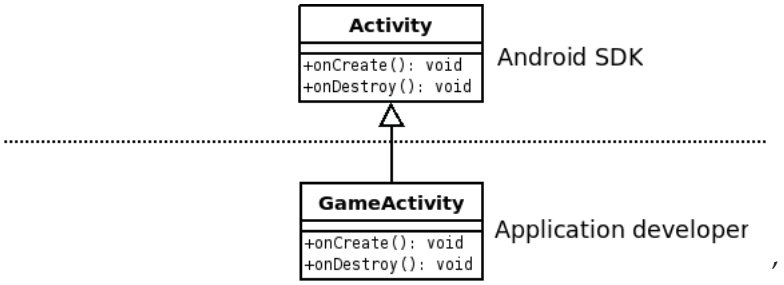

Figure 3.2: Example GameActivity extends Activity

There are default implementations for the `onCreate`, `onStart`, `onResume` methods and so on. This means that a developer would only need to override those methods he/she wants to change. For instance, if a developer only wants some buttons to show up when the activity runs, it is sufficient to override `onCreate`. This is because all layout initializations are done there. When `onResume` is called later on the layout, as set in `onCreate`, is merely shown. The latter will be performed even if `onResume` is not overridden.

Service

A service has no GUI. It runs in the background for an indefinite amount of time. Even though activities and services can share the same threads, most of a service's execution time is in one of the application's Binder threads, see Binder section below. An activity can use them as well when it does some Binder tasks, but it is mostly the service that will use them.

Communication with the service is done using Binder, the

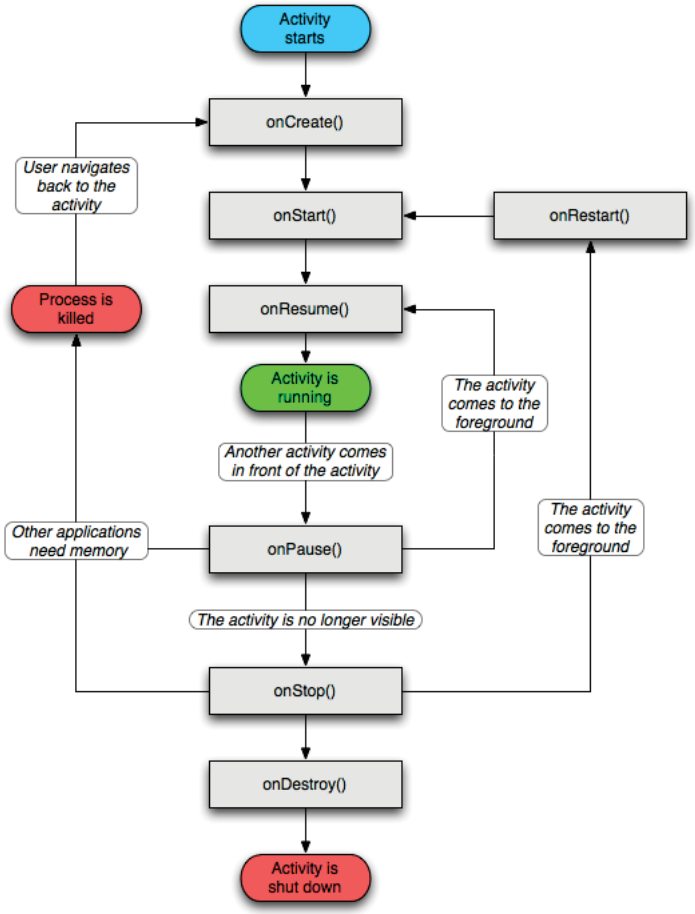

Figure 3.3: Life cycle of an activity. Ref. [11]

Inter Process Communication (IPC) system in Android. Using Binder, an activity or some other application component can "bind" to a service, if the service allows it, in the same or another application[2]. When a component is bound to a service it can send commands to that service. It is much like remote method invocation (RMI) in Java.

A service only have `onCreate`, `onStart` and `onDestroy` as its life cycle methods, see Figure 3.4. The time between calls to `onCreate` and `onDestroy` are the entire life cycle of the service. The active life cycle begins when `onStart` is being called. To be able to bind to a service, one has to start the service first with the `startService` method. That method will then call `onStart` on the service.

[2]Application in this case means an `.apk` file

A service has to define what methods can be called on it. The developer does this by writing a specification (much like Java) in an `.aidl` file. The `.aidl` file then gets translated to different classes that are needed. The connecting activity will get a reference to one of those instanced classes when it does "bind" on the service. It can then use that reference to communicate with the service.

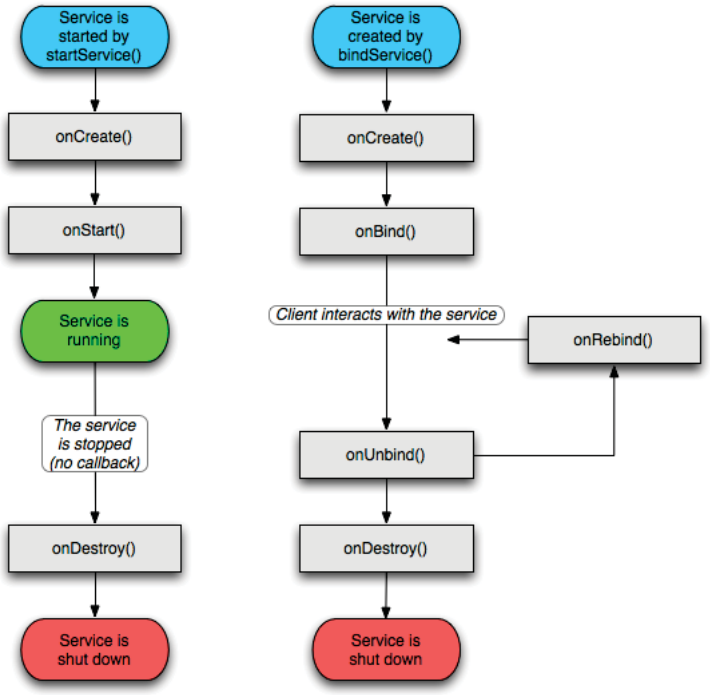

Figure 3.4: Life cycle of a service. Ref. [11]

Binder

Binder is the IPC system used in Android. It was developed at Be inc and Palm inc with Dianne Hackborn, who is now one of Android's framework designers [40][23][42].

Binder is quite fast since it is implemented as a kernel module [41]. According to Harald Gustafsson and Song Yuan at Ericsson Research, the registration process for the ACTORS RM using another common IPC system, D-Bus, can take up to 1 second. The measurements performed during this thesis project has shown that the same registration process will take approximately 10 ms using Binder. Hence, in this field of application Binder is approximately a factor of 100 times faster than D-Bus.

In Android there are wrappings in Java to access Binder and it is done seam-

lessly since the Android SDK has tools that will generate classes in Java, based on the simple `.aidl` file that described what kind of methods can be called on different services.

Another important part of Binder are Binder threads [22]. Each Android application has a pool of Binder threads. These threads are passively waiting for incoming IPC calls from other processes. If an IPC call is dispatched to a process, one of the Binder threads in the pool of that process becomes active in order to process the IPC call. There are a couple of different ways an IPC call can arrive to a process.

3.3 Resource management

In a regular Linux system resource management is more or less equal to scheduling. Therefore Subsection 3.3.1 below has been named *Scheduling in the Linux kernel*. There the basics needed to understand scheduling in Linux and subsequently in Android have been explained. The second subsection, 3.3.2, mainly describes the Wake Lock concept introduced with Android. Lastly Subsection 3.3.3 describes resource management from the ACTORS RM.

3.3.1 Scheduling in the Linux kernel

This section is dedicated to explaining some basic facts around how Linux scheduling works, both in general and in some detail. The overall idea behind scheduling is quite simple. At any given time the scheduler in a system needs to decide which process in that system should be the next one to run [33]. All processes that currently want to run are in the state *waiting to run*. Of course, if the system has multiple cores, many processes can run in parallel.

Preemptive multitasking

There are (at least) two different approaches to running multiple tasks at the same time, preemptive multitasking and cooperative multitasking. Linux is a good example of an operative system doing preemptive multitasking. The preemptive multitasking means that the scheduler will decide when a process is allowed to run as well as when it should cease running. The time a process is going to run is most commonly predetermined and is called that process' timeslice.

Cooperative multitasking

In a system doing cooperative multitasking the scheduler has no possibility to tell a process to stop running. Instead the system completely relies on that processes voluntarily suspends themselves, also known as yielding. This concept relies on processes yielding rather often which should result in each process getting a decent size timeslice. Unfortunately frequent yielding is not necessarily the

case and the worst case scenario is that a process actually never yields resulting in a system crash. Most modern operative systems are preemptive. Examples of cooperative multitasking systems are typically Windows 3.1 and earlier as well as Mac OS 9 and earlier. Cooperative multitasking is also known as non-preemptive multitasking.

Policy and scheduling classes

A scheduling policy in a Linux system defines the behaviour of the scheduler in terms of what runs when. The policy is responsible for the overall responsiveness and throughput of a system, hence it is really important. The policy consists of a couple of different things where priority and timeslice are two of the more significant.

Priority
A common way of scheduling is priority-based scheduling [33]. This type of scheduling is based on two factors in Linux and Unix systems. These are real-time priority or nice values. The nice value ranges from -20 to +19 where a lower number corresponds to a higher priority and the other way around. The real-time priority ranges by default from 0-99. For real-time processes a higher real-time value corresponds to higher scheduling priority.

Timeslice
A timeslice is a numeric value which represents the amount of time a task should be allowed to run before it is preempted. A too long timeslice will give that process a poor interactive performance, whereas a too short timeslice will cause too much CPU time to be lost on the overhead of context switching. The default timeslice value is 10 ms or less. This is to prevent the system from getting a poor interactive performance.

Scheduling algorithms

A Linux system is very modular when it comes to utilizing different algorithms and schedulers. It allows different schedulers, also known as scheduler classes, to be scheduling different types of processes. These different scheduler classes are very much allowed to co-exist in the system. It is the base scheduler (found in *kernel/sched.c*) that iterates in order over the schedulers in the system to find which scheduler will be running next. The highest scheduler class with a runnable thread wins.

The scheduler class used for normal processes is the Completely Fair Scheduler (CFS). This is also the scheduler that is utilized in this thesis project. It should also be mentioned that for the needs within the ACTORS project however, the CFS did not have the right properties [50]. So instead it was decided to develop a new scheduler which should implement the Earliest Deadline First (EDF) algorithm [49].

It would have been interesting to also try using either the SCHED_EDF or SCHED_DEADLINE, described below, as part of the thesis project. However there was not enough time to do so.

Completely Fair Scheduler (CFS)
The CFS was merged into the Linux kernel starting with version 2.6.23 [33][20]. It replaced the previous O(1) scheduler [33][36] as the standard scheduling class.

The CFS class aims at maximizing both CPU utilization and interactive performance. Instead of using run queues CFS keeps track of all processes in a runnable state by using a single red-black tree [34]. The CFS looks in the node farthest to the left in the tree to find the process most entitled to run. The red-black tree is ordered by a key assigned to each process. When a task is placed in the run queue of the system, the CFS keeps track of the CPU time that the task had the right to have. This time, with some adjustments for higher-priority processes, is the key for that specific task.

The CFS also uses the nice value to calculate a timeslice, so that a process with lower nice value (higher priority) will get a proportionally larger timeslice compared to a process with higher nice value (lower priority).

This way of taking both nice values and utilization so far into account makes each application run for a fair amount of time.

The CFS does not assign a set time period but rather a proportion of the CPU. The amount of CPU time that a process receives is a function of the load of the system. Then that proportion is further affected by the process's nice value.

SCHED_EDF and SCHED_DEADLINE
SCHED_EDF [50][51] is the name of a new scheduler class for Linux created within the ACTORS project. It implements the real-time scheduling algorithm EDF. This initial version was targeted to fulfill every need of the ACTORS project as a whole. The scheduler class later forked into a separate implementation, SCHED_DEADLINE [49] which was essentially implementing the same algorithm as the EDF, however targeted for the broader Linux community.

One of the key features of these scheduler/-s is that it ensures *temporal isolation*. This means that the ability for any process to meet its deadline is not affected by any other task in the system. In this way a task that does not meet its deadline or misbehaves in any other way has no possibility to affect any other part of the system.

3.3.2 Resource management in Android

Android has its own power management which works as a complement to the standard Linux power management [16]. As described in the Android documentation, the Android power management was designed with the goal that a CPU should not consume power if there is no application or service requiring it to do so.

Wake lock

The constant goal for the system is to be able to shut down the CPU, which means that a service or an activity that wants to run has to set a so called *wake lock* to request CPU resources from the system. If the system has no active wake locks it will inevitably shut down the CPU. The Figure 3.5 below shows the architecture of the power management mechanism. *Solid elements represent Android blocks and dashed elements represent partner-specific blocks* [16]

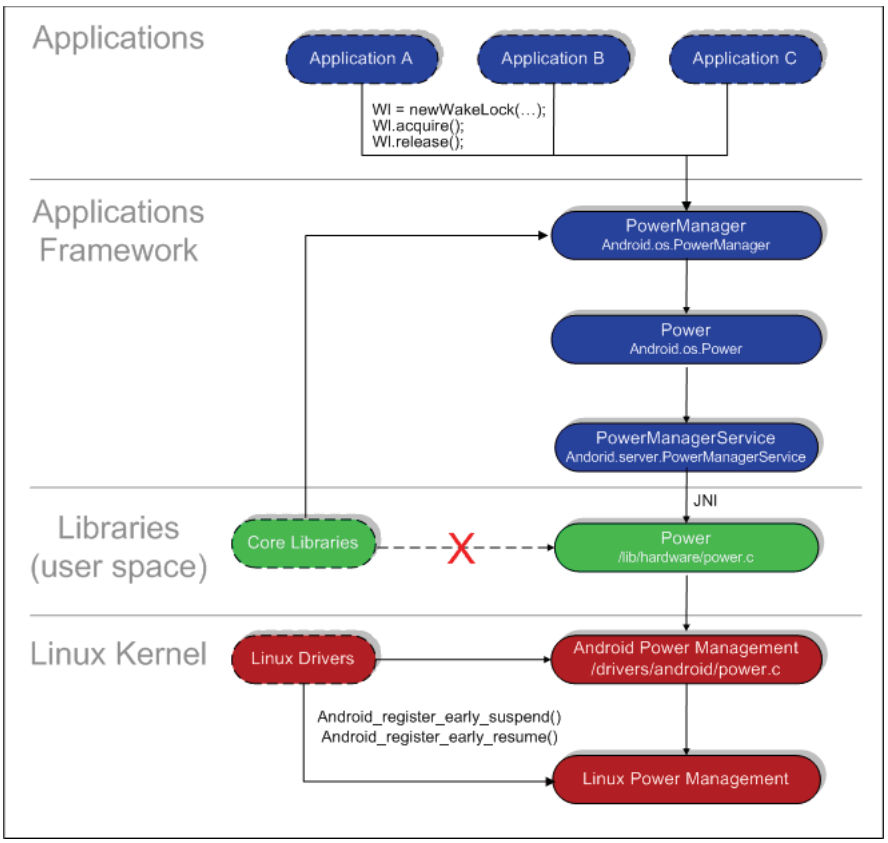

Figure 3.5: Wake lock schematics. Ref. [16]

A set wake lock prevents the system from changing into suspend or low-power state. There are two different settings, WAKE_LOCK_IDLE and WAKE_LOCK_SUSPEND:

> WAKE_LOCK_SUSPEND: prevents a full system suspend.
> WAKE_LOCK_IDLE: low-power states, which often cause large interrupt latencies or that disable a set of interrupts, will not be entered from idle until the wake locks are released.

The wake lock mechanism has not been changed in this thesis project, it is still working as before.

Android cgroups

In Section 3.1.3, the cgroups file system is explained in general and briefly how it is used. In Android there is a standard mount of the cgroup mechanism in */dev/cpuctl*. In this standard setup there are two groups with corresponding subfolders set up to handle all processes in the system. The two groups are called *fg_boost* and *bg_non_interactive*. All processes that need to have an extra boost while running in the foreground are placed in the *fg_boost* cgroup. And in the *bg_non_interactive* cgroup all processes placed that are not currently running or requiring an extra boost. There are also processes located in the root directory, which are typically system processes.

3.3.3 Actors resource manager

Ericsson Research has developed a resource manager [1][2] together with partners in the ACTORS EU-project [3]. One of the partners is the Department of Automatic Control at Lund University [21].

The purpose of this resource manager is described in the introduction chapter of one of the deliverables of the project [1]:

> The resource manager is the core of the ACTORS' adaptive resource management framework. The resource manager constitutes the interface between the applications executing within the system and the underlying reservation-based operating system scheduler. It decides how the CPU resources should be divided between competing applications, both in terms of CPU bandwidth and how the execution threads of the applications are allocated onto cores. It also decides how the computing resource allocated to an application should be divided among the threads of that application. Finally, it decides at which quality or service level that the individual applications should execute at in order to obtain the best global performance.
>
> The resource manager is implemented as a multi-threaded Linux user-level C++ module. It communicates with the applications, both CAL applications and legacy applications using a D-Bus interface. It communicates with the underlying Linux scheduler using the cgroups file system interface. The Linux scheduler either consists of the SCHED_EDF scheduler developed within ACTORS, the existing Linux Completely Fair Scheduler (CFS), or a combination of both. The latter case is made possible by the hierarchical nature of the Linux scheduler, in which tasks scheduled under SCHED_EDF are executing at the highest level and ordinary Linux tasks are scheduled at a lower priority.

How then does the resource manager know what resources (read percentage of CPU bandwidth) an application should have? This is communicated through the use of service levels.

Service level

Each application has several service levels it can operate on. A service level consists of the following information:

- **Quality of service** - This is how well the client can perform on this service level. Values range 0-100.
- **Bandwidth** - This is the percentage of CPU the resource manager allocates to the client. In a single core system values range 0-100. In a dual core (as the MOP500 that is used in this thesis project) values range 0-200.
- **Granularity** - This is the period in microseconds this client should be scheduled on. This is not used in the thesis project since the CFS is used which does not use explicit periods.
- **Bandwidth distribution specifier** - This is used to describe how the values in the Bandwidth distribution map are to be interpreted. Hence, whether they should be interpreted as absolute values or relative to each other. There is also the possibility to say that no special distributions needs to be done. That is what has been used in this thesis project.
- **Bandwidth distribution count** - This tells the resource manager how many values there are in the bandwidth distribution map. It is not used in this thesis project.
- **Bandwidth distribution map** - This is a map of thread ids and the amount of the bandwidth they should have. It is not used in this thesis project.

To sum up, in this thesis project only quality of service and bandwidth were used.

An application that demands a lot of CPU bandwidth may have a high quality of service on the service levels where the bandwidth is high and a low quality of service on the service levels where the bandwidth is low. A game for instance might have the following service levels:

Service level number	Quality of service	Bandwidth (percentage of one CPU)
0	100	80
1	50	70
2	20	50

These service levels tell the resource manager that the game will perform well if it gets 80 percent of the CPU bandwidth and that it will perform half as good if

it has to run on service level 1. If the game is forced to run on service level 2, its performance will only be one fifth compared to running on service level 0.

A background application that is not prioritized on the other hand, e.g. an File Transfer Protocol (FTP) server that transfers files, might have the following settings:

Service level number	Quality of service	Bandwidth (percentage of one CPU)
0	100	80
1	95	10
2	90	5

This application will perform well if it gets 80 percent of the CPU bandwidth. In contrast to the game described above, it will perform almost as well even if it only gets 10 percent. So when the CPU bandwidth is not enough to satisfy both applications to run on service level 0, the resource manager should change the service level to 1 for the FTP server since it will still perform quite good but the game on the other hand would perform bad if it had to change its service level to 1.

The resource manager interface

To make the ACTORS RM able to know what applications it should control, each client must register with it. This is done via an IPC system called D-Bus. The resource manager implements a D-Bus interface consisting of several methods an application can call. The methods are the following:

- `registerApp`
- `announceCPUCategory`[3]
- `announceCPUPowerCategory`[4]
- `announceContinousQualityFunction`
- `announceServiceLevels`
- `createThreadGroup`
- `addThreadsToGroup`
- `moveThreadsToGroup`
- `deleteThreadsFromGroup`
- `keepGroupsSeperate`
- `commit`

[3]Discussed further in Section 3.1.3 since changes have been made to this method
[4]Discussed further in Section 3.1.3 since this is an addition to the interface

- `unregister`
- `reportHappiness`

All of the methods will have `clientId` as a parameter. This is a unique name specifying what client should be operated on by the resource manager. Most methods return a value to the application trying to register with the resource manager. The result is always 0 if the call was successful, otherwise non-zero.

int registerApp(String clientId, int applicationId)
This method will create a client object with default values. It will not be put into use by the resource manager until after the commit method has been called with a successful return value.

int announceCPUCategory(String clientId, int cpuCategory)
This method will set the CPU category for the specified client. CPU category is used to choose what margin should be used[5]. CPU category can have the following settings:

- Low
- Medium
- High

int announceCPUPowerCategory(String clientId, int cpuPowerCategory)
This method will set the CPU power category for the specified client. CPU power category is used to choose what ceiling should be used and whether or not the fast flag should be set[6]. CPU power category can have the following settings:

- Tight
- Max

int announceContinousQualityFunction(String clientId, FunctionType type, int param1, int param2, int param3, int param4)
This method is used to tell the resource manager what function should be used to map quality of service to resources. It is only used by clients that do not offer discrete service levels. Hence, this function is not used in this thesis project.

int announceServiceLevels(String clientId, int initialServiceLevel, int serviceLevelCount, ServiceLevel[] serviceLevels)
This method is used to let the resource manager know what service levels the client has.

int createThreadGroup(String clientId, int groupId)
This method will create a thread group for the specified client. All of that client's

[5]More on that in Section 3.1.3 above
[6]More on that in Section 3.1.3 above

threads will be placed in that thread group. A client must have a thread group to be able to run. Only one thread group per application is used in the adapted open source applications.

int addThreadsToGroup(String clientId, int groupId, int threadIdCount, int[] threadIds)
This method will add the specified threads to the specified thread group.

int moveThreadsToGroup(String clientId, int sourceGroupId, int destination-GroupIt, int threadIdCount, int[] threadIds)
This method will be used when moving threads to another thread group. It is not used in this thesis project.

int deleteThreadsFromGroup(String clientId, int groupId, int threadIdCount, int[] threadIds)
This method will be used when deleting threads from a thread group. It is not used in this thesis project.

int keepGroupsSeperate(String clientId, int groupIdCount, int[] groupIds)
This method will make sure different thread groups will not execute on the same physical core. It is not used in this thesis project.

int commit(String clientId)
This method will be executed when all necessary methods have been called by the connecting client. If this method returns 0, then the client is accepted and the resource manager will allocate resources for it.

void unregister(String clientId)
This method will unregister a client from the resource manager. All resources will be free again and available to the other applications if they need resources.

void reportHappiness(String clientId, int happiness)
This method will let the resource manager know if the client is happy about its resources. It is used as a way for the clients to give feedback to the resource manager so that it can make better resource distributions. If the client is happy with its resources it should call `reportHappiness` with `happiness` set to 1, otherwise 0.

This method should have been used in the project. However, there were problems finding a good general method to figure out whether or not an application is happy. Hence, this method was not used.

The client interface

The clients also must implement some methods to make it possible for the resource manager to call methods on the client. The methods the client must implement are:

- `changeServiceLevel`
- `changeContinous`

void changeServiceLevel(int serviceLevel)
This method is called by the resource manager to let the client know when its service level has been changed. It is up to the client to do what it wants with the information.

void changeContinous(int value)
This is called by the resource manager to let the client know the new amount of resources that is available for it. It is used if the client announced a continuous resources to quality mapping function. That has not been done in this thesis project so this method is not used.

Registration sequence

When an application wants to register as a client to the resource manager it will call the methods of the resource manager in the following order:

- `registerApp(clientId)`
- `announceCPUCategory(clientId, cpuCategory)`
- `announceCPUPowerCategory(clientId, cpuPowerCategory)`
- `announceServiceLevels(clientId, initialServiceLevel, serviceLevels.size(), serviceLevels)`
- `createThreadGroup(clientId, threadGroupId)`
- `addThreadsToGroup(clientId, threadGroupIt, threadsToAdd.size(), threadsToAdd)`
- `commit(clientId)`

If all goes well, the `commit` method will return 0 and the application is now a client of the resource manager. The client will be assigned resources by the resource manager based upon the service level values, the CPU category and the CPU power category.

The importance file

Apart from using quality of service to help the resource manager better decide what application it should prioritize, the resource manager will also read an importance file on startup. The importance file is called `actrm.imp`. Each line in the file has the format "clientName = value" where clientName is the name the application will register with the resource manager and the value is an integer value from 1 and up. The higher the integer value is, the higher priority the client has.

3.4 Power management

There are many different ways of saving power in a computer system, i.e. by lowering the utilization of the CPU, I/O or radio modules such as Wi-Fi. However, in this thesis project the focus was on only one of them: lowering the total CPU-utilization of the system.

The formula for calculating power consumption is:

$$p = CV_{DD}^2 f \tag{3.2}$$

Where p is power, C is capacitance, V_{DD} is voltage supply and f is the CPU clock frequency [43]. If only scaling the CPU clock frequency this would lead to a linear reduction/increase in power. However, using dynamic voltage scaling[7] the power reduction is even larger when clocking down the CPU. There are different suggestions on how much this affects power consumption, e.g. if the reduction is an order of two or three [44]. For this thesis, an order of two has been chosen so that no exaggerations on the power reductions are made.

Draining a battery slowly will deliver more energy than flash discharge [45]. If the CPU runs on a lower clock frequency it will drain the battery slower and hence get more energy out of the battery. Since Android in most cases is run on battery powered devices, draining the battery slowly is the preferred behaviour.

3.5 Platforms

During the masters thesis work three different platforms have been used to run Android. Their hard facts can be seen below or by visiting the web-pages referred to in the table.

The first one, maybe also the most obvious one is the Android SDK Emulator [52]. It emulates an ARM v5 CPU with a corresponding memory-management unit. This first platform was used during the first phase of the thesis work to learn how to program for Android. It has also been a great help in the later parts

[7] Dynamic voltage scaling is when the voltage supply is increased/decreased simultaneously with the clock frequency

of the thesis project while there has been parallel development done, but only one development board available.

The second platform used was the (Original) BeagleBoard [53][62]. The BeagleBoard is a USB-powered, fan-less single board computer which can run a number of different operating systems such as Windows CE [8], RTOS [9] and Android [10]. Since Android was a prerequisite and Beagledroid, which is a port of Android to BeagleBoard, had been tried out in Ericsson Research before, it was an easy decision to pick that one.

The BeagleBoard was used in the project only for a short while, up until it was decided to change to the ST-Ericsson MOP500 development platform based on the ST-Ericsson U8500 smartphone platform [55]. The reason for the change of platform was partly that it allowed running the latest version of Android, 2.2 [11], and a much newer Linux kernel, 2.6.34 instead of 2.6.29. It was partly also due to the fact that Ericsson Research saw a greater value in using a mobile industry platform instead of a platform targeted for education. The ST-Ericsson MOP500 was the platform all tests were performed on.

	BeagleBoard (original)	**MOP500**
Chipset	TI OMAP3530 [62][56]	ST-Ericsson U8500 [54]
Processor	550 MHz ARM Cortex™ A8	600 MHz ARM dual Cortex™ A9
# of cores	1	2
RAM	256 MB	512 MB (352 MB to OS)
Console	USB and serial cable	USB and serial cable
Connectivity	Ethernet over USB, Wi-Fi and Bluetooth via peripherals	Ethernet over USB, Wi-Fi, Bluetooth and GPS via peripherals

Table 3.1: Hardware comparison between BeagleBoard [53] and MOP500 [55]

[8] Windows Embedded CE 6.0 [63]

[9] QNX Neutrino RTOS on OMAP [46]

[10] Beagledroid [12]

[11] At the beginning of the masters thesis project Android version 2.2 was the latest released version

Chapter 4
Design

During the creative design process the focus has no doubt been the requirements as stated in the problem formulation. To maybe repeat the obvious, it should be as easy and straight forward as possible for an Android application developer to make his or her application aware of the ACTORS RM. The lowest threshold for an application developer should be really low. In other words it is preferable with some sort of standard settings for an application where the developer has not given any information to the ACTORS RM.

An overview of the system can be seen in Figure 4.1. When using the resource manager, the application will register with the resource manager when it starts. When it registers it will supply service level settings which is basically information saying how much CPU bandwidth the application needs on different levels. It also tells the resource manager how good it can perform on those different service levels.

The resource manager takes all of the applications' service levels into account and makes reservations for the applications. The scheduler uses this information (and some other factors) to decide how much the threads of that application will run on the CPU.

4.1 Overview

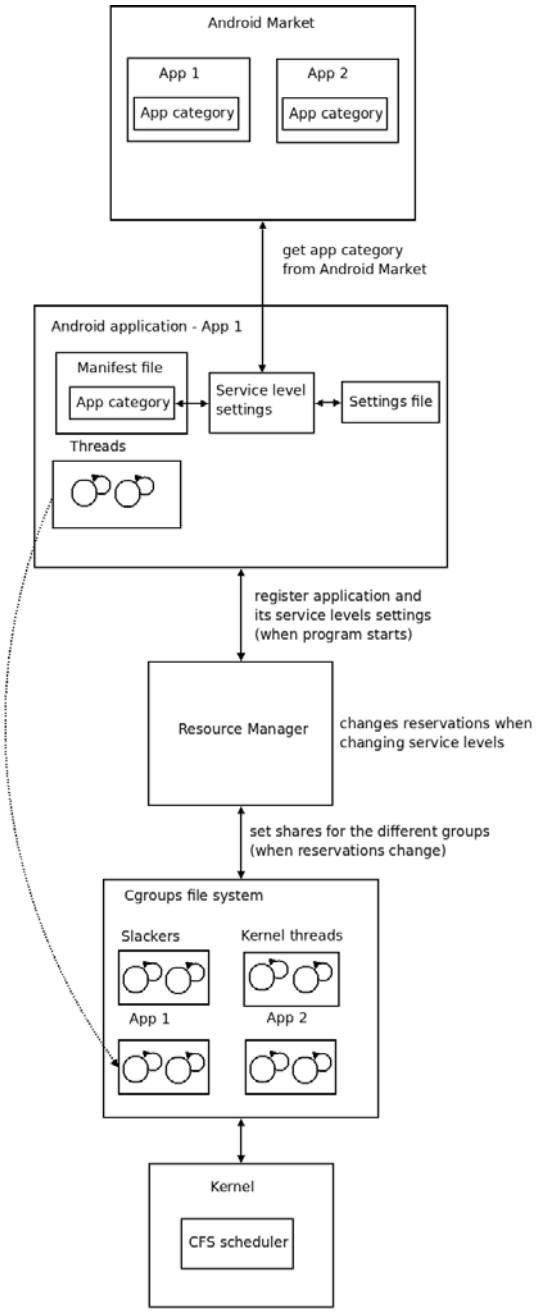

Figure 4.1: Overview of the system

Below follows a more detailed explanation of the system.

- **Default category** - All applications downloaded from the Android Market are categorized as one of 26 different categories. As described in Section 5.4 these categories are used in the implementation. They are, however, not automatically parsed from the Android Market information which would be the ideal case.
- **Custom category** - If the developer does not want to use the default category and would like to set it manually, that should also be possible.
- **Custom service levels** - If the developer wants to supply more detailed information, service levels settings can be supplied to the program using the settings file. This is the most detailed information the developer can supply to the framework.
- **Registration/Unregistration** - Registration of an application with the ACTORS RM should be done automatically when the application launches. The application should stay registered as long as it is running. When an application is closed (e.g. when the `onDestroy` method in Android's `Activity.java` is called in the last component of the application) it will be unregistered from the resource manager.
- **Program running alone** - When an application is running as the only client registered with the resource manager it will be supplied as much as possible of its desired resources. Basically this is because there is no other application around to steal resources from it.
- **Multiple programs running** - If there is one application running alone, as in the previous scenario, and another application starts, the resource manager will take all settings mentioned in Section 3.3.3 from those two applications into account when deciding if the second application will be allowed to run. To be able to fit the second application, the resource manager might have to change the service levels of one or several of the registered applications (or of the registering application). This evaluation, or rather re-evaluation is performed every time a new client tries to register or unregister.

4.1.1 Service dependency

When an Android application runs, it makes use of other services running on the device. These services should ideally be "baked in" together with the application so that they get scheduled in the same fashion that the application get scheduled. In other words, if the application has a lot of resources then the services it depends on should also have a lot of resources.

The *SurfaceFlinger* service (which consists of two threads) for instance is used to draw the screen. The system should work conceptually as the Figure 4.2 shows.

If *SurfaceFlinger* would have dedicated threads to each GUI application then those threads could be added to that application's cgroup.

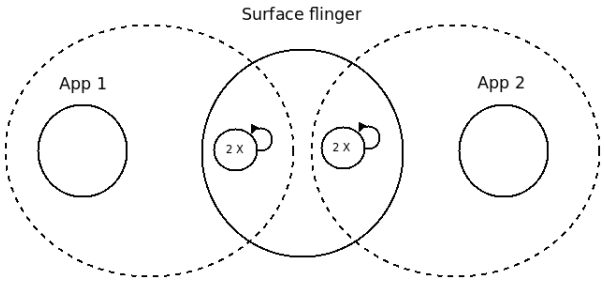

Figure 4.2: How services should be used

Unfortunately this is not the case. *SurfaceFlinger* uses the same threads when drawing different applications. Therefore, the resource manager can not put *SurfaceFlinger*'s threads in an applications cgroup since *SurfaceFlinger* is used to draw the screen for all applications. The real case looks more like Figure 4.3.

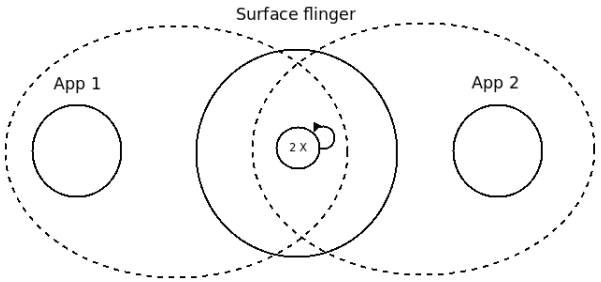

Figure 4.3: How services are used

This caused some problems when the resource manager puts the system tasks in different cgroups in its startup phase. All of those tasks which other applications might be dependent upon were put in a special cgroup called slackers, */dev/cgroup/actrm-internal-slacker-internal-slacker-0*. Since maybe some high priority application needed one of those task, they were allowed to run freely and were not limited. That was of course not ideal in all cases since maybe a low priority application will activate a task in slackers and make it use a lot of CPU power.

The same goes with kernel tasks. In its startup phase the resource manager puts all kernel tasks in the cgroups root group, */dev/cgroup*. Since they are in the root cgroup they will be weighed heavier against all other cgroups when the scheduler operates.

Chapter 5
Implementation

To implement the design using the resource manager the first thing that had to be done was porting it to Android. This initial port was was done by Song Yuan. The resource manager depends on some C++ libraries and the GLPK library [25]. Yuan linked the resource manager with the C++ implementation of `Standard Template Library (stl)` [**?**] called `stlport` which is shipped with Android. He also downloaded the source code for GLPK and added Android makefiles to make it build.

At an early stage it was discovered that using D-Bus as the IPC system was not optimal. Since Android uses Binder to do IPC when Android applications communicate with each other, it was decided that using Binder instead of D-Bus would be much better. Another benefit is that, as mentioned in 3.2.2, Binder is approximately a factor of 100 times faster than D-Bus. Having decided to use Binder, a Binder interface to the resource manager was needed. This was solved by creating an Android application called MiniRM.

The MiniRM application was created to make it easy for Android applications to talk with the resource manager. The MiniRM package consists of two parts. The first is a service that other Android applications can connect to. The second is an activity (GUI) that will start the service, bind to it and call setup methods that the service provides. The reason why this Java wrapper was needed was that it would function as a middleman between Android applications and the resource manager. The resource manager is implemented in C++ and was previously designed to be compiled into a binary executable and run with root privileges.

5.1 Binder in C++ or in Java?

The resource manager is written in C++ so it would be much nicer to avoid using Java at all. It is possible to write applications in C++ that will use Binder as communication. There are examples of that on the internet [24]. Using Binder in C++, a system where Android Java application would communicate with the resource manager's C++ Binder service could be obtained. A discussion was held with Dianne Hackborn on the *android-porting google group* on how that should be

done [13]. First she proposed that a Java wrapper should be used, as also was the final choice [14]. She also said it could be possible to export the C++ IBinder interface into Java using JNI calls [15]. JNI makes it possible to call C and C++ methods from Java [30]. A JNI method call is a method call from Java to a C or C++ method.

Both approaches were tested. First the second proposal was tested since that would allow for more programming to be done in C++ and doing the Binder communication in C++. However, a Java service would still have had to be used to export the IBinder interface from C++. Since some problems with exporting the IBinder interface were discovered the first proposal using a Java service was eventually decided upon.

The service will have all the methods the resource manager has in its D-Bus interface, see 3.3.3. The resource manager is compiled into a shared library instead of being compiled into a binary executable. JNI calls are used by the Java service to call the appropriate method in the resource manager. To make it possible to call methods in the resource manager library, it was necessary to create a static C++ JNI library that would call the correct C++ methods in the resource manager.

So when an application wants to register as a client to the resource manager it will bind to the Java service and call the service's `registerApp` method. That `registerApp` method will call a JNI method called `jniRegisterApp` that will in turn call the correct library method of the resource manager. This is sketched in Figure 5.1.

It was necessary to Figure out what the D-Bus interface did with the calls made to it. That is, what code executed when an application registered with the resource manager using D-Bus. The same code had to be executed in the MiniRM JNI library doing the call from Binder.

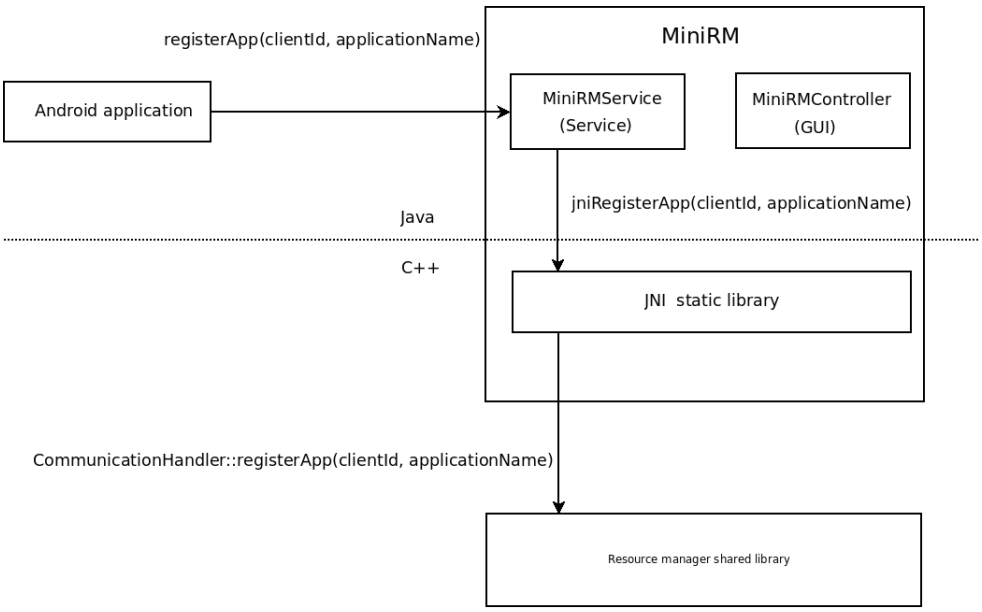

Figure 5.1: Communication from Binder down to C++ method call

5.2 MiniRM GUI (MiniRMServerController)

For debug reasons it was desirable to control when the MiniRM service starts and stops. Therefore a simple GUI was developed that would have two buttons: start and stop. The start button starts the service (using the `startService` method in the Android SDK), then binds to the service and calls the `startRM` method on the resource manager that does some setup. After that call, it will unbind from the service.

The stop button simply stops the service by calling the `stopService` method in the Android SDK. A screenshot of the GUI can be seen in Figure 5.2.

5.3 Alterations to the resource manager

To make the resource manager function as a library instead of an executable some changes had to be made to the resource manager code. A substantial amount of time in this thesis work was put into removing the D-Bus code and make functions callable from JNI. The D-Bus communication was not modularized and separated from the resource manager which made replacement hard. Some classes could be removed but some methods and data (that were not D-Bus specific) had to be moved to other classes.

Apart from the difficulties with the Android build system because of insuf-

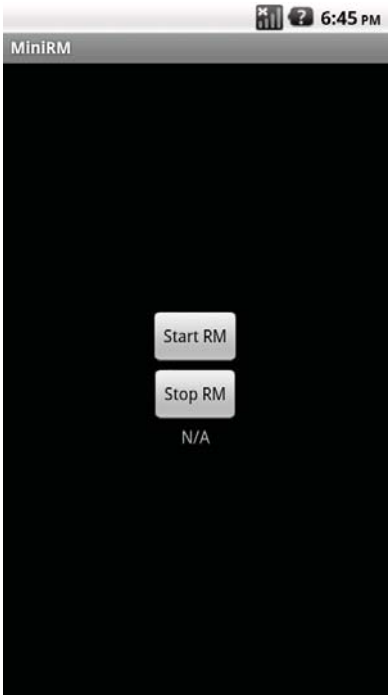

Figure 5.2: Screenshot of the MiniRM GUI

ficient documentation, the only change that needed to be done to make the resource manager build as a library was to rename the `main` method to `initrm`. Since it would not be a standalone program executed from the command line, the `main` method would not be needed.

However the functionality of the `main` method was of course still necessary hence the creation of the `initrm` method. This method has to be executed before applications can start registering. Therefore, it is executed in the `startRM` method which the MiniRM GUI calls after starting the MiniRM service. The `startRM` also starts some Java threads that calls JNI callback methods that are needed when D-Bus is gone.

The system was also configured to always distribute the threads of an application evenly between the two cores in the system. Hence, the affinity set in each cgroup, `cpuset.cpus`, was set to `0-1`. This was to give all threads the possibility to run equal amounts of time. Another reason was to let the system decide which core to put each thread on.

5.3.1 Replacing D-Bus with Binder and JNI

The greatest challenge was to replace D-Bus with Binder. The resource manager was designed to work with D-Bus so a lot of changes had to be made to the source

code files.

As already mentioned, clients can communicate with the resource manager over Binder using JNI calls down to a C++ library. That library will then call the corresponding methods in the resource manager library. It also had to be possible for the resource manager to return the result of commands.

The resource manager must also be able to call some methods on the client, e.g. `changeServiceLevel`, see Section 3.3.3. This was solved by adding an extra argument to the `registerApp` method. The argument is the client Binder interface instance for that client. The Binder interface first had to be created on the client. In `registerApp`, the MiniRM service will have a map called "clients" linking client ID:s to client interface instances. So when the resource manager wants to call a method on a client, it simply looks up the client ID in the "clients" map and it will receive the Binder interface.

The calls to the resource manager is not preventing other clients to make calls. When the resource manager gets a call, e.g. `registerApp`, it will create a message with the right information (e.g. which client made the call and what kind of call it was), post it to a mailbox and then exit the method. It is the responsibility of a separate thread to handle the messages in this mailbox. *It is only after the the message is handled by this separate thread that the resource manager has a result it can return*. With D-Bus this was easily solved by putting the connection ID of the D-Bus connection in the message. Later when the message was handled, the method that knew the result could directly call the D-Bus client using the connection ID.

This is not the case with Binder in Java. The solution was to let the resource manager call a client method when the result from that client's method call was done as shown in Figure 5.3. In the background Chapter 3.3.3 it was described what methods the client had to implement so that the resource manager could call methods to e.g. let the client know its service level had been changed. Yet another method had to be introduced for the resource manager to be able to return values to a client after it had called a method in the resource manager.

At the same place in the source code where the resource manager earlier would have used the D-Bus connection ID for the client, the resource manager would instead create a message and post it to a mailbox which only would contain return value messages. This message would consist of the client ID and the integer value that should be returned. Another thread (one of those that got started from the `startRM` method) handles the messages in that mailbox. It waits until there is a message in the mailbox. When a new message arrives it does a Java JNI call (but from C++ to Java instead) to a method in the MiniRM service called `returnValue`. The `returnValue` method takes client ID and the integer value as parameters. This `returnValue` method is not a Binder method but just a normal Java method. This method will then look up the client ID in the "clients" map, get the corresponding Binder interface and then call the `returnValue` method with the return value.

The same solution was used when notifying clients that their service levels had been changed. A separate mailbox was used for that and a separate thread to handle that mailbox (also started from the `startRM` method). Additionally another Java method had to be added to the MiniRM service that the resource

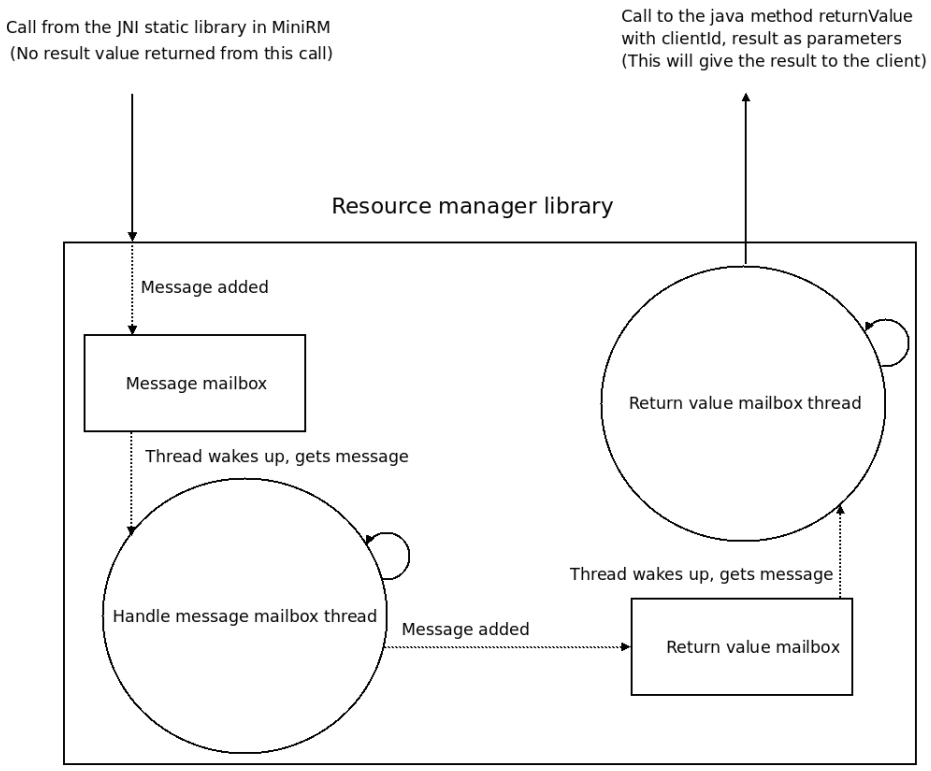

Figure 5.3: How result values are solved

manager library could call using JNI (from C++ to Java). That method would look up the client ID in the "clients" map and get a Binder interface. It could then call the `changeServiceLevel` method on that interface. The resource manager could do this whenever it wanted to notify a client that its service level had been changed.

5.3.2 Handling of CPU category and CPU power category

Another thing that was changed in the resource manager was the handling of the CPU category and CPU power category that were sent from the client. The idea is that when a client registers, it sends some settings that consists of:

- Service levels, see Section 3.3.3
- CPU category, see Section 3.3.3
- CPU power category, see Section 3.3.3

The service levels are simple, the bandwidth is directly mapped to the percentage of the CPU the client may use by setting the share value for the cgroup

share. The CPU category and the CPU power category set the ceil, margin and the fast flag for the client's cgroup. All of these cgroups settings are described in Section 3.1.3.

When setting the CPU category low it will result in a low margin and when setting it high it will result in a higher margin.

When setting the CPU power category to tight it will result in setting the fast flag to 0 and setting the margin to be absolute values. The values are 1, 3 or 10 percent of the maximum CPU frequency and are dependent on CPU category. When setting the CPU power category to max it will result in setting the fast flag to 1 and setting the margin to relative values. The margin is relative to the ceiling which in turn is relative to the bandwidth for the current service level. The values are 5, 10 or 20 percent of the ceiling for the client and are dependent on CPU category.

5.4 Changes made to Android

So far it has only been discussed how applications could register as clients with the resource manager. But as stated in the problem formulation and in the design chapter, it should be made as easy as possible for Android application developers to make use of the resource manager. That should include a convenient way to automatically connect to the resource manager and in an easy way supply settings for an application without needing detailed knowledge of how the resource manager works.

To emphasize the changes made to Android in order for getting the ACTORS RM up and running, they have been placed in its own sub-section. The second sub-section below contains a description of all other alterations made to Android for getting proper results, tracing, using cgroups etc.

5.4.1 Integration of the Actors RM into Android

Registration should be performed when the application is launched. When an Activity (i.e. a GUI) starts, its `onCreate` method will be called. There are default implementations of the life cycle methods described in 3.2.2. As stated on the Android developer website [11], these should always call its super class implementation:

> An implementation of any activity life cycle method should always first call the superclass version. For example:

```
protected void onPause() {
    super.onPause();
    . . .
}
```

Therefore it is fairly certain that the default implementations always will be run. Indeed, if a developer were to add the Android Development Tools (ADT)

plugin to Eclipse to automatically override one of those life cycle methods, the call to the superclass version would be inserted. If the developer has followed this rule, the default implementations will always be run. Changes described in this section have been made in those methods.

OurActivity and OurService

As mentioned in Section 3.2.2, focus has been on the Android components Activity and Service. Both Activity and Service have the `onCreate` and `onDestroy` methods. It is in these methods either a registration or an unregistration with the resource manager will be made. Ideally the necessary changes would have been inserted directly into the base classes `Activity.java` and `Service.java` in Android. Then all Android applications would register with the resource manager. But since full control over what was happening was preferable, two intermediate classes called OurActivity and OurService were created.

These classes extend Activity and Service respectively and override the `onCreate` and `onDestroy` methods. Then a test application would extend those classes instead of extending the Activity and Service classes. Figure 5.4 explains the relationship between an application and the previously mentioned classes.

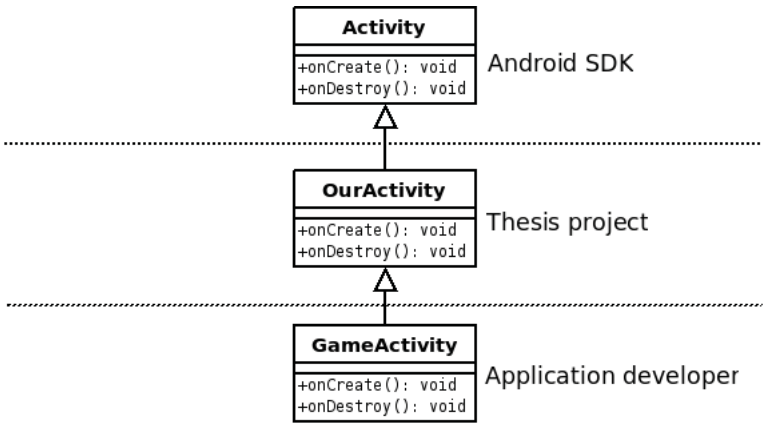

Figure 5.4: Example GameActivity extends OurActivity

It is not desirable to have an application unregistering each time an `onDestroy` method is being called. Rather, the application should unregister when the *last* `onDestroy` method is being called.

A program can have multiple OurActivity classes. Actually it would be quite common to in a game have one OurActivity for e.g. settings and one OurActivity for the actual playing of the game. When a new OurActivity launches the previous OurActivity's `onPause` method is called. When the new OurActivity finishes its `onPause` and `onDestroy` methods will be called as can be seen in Figure 3.3. The old OurActivity's `onResume` (and maybe `onRestart` and `onStart`) method will be called since it is that OurActivity's turn to execute again.

An application can have both an OurActivity and an OurService so the activities and services that are active must be kept track of. An application should not be unregistered when the last OurActivity's `onDestroy` method is called if there still is an OurService running. This was solved by creating a new class called OurApplication to keep track of it all.

In the first phase of the project it was considered to let each OurActivity and OurService register as separate clients to the resource manager. But that proved to be a quite complicated since they would share the same threads. It was decided that an application as a whole will register as a client not the different parts of it.

There was also the idea that an application could re-register with the resource manager to change its settings. It would still be the same client but with different settings. The reason for this was that some applications change their behaviour during runtime. An example would be a CPU intensive game. When playing, the game needs a lot of CPU bandwidth but when the game is in the game menu it does not need as much.

However some problems were encountered with re-registering an application so they no longer do that. If a client registers, re-registers and unregisters, the resource manager would get a segmentation fault. It was decided not to prioritize re-registering, hence the feature was disabled.

OurApplication

Since an application can consist of many OurActivity:s and OurService:s the application has to keep track of them. The application should register as one client whether it is an OurActivity or an OurService that launches first. The OurActivity:s and OurService:s must therefore have access to the same OurApplication instance. This is solved by letting the OurApplication class extend the Application class in Android.

When an OurActivity launches, its `onCreate` method is called. That method will call methods in OurApplication to let it know it has one more OurActivity or OurService. If it is the first one, OurApplication will try to register with the resource manager. If there is already an OurActivity or OurService created, the registration process is already done and hence there is no need to do it again.

Similarly, when an OurActivity finishes, its `onDestroy` method is called. That method will call methods in OurApplication to let it know it is done. If it is the last OurActivity or OurService, then there is no need to still be registered with the resource manager so OurApplication will unregister with the resource manager.

Java packages

The OurActivity, OurService and OurApplication classes, among some other help classes, were put in a Java package named `com.example.MiniRMOur`. Other files such as `.aidl` files were put in another Java package named `com.example.MiniRMOur`.

If a developer wants to use the changes to Android described, he or she must include these Java packages into the application.

AndroidManifest

It has been stated several times that it should be easy for a developer to supply settings to the resource manager. This was solved by introducing the "app_category" meta data to the AndroidManifest. A developer could then enter the following line in the `AndroidManifest.xml` file:

```
<meta-data android:name="app_category"
        android:value="tools"/>
```

Before OurApplication registers the application with the resource manager, the `AndroidManifest.xml` file is parsed to extract this meta data. If the meta data is found, "tools" in this example, the found category will be translated into the correct settings. It seemed reasonable to map the categories to something already existing, and since the Android market categorizes all of its applications it was decided that using those categories would be a good idea. The Android market was visited on the web to collect all of the application categories. The result can be seen in the list below:

- Game
- Game arcade
- Game action
- Game brain
- Game puzzle
- Game card
- Game casino
- Game casual
- Finance
- Health
- Communication
- Lifestyle
- Multimedia
- News
- Weather
- Productivity
- Reference

- Travel
- Comics
- Shopping
- Social
- Sports
- Entertainment
- Tools
- Demo
- Software libraries

The point of doing this was that if the developer did not supply any settings in the `AndroidManifest.xml` file, information about what kind of category the application belonged to would be fetched from the Android market automatically. This has not been a prioritized feature and it was therefore never implemented.

If an application developer would create a black jack game for instance, he or she would then have "game/card" as value to the "app_category" meta data in the `AndroidManifest.xml` file. This value would then be parsed before registering the application with the resource manager so that fitting settings for a card game would be created.

The idea is that a card game could be assumed to probably need less resource in terms of CPU bandwidth than what an action game in 2D or 3D maybe would need. This mapping between application categories and settings does of course have some flaws. For instance, the card game could be in 3D and the action game could be a simple text based adventure game such as NetHack [59] [26].

Another change that had to be done is to make the application use OurApplication instead of the default Application class. This is done by adding the following text to the application tag in the `AndroidManifest.xml` file:

```
android:name="com.example.MiniRMOur.OurApplication"
```

The settings file
When in the testing phase it was discovered that it is tedious having to rebuild the test applications every time the settings were tuned. Therefore yet another way to make it possible for the developer to supply settings for the application was included. Using a settings file named `settings.txt`, the developer can supply detailed information about the application such as service levels, CPU category and CPU power category. The file is small and would typically look like this:

```
ServiceLevel 0
quality 100
bandwidth 50
```

```
granularity 50000

ServiceLevel 1
quality 95
bandwidth 40
granularity 50000

ServiceLevel 2
quality 90
bandwidth 20
granularity 50000

Category low

PowerCategory tight
```

The structure of a service level is the same as described in Section 3.3.3. CPU category and power category are described in Section 3.3.3.

Implementing service level changes

If the application developer wants to make the application use its allocated resources in a better fashion, he or she can implement different service level methods. There are three different service level methods available to be overridden:

- `void serviceLevel0`
- `void serviceLevel1`
- `void serviceLevel2`

When the application runs in its highest service level, it is in `serviceLevel0`. If the resource manager decides to decrease the resources for a client, it will send a message containing the service level number it has changed the client to.

Which service level the application should run on is quite useful information to it. The application developer should want the user to have as good experience as possible when using the application. So when the resources are decreased for an application, maybe somethings can be switched off or be done later. A weather application for instance could maybe decrease the frequency with which it updates the weather information. A game could i.e. lower the animation or bit depth. There are a lot of settings that could be changed to make applications run better. It is up to the developer to decide what code should be executed when a service level has been changed.

There was work done on finding default changes that always could be done for each service level. Unfortunately it was not possible to find a way to do that since too little information is known about generic applications.

These service level methods will override the default service level methods in OurActivity and OurService. When the resource manager sends a message to the client that its service level has been changed, OurApplication gets that information and calls the implemented method in either OurActivity or OurService.

Unfortunately a developer must add the service level implementation in both OurActivity and OurService since OurApplication does not know if the application always contains e.g. an OurService. If there is no OurService in the application nothing would happen if OurApplication would call the `changeServiceLevel0` method for instance. Therefore, the developer has to put the implemented methods in both OurActivity and in OurService.

Notes on integration

As stated in Section 5.4.1 the changes from the thesis project should ideally be integrated into the base classes, Activity and Service. It is possible to do that integration, but since a controlled environment was important the integration was never carried out. The original plan was to make the integration in the end of the project but there was simply not enough time.

To make an application ACTORS aware, so that it will register with the resource manager, the minimum changes a developer has to do are the following:

- Include the Java packages `com.example.MiniRM` and `com.example.MiniRMOur` in the project
- Make all Activity and Service sub classes extend OurActivity and OurService respectively
- Add the `android:name` attribute to the application tag in the `AndroidManifest.xml` file

A script was created that did this for all of the test applications. The script does the changes mentioned above automatically.

To make the application use the resource manager even better the developer can add app_category settings in the `AndroidManifest.xml` file and also implement service level changes.

5.4.2 Other alterations

The only modification that had to be done to Android was adding functionality that wrote the system frame rate (FPS) to the system's trace file. The FPS debug functionality was removed from `/frameworks/base/libs/surfaceflinger/SurfaceFlinger.cpp` in the used version of Android, 2.2. However, an older version [18] of the file was found which contained all the necessary changes.

As described in Section 3.3.3 the ACTORS RM uses the cgroup file system interface (see Section 3.1.3) to handle all applications registered with the ACTORS

RM. To enable the resource manager to do so the standard cgroup setup with `fg_boost` and `bg_non_interactive` has been replaced with an ACTORS RM specific cgroup setup.

5.5 Changes made to the Linux kernel

In a similar way as with Section 5.4, the changes made to the Linux kernel have been divided into two subsections. One that includes all changes made to make the Actors RM work, and one section with all the other changes.

5.5.1 Integration of the Actors RM with the Linux kernel

The Actors RM cgroup is set up in `/dev/cgroups`. To create the new cgroup structure cgroup support and support for the group on demand governor had to be added when compiling the Linux kernel. Changes also had to be made in Android's `init.rc` file so that the cgroup structure would actually be set up at system boot.

The cgroup structure that is used is shown simplified in Figure 5.5. All of a cgroups' thread ID:s are written to the `tasks` file. The `cpu.shares` file is one of the factors making up the weight that the scheduler uses to decide what thread should run next. The threads in the root cgroup are higher up in the hierarchy and each thread in the root cgroup is scheduled alongside with the other cgroups.

The slackers cgroup initially has 20,000 as its CPU share. When another cgroup is created and gets a CPU shares value, that value is subtracted from the slacker's CPU share. The purpose of this is that all the cgroups (apart from the root cgroup) will get the 20,000 to divide among themselves. The division is done by the resource manager and is proportional to the reservation for an application.

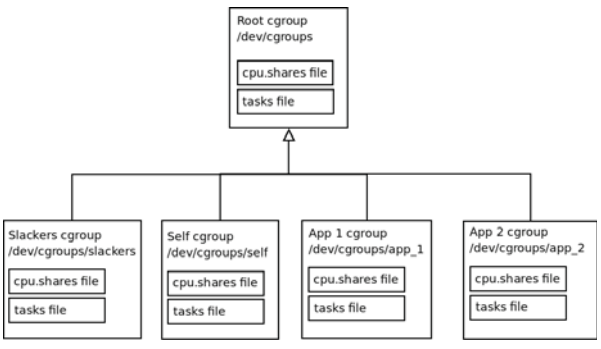

Figure 5.5: Cgroups hierarchy

5.5.2 Other alterations

Another change that also had to be made both to the Linux kernel and added to `init.rc` was the support for debugging using `ftrace` and `performance events`. The debug folder structure was mounted in `/debugfs`.

To get all information about the system performance needed, CPU frequency, current load, current margin and whether the fast flag was set or not, a new trace function called `power_grpondemand` was added to the Linux kernel (`kernel/include/trace/events/power.h`). The trace method was used in `/drivers/cpufreq/cpufreq_groupondemand.c`.

Chapter 6

Adapted open source applications

To see if the resource manager actually made a difference, it had to be tested with some applications. It was decided to test two major types of programs, applications that were purely GUI (using an Activity) and applications that were executed in the background (using a Service). The reason was that the user's requirements for these two types of applications, in the sense of user experience, are different. A user is more apt to be annoyed with a GUI getting less resources than a background application getting less resources.

It was decided to find four applications that would fit the descriptions below:

- Activity application which is mostly active
- Activity application which is mostly idle
- Service application which is mostly active
- Service application which is mostly idle

An idle application is an application that most of the time does not have to run. An active application is on the other hand an application that has to run a lot.

Four applications matching the descriptions above were found:

- Missile Intercept [35] - Activity application which is mostly active
- Android native calculator [8] - Activity application which is mostly idle
- SwiFTP [27] - Service application which is mostly active
- NoiseAlert [28] - Service application which is mostly idle

The first three applications were used in the scenarios and are described below. NoiseAlert was an interesting program but there were problems making it work properly even without making it use the resource management framework. This was probably due to broken drivers on the MOP500 or simply poorly written code in the NoiseAlert application.

It was first decided to also try the native browser since it is often used on a phone. Consequently making it work well with the resource manager would be very satisfying. However, the browser made use of ListActivity which extends Activity.

If a full integration had been done, the changes in `Activity.java` (which now resides in `OurActivity.java`) would be included in its subclasses such as ListActivity. In other words, in order to make the browser ACTORS aware, without doing a full integration, a substitution of ListActivity would have had to be made (named e.g. OurListActivity). There was not enough time to carry out this work.

6.1 Missile Intercept

Missile intercept is a re-interpretation of the classic game Missile Command from Atari [60]. It is not using extremely much of the CPU when playing but it is still noticeable. And a player can really see a difference if the game is running alone or if there is something else disturbing in the background. A screenshot from the game can be seen in Figure 6.1.

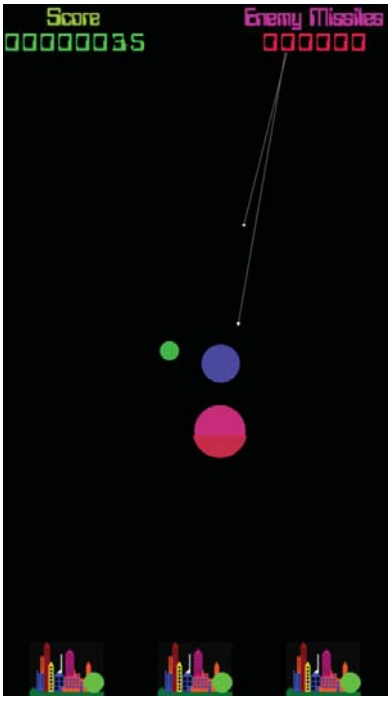

Figure 6.1: Screenshot of Missile Intercept

6.2 SwiFTP

SwiFTP is a truly well written and easy to use application. It offers just about enough features in respect of what you would like to have an FTP server in your mobile device doing. It gladly uses a lot of resources if not constrained by the system it is running under. A screenshot from the application can be seen in Figure 6.2.

The SwiFTP application was the only application for which it was decided to implement different service levels, see Sections 3.3.3, 7.2.7 and 7.1.

Figure 6.2: Screenshot of the SwiFTP GUI

6.3 Android native Calculator

There is not much to say about the Calculator other than the fact that it is a quite small application in terms of functionality, lines of code and CPU-utilization. A screenshot from the application can be seen in Figure 6.3.

Figure 6.3: Screenshot of the Calculator app

Chapter 7
Result

The results section is split into a couple of different subsections. The first one, Section 7.1, describes the different settings used for the applications. Then comes the user scenario overview in Section 7.2 where the scenarios that has set up and used are described in general. The third section, Section 7.3, contains descriptions and discussions around the most interesting scenarios and runs.

7.1 Different application settings

To see how the user scenarios would make the system behave with different settings, three different settings were used for the applications.

7.1.1 Game

This setting is used to tell to the ACTORS RM that it should prioritize this application. By having quality set to 50 in service level 1 the application will tell the resource manager it really will not be happy with being pushed down one service level. Also by using *high* as CPU category the application informs the ACTORS RM that it wants a high margin.

By using *max* as CPU power category the application informs the resource manager that it wants to have its margin relative to its ceil (which is proportional to the bandwidth). Lastly the game setting defines that the fast flag should be set.

All in all, this setting will let the application be prioritized and let the CPU run on the maximum frequency.

The game setting is described by the `game.txt` settings file:

```
ServiceLevel 0
quality 100
bandwidth 130
granularity 50000

ServiceLevel 1
```

```
quality 50
bandwidth 120
granularity 50000

ServiceLevel 2
quality 10
bandwidth 80
granularity 50000

Category high

PowerCategory max
```

The reason for the game setting to have the bandwidth set to 130 percent instead of the maximum 200 percent is that the ACTORS RM will never be able to assign all of the CPU bandwidth to one process. E.g. the ACTORS RM itself will consume some bandwidth. There will also always be system processes running consuming some of the bandwidth.

7.1.2 Tools static

This setting will make sure that the application will not get a lot of bandwidth. CPU category *low* will make sure there is a low margin as well. CPU power category *tight* will make sure the fast flag is not set and that the margin is absolute.

This setting will make sure that the application can not contribute to making the CPU clock up to a higher frequency.

The tools static settings is described by the `tools_static.txt` settings file:

```
ServiceLevel 0
quality 100
bandwidth 10
granularity 50000

ServiceLevel 1
quality 95
bandwidth 5
granularity 50000

ServiceLevel 2
quality 90
bandwidth 3
granularity 50000

Category low

PowerCategory tight
```

7.1.3 Tools dynamic

This setting will tell the resource manager that the application in question would like to get a lot of bandwidth. However, if there is another application that wants the bandwidth more, the application will be almost as happy (quality 90) to run on a lot less bandwidth. CPU category *low* will make sure there is a low margin as well. CPU power category *tight* will make sure the fast flag is not set and that the margin is absolute.

This setting will be used by an application for which it is not that important to run on full speed but would happily do that if no other program needs the bandwidth.

The tools dynamic setting is described by the `tools_dynamic.txt` settings file:

```
ServiceLevel 0
quality 100
bandwidth 100
granularity 50000

ServiceLevel 1
quality 90
bandwidth 10
granularity 50000

ServiceLevel 2
quality 80
bandwidth 5
granularity 50000

Category low

PowerCategory tight
```

7.2 User scenario overview

In this section all the user scenarios are presented both in text and in the shape of a table, Table 7.1. The textual descriptions briefly describe which programs were run during each scenario whereas the table describes the settings used for different runs of the same scenario. The table also contains references to all graphs belonging to each scenario.

All settings vary between different runs except the importance of each program, which is held constant in the *importance file*. Basically the *importance file* declares that SwiFTP has a lower priority than the rest of the applications.

Another general decision made was to keep the time for downloading files constant rather than keeping the downloaded amount of data constant. This de-

cision was based on the fact that focus was on the performance of the Calculator and Missile Intercept.

7.2.1 Scenario 1

In scenario 1, the programs SwiFTP, Missile Intercept and the Calculator are run. SwiFTP is started and a desktop computer will start a repeated series of downloads[1] from the FTP server. Then Missile intercept will run for a while and an *Application Exerciser Monkey* [7] uses the program. When that is done the game will exit and then the Calculator will be run. This program will also be used by an *Application Exerciser Monkey*. After the calculator exits the scenario ends.

This scenario aims at establishing whether the gaming experience is crippled with or without the resource manager.

7.2.2 Scenario 2

In scenario 2, SwiFTP is run alone[2] in the system. SwiFTP is started after approximately one second and an a desktop computer will start a repeated series of downloads from the FTP server. SwiFTP will be running alone in the background during the entire scenario.

This scenario aims at determining how the program acts while being the only one to run.

7.2.3 Scenario 3

In scenario 3, the Calculator is run alone in the system. The program is started after approximately one second and an *Application Exerciser Monkey* uses the program. The Calculator will be running alone during the entire scenario.

This scenario aims at determining how the program acts while being the only one to run.

7.2.4 Scenario 4

In scenario 4, Missile Intercept is run alone in the system. The program is started about one second into the execution and an *Application Exerciser Monkey* uses the program. Missile Intercept will be running alone during the entire scenario.

[1] Repeatedly downloading a 10 Mb file. One could argue that a larger file would have given a better result since it would've resulted in less overhead. However there was not enough free disk space to fit a larger file on the platform.

[2] Alone in the sense that it is the only program started by a user

This scenario aims at determining how the program acts while being the only one to run.

7.2.5 Scenario 5

In scenario 5, the programs SwiFTP and the Calculator are run. SwiFTP is started approximately one second into the execution and a desktop computer will start a repeated series of downloads from the FTP server. Then six seconds into the execution the Calculator will be started. This program will be used by an *Application Exerciser Monkey*. After approximately 21 seconds the Calculator exits and after another second the scenario ends.

This scenario aims at establishing how an application which is mostly idle will work together with a service application which is mostly active.[6]

7.2.6 Scenario 6

In scenario 6, the programs SwiFTP and Missile Intercept are run. SwiFTP is started about one second into the execution and a desktop computer will start a repeated series of downloads from the FTP server. Then five seconds into the execution Missile Intercept will be started. This program will be used by an *Application Exerciser Monkey*. After approximately 33 seconds the scenario ends.

This scenario aims at establishing how an application which is mostly active will work together with a service application which is mostly active.[6]

7.2.7 The scenario table

As mentioned before the scenarios have all been listed in Table 7.1 below. The table is quite self-explanatory but the columns might need a couple of comments.

- **User scenario** - Contains [number of the scenario].[number of the run]
- **With RM** - *Yes* means the run was performed on a MOP500 with the ACTORS RM running. *No* means the ACTORS RM was not running and that the system was an as standard Android version as possible running on a MOP500.
- **Application Settings** - Describes which program had what setting from Section 7.1.
- **Graphs** - Contains references to all graphs belonging to that particular run.

Table 7.1: Complete list of user scenarios and their relevant settings

User scenario	With RM	Application Settings	Graphs
1.1	No	-	fig. B.5, B.2, B.4, B.1, B.3
1.2	Yes	SwiFTP=tools dynamic MissileIntercept=game Calculator=tools static	B.11, B.10, B.8, B.6, B.12, B.13, B.9, B.7
1.3	Yes	SwiFTP=tools dynamic MissileIntercept=tools dynamic Calculator=tools dynamic	B.19, B.18, B.16, B.14, B.20, B.21, B.17, B.15
1.4	Yes	SwiFTP=tools dynamic MissileIntercept=game Calculator=tools static	B.27, B.26, B.24, B.22, B.28, B.29, B.25, B.23
1.5	Yes	SwiFTP=tools dynamic MissileIntercept=tools dynamic Calculator=tools dynamic	B.35, B.34, B.32, B.30, B.36, B.37, B.33, B.31
2.1	No	-	fig. B.42, B.39, B.41, B.38, B.40
2.2	Yes	SwiFTP=tools dynamic	B.43, B.45, B.46, B.47, B.48, B.44
2.3	Yes	SwiFTP=tools static	B.49, B.51, B.52, B.53, B.54, B.50
2.4	Yes	SwiFTP=game	B.55, B.57, B.58, B.59, B.60, B.56
3.1	No	-	fig. B.65, B.62, B.64, B.61, B.63
3.2	Yes	Calculator=tools dynamic	B.66, B.68, B.69, B.70, B.71, B.67
3.3	Yes	Calculator=tools static	B.72, B.74, B.75, B.76, B.77, B.73
3.4	Yes	Calculator=game	B.78, B.80, B.81, B.82, B.83, B.79
4.1	No	-	fig. B.88, B.85, B.87, B.84, B.86
4.2	Yes	Missile Intercept=tools dynamic	B.89, B.91, B.92, B.93, B.94, B.90
4.3	Yes	Missile Intercept=tools static	B.95, B.97, B.98, B.99, B.100, B.96
4.4	Yes	Missile Intercept=game	B.101, B.103, B.104, B.105, B.106, B.102
5.1	No	-	fig. B.111, B.108, B.110, B.107, B.109
5.2	Yes	SwiFTP=tools dynamic Calculator=tools dynamic	B.112, B.114, B.115, B.116, B.117, B.118, B.113

Continued on next page

Table 7.1 – continued from previous page

User scenario	With RM	Application Settings	Graphs
5.3	Yes	SwiFTP=tools dynamic Calculator=game	B.119, B.121, B.122, B.123, B.124, B.125, B.120
5.4	Yes	SwiFTP=tools dynamic Calculator=tools dynamic	B.126, B.128, B.129, B.130, B.131, B.132, B.127
5.5	Yes	SwiFTP=tools dynamic Calculator=game	B.133, B.135, B.136, B.137, B.138, B.139, B.134
6.1	No	-	fig. B.144, B.141, B.143, B.140, B.142
6.2	Yes	SwiFTP=tools dynamic Missile Intercept=tools dynamic	B.145, B.147, B.148, B.149, B.150, B.151, B.146
6.3	Yes	SwiFTP=tools dynamic Missile Intercept=game	B.156, B.152, B.154, B.155, B.157, B.158, B.153
6.4	Yes	SwiFTP=tools dynamic Missile Intercept=tools dynamic	B.163, B.159, B.161, B.162, B.164, B.165, B.160
6.5	Yes	SwiFTP=tools dynamic Missile Intercept=game	B.170, B.166, B.168, B.169, B.171, B.172, B.167

7.3 The most interesting user scenarios

Above, all the scenarios and their settings have been listed. In short, scenarios 2-4 contains only one program each which makes them quite empty and not particularly interesting to discuss. However there are some other interesting comparisons that are worth mentioning and these are listed below.

All listed runs below have a couple of rows in common; settings for the applications and values for the two compared runs. The values for “Frames shown“ and “Megabytes transferred“ is the same as the calculated area underneath the graph, shown in Figure 7.1 as the grayed area. Values for “Frames shown“ and “Megabytes transferred“ for runs not listed below can be found in Appendix A.

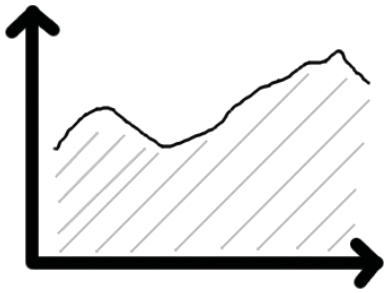

Figure 7.1: Area underneath a graph

The fourth column ”Energy weighted usage“[3] is an approximation of the power consumed by the CPU in each run. It is calculated by the algorithm in Section 3.4 which could also be described by the following pseudo code.

```
for each CPU frequency level :
      1. sum up the total time the CPU has
         run on this frequency.
      2. take the frequency to the power
         of two and multiply that with the
         time from the previous step.
done

the sum of the products received in step two is the
energy weighted usage.
```

The energy weighted usage is an extra value which helps when evaluating the different runs. The relative difference between the energy weighted usage for two runs is a reasonable measurement on how much the energy consumption has increased or decreased.

One last observation that has been made during the different runs is that all of the three programs, Missile Intercept, SwiFTP and the Calculator, seem to act as if they were single threaded. The programs do have multiple threads, but only one of them at a time seem to be consuming a lot of resources.

This single threaded behaviour could e.g. be observed in graph B.94 where it is clear that the program almost only runs on one core at a time, compare the load for CPU 0 and 1. The fact that the programs act as if they were single threaded could result in a program not being able to use all of its assigned resources.

As an example, say that a program is assigned 75 % of the total CPU bandwidth in a system with two cores. Then the reservation would be the green area in Figure 7.2 below. However, if the program is single threaded, it will only be

[3]The energy weighted usage is not shown in the graphs but is based on the CPU-utilization

able to utilize 50 % of the total bandwidth, the striped area of CPU 0, since one thread can only be run on one CPU.

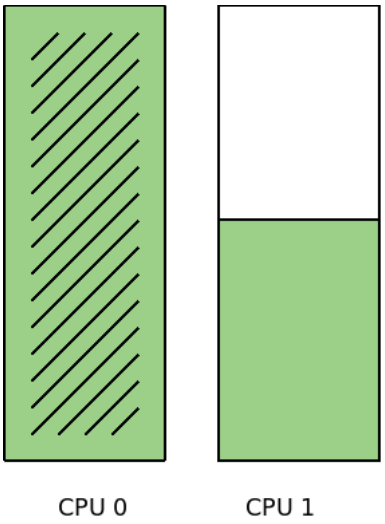

Figure 7.2: Single threaded program on multicore CPU

7.3.1 Run 1.1 to 1.5

This is indeed an interesting scenario. However it consists of a combination of scenario 5 and 6. Therefore the only extra result scenario 1 adds is the fact that the resource manager will switch back to a higher service level if possible after a client has unregistered, see the reservations changes in Figure B.12.

7.3.2 Run 2.1 vs 2.2

Settings - SwiFTP: *tools dynamic*, SwiFTP has no service level implementation

Data type	System CPU	Frames shown	Megabytes transferred	Energy weighted usage
Run 2.1	8181	58	45	4450770
Run 2.2	6294	29	36	2873810
Relative difference	77 %	50 %	80 %	65 %

The run 2.2 has lower SwiFTP data transfer (80 % of the SwiFTP data transfer of run 2.1). There is also a large decrease (50 %) in system frames shown. This is not bad though since there is no vital GUI in this scenario. This can be observed in Figure B.43 and B.39.

The run 2.2 also has a lower CPU-utilization, 77 % of the CPU-utilization of run 2.1 as can be seen in Figure B.46 and B.38. The power consumption in 2.2 is estimated at 65 % of that in 2.1 which is significant decrease.

These changes are expected with the tools dynamic settings.

7.3.3 Run 2.1 vs 2.3

Settings - SwiFTP: *tools static*, SwiFTP has no service level implementation

Data type	System CPU	Frames shown	Megabytes transferred	Energy weighted usage
Run 2.1	8181	58	45	4450770
Run 2.3	4515	31	27	1569636
Relative difference	55 %	54 %	61 %	35 %

As seen in Figures B.38 and B.52, the resource manager does a good job keeping down the CPU-utilization for SwiFTP. The lowering of the power consumption to 35 % in run 2.3 shows that the tools static setting does make a large difference to the overall power consumption of the system. These results are expected with the tools static setting.

However it comes with a price, the SwiFTP data transfer is significantly lower for run 2.3, Figure B.49, about 61 % of the data transfer in run 2.1, Figure B.39.

7.3.4 Run 3.1 vs 3.3

Settings - Calculator: *tools static*

Data type	System CPU	Frames shown	Megabytes transferred	Energy weighted usage
Run 3.1	6336	197	N/A	2695417
Run 3.3	5785	179	N/A	2246371
Relative difference	91 %	91 %	N/A	83 %

As seen in the table and in Figures B.72 and B.62, 91 % of the frames shown in run 3.1 are also shown in 3.3. There is also a decrease of energy consumption to 83 % and a decrease of CPU-utilization to 91 %, see Figures B.61 and B.75.

This is expected with the tools static settings. The gain here is avoiding any high CPU frequencies.

7.3.5 Run 3.1 vs 3.4

Settings - Calculator: *game*

Data type	System CPU	Frames shown	Megabytes transferred	Energy weighted usage
Run 3.1	6336	197	N/A	2695417
Run 3.4	7424	192	N/A	3681547
Relative difference	117 %	98 %	N/A	137 %

The frames shown in run 3.1, B.62, and run 3.4, B.78, are almost the same (98 % in run 3.4). However the CPU-utilization in run 3.4, B.81, is much higher than in run 3.1 (17 % higher), see Figure B.61. The power consumption is also higher, some 37 %. This is easily explained by the *game* application setting which more or less allows the application, in this case the calculator, to use as much power as it possibly could want.

Hence, the *game* setting is not appropriate for this type of application, *activity application which is mostly idle* (as described in Section 6).

7.3.6 Run 4.1 vs 4.2

Settings - Missile Intercept: *tools dynamic*

Data type	System CPU	Frames shown	Megabytes transferred	Energy weighted usage
Run 4.1	13249	485	N/A	6500965
Run 4.2	12709	444	N/A	6092018
Relative difference	96 %	92 %	N/A	94 %

In run 4.2 the CPU-utilization is 96 % of that of run 4.1. (See Figure B.84 and B.92). The frames shown though is only 92 % so that is not really a benefit.

This is due to the fact that Missile Intercept is limited in run 4.2 and that it can run without restrictions in 4.1. There is also a lower margin in 4.2 which prevents the CPU from changing frequency until the load is closer to the current frequency. In 4.1 the frequency will change earlier.

Running with the resource manager, the lower margins are likely to introduce delays to the *SurfaceFlinger* service.

To conclude, no resource manager means a higher margin which means higher frame rate which means better responsiveness.

7.3.7 Run 5.1 vs 5.2

Settings - SwiFTP: *tools dynamic*, Calculator: *tools dynamic*, SwiFTP has no service level implementation

Data type	System CPU	Frames shown	Megabytes transferred	Energy weighted usage
Run 5.1	12212	183	44	6749274
Run 5.2	9814	137	43	5052536
Relative difference	80 %	75 %	98 %	75 %

This comparison is interesting in respect of showing flaws with using the CFS scheduler described in Section 3.3.1. In this scenario a light program, the calculator, is running in the foreground and the SwiFTP application is running in the background. The SwiFTP application has a lower priority in the importance file which is expected to result in a lower priority of the application. However if one would have a look at Figure B.118 it is quite clear that SwiFTP uses way more CPU than expected.

The outcome of run 5.2 could also be explained by the programs being single threaded as discussed in the beginning of this section, Section 7.3.

The share is set to 5 %[4]. If a scheduler with hard reservations had been used, SwiFTP would never have risen above the share. However since CFS is used, which uses soft reservations, SwiFTP is allowed to overrun its reservation. This combined with the program being single threaded, is why the frames shown in run 5.2, B.112, are lower than in run 5.1, B.108.

7.3.8 Run 5.1 vs 5.4

Settings - SwiFTP: *tools dynamic*, Calculator: *tools dynamic*, SwiFTP has service level implementation

Data type	System CPU	Frames shown	Megabytes transferred	Energy weighted usage
Run 5.1	12212	183	44	6749274
Run 5.4	9648	174	36	4881196
Relative difference	79 %	95 %	82 %	72 %

In run 5.4 different service levels have been implemented in SwiFTP. In service level 0 (the default level) no changes have been made to the program. In service level 1 and 2 there have been a *sleep()* added to the thread in charge of transferring data. The sleep length for service level 1 and 2 are 10 and 100 ms respectively between each 64 KB block transfers. SwiFTP will receive a service level 1 change during executing.

When comparing the performance for run 5.1, see Figure B.108, and 5.4, see Figure B.126, it shows that the data transfer of run 5.4 is 82 % of the run 5.1. The CPU-utilization for run 5.4, see Figure B.107, is 79 % of run 5.1, see Figure B.107.

The frames shown however is not decreased that much, the frames shown of run 5.4 is 95 % of the run 5.1.

The service level implementations results in something that could be described as voluntary cooperative multitasking[5] for the FTP server. If SwiFTP would have done a *yield()* instead of a sleep, that would likely have given the same result.

The result of run 5.4 is really good. It lowers the CPU-utilization for the entire system to 79 % compared to not using the resource manager, but it does so with

[4]Service level 2 in setting tools dynamic from Section 7.1

[5]See 3.3.1

only decreasing the frames shown to 95 %. At the same time the weighted energy usage has approximately been decreased by 28 % which is also good. The data transfer however is decreased to 82 % but that is because SwiFTP is not prioritized.

7.3.9 Run 6.1 vs 6.2

Settings - SwiFTP: *tools dynamic*, Missile Intercept: *tools dynamic*, SwiFTP has no service level implementation

Data type	System CPU	Frames shown	Megabytes transferred	Energy weighted usage
Run 6.1	18117	390	69	9933172
Run 6.2	15691	335	67	8258760
Relative difference	87 %	86 %	97 %	83 %

In run 6.2 the difference compared to run 6.1 is that SwiFTP has got a slightly lower data transfer (97 % as good as 6.1), compare B.145 and B.141. Apart from that it should also be noticed that the slacker group, B.147, has a quite high CPU-utilization. This is likely the result of the behaviour explained in Section 4.1.1.

The frames shown is decreased to 86 %. This clearly demonstrates the problem discussed in Section 4.1.1. The threads that handle frame rendering (the threads in the *SurfaceFlinger* service) are put in the slackers cgroup. Ideally, *SurfaceFlinger* would have dedicated threads for each GUI application so that those thread ID:s could be moved into corresponding cgroup, Missile Intercept's cgroup in this case. Then the *SurfaceFlinger* threads would run just as good as the rest of the game and the frames shown would not decrease. The case now is that *SurfaceFlinger* threads has to be put in the slackers cgroup which will get less resources when other cgroups get more.

Overall the CPU-utilization is less in 6.2 (87 % of that in 6.1), which of course is good, see Figures B.148 and B.140. The utilization would however have been even less if the original design in Section 4.1.1 would have been possible. It is at the same time possible, here as well as in Section 7.3.7, that the behaviour can be caused by SwiFTP and Missile Intercept being single threaded. A scheduler using hard reservations could also have made an improvement here.

7.3.10 Run 6.1 vs 6.4

Settings - SwiFTP: *tools dynamic*, Missile Intercept: *tools dynamic*, SwiFTP has service level implementation

Data type	System CPU	Frames shown	Megabytes transferred	Energy weighted usage
Run 6.1	18117	390	69	9933172
Run 6.4	14596	405	47	7431089
Relative difference	81 %	104 %	68 %	75 %

This is one of the best examples there is of the Actors resource manager doing a great work making a real difference to both performance, CPU-utilization and power consumption. The data transfer for SwiFTP in run 6.4, B.163, is 68 % of that in run 6.1, see B.141. But the frames shown are 4 % higher in 6.4 than in 6.1.

The system CPU-utilization in 6.4 is 81 % of the CPU-utilization in 6.1, see Figure B.140 and B.161. On top of that the power consumption is approximately 25 % less than in run 6.1.

In other words the system uses less CPU bandwidth and the power consumption is even lower with the resource manager running than it without it. It also delivers really good performance to the foreground application, Missile Intercept, and acceptable performance to the lower prioritized background app, SwiFTP.

7.3.11 Run 6.1 vs 6.5

Settings - SwiFTP: *tools dynamic*, Missile Intercept: *game*, SwiFTP has service level implementation

Data type	System CPU	Frames shown	Megabytes transferred	Energy weighted usage
Run 6.1	18117	390	69	9933172
Run 6.5	15329	447	23	7991207
Relative difference	85 %	115 %	34 %	80 %

The outcome of run 6.5 is similar to 6.4 but with even more frames shown for the game (increase with 15 % compared to 4 %) and even worse data transfer for SwiFTP, see B.170 compared to B.141 and B.163. The increased value on frames shown is explained by the *game* setting which is also how that setting is expected to affect an application.

The *game* setting for Missile Intercept is also the underlying reason for the decreased data transfer of SwiFTP. Because the reservation (share) for a *game* is so high, B.171, SwiFTP is forced to change into service level 2 which severely limits its possibilities to perform well, B.172.

However, the overall system performance is not all that bad. In fact the total system CPU-utilization in run 6.5, B.168, is 85 % of that in 6.1, B.168. On top of that the power consumption is approximately 20 % less than in 6.1. One might say that the system runs really well but that SwiFTP takes a big battering [6] on behalf of Missile Intercept.

[6] Voluntarily though because of the implemented service levels

Chapter 8
Conclusions

8.1 Summary of achievements

Using the dynamic setting on all applications in combination with low importance value in the importance file for SwiFTP has given the best result. This is reflected both in terms of lowered CPU usage and even more lowered energy weighted usage combined with only a slight decrease in frames shown and data transfered. See e.g. run 5.4 (Section 7.3.8) which has CPU usage: 79 %, weighted energy usage: 72 % frames shown: 95 % and data transfered: 82 %.

If the background task, SwiFTP, is considered to have a lower priority, a performance decrease for its tasks by 18 % is good considering that the weighted energy consumption is 28 % lower. Especially considering that the performance of the more prioritized application, Missile Intercept is nearly the same (95 %).

In many of the successful scenario runs from the result chapter, Section 7.3 the system CPU-utilization has actually been lowered. The weighted energy consumption has in all those cases been lowered even more. This is due to the fact that power consumption increases quadratically with the clock frequency as described in Section 3.4. The power consumption has been lowered but Missile Intercept and Calculator still keeps good performance in terms of frames shown. The data transfer is decreased in all of these cases because of the decreased reservation for SwiFTP.

The results achieved are due to trimming of service levels and importance value. Having this said there still is a lot of work that has to be done with the different categories described in Section 5.3.2 before they can be used in a more general Android system.

From the different runs one can also conclude that when a service level is well chosen, as in run 5.4 and 6.4 where all programs are set to *tools dynamic*, the resource manager is able to do a good job. When the well chosen service levels on top of that is combined with a simple implementation of service level 0, 1 and 2 in SwiFTP, the result is really satisfying.

In run 6.4, the frames shown increases to 104 % and the data transfer decreases to 68 % compared to not using the resource management framework.

Considering that the weighted energy consumption also decreases to 75 %, this must be considered a very good result. Especially if the performance of SwiFTP is secondary.

The CFS has produced some unwanted behavior. This was especially well noticed in run 5.2 and 6.2. The soft reservations used by the CFS cripples the ACTORS RM in such a way that the shares it sets does not actually make a significant difference. This is also the reason why i.e. 5.4 and 6.4 give such a good result compared to 5.2 and 6.2 respectively.

The weak result in run 5.2 and 6.2 is due to the fact that the ACTORS RM is running on a multicore system with CFS. The ACTORS RM was designed primarily for hard reservations. If a thread is running alone on a core it will consume all of the CPU if it needs to. This is thanks to CFS which will allow the program to overrun its reservations since no other thread needs the CPU time.

The quality of service implementation as described in Section 3.3.3 did not actually make any difference. As described in Section 3.3.3, the resource manager should first try to lower the service levels of applications with a high quality of service on its lower service levels, rather than lowering the service levels for applications with low quality of service on its lower service levels.

This is the way quality of service was used in the settings files described in Section 7.1. However, if SwiFTP did not get a low priority in the `actrm.imp` file, it would not change service in run 6.5, see Section 7.3.11. Despite the fact that the game setting has higher quality of service in its service level 1 compared to that of the tools dynamic setting. This was discovered late in the project so there was not enough time to investigate why it did not work as expected.

The integration of changes made to the `Application.java`, `Activity.java` and `Service.java` classes into Android was only partial. The changes were kept in the mid-layer classes `OurApplication.java`, `OurActivity.java` and `OurService.java`.This was a design decision so that there would be a controlled environment. The plan was initially that there would be a full integration made later on so that all Android applications would benefit from the framework. This would however have presented a new set of challenges which were outside the scope of the project. Hence, the full integration was never performed.

8.2 Follow-up of the problem formulation

During the entire thesis project the problem formulation has been the focus. Were the goals stipulated in the problem formulation reached?

A resource reservation manager in Android has been introduced. The system overall power consumption has been lowered. It is now easy for an application developer to use the resource management framework. There are default settings so the minimum a developer has to do is running the script that makes an application ACTORS aware.

Most of the properties described in the problem formulation have been implemented. The numbered list below matches the list in Chapter 2, Problem for-

mulation:

1. Service level implementation is easy.
2. A method has been introduced that can be overridden to give application feedback on how well the application is fulfilling the service level. However, the feedback to the resource manager has not been done automatically and this method was not used when the open source applications were adapted to use the resource management framework.
3. Applications have been given access to reserve processing time in order to meet their requirements.
4. System services used by an application were not included in the service level handling of the framework. This was too complex and is described in Section 4.1.1. However, services that are a part of the application are included in the service level handling.
5. Default handling that is suitable for many applications has been developed based on the categories found in the Android Market. However, no default service level implementation has been developed.
6. The usage of the resource management framework has been demonstrated by adopting three existing open source Android applications of different types.

8.3 Future work

Below is a list of bullets that might be of interest to investigate further in future work.

- It would be interesting to further investigate how much I/O effects the result from the user scenarios. The SwiFTP application might be limited not only by CPU utilization but also by I/O since it does a lot of reading from the file system and sending data on the network.
- More work could be done with the importance file. I.e. a service should not necessarily have the same importance value as an activity, even if they belong to the same application.
- Investigate if the work from this masters thesis has anything to gain from being combined with the resent Linux kernel patch by Mike Galbraith [48].
- If the ACTORS RM were to become a native part of Android it is crucial to investigate what changes had to be done to make the resource manager aware of all system processes.
- Investigaste how peripherals, such as Wi-Fi, affect energy consumption. Peripherals might consume a significant amount of the system's resources.

8.4 Final words from the authors

During this masters thesis project we have successfully carried out a number of both trivial and non-trivial tasks. A resource management framework for Android devices has been created both with brand new parts and with some borrowed and modified parts. If we were to describe the framework with a few lines this would be them:

- Porting of ACTORS RM to Android - Done
- Replacing D-Bus with Binder to increase speed - Done
- Adapting open source applications to the ACTORS RM - Done and it is really easy
- Maintaining, even improving, performance of test applications while lowering CPU-utilization and power consumption - Done

Do we believe the project has been successful? Yes! Of course there is more that *can* be done. And if our work is to be used in industry, of course there is more that *should* be done. However we are really satisfied with the result an hope it will come to good use in the future.

Bibliography

[1] *ACTORS* [2011a]. Adaptivity and Control of Resources in Embedded Systems (d3b).
URL: *http://www.control.lth.se/user/karlerik/Actors/M24/d3b-main.pdf*

[2] *ACTORS* [2011b]. Adaptivity and Control of Resources in Embedded Systems (d3c).
URL: *http://www.control.lth.se/user/karlerik/Actors/M24/d3c.pdf*

[3] *ACTORS* [2011c]. Actors website.
URL: *http://www.actors-project.eu*

[4] *Actors EU-project* [2011]. Official website for the Actors EU-project.
URL: *http://www.actors-project.eu*

[5] *ACTORS project* [2011a]. ACTORS deliverable d3b.
URL: *http://www.control.lth.se/user/karlerik/Actors/M24/d3b-main.pdf*

[6] *ACTORS project* [2011b]. ACTORS deliverable d3c.
URL: *http://www.control.lth.se/user/karlerik/Actors/M24/d3c.pdf*

[7] *Android Application Exerciser Monkey* [2011].
URL: *http://developer.android.com/guide/developing/tools/monkey.html*

[8] *Android at kernel.org* [2011]. Calculator source code.
URL: *http://android.git.kernel.org/?p=platform/packages/apps/Calculator.git;a=summary*

[9] *Android Open Source Project* [2011a]. Announcement of Android by Google and Open Handset Alliance.
URL: *http://source.android.com*

[10] *Android Open Source Project* [2011b]. Android fundamentals.
URL: *http://developer.android.com/guide/topics/fundamentals.html*

[11] *Android Open Source Project* [2011c]. Android fundamentals - Component lifecycles.
URL: *http://developer.android.com/guide/topics/fundamentals.html#lcycles*

[12] *Android OS port for BeagleBoard* [2011].
URL: *http://labs.embinux.org/index.php/Android_Porting_Guide_to_Beagle_Board*

[13] *Android-porting discussion* [2011a]. How to connect to a binder c++ service?
URL: *http://www.mail-archive.com/android-porting@googlegroups.com/msg12714.html*

[14] *Android-porting discussion* [2011b]. How to connect to a binder c++ service?
URL: *http://www.mail-archive.com/android-porting@googlegroups.com/msg12731.html*

[15] *Android-porting discussion* [2011c]. How to connect to a binder c++ service?
URL: *http://www.mail-archive.com/android-porting@googlegroups.com/msg12745.html*

[16] *Android Power Management* [2011].
URL: *http://source.android.com/porting/power_management.html*

[17] *Android software stack* [2011].
URL: *http://developer.android.com/guide/basics/what-is-android.html*

[18] *A version of SurfaceFlinger.cpp which contains code for calculating and showing fps* [2011].
URL: *http://www.netmite.com/android/mydroid/frameworks/base/libs/surfaceflinger/SurfaceFlinger.cpp*

[19] *cgroups on Wikipedia* [2011].
URL: *http://en.wikipedia.org/wiki/Cgroups*

[20] *Completely Fair Scheduler on Wikipedia* [2011].
URL: *http://en.wikipedia.org/wiki/Completely_Fair_Scheduler*

[21] *Department of Automatic Control* [2011]. Adaptivity and Control of Resources in Embedded Systems (ACTORS).
URL: *http://www.control.lth.se/project/ACTORS*

[22] *Dianne Hackborn explaining Binder Thread* [2011].
URL: *http://groups.google.com/group/android-developers/browse_thread/thread/5d8059d5dfcba2f8*

[23] *Dianne Hackborn's homepage* [2011]. Presentation.
URL: *http://www.angryredplanet.com/ hackbod*

[24] *Github - Project Android-HelloWorldService* [2011]. Binder service and client in C++.
URL: *https://github.com/mcr/Android-HelloWorldService*

[25] *GNU Linear Programming Kit (GLPK)* [2011]. Website for GLPK.
URL: *http://www.gnu.org/software/glpk/*

[26] *Google code* [2011a]. NetHack for Android.
URL: *http://code.google.com/p/nethack-android/*

[27] *Google code* [2011b]. SwiFTP.
URL: *http://code.google.com/p/swiftp*

[28] *Google code* [2011c]. NoiseAlert.
URL: *http://code.google.com/p/android-labs/wiki/NoiseAlert*

[29] *IBM website* [2011]. Reduce Linux power consumption, Part 1: The CPUfreq subsystem.
URL: *http://www.ibm.com/developerworks/linux/library/l-cpufreq-1*

[30] *Java website* [2011]. Java Native Interface: Programmer's Guide and Specification.
URL: *http://java.sun.com/docs/books/jni/html/intro.html#1811*

[31] *Linux kernel archives* [2011]. Linux kernel documentation - CPUfreq governors.
URL: *http://www.kernel.org/doc/Documentation/cpu-freq/governors.txt*

[32] Love, R. [2010a]. *Linux Kernel Development - A thorough guide to the design and implementation of the Linux kernel*, 3rd edn, Addison-Wesley, chapter 3, pp. 23–24.

[33] Love, R. [2010b]. *Linux Kernel Development - A thorough guide to the design and implementation of the Linux kernel*, 3rd edn, Addison-Wesley, chapter 4, pp. 41–50.

[34] *lwn.net article on CFS* [2011].
URL: *http://lwn.net/Articles/230574/*

[35] *Missile Intercept* [2011]. Missile Intercept website.
URL: *http://www.kirit.com/Missile%20intercept*

[36] *O(1) scheduler* [2011].
URL: *http://en.wikipedia.org/wiki/O(1)_scheduler*

[37] *Official website for the Android open source project* [2011].
URL: *http://source.android.com*

[38] *onCreate() for a service* [2011a].
URL: *http://developer.android.com/reference/android/app/Service.html*

[39] *onCreate() for a service* [2011b].
URL: *http://developer.android.com/reference/android/app/Activity.html*

[40] *OpenBinder website* [2011a]. OpenBinder website first page.
URL: *http://www.angryredplanet.com/ hackbod/openbinder/docs/html/index.html*

[41] *OpenBinder website* [2011b]. Binder IPC Mechanism.
URL: *http://www.angryredplanet.com/ hackbod/openbinder/docs/html/BinderIPCMechanism.html*

[42] *OS News* [2011]. Introduction to OpenBinder and Interview with Dianne Hackborn.
URL: *http://www.osnews.com/story/13674/Introduction-to-OpenBinder-and-Interview-with-Dianne-Hackborn*

[43] Rabaey, J. [2009a]. *Low power design essentials*, Springer, chapter 1, p. 16.

[44] Rabaey, J. [2009b]. *Low power design essentials*, Springer, chapter 10, p. 254.

[45] Rabaey, J. [2009c]. *Low power design essentials*, Springer, chapter 1, p. 7.

[46] *RTOS on BeagleBoard* [2011].
URL: *http://www.qnx.com/products/reference-design/ti-reference-design.html*

[47] *Samsung website* [2011]. Samsung Xcover E2370 product page.
URL: *http://www.samsung.com/se/consumer/mobile/mobilephones/mobilephones/GT-E2370FSAXEE/index.idx?pagetype=prd_detail*

[48] *sched: automated per tty task groups* [2011].
URL: *https://lkml.org/lkml/2010/10/19/123*

[49] *SCHED_DEADLINE* [2011].
URL: *http://www.evidence.eu.com/content/view/313/390/*

[50] *SCHED_EDF* [2011].
URL: *http://www.actors-project.eu/index.php?page=sched_edf-dissemination*

[51] *SCHED_EDF on Ericsson Labs* [2011].
URL: *https://labs.ericsson.com/developer-community/blog/making-linux-more-real-time*

[52] *Specification of the Emulator included in Android SDK* [2011].
URL: *http://developer.android.com/guide/developing/tools/emulator.html*

[53] *Specification of the (Original) BeagleBoard.* [2011].
URL: *http://beagleboard.org/hardware*

[54] *Specification of the ST-Ericsson u8500 chipset* [2011].
URL: *http://www.malideveloper.com/developer-resources/development-boards/st-e-mop500-development-platform.php*

[55] *Specification of the ST - Ericsson U8500 smartphone platform also known as the MOP500.* [2011].
URL: *http://www.stericsson.com/platforms/U8500.jsp*

[56] *Specification of the Texas Instruments OMAP3530 Processor* [2011].
URL: *http://en.wikipedia.org/wiki/Texas_Instruments_OMAP*

[57] *The Linux Foundation* [2011].
URL: *http://www.linuxfoundation.org/*

[58] *Wikipedia* [2011a]. Feature phone article.
URL: *http://en.wikipedia.org/wiki/Feature_phone*

[59] *Wikipedia* [2011b]. Wikipedia article on NetHack.
URL: *http://en.wikipedia.org/wiki/NetHack*

[60] *Wikipedia* [2011c]. Missile Command article.
URL: *http://en.wikipedia.org/wiki/Missile_command*

[61] *Wikipedia definition of Linux* [2011].
URL: *http://en.wikipedia.org/wiki/Linux*

[62] *Wikipedia specification of the (Original) BeagleBoard* [2011].
URL: *http://en.wikipedia.org/wiki/BeagleBoard*

[63] *Windows CE port for BeagleBoard* [2011].
URL: *http://www.mpcdata.com/ti-beagleboard-windows-embedded-ce-bsp/*

Chapter 9
Acronyms

List of Acronyms

CPU Central Processing Unit

IPC Inter Process Communication

ACTORS RM ACTORS Resource Manager

SDK Software Development Kit

IDE Integrated Development Environment

GPL GNU General Public License

GUI Graphical User Interface

FTP File Transfer Protocol

CFS Completely Fair Scheduler

EDF Earliest Deadline First

ACTORS Adaptivity and Control of Resources in Embedded Systems

ms milliseconds

GTK GIMP Toolkit

ADT Android Development Tools

Appendix A

Areas

Integral values for all runs

1.1 - FPS area: 555
1.1 - BPS area: 195
1.1 - CPU area: 30704
1.1 - CPU energy weighted usage: 16703799

==========

1.2 - FPS area: 516
1.2 - BPS area: 159
1.2 - CPU area: 24609
1.2 - CPU energy weighted usage: 12607619

==========

1.3 - FPS area: 531
1.3 - BPS area: 149
1.3 - CPU area: 24430
1.3 - CPU energy weighted usage: 11935796

==========

1.4 - FPS area: 607
1.4 - BPS area: 84
1.4 - CPU area: 26255
1.4 - CPU energy weighted usage: 13504353

==========

1.5 - FPS area: 646
1.5 - BPS area: 117
1.5 - CPU area: 26282
1.5 - CPU energy weighted usage: 13687137

==========

2.1 - FPS area: 58
2.1 - BPS area: 45
2.1 - CPU area: 8181
2.1 - CPU energy weighted usage: 4450770

==========

2.2 - FPS area: 29
2.2 - BPS area: 36
2.2 - CPU area: 6294
2.2 - CPU energy weighted usage: 2873810

==========

2.3 - FPS area: 31
2.3 - BPS area: 27
2.3 - CPU area: 4515
2.3 - CPU energy weighted usage: 1569636

==========

2.4 - FPS area: 33
2.4 - BPS area: 41
2.4 - CPU area: 7168
2.4 - CPU energy weighted usage: 3636130

==========

3.1 - FPS area: 197
3.1 - BPS area: None
3.1 - CPU area: 6336
3.1 - CPU energy weighted usage: 2695417

==========

3.2 - FPS area: 186
3.2 - BPS area: None
3.2 - CPU area: 6392
3.2 - CPU energy weighted usage: 2787406

==========

3.3 - FPS area: 179
3.3 - BPS area: None
3.3 - CPU area: 5785
3.3 - CPU energy weighted usage: 2246371

==========

3.4 - FPS area: 192
3.4 - BPS area: None
3.4 - CPU area: 7424
3.4 - CPU energy weighted usage: 3681547

==========

4.1 - FPS area: 485
4.1 - BPS area: None
4.1 - CPU area: 13249
4.1 - CPU energy weighted usage: 6500965

==========

4.2 - FPS area: 444
4.2 - BPS area: None
4.2 - CPU area: 12709
4.2 - CPU energy weighted usage: 6092018

==========

4.3 - FPS area: 454
4.3 - BPS area: None
4.3 - CPU area: 11301
4.3 - CPU energy weighted usage: 4908311

==========

4.4 - FPS area: 468
4.4 - BPS area: None
4.4 - CPU area: 14321
4.4 - CPU energy weighted usage: 7608210

==========

5.1 - FPS area: 183
5.1 - BPS area: 44
5.1 - CPU area: 12212
5.1 - CPU energy weighted usage: 6749274

==========

5.2 - FPS area: 137
5.2 - BPS area: 43
5.2 - CPU area: 9814
5.2 - CPU energy weighted usage: 5052536

==========

5.3 - FPS area: 152
5.3 - BPS area: 44
5.3 - CPU area: 9925
5.3 - CPU energy weighted usage: 5208862

==========

5.4 - FPS area: 174
5.4 - BPS area: 36
5.4 - CPU area: 9648
5.4 - CPU energy weighted usage: 4881196

==========

5.5 - FPS area: 207
5.5 - BPS area: 24
5.5 - CPU area: 9669
5.5 - CPU energy weighted usage: 4977775

==========

6.1 - FPS area: 390
6.1 - BPS area: 69
6.1 - CPU area: 18117
6.1 - CPU energy weighted usage: 9933172

==========

6.2 - FPS area: 335
6.2 - BPS area: 67
6.2 - CPU area: 15691
6.2 - CPU energy weighted usage: 8258760

==========

6.3 - FPS area: 367
6.3 - BPS area: 63
6.3 - CPU area: 16176
6.3 - CPU energy weighted usage: 8678504

==========

6.4 - FPS area: 405
6.4 - BPS area: 47
6.4 - CPU area: 14596
6.4 - CPU energy weighted usage: 7431089

==========

6.5 - FPS area: 447
6.5 - BPS area: 23
6.5 - CPU area: 15329
6.5 - CPU energy weighted usage: 7991207

==========

Appendix B
Graphs

In this chapter all graphs from all the runs of all the six scenarios can be found. Each one of them has a caption built up by a graph name and a run number. An example could be *System, run 1.1*. Below all the graph names are briefly explained.

The data for all the graphs is collected by a user-space application, with one exception. The exception is the System graphs and the recurring red-colored CPU frequency where the data is collected directly from the Linux kernel. The user-space collection of the rest of the data might result in a slight drift of these values in time compared to the System and/or the CPU frequency graph.

Graphs generated for runs both with and without the ACTORS Resource Manager

- **System** - A graph with this name shows the total CPU-utilization of the system
- **FPS, BPS** - These graphs show the system's frame rate in frames per second (FPS). If applicable it also shows the transfer speed of SwiFTP in Megabytes per second.
- **Root** - Root means the root cgroup whether it is `/dev/cgroup` or `/dec/cputl`.

Graphs generated for runs only without the ACTORS Resource Manager

- **fg_boost** - This graph shows CPU-utilization for tasks in the fg_boost cgroup
- **bg_non_interactive** - This graph shows CPU-utilization for tasks in the bg_non_interactive

Graphs generated for runs only with the ACTORS Resource Manager

- **Slacker and Self** - This graph shows CPU-utilization for tasks in the slacker or self cgroup respectively. These cgroups are both part of the ACTORS Resource Manager default setup
- **Calculator, SwiFTP or and Missile Intercept** - Graphs with either of these three names show the CPU-utilization for the corresponding program

B.1 User scenario 1

See Section 7.2.1 for a description of this scenario.

B.1.1 Run 1.1

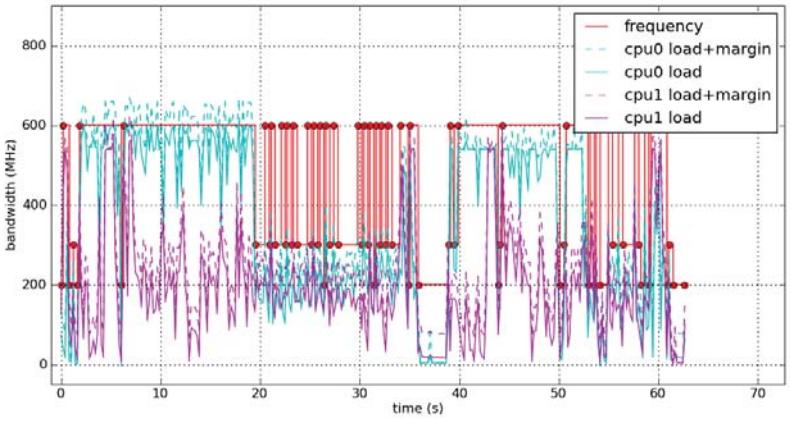

Figure B.1: System, run 1.1

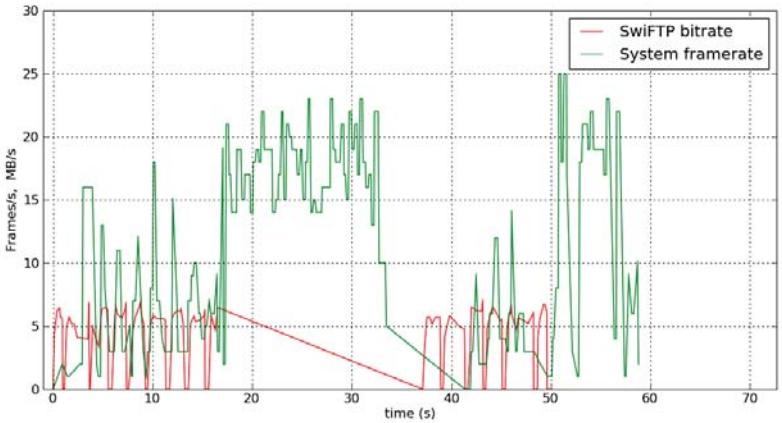

Figure B.2: FPS, BPS, run 1.1

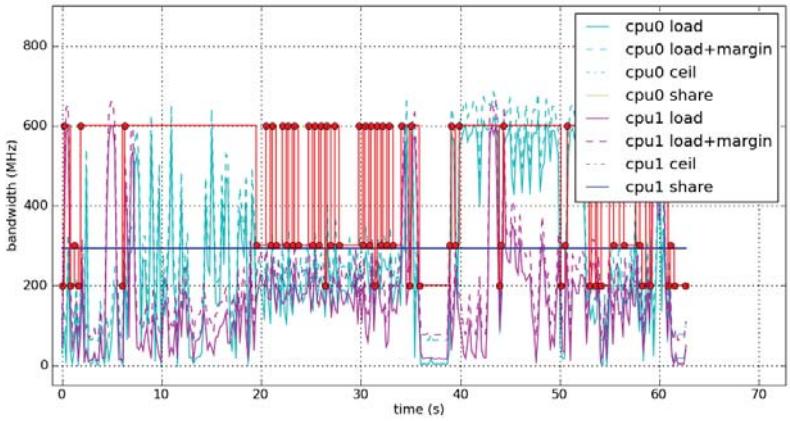

Figure B.3: Root, run 1.1

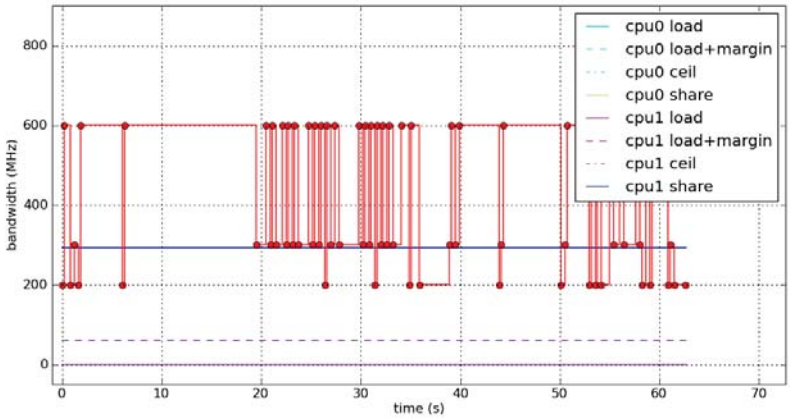

Figure B.4: fg_boost, run 1.1

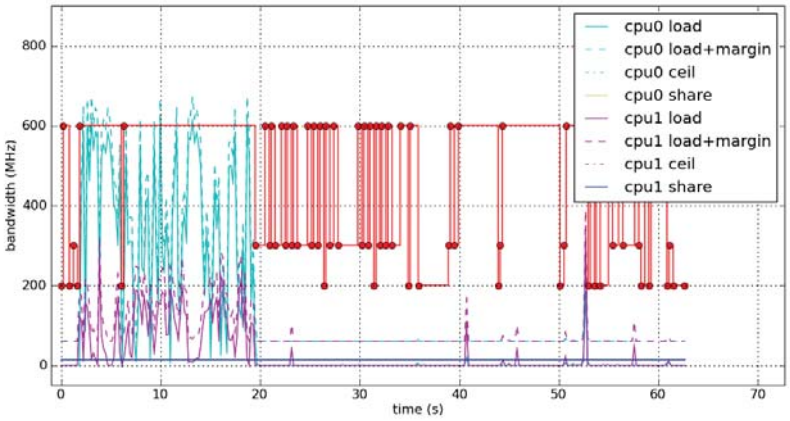

Figure B.5: bg_non_interactive, run 1.1

B.1.2 Run 1.2

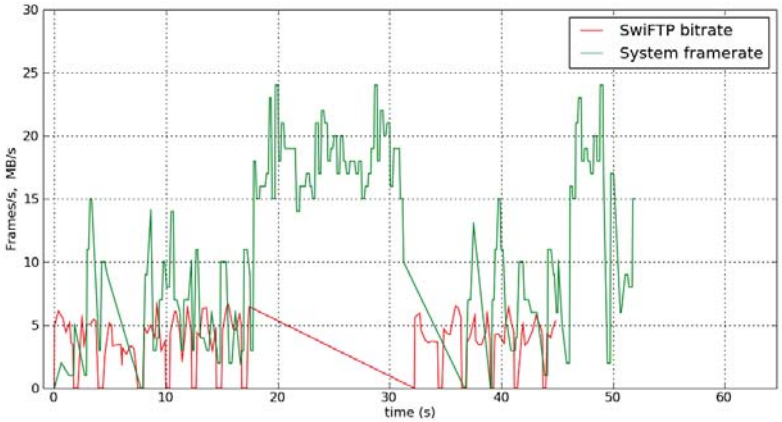

Figure B.6: FPS, BPS, run 1.2

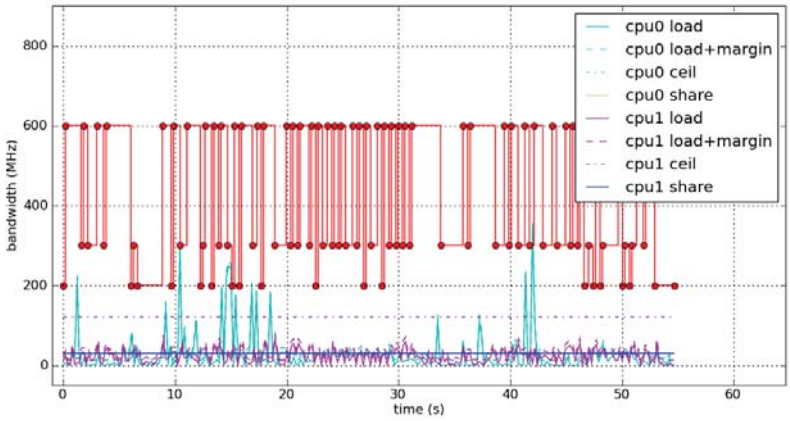

Figure B.7: Root, run 1.2

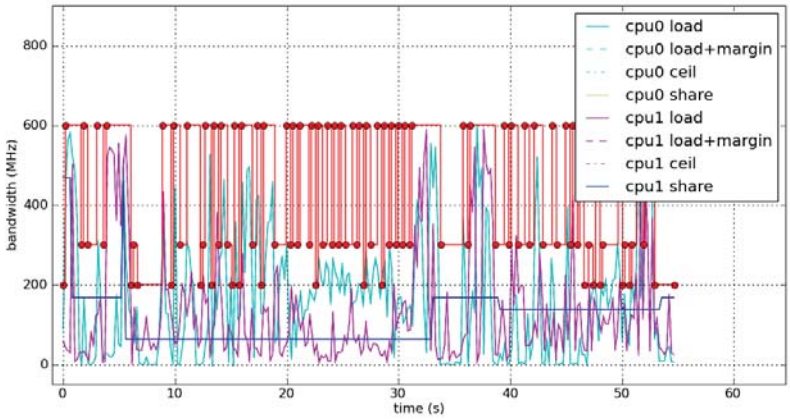

Figure B.8: Slacker, run 1.2

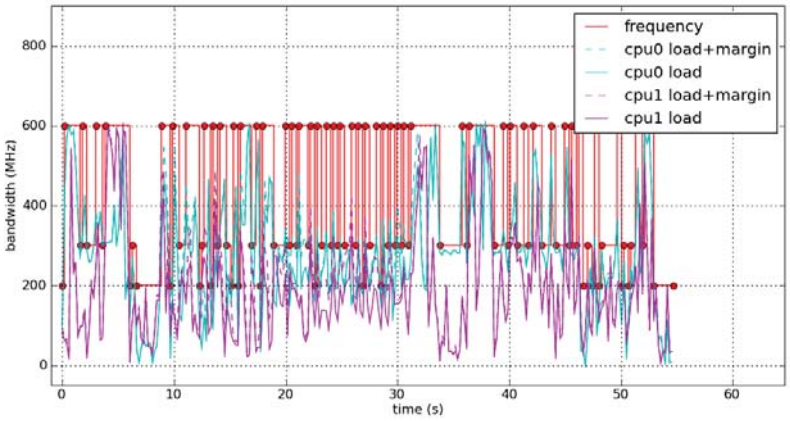

Figure B.9: System, run 1.2

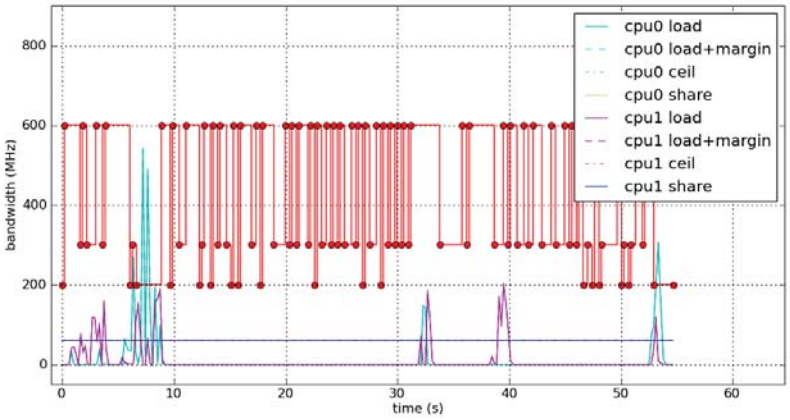

Figure B.10: Self, run 1.2

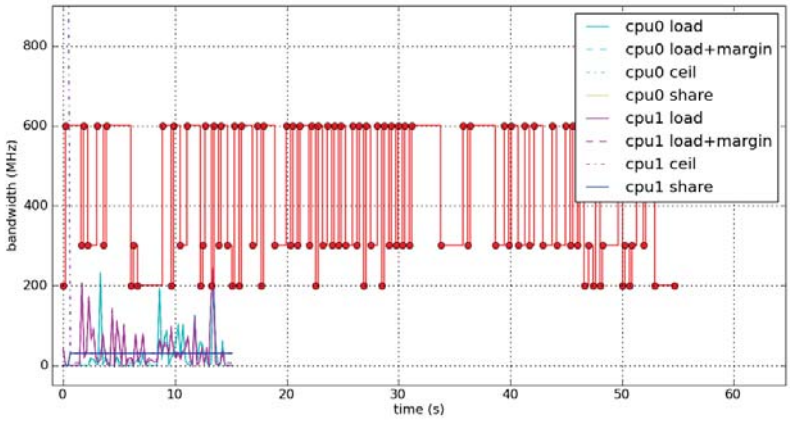

Figure B.11: Calculator, run 1.2

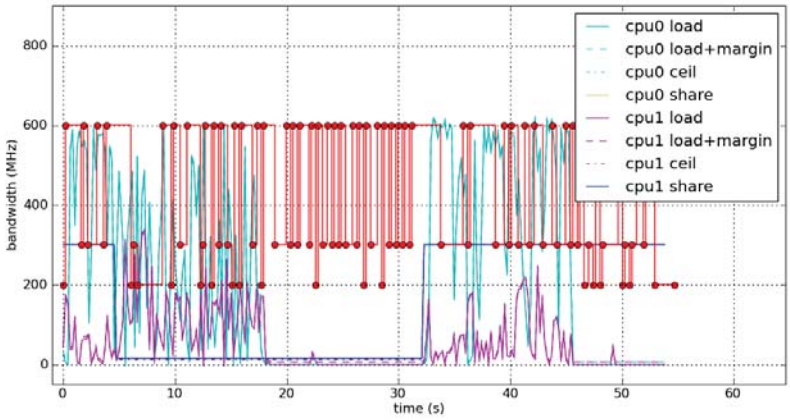

Figure B.12: SwiFTP, run 1.2

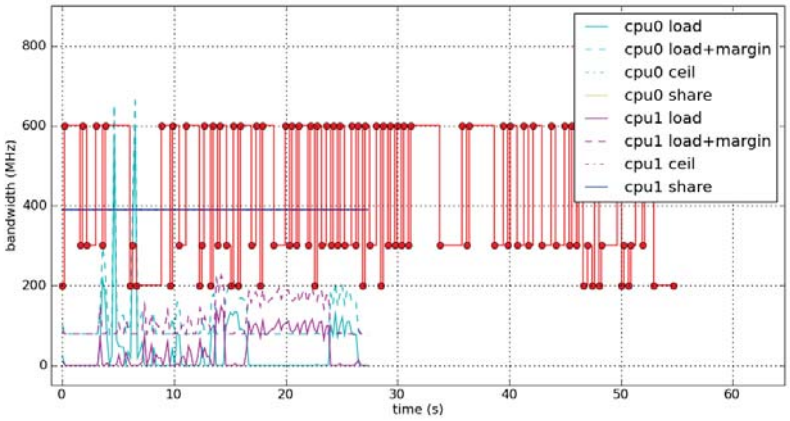

Figure B.13: Missile Intercept, run 1.2

B.1.3 Run 1.3

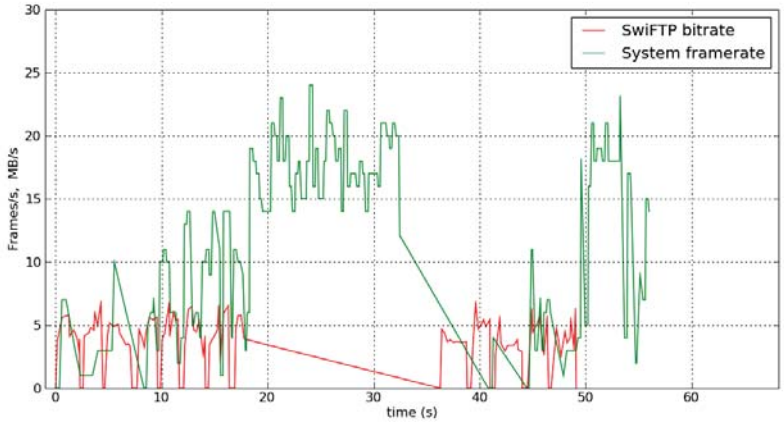

Figure B.14: FPS, BPS, run 1.3

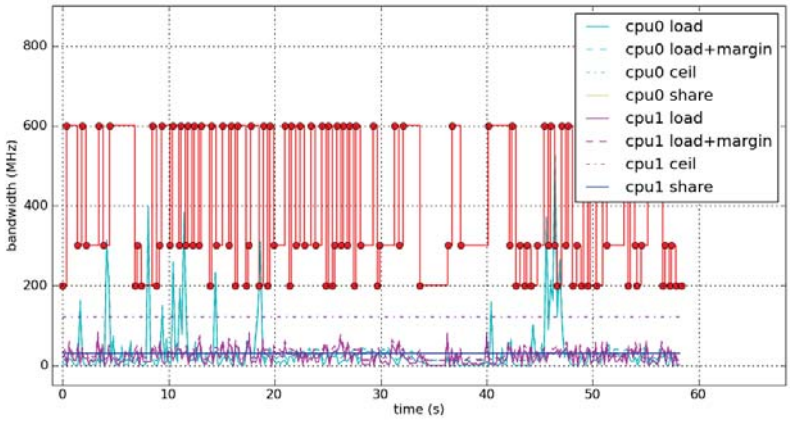

Figure B.15: Root, run 1.3

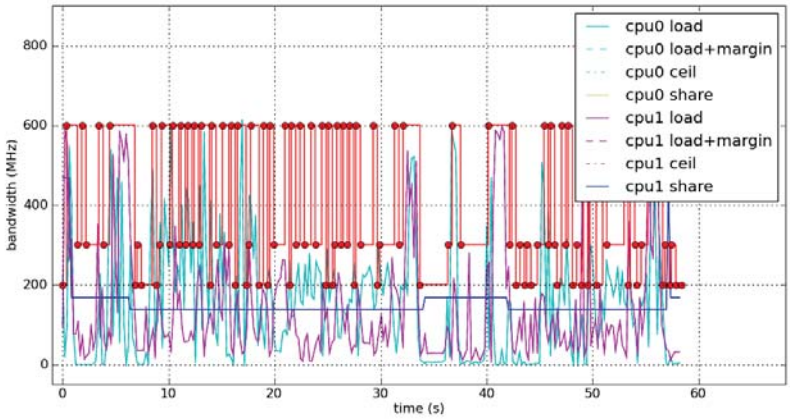

Figure B.16: Slacker, run 1.3

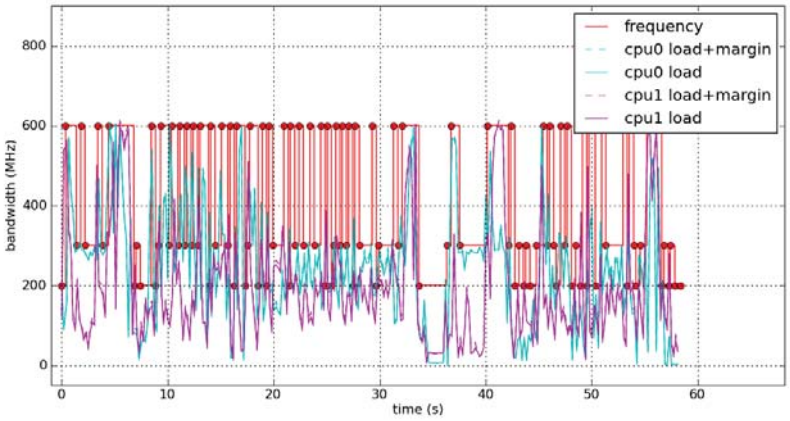

Figure B.17: System, run 1.3

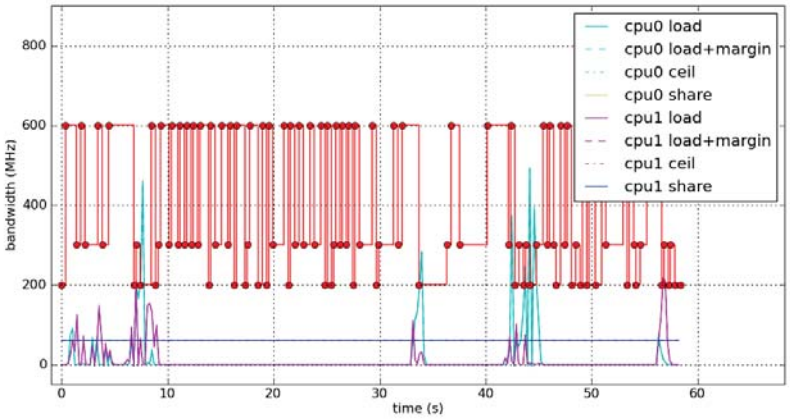

Figure B.18: Self, run 1.3

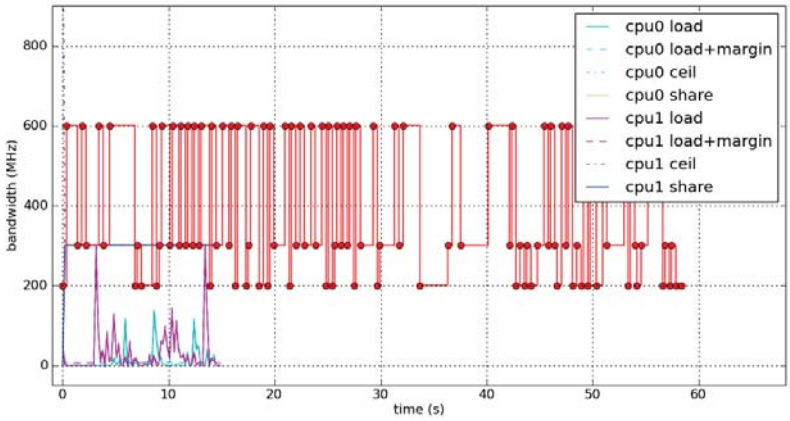

Figure B.19: Calculator, run 1.3

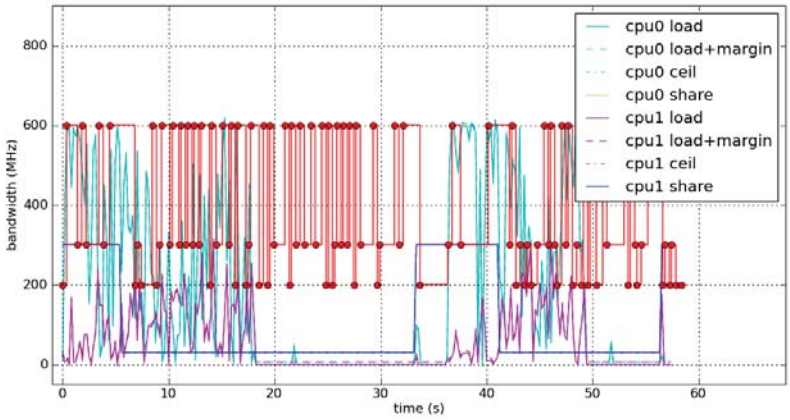

Figure B.20: SwiFTP, run 1.3

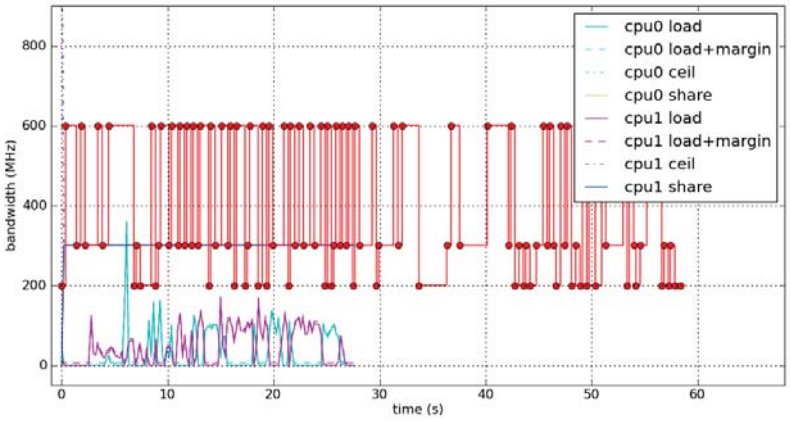

Figure B.21: Missile Intercept, run 1.3

B.1.4 Run 1.4

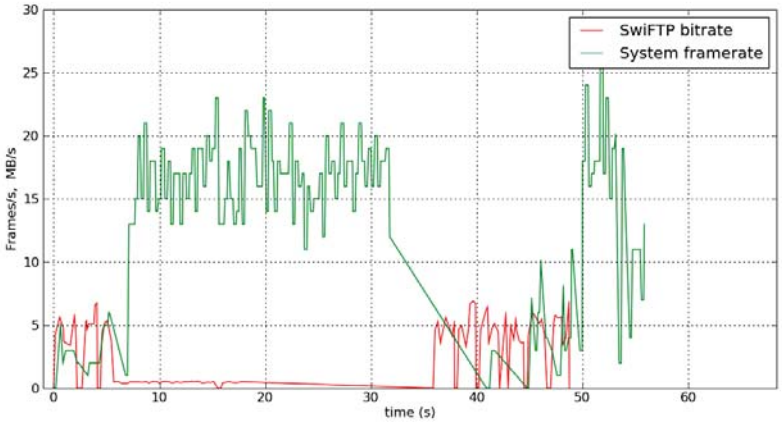

Figure B.22: FPS, BPS, run 1.4

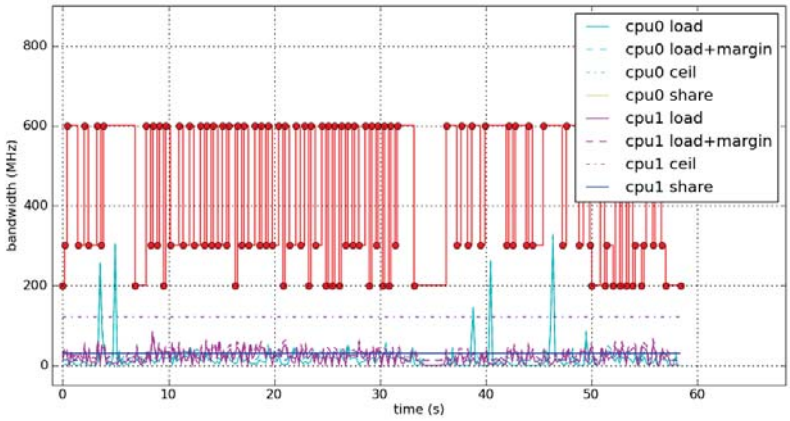

Figure B.23: Root, run 1.4

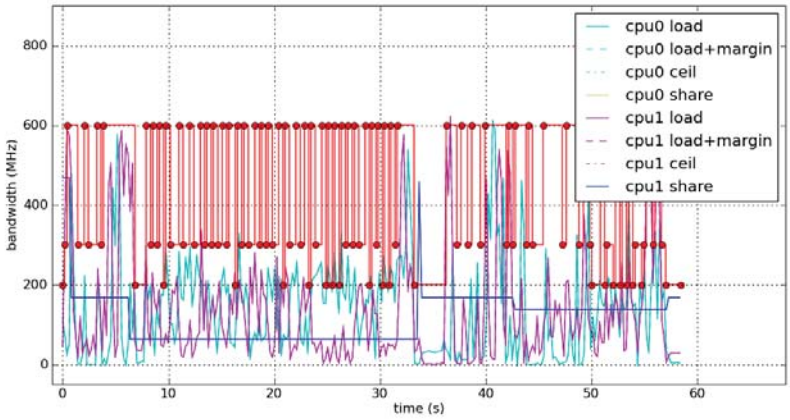

Figure B.24: Slacker, run 1.4

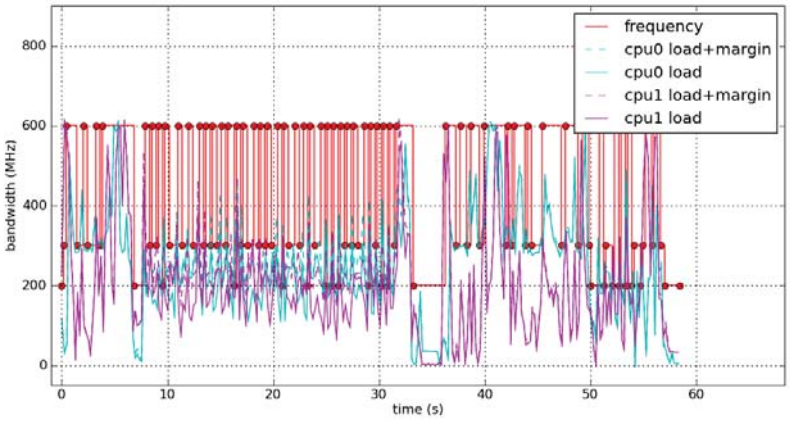

Figure B.25: System, run 1.4

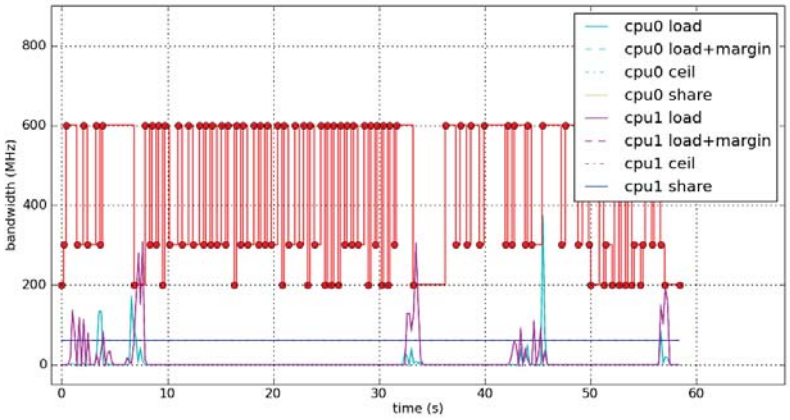

Figure B.26: Self, run 1.4

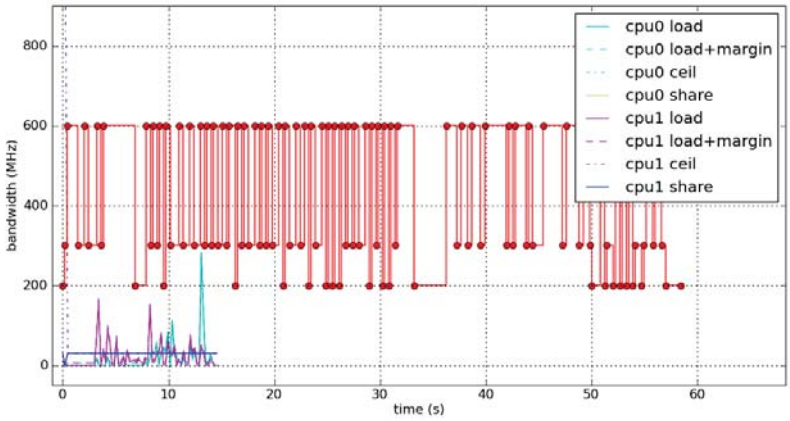

Figure B.27: Calculator, run 1.4

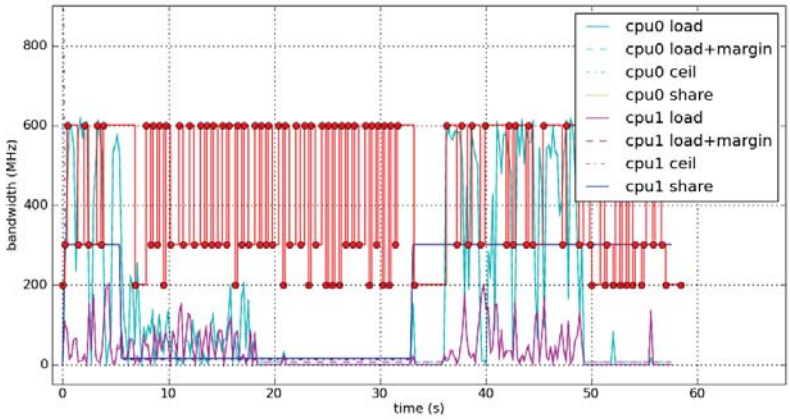

Figure B.28: SwiFTP, run 1.4

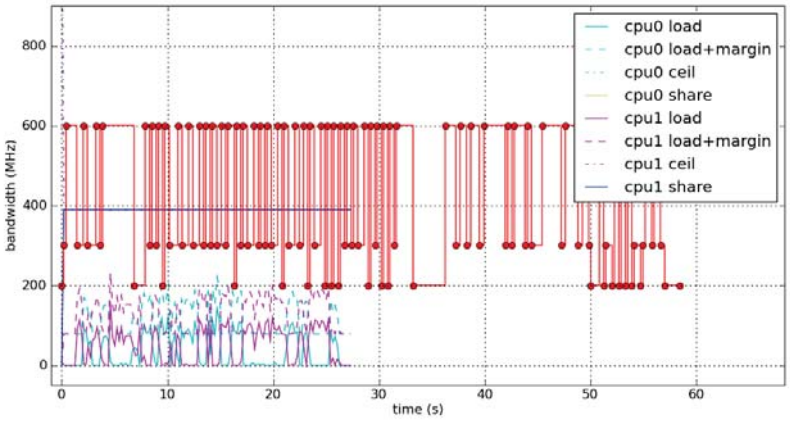

Figure B.29: Missile Intercept, run 1.4

B.1.5 Run 1.5

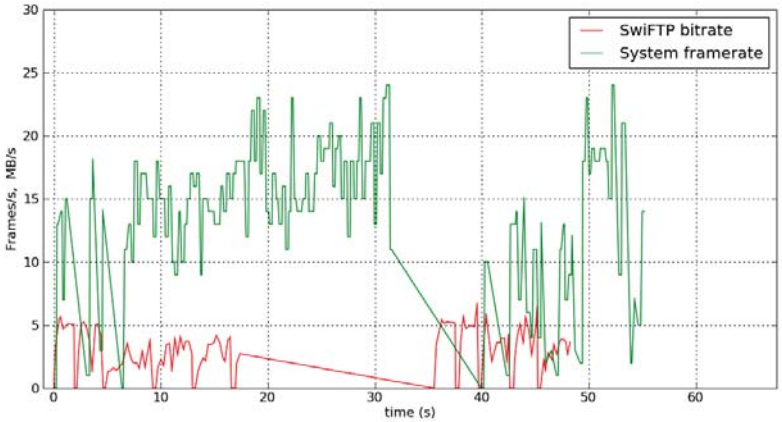

Figure B.30: FPS, BPS, run 1.5

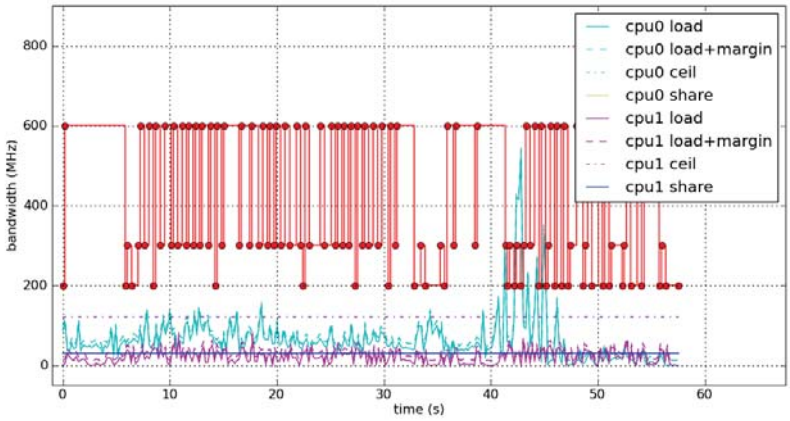

Figure B.31: Root, run 1.5

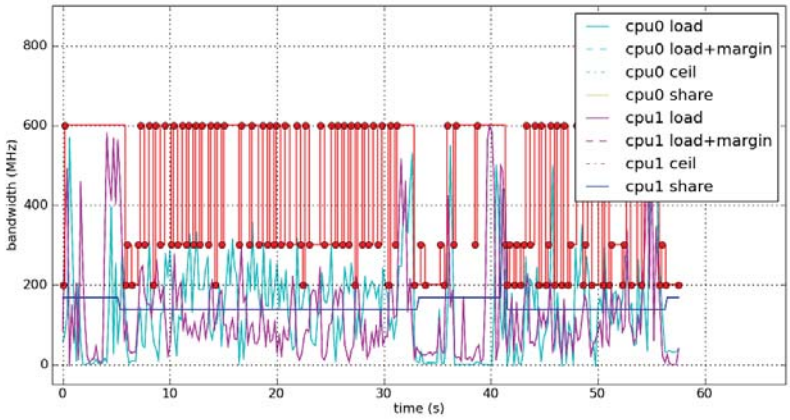

Figure B.32: Slacker, run 1.5

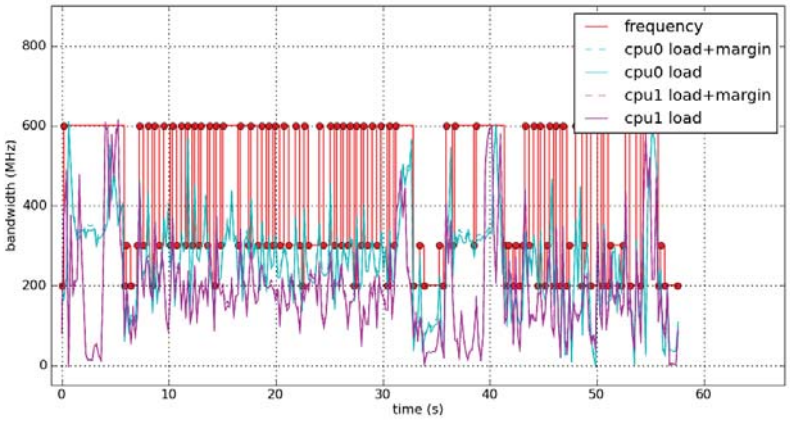

Figure B.33: System, run 1.5

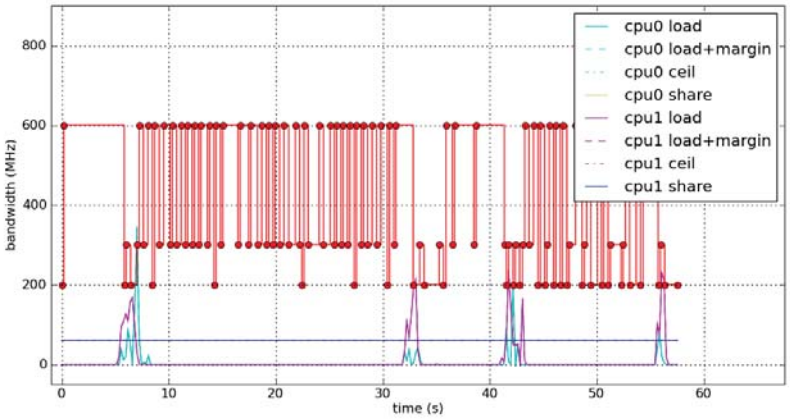

Figure B.34: Self, run 1.5

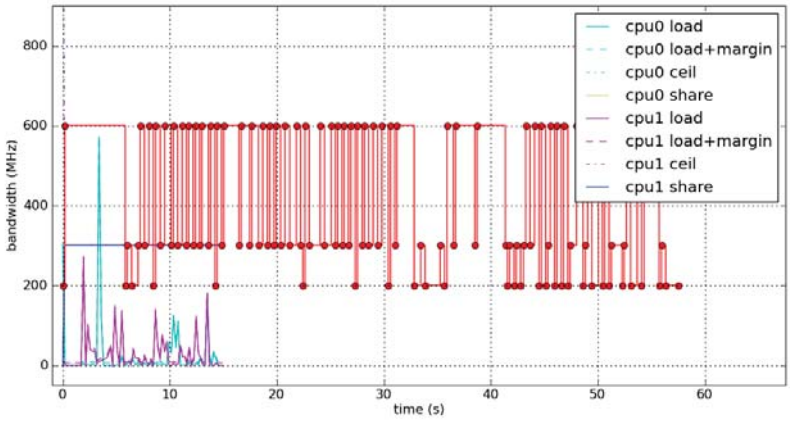

Figure B.35: Calculator, run 1.5

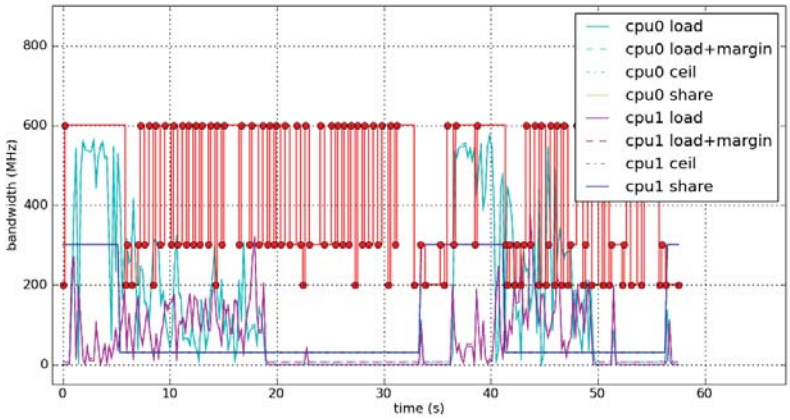

Figure B.36: SwiFTP, run 1.5

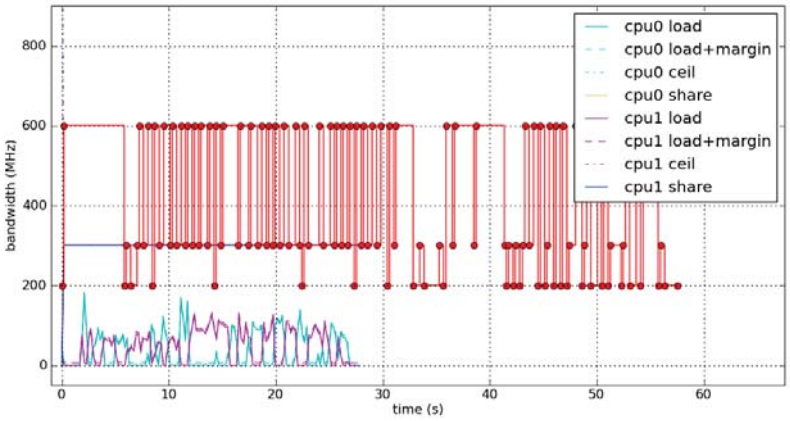

Figure B.37: Missile Intercept, run 1.5

B.2 User scenario 2

See Section 7.2.2 for a description of this scenario.

B.2.1 Run 2.1

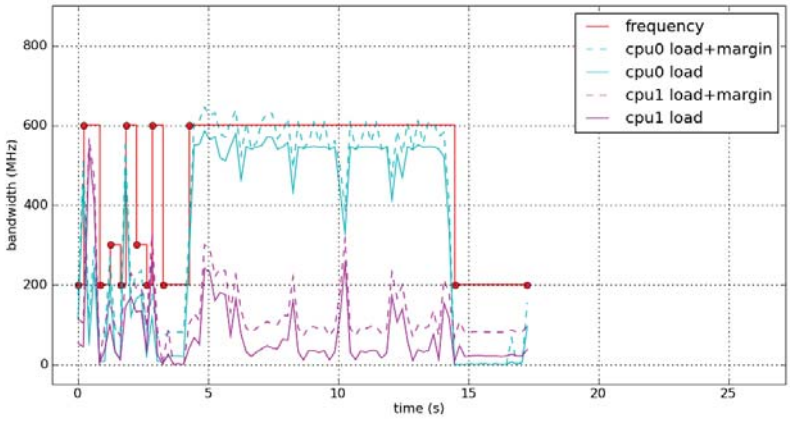

Figure B.38: System, run 2.1

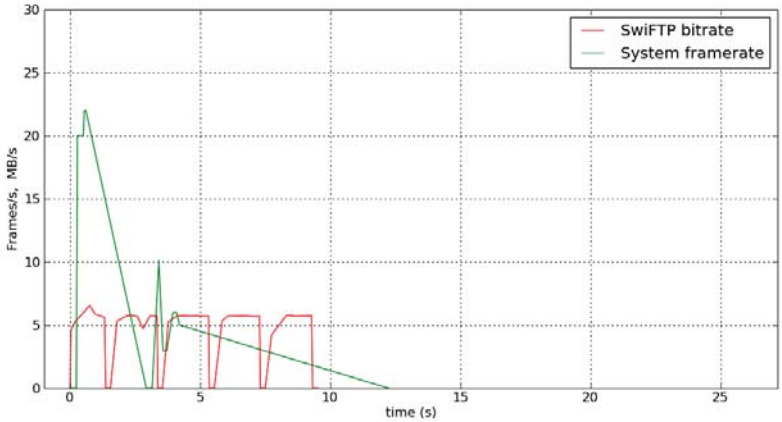

Figure B.39: FPS, BPS, run 2.1

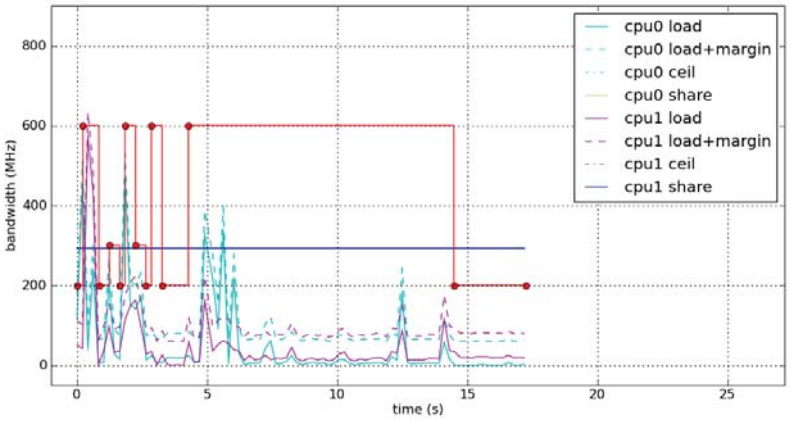

Figure B.40: Root, run 2.1

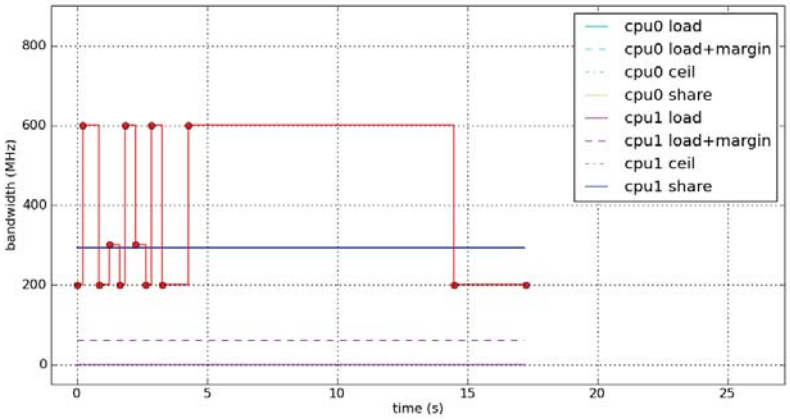

Figure B.41: fg_boost, run 2.1

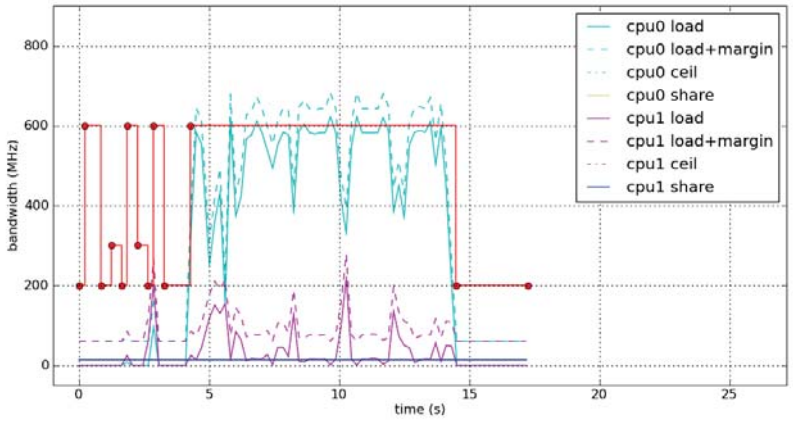

Figure B.42: bg_non_interactive, run 2.1

B.2.2 Run 2.2

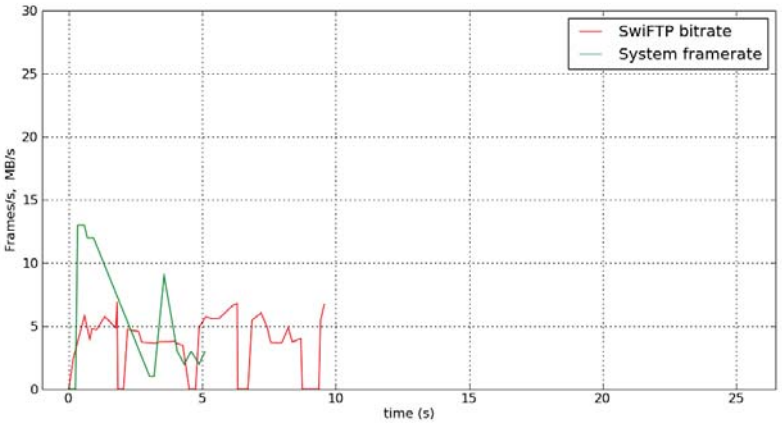

Figure B.43: FPS, BPS, run 2.2

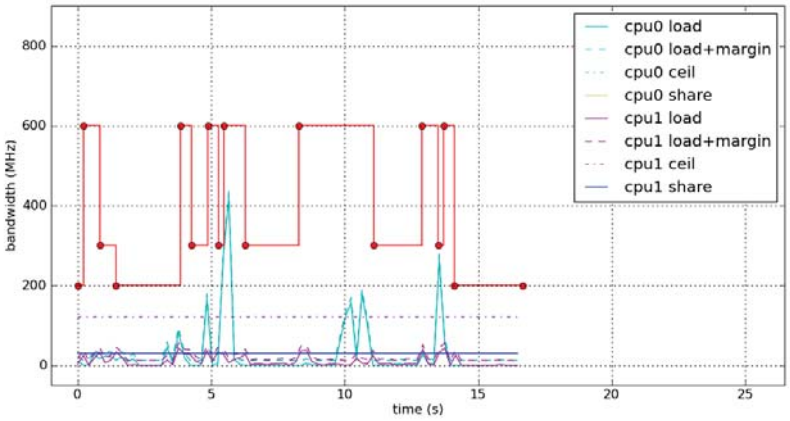

Figure B.44: Root, run 2.2

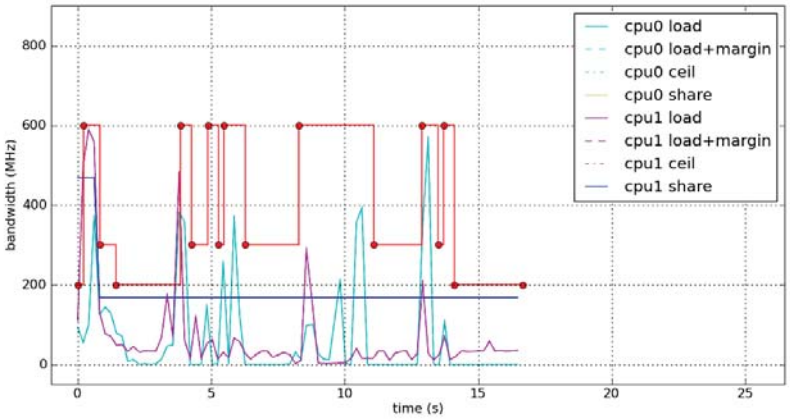

Figure B.45: Slacker, run 2.2

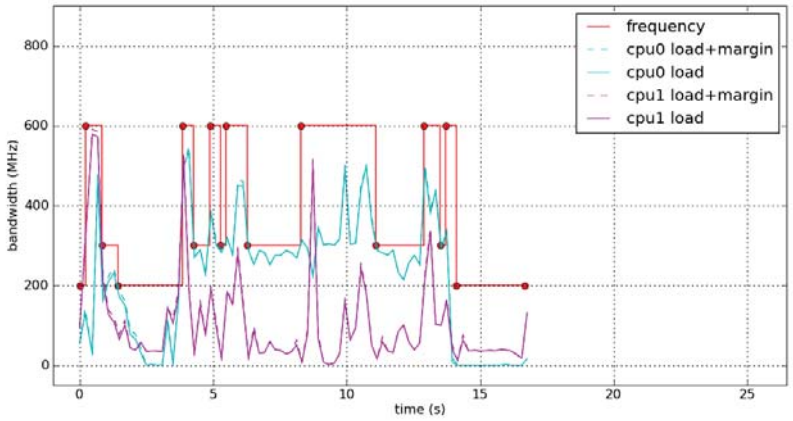

Figure B.46: System, run 2.2

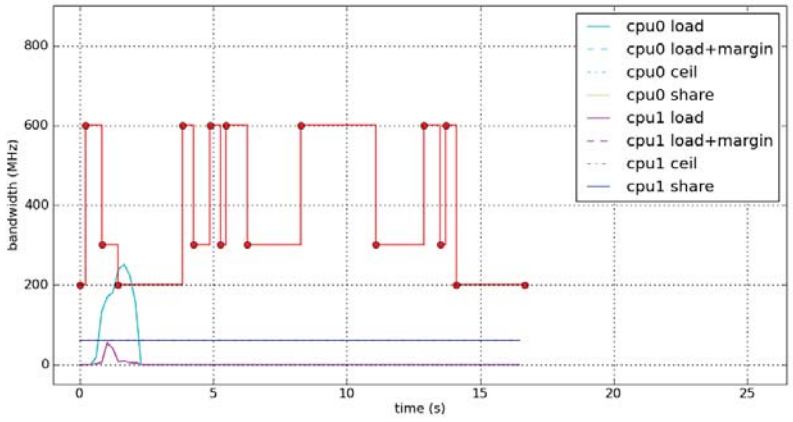

Figure B.47: Self, run 2.2

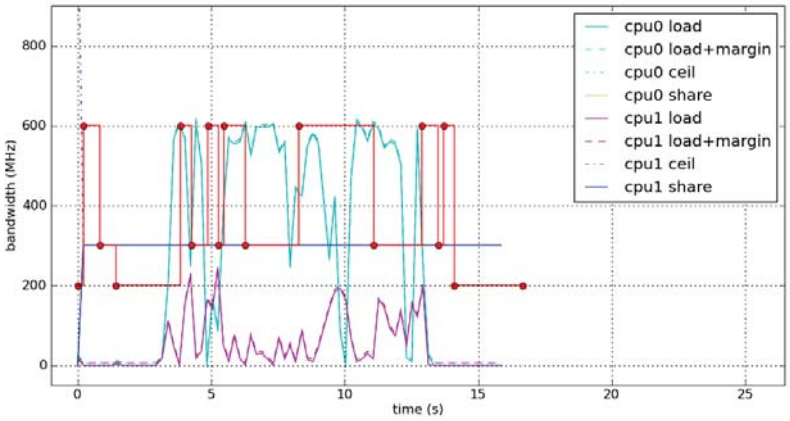

Figure B.48: SwiFTP, run 2.2

B.2.3 Run 2.3

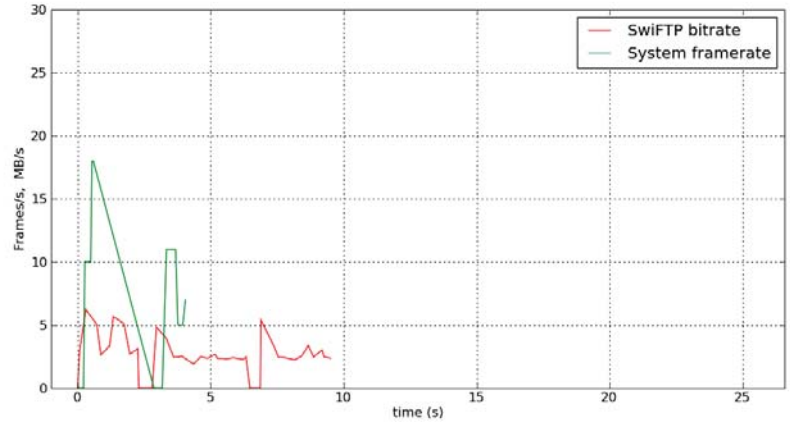

Figure B.49: FPS, BPS, run 2.3

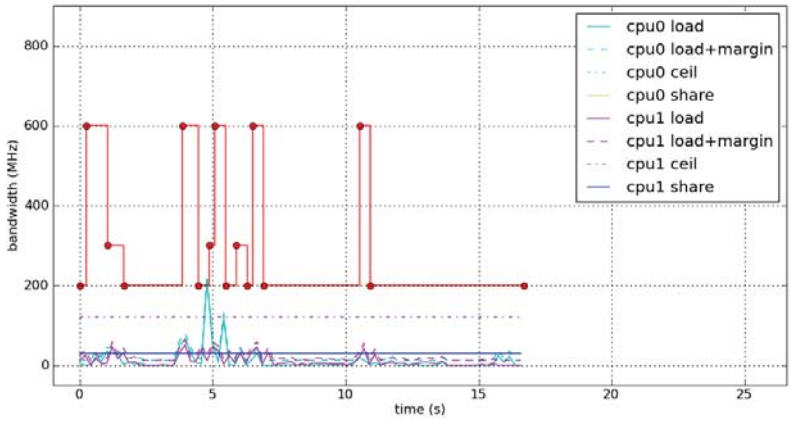

Figure B.50: Root, run 2.3

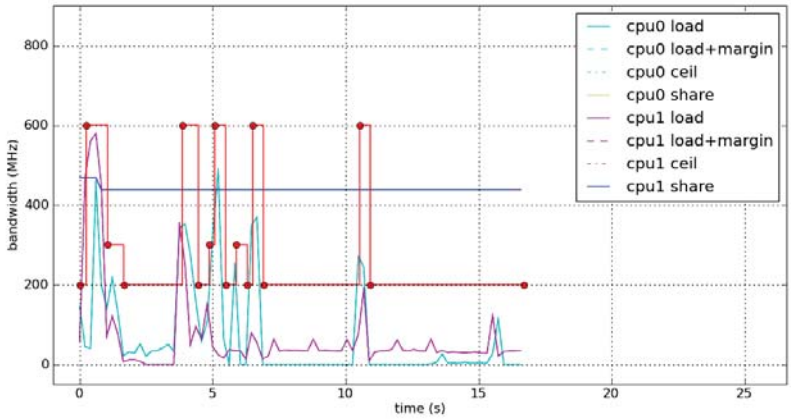

Figure B.51: Slacker, run 2.3

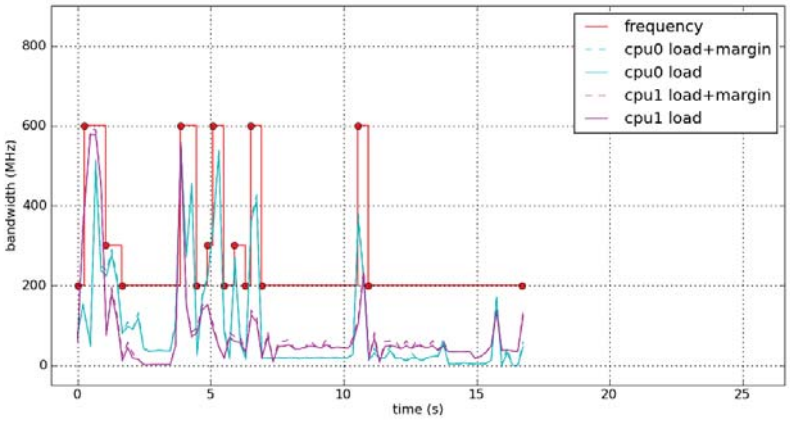

Figure B.52: System, run 2.3

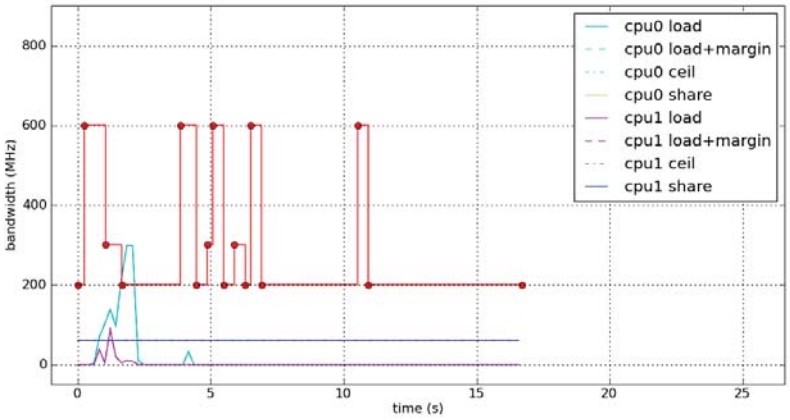

Figure B.53: Self, run 2.3

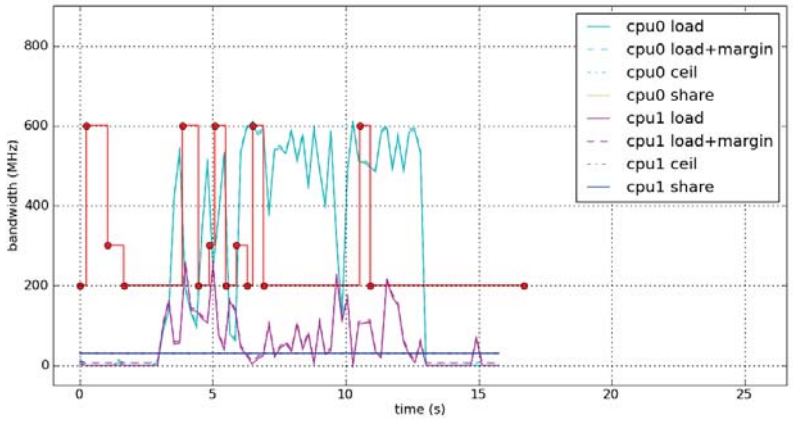

Figure B.54: SwiFTP, run 2.3

B.2.4 Run 2.4

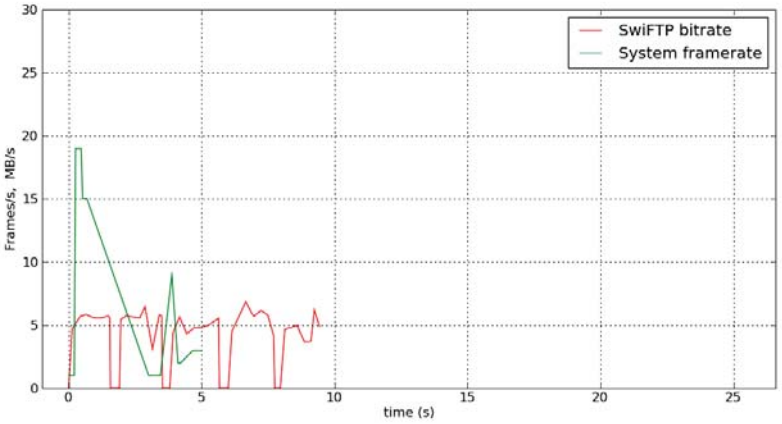

Figure B.55: FPS, BPS, run 2.4

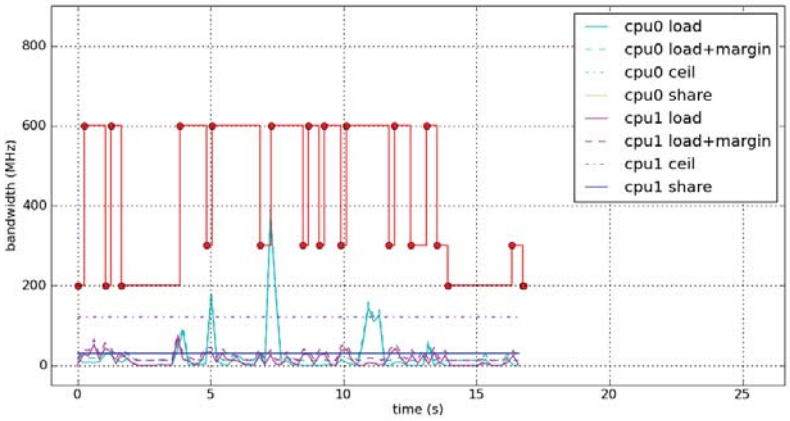

Figure B.56: Root, run 2.4

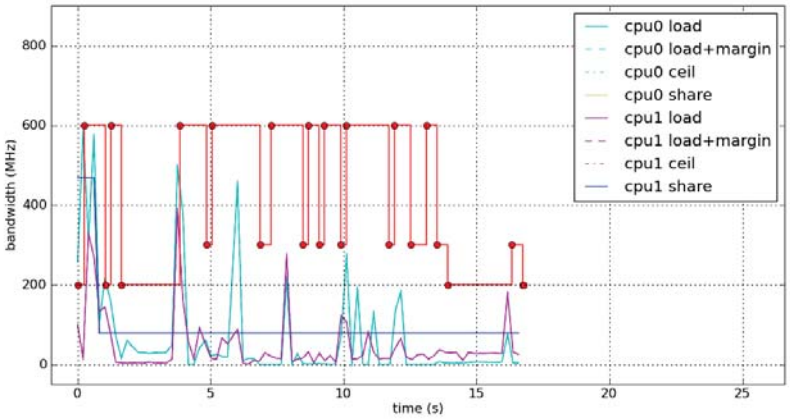

Figure B.57: Slacker, run 2.4

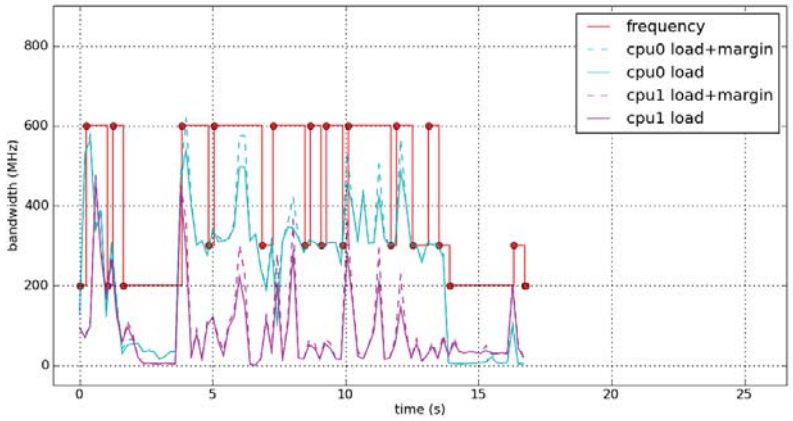

Figure B.58: System, run 2.4

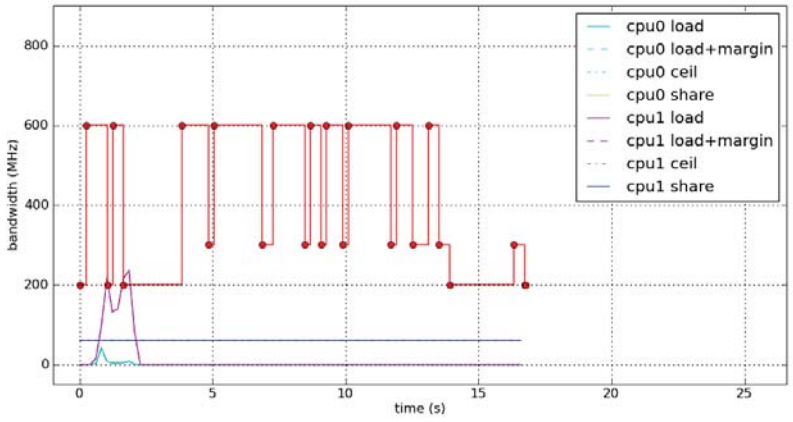

Figure B.59: Self, run 2.4

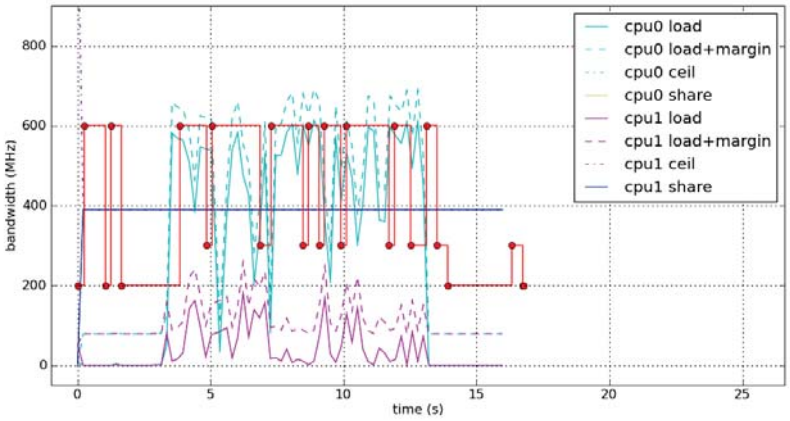

Figure B.60: SwiFTP, run 2.4

B.3 User scenario 3

See Section 7.2.3 for a description of this scenario.

B.3.1 Run 3.1

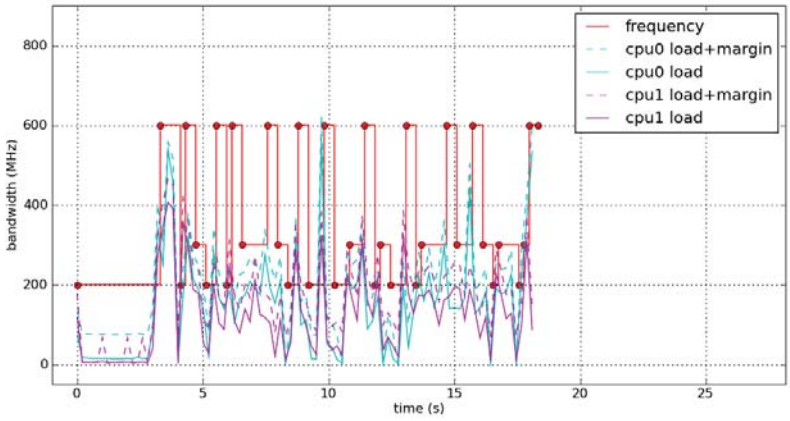

Figure B.61: System, run 3.1

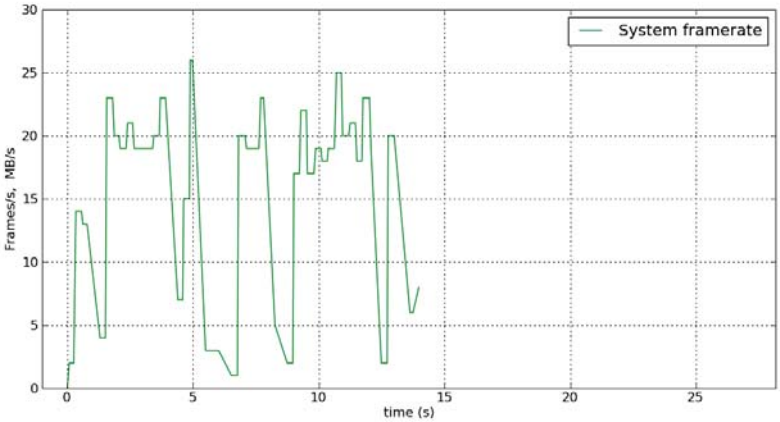

Figure B.62: FPS, BPS, run 3.1

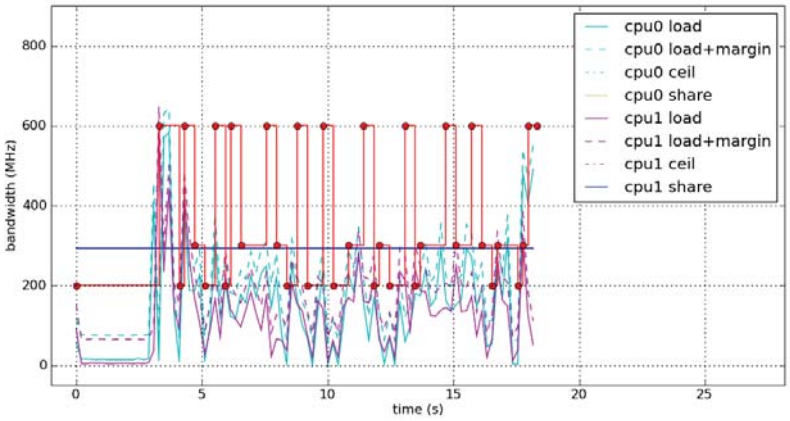

Figure B.63: Root, run 3.1

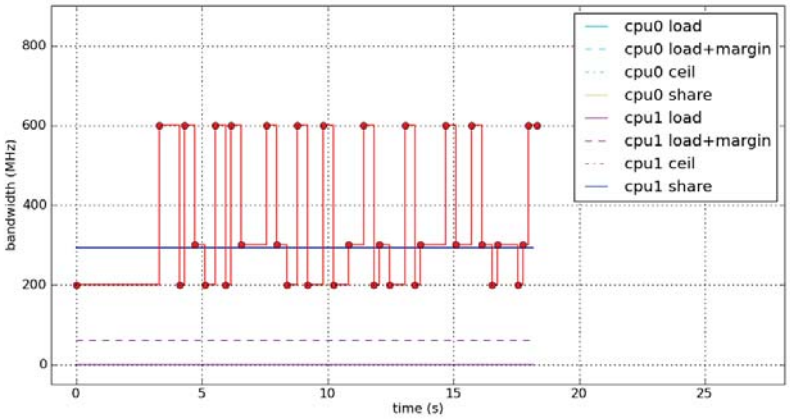

Figure B.64: fg_boost, run 3.1

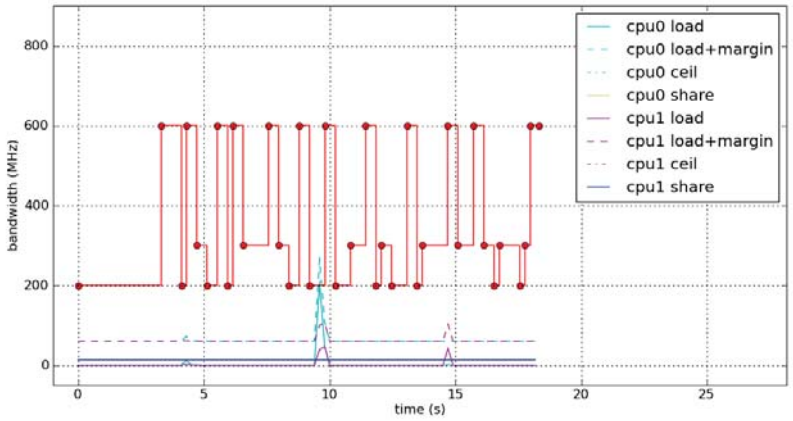

Figure B.65: bg_non_interactive, run 3.1

B.3.2 Run 3.2

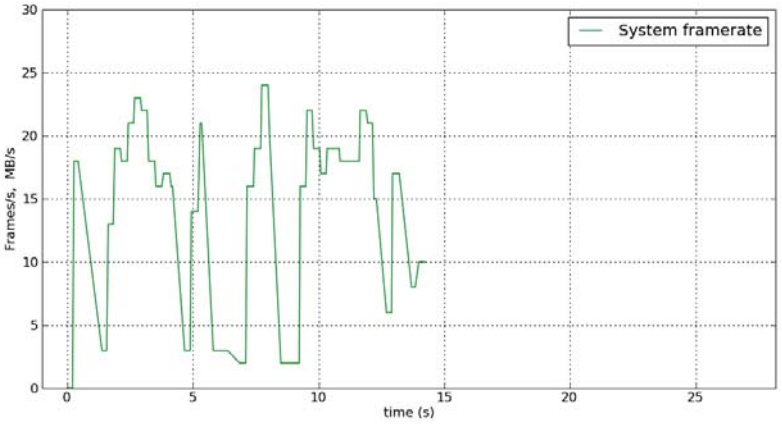

Figure B.66: FPS, BPS, run 3.2

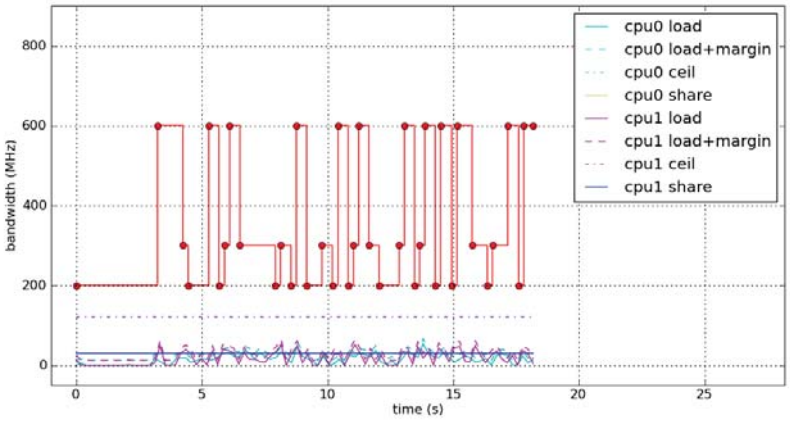

Figure B.67: Root, run 3.2

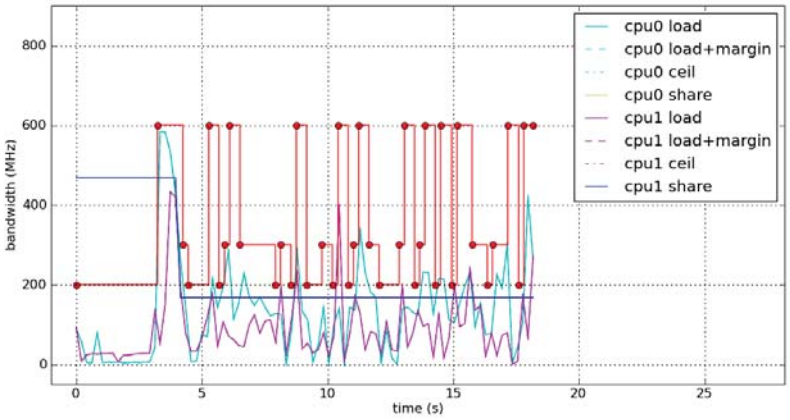

Figure B.68: Slacker, run 3.2

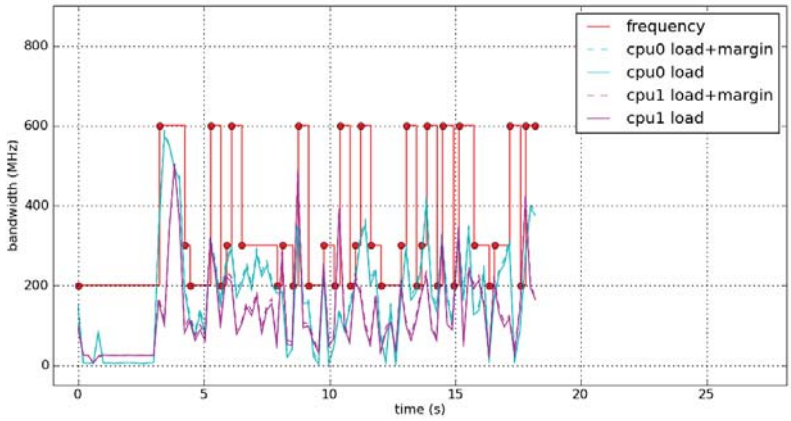

Figure B.69: System, run 3.2

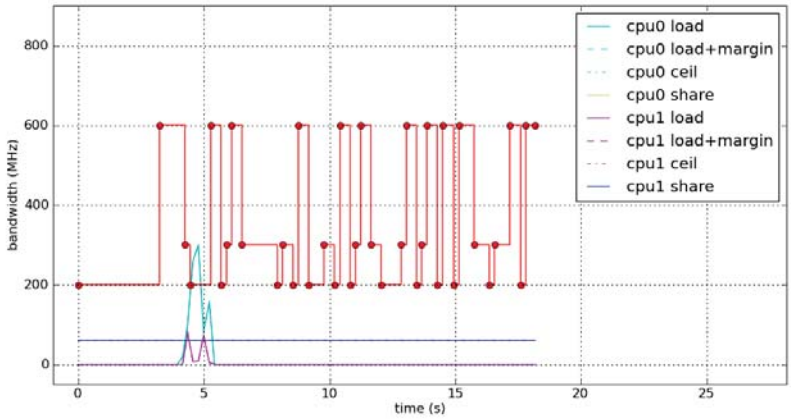

Figure B.70: Self, run 3.2

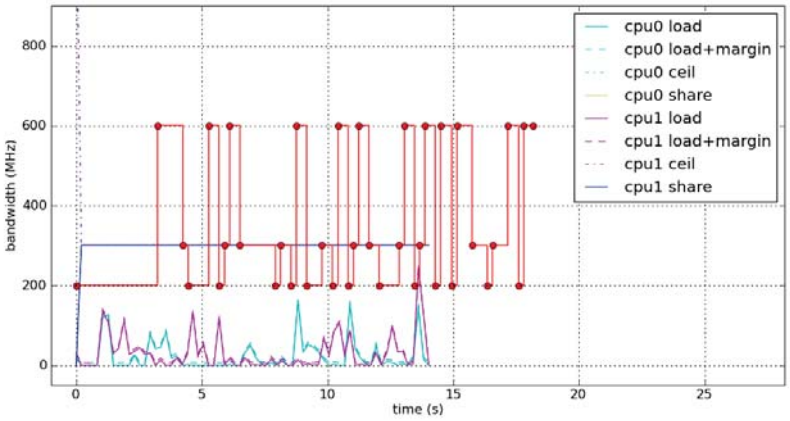

Figure B.71: Calculator, run 3.2

B.3.3 Run 3.3

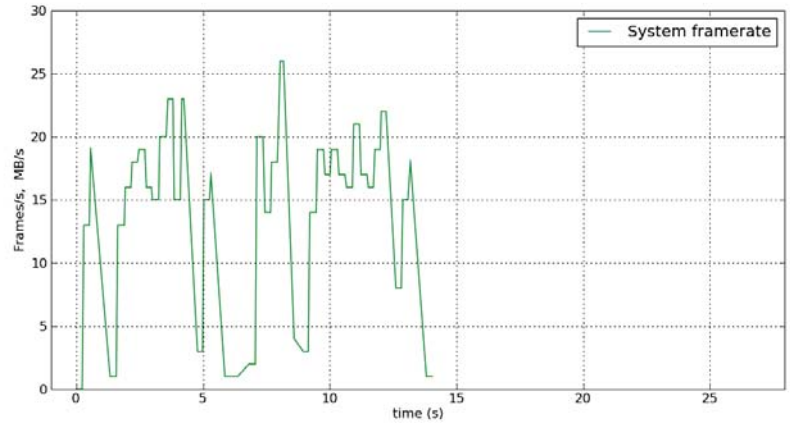

Figure B.72: FPS, BPS, run 3.3

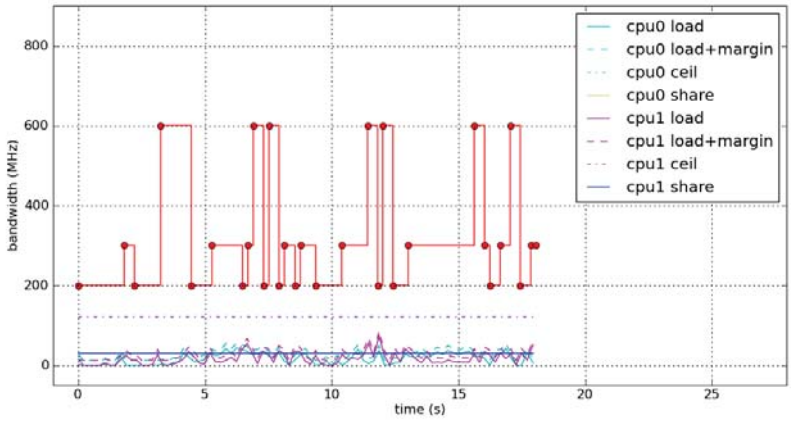

Figure B.73: Root, run 3.3

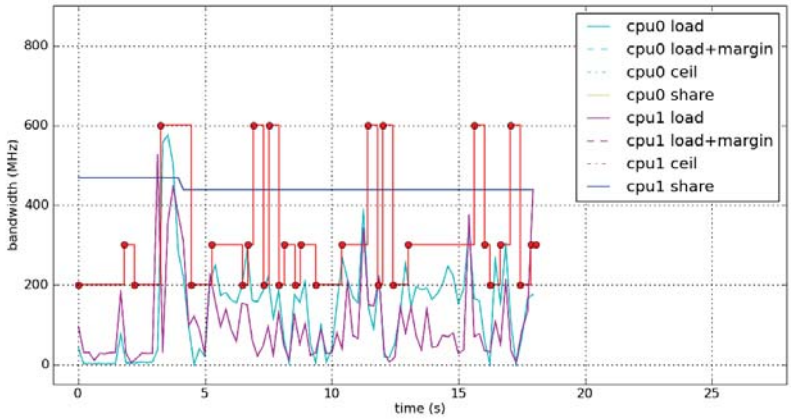

Figure B.74: Slacker, run 3.3

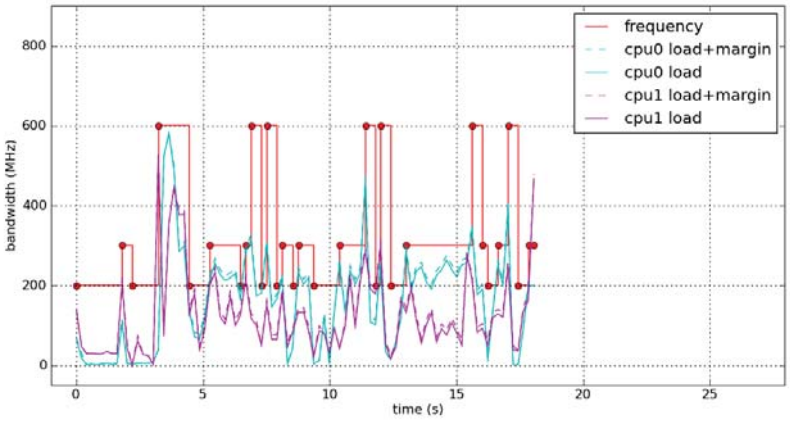

Figure B.75: System, run 3.3

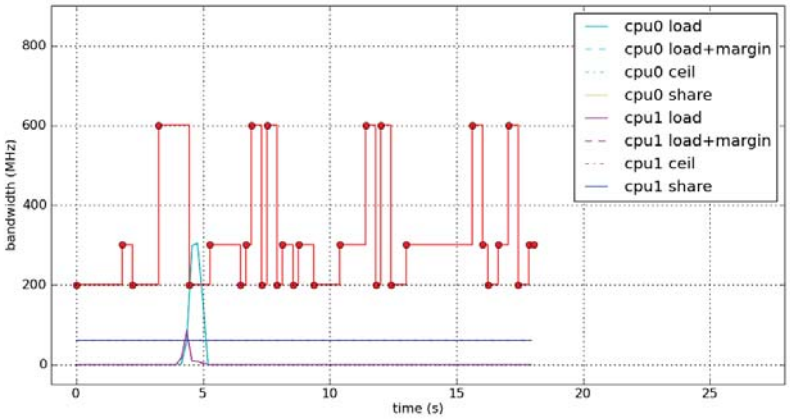

Figure B.76: Self, run 3.3

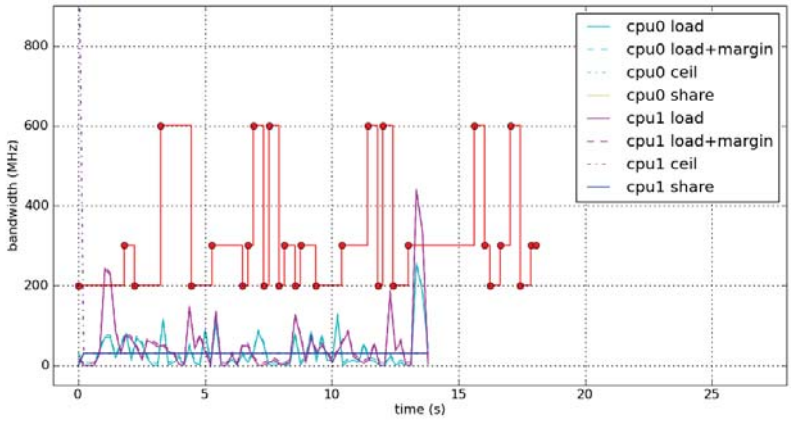

Figure B.77: Calculator, run 3.3

B.3.4 Run 3.4

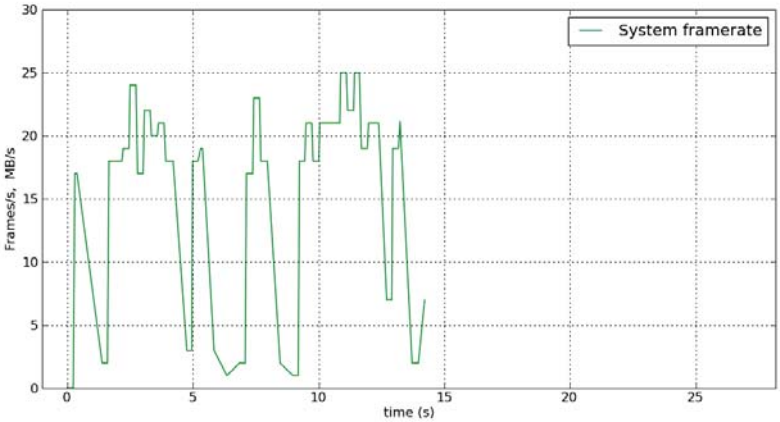

Figure B.78: FPS, BPS, run 3.4

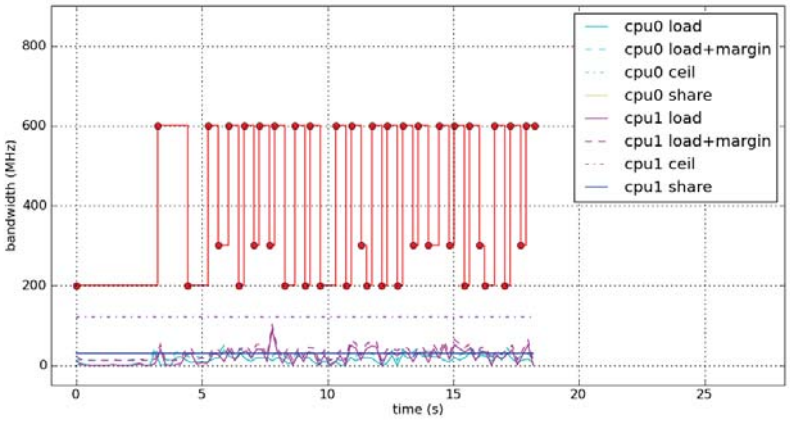

Figure B.79: Root, run 3.4

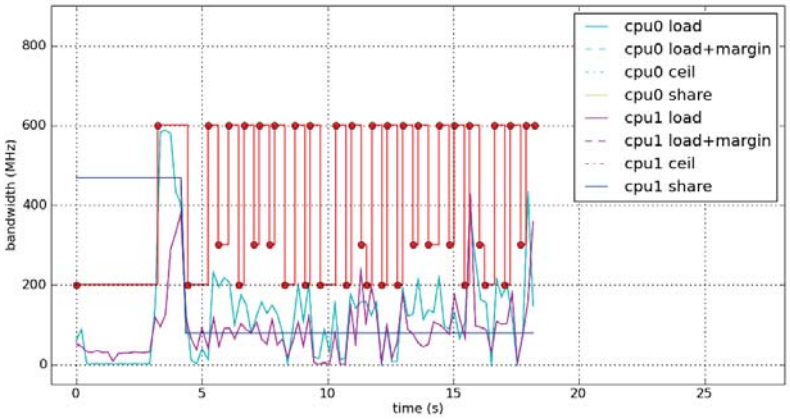

Figure B.80: Slacker, run 3.4

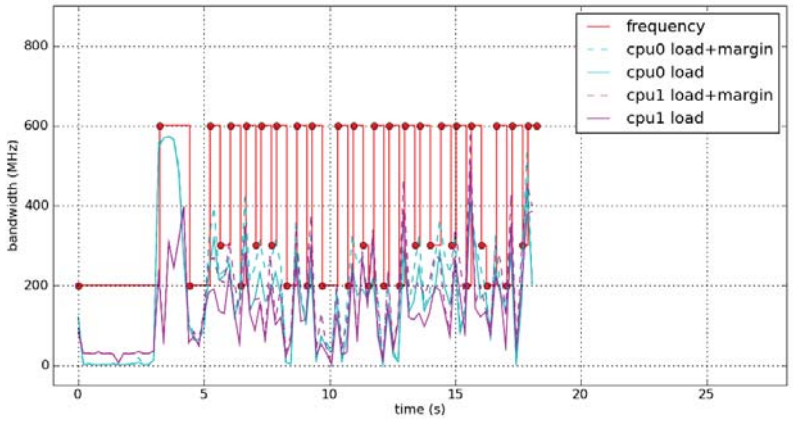

Figure B.81: System, run 3.4

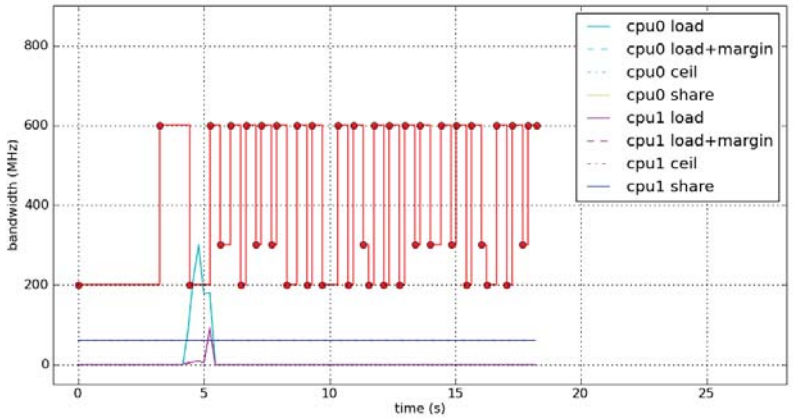

Figure B.82: Self, run 3.4

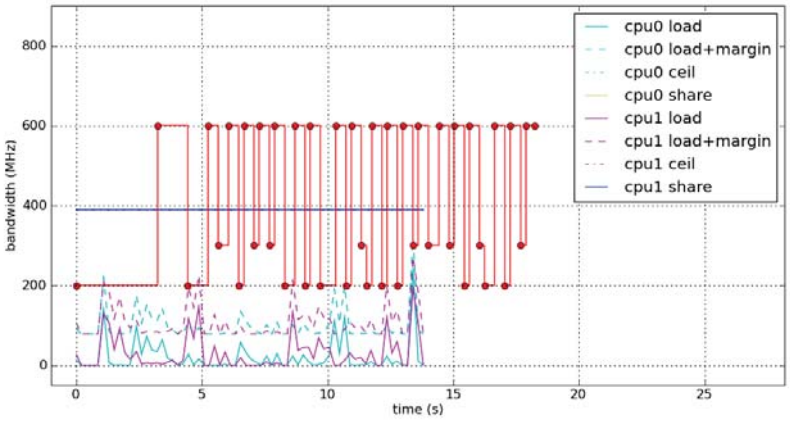

Figure B.83: Calculator, run 3.4

B.4 User scenario 4

See Section 7.2.4 for a description of this scenario.

B.4.1 Run 4.1

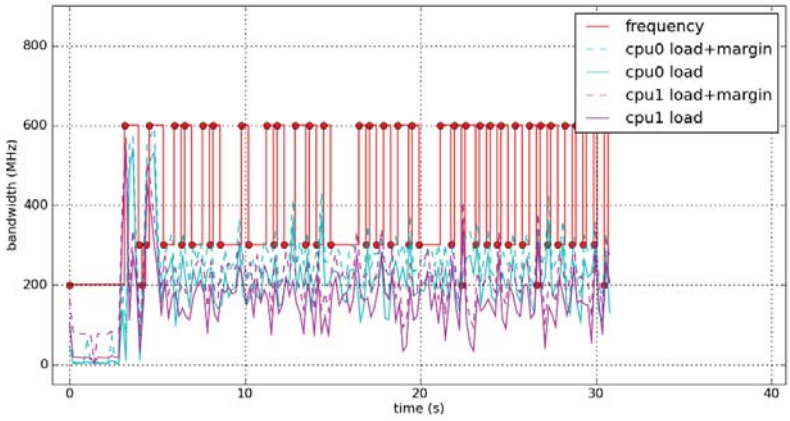

Figure B.84: System, run 4.1

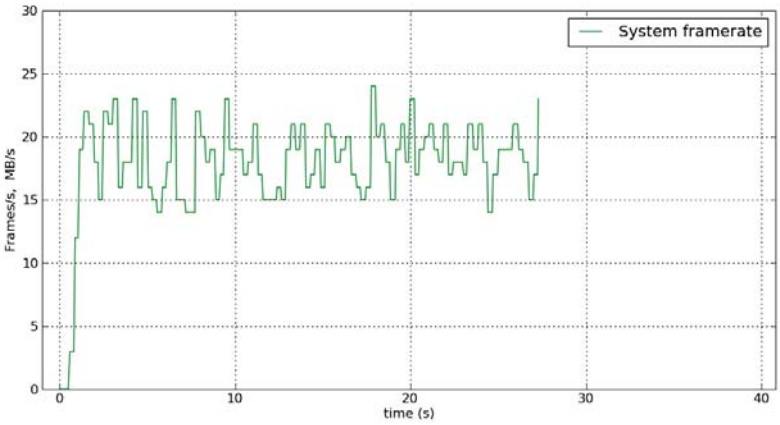

Figure B.85: FPS, BPS, run 4.1

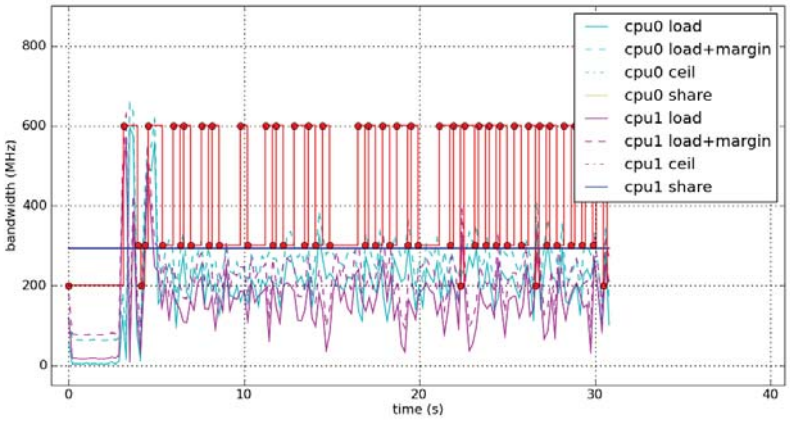

Figure B.86: Root, run 4.1

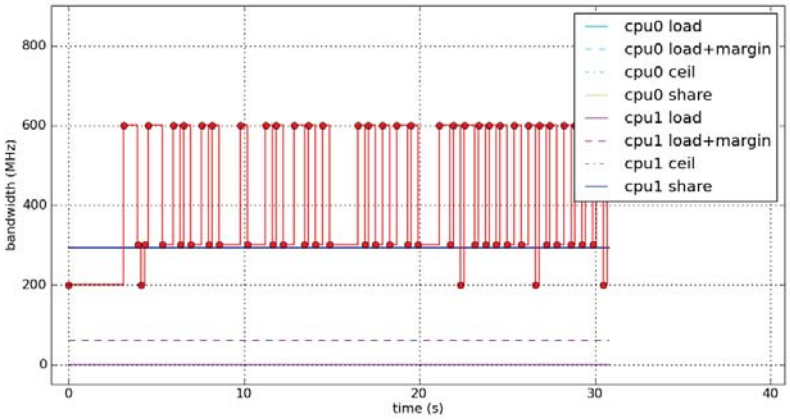

Figure B.87: fg_boost, run 4.1

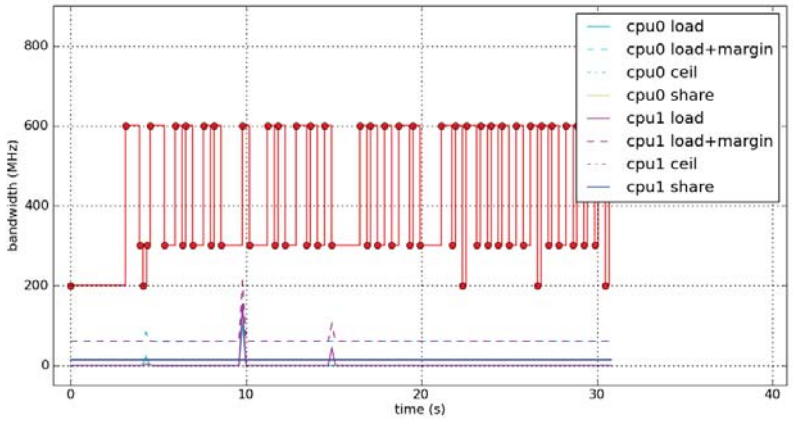

Figure B.88: bg_non_interactive, run 4.1

B.4.2 Run 4.2

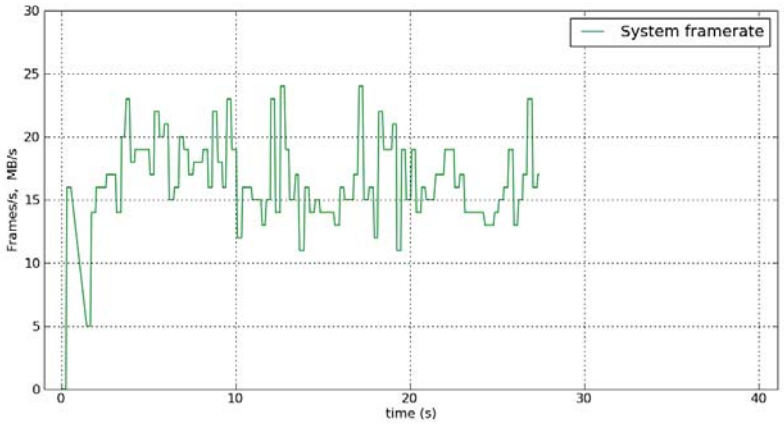

Figure B.89: FPS, BPS, run 4.2

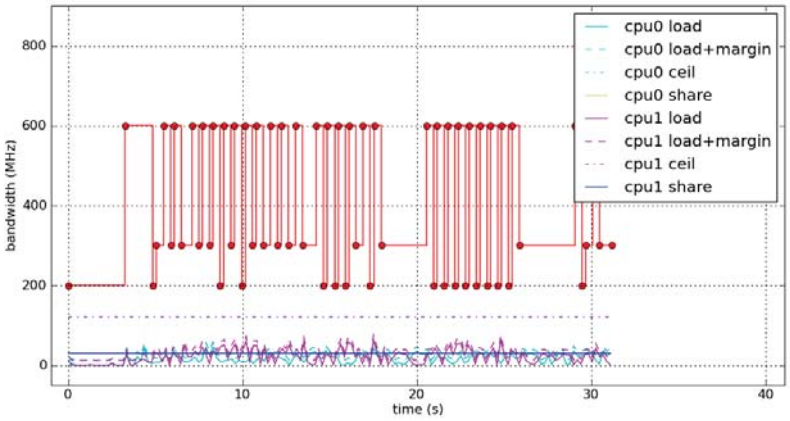

Figure B.90: Root, run 4.2

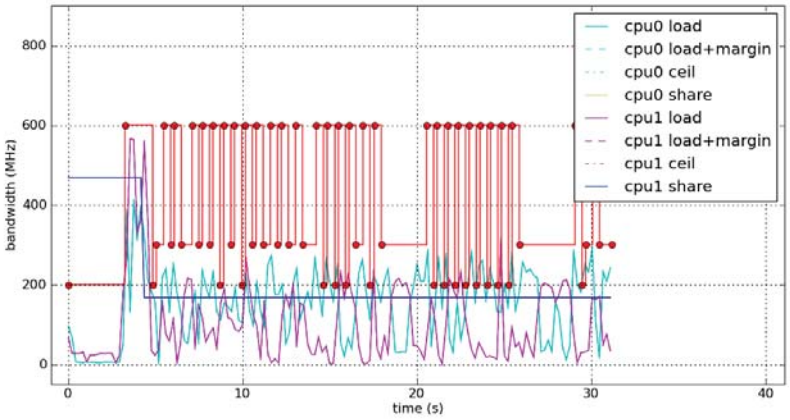

Figure B.91: Slacker, run 4.2

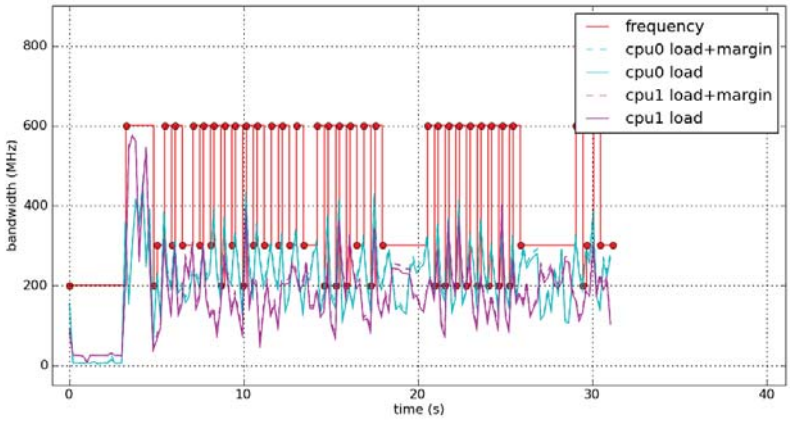

Figure B.92: System, run 4.2

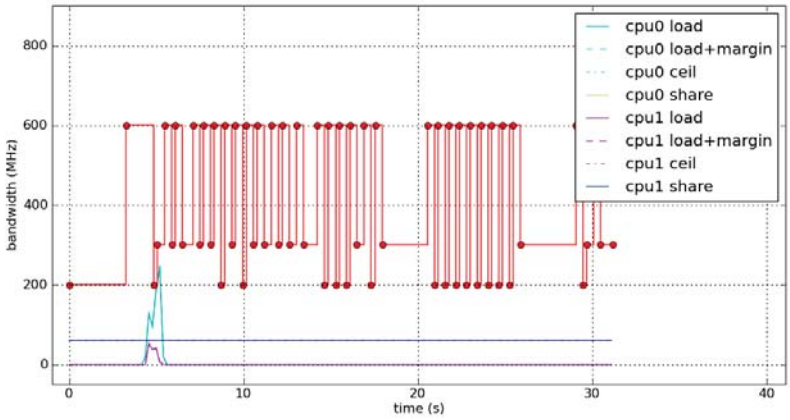

Figure B.93: Self, run 4.2

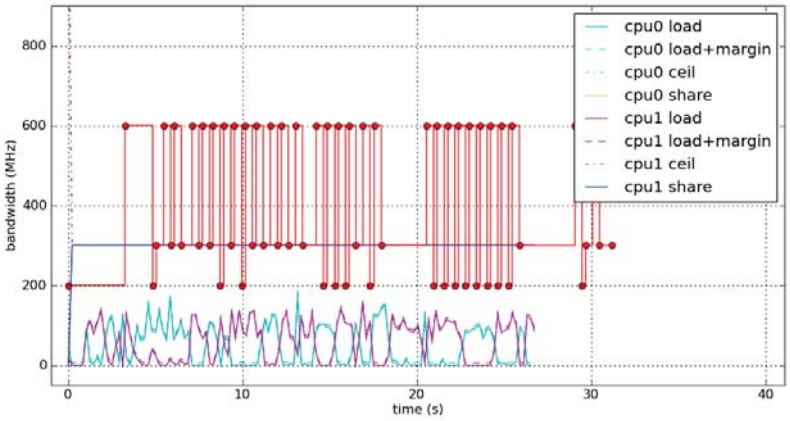

Figure B.94: Missile Intercept, run 4.2

B.4.3 Run 4.3

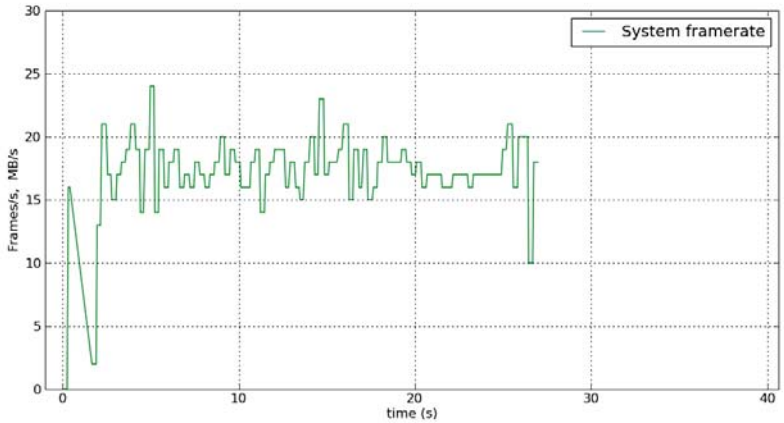

Figure B.95: FPS, BPS, run 4.3

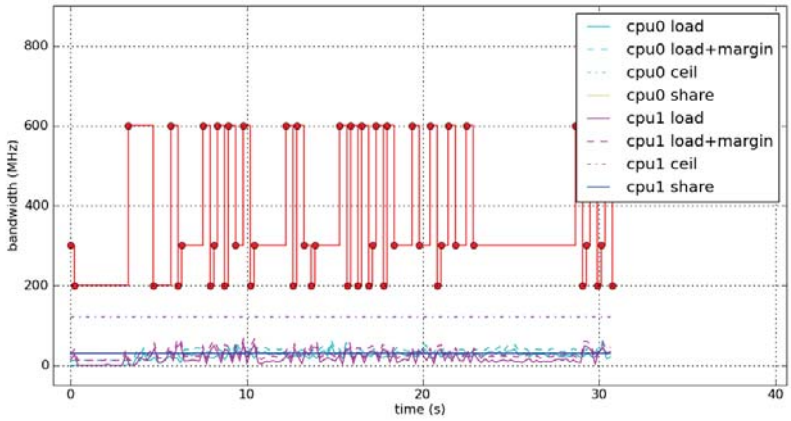

Figure B.96: Root, run 4.3

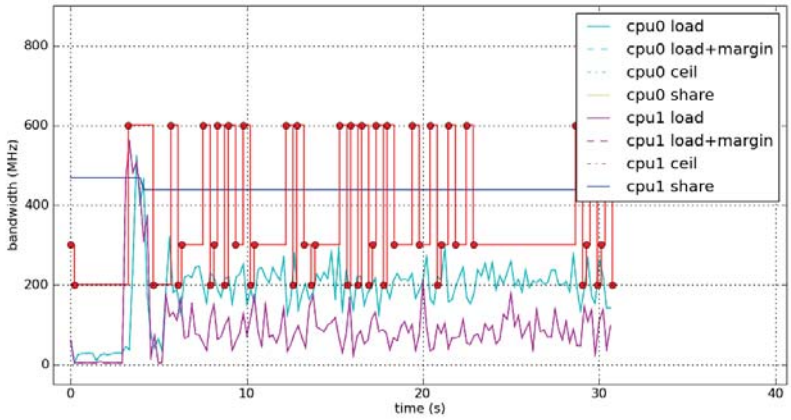

Figure B.97: Slacker, run 4.3

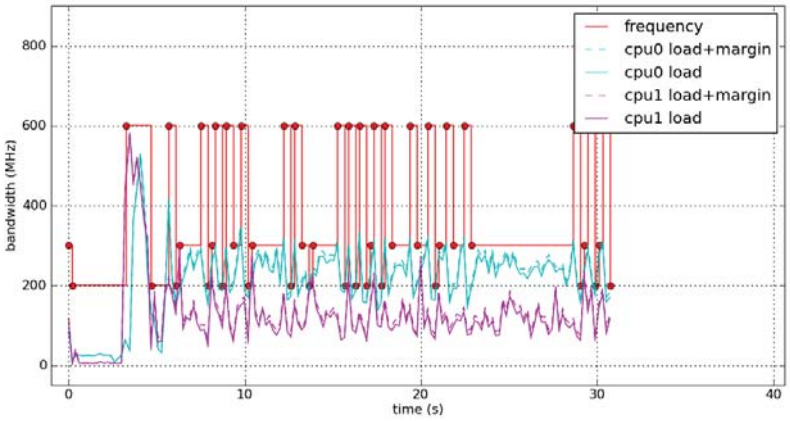

Figure B.98: System, run 4.3

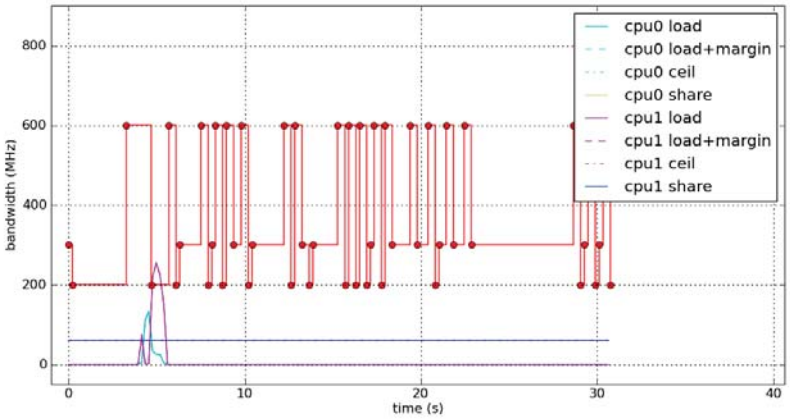

Figure B.99: Self, run 4.3

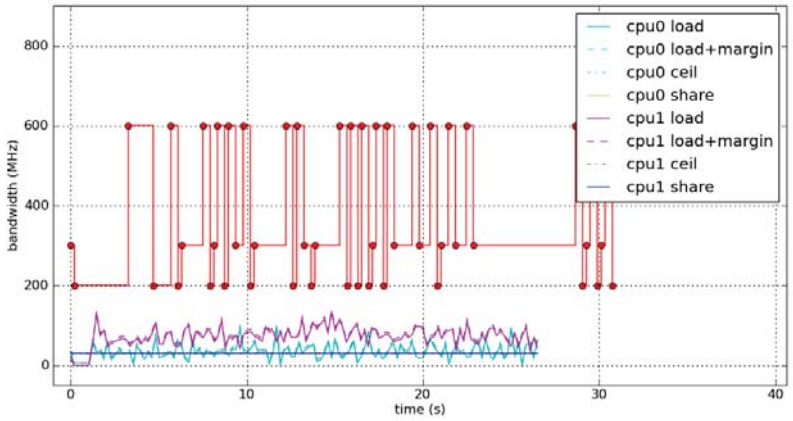

Figure B.100: Missile Intercept, run 4.3

B.4.4 Run 4.4

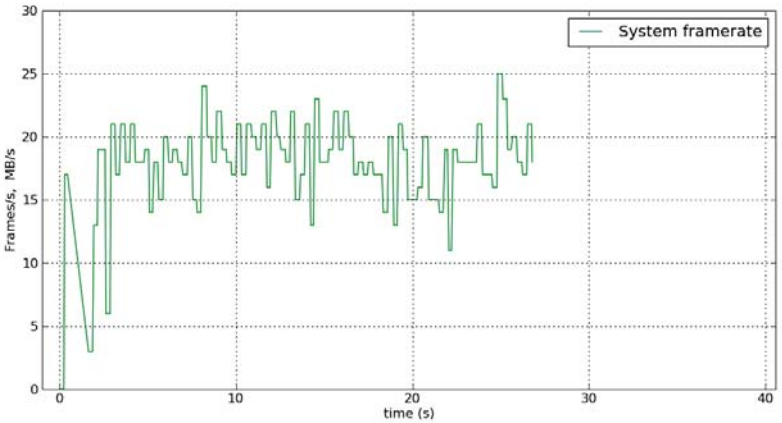

Figure B.101: FPS, BPS, run 4.4

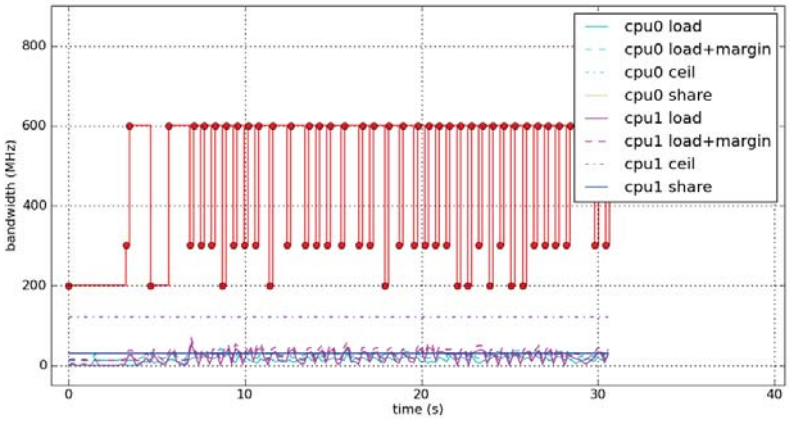

Figure B.102: Root, run 4.4

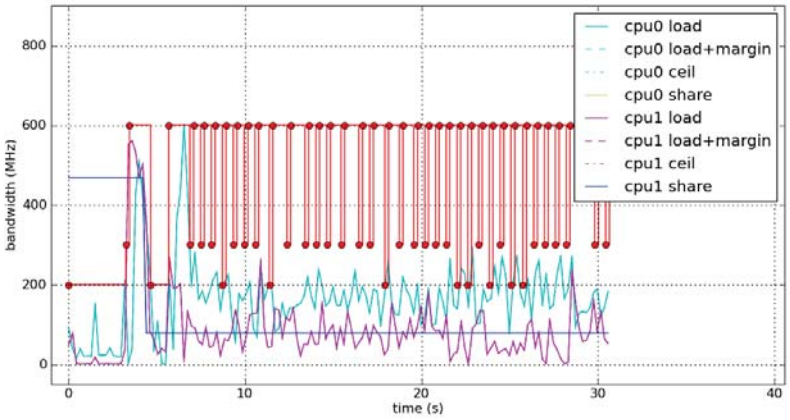

Figure B.103: Slacker, run 4.4

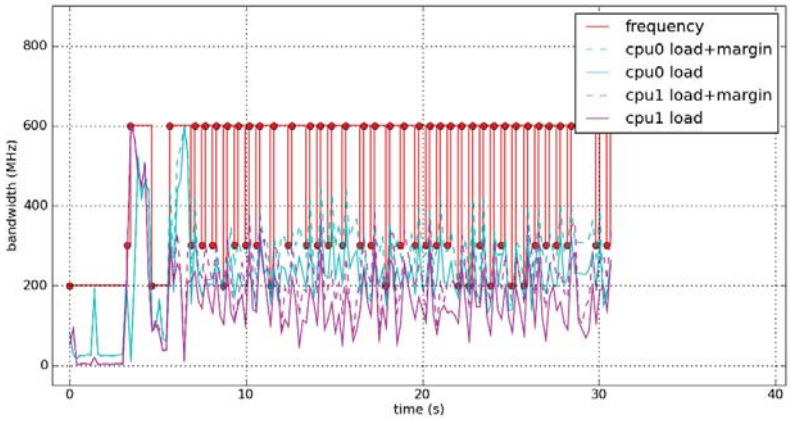

Figure B.104: System, run 4.4

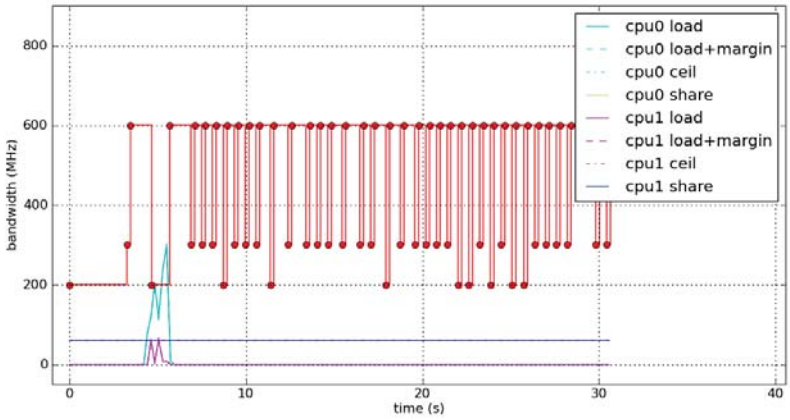

Figure B.105: Self, run 4.4

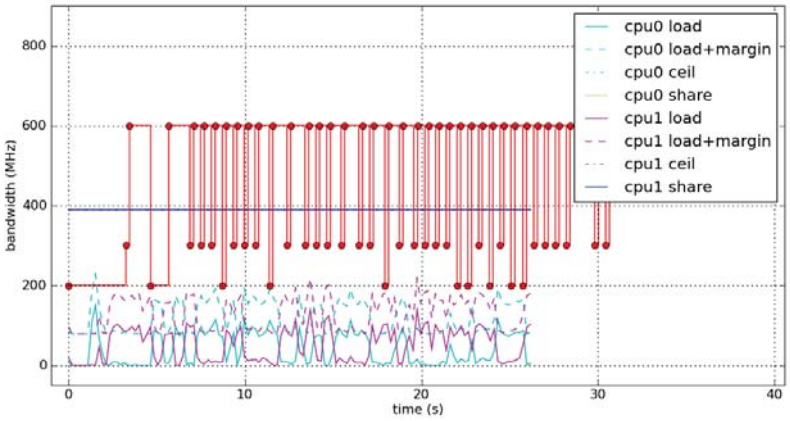

Figure B.106: Missile Intercept, run 4.4

B.5 User scenario 5

See Section 7.2.5 for a description of this scenario.

B.5.1 Run 5.1

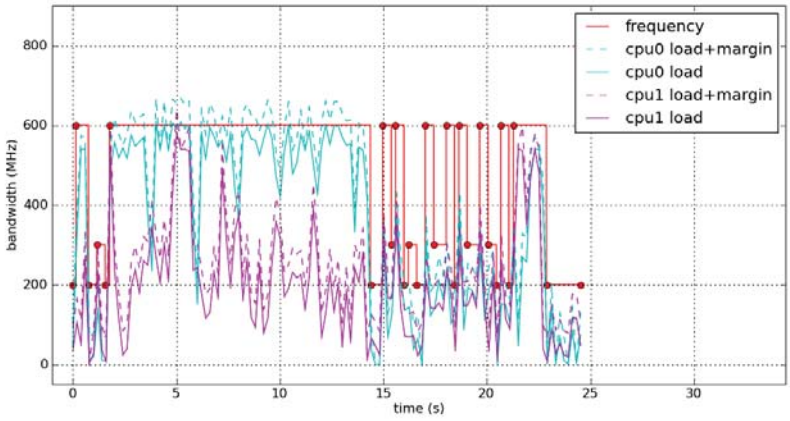

Figure B.107: System, run 5.1

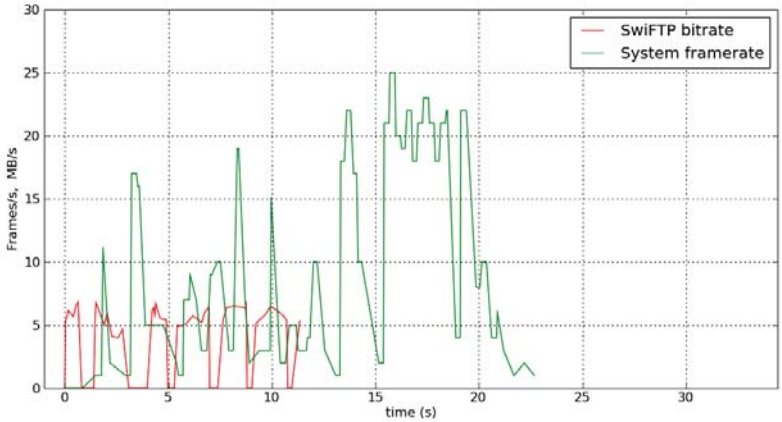

Figure B.108: FPS, BPS, run 5.1

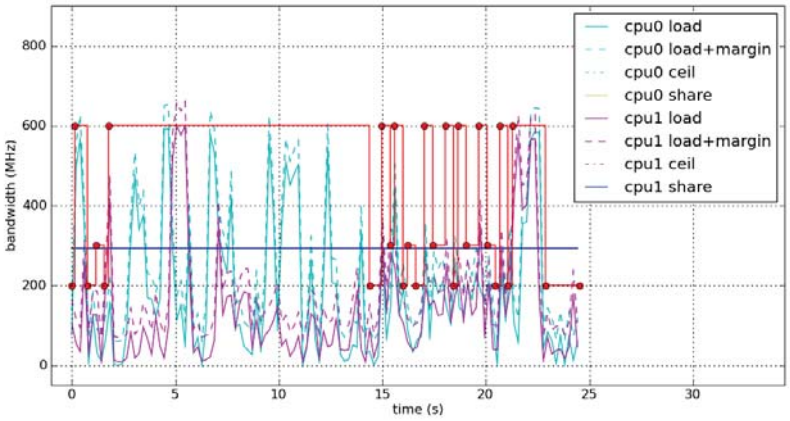

Figure B.109: Root, run 5.1

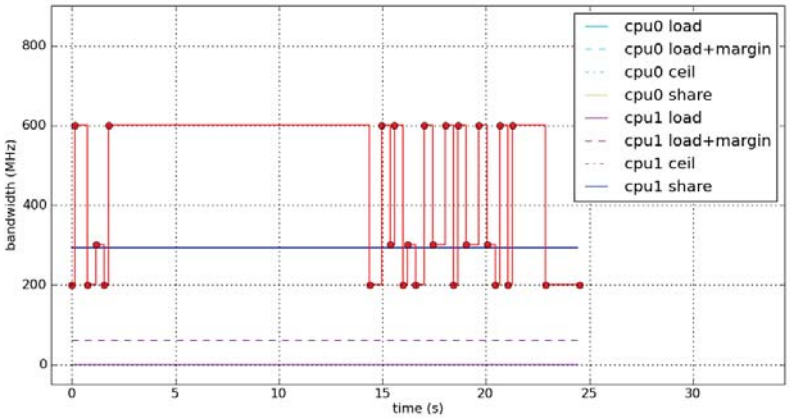

Figure B.110: fg_boost, run 5.1

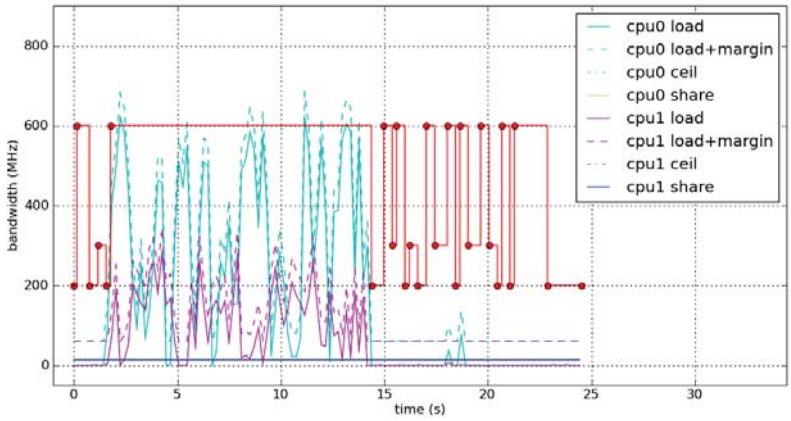

Figure B.111: bg_non_interactive, run 5.1

B.5.2 Run 5.2

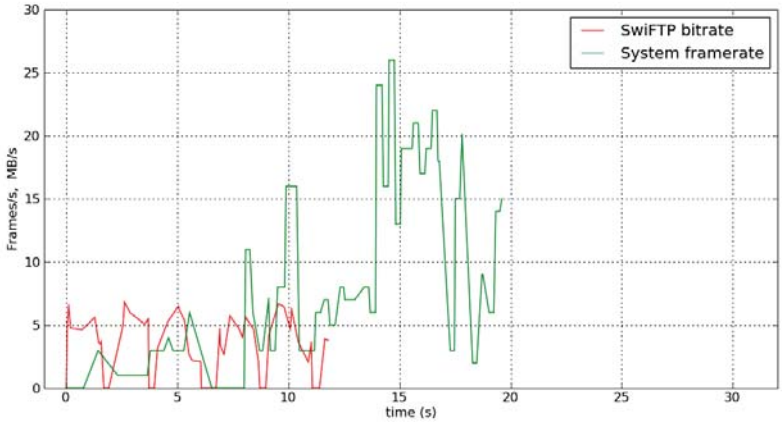

Figure B.112: FPS, BPS, run 5.2

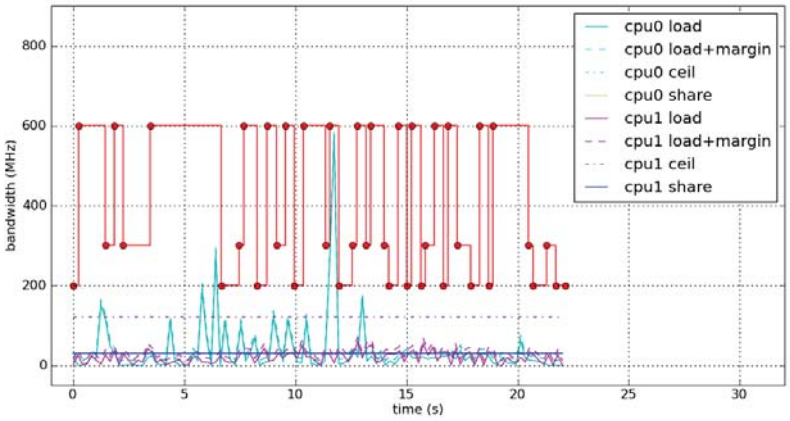

Figure B.113: Root, run 5.2

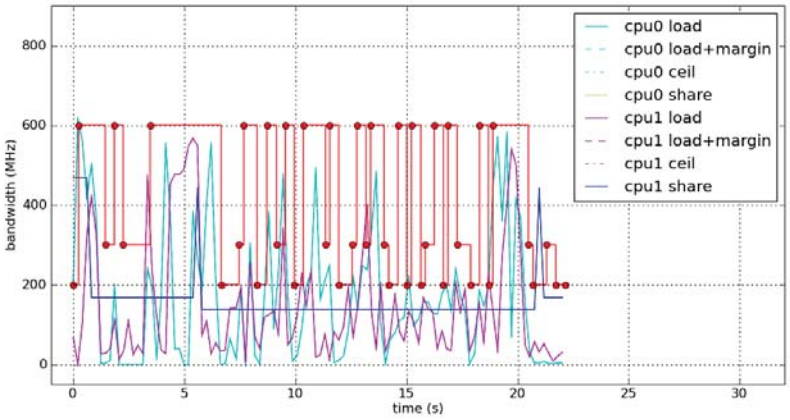

Figure B.114: Slacker, run 5.2

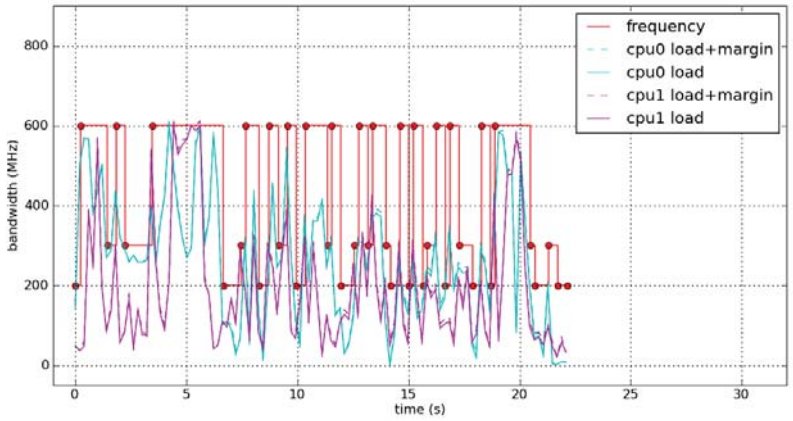

Figure B.115: System, run 5.2

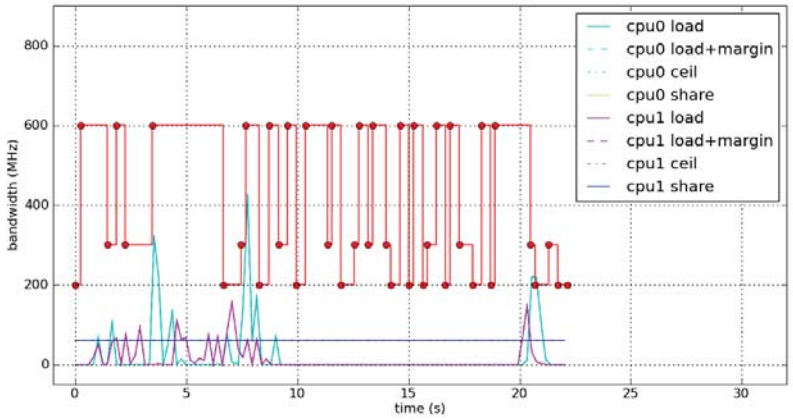

Figure B.116: Self, run 5.2

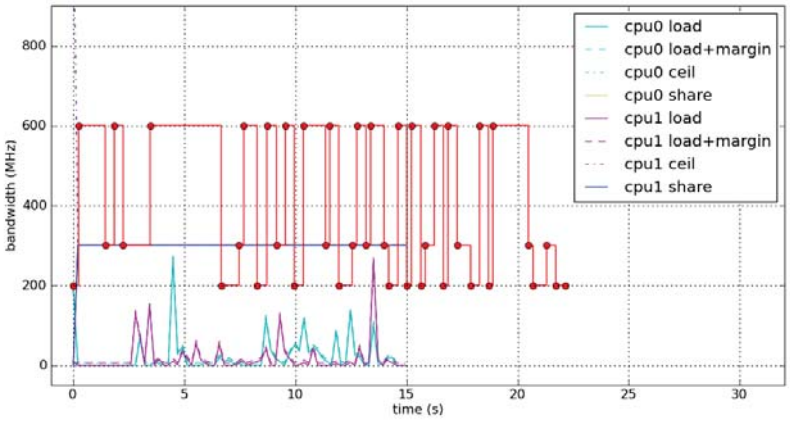

Figure B.117: Calculator, run 5.2

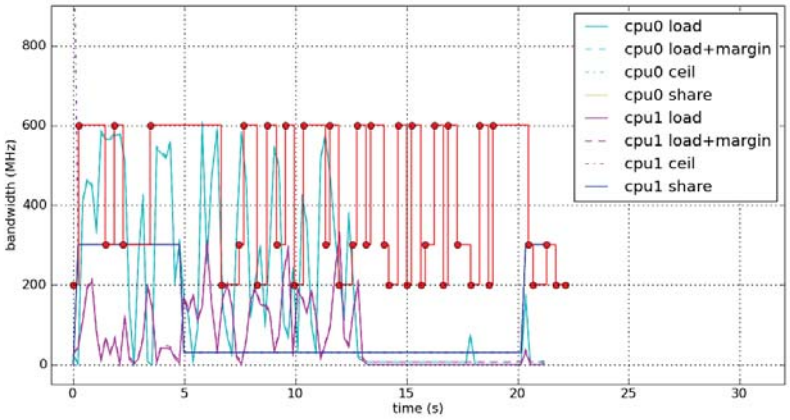

Figure B.118: SwiFTP, run 5.2

B.5.3 Run 5.3

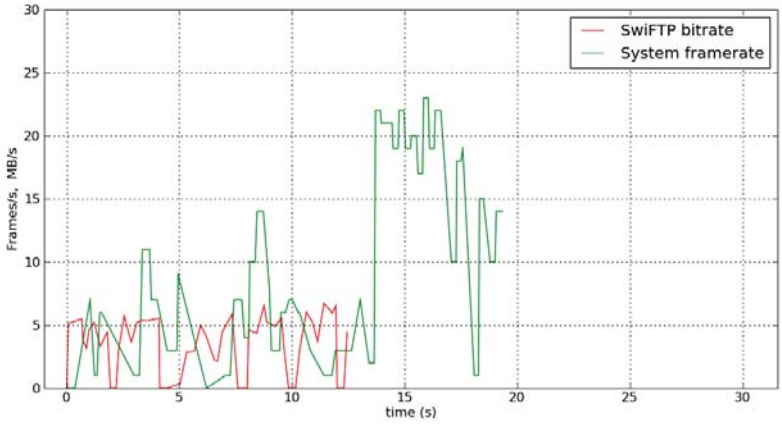

Figure B.119: FPS, BPS, run 5.3

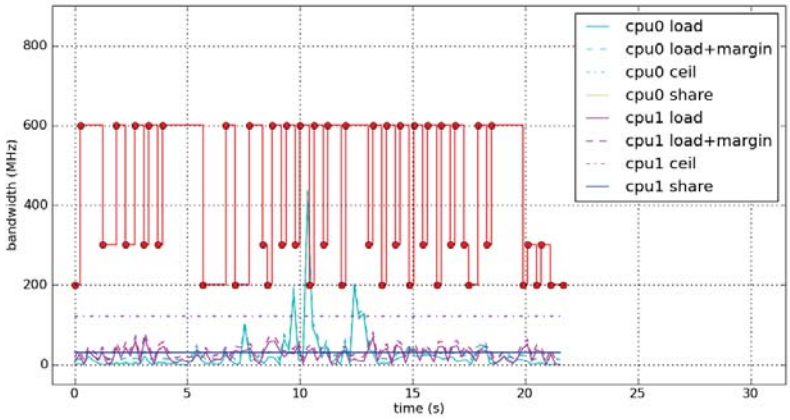

Figure B.120: Root, run 5.3

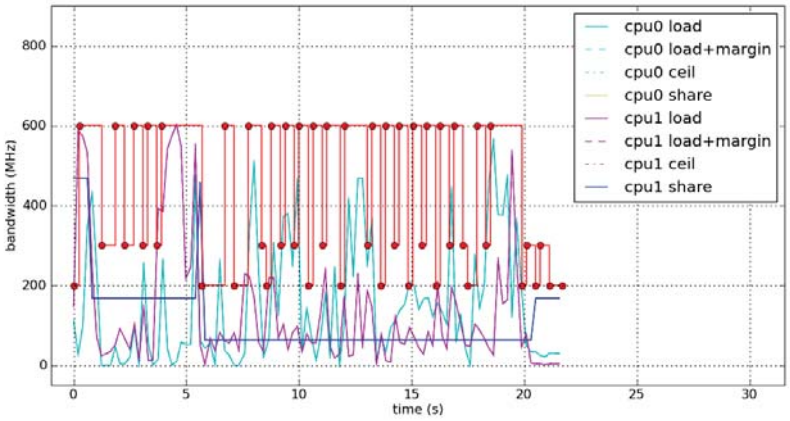

Figure B.121: Slacker, run 5.3

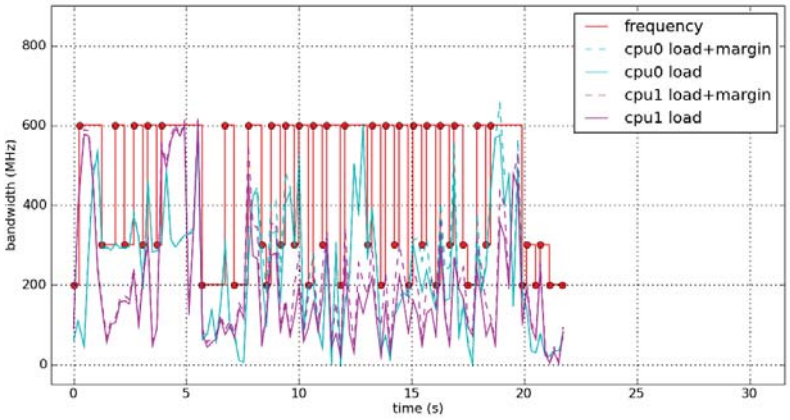

Figure B.122: System, run 5.3

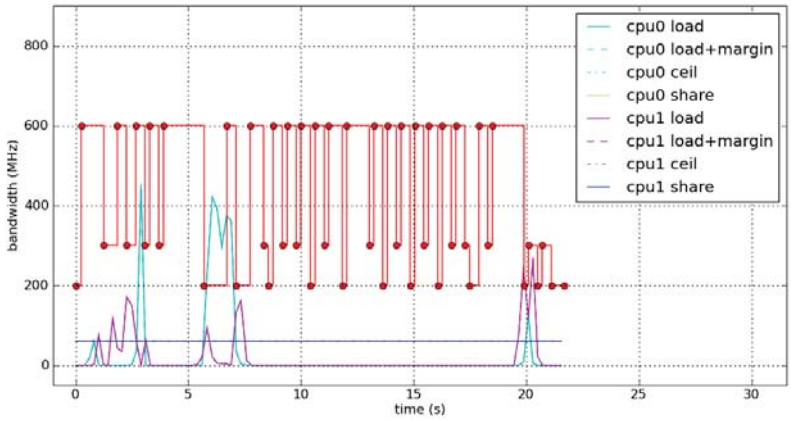

Figure B.123: Self, run 5.3

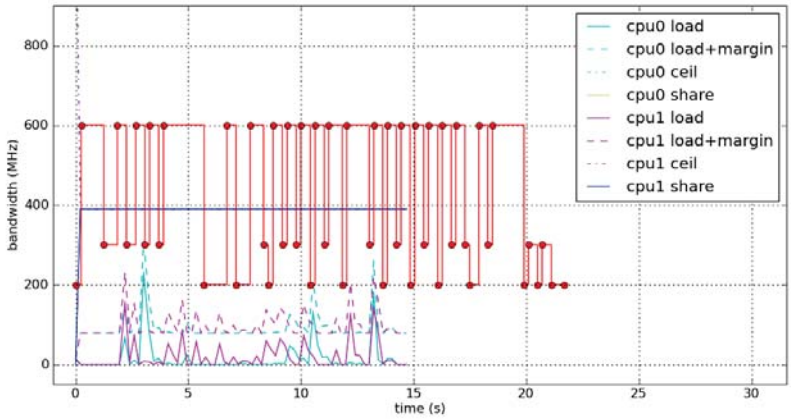

Figure B.124: Calculator, run 5.3

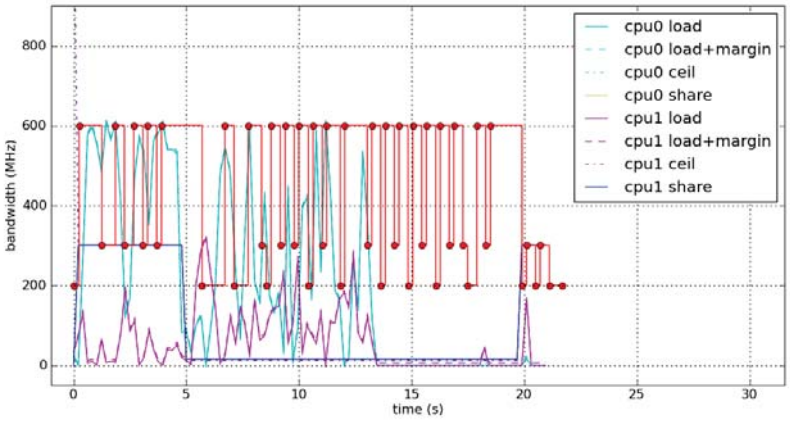

Figure B.125: SwiFTP, run 5.3

B.5.4 Run 5.4

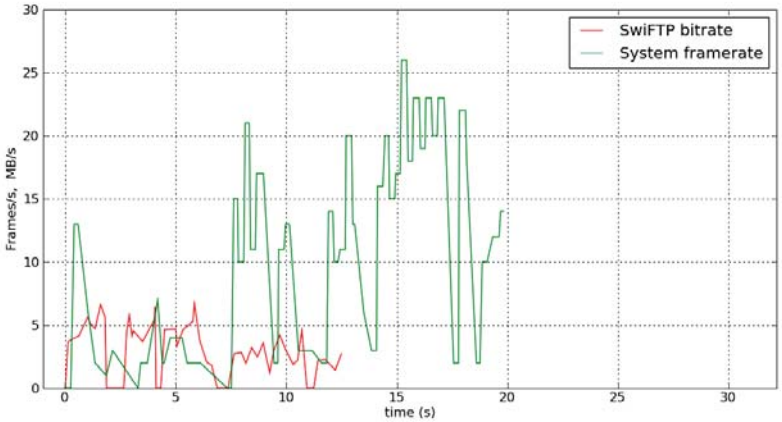

Figure B.126: FPS, BPS, run 5.4

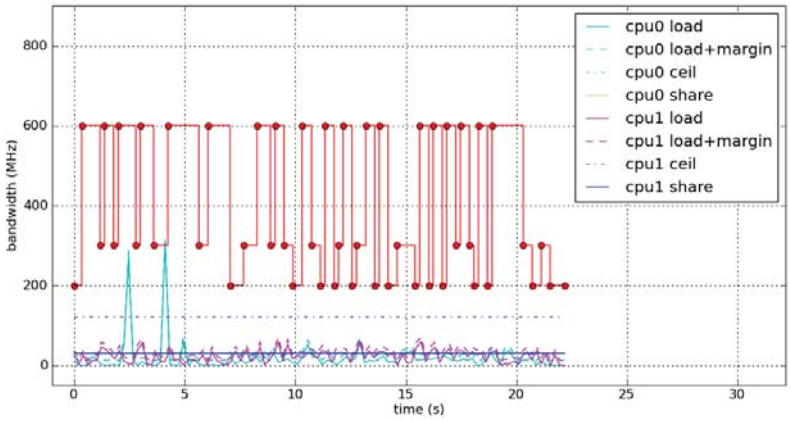

Figure B.127: Root, run 5.4

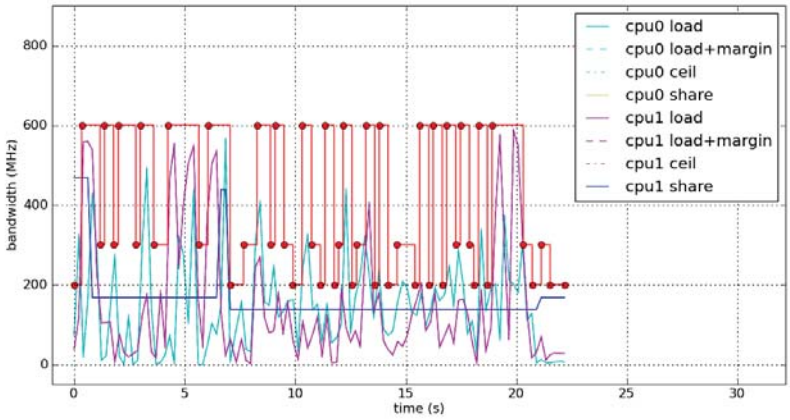

Figure B.128: Slacker, run 5.4

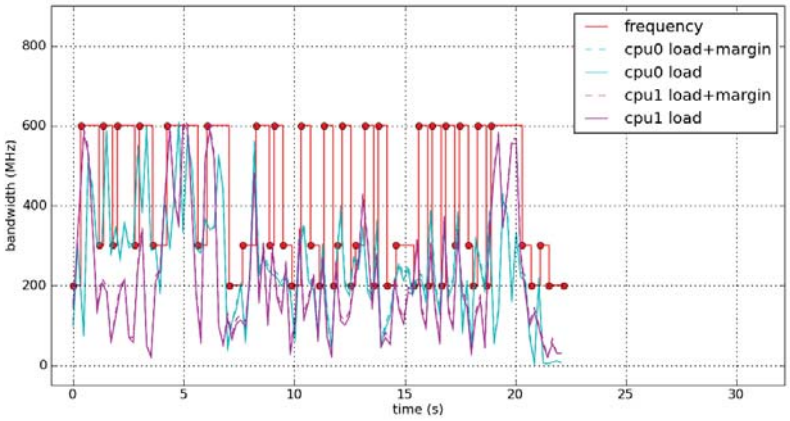

Figure B.129: System, run 5.4

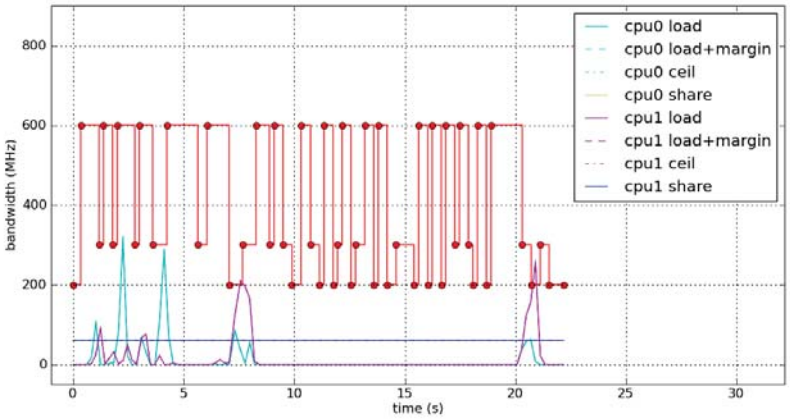

Figure B.130: Self, run 5.4

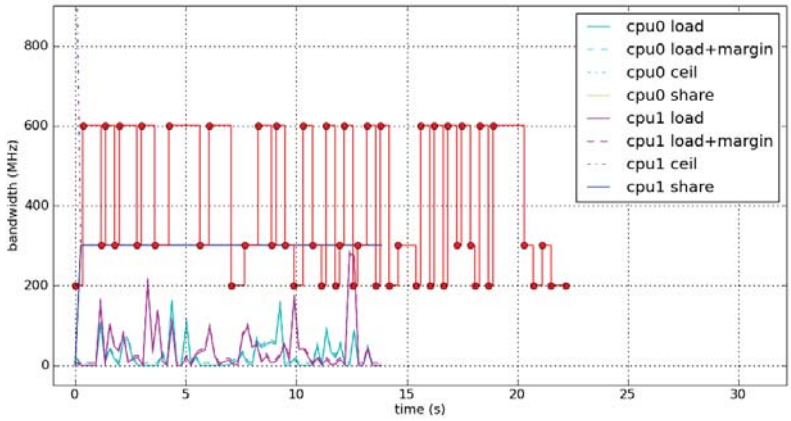

Figure B.131: Calculator, run 5.4

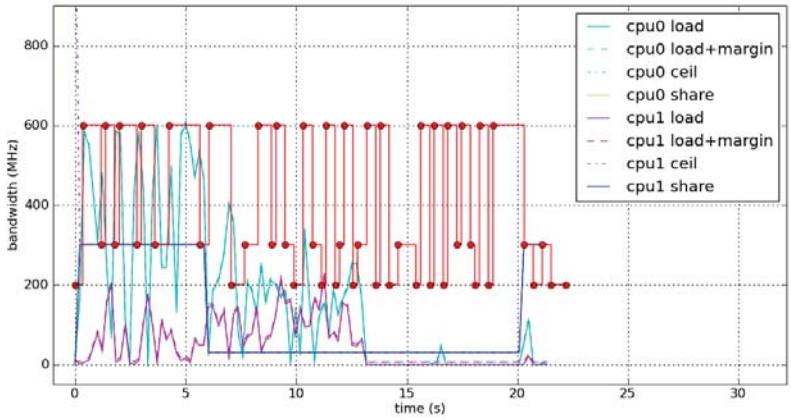

Figure B.132: SwiFTP, run 5.4

B.5.5 Run 5.5

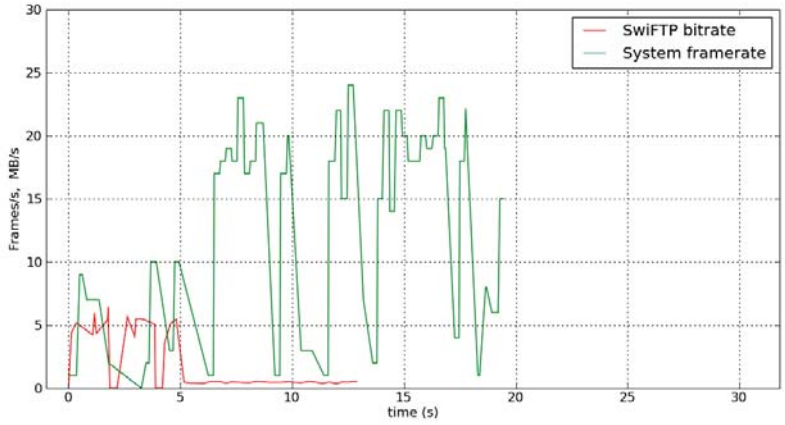

Figure B.133: FPS, BPS, run 5.5

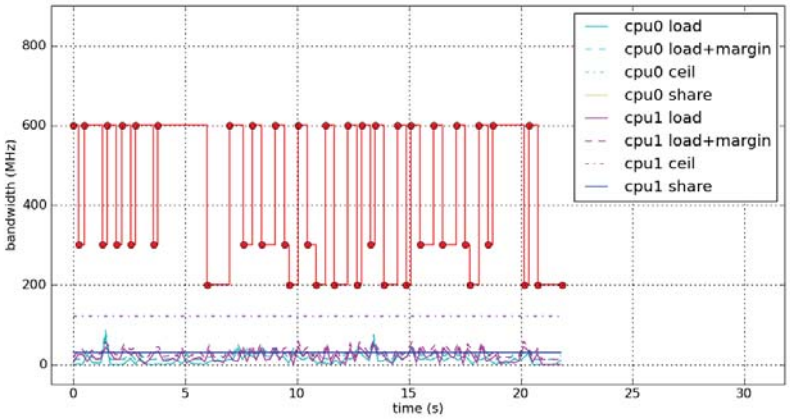

Figure B.134: Root, run 5.5

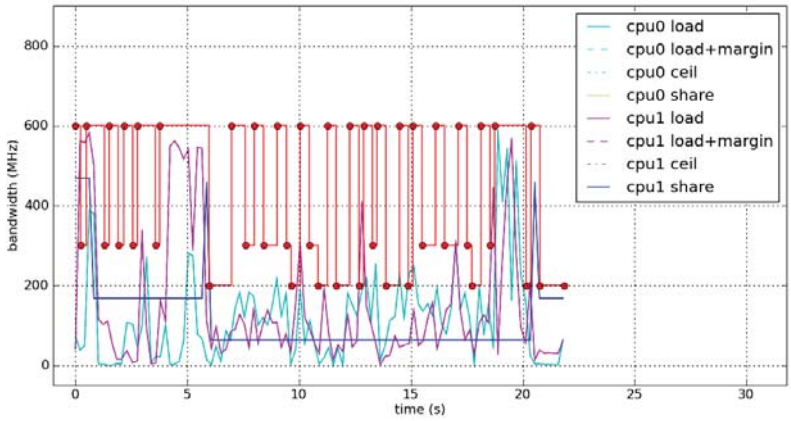

Figure B.135: Slacker, run 5.5

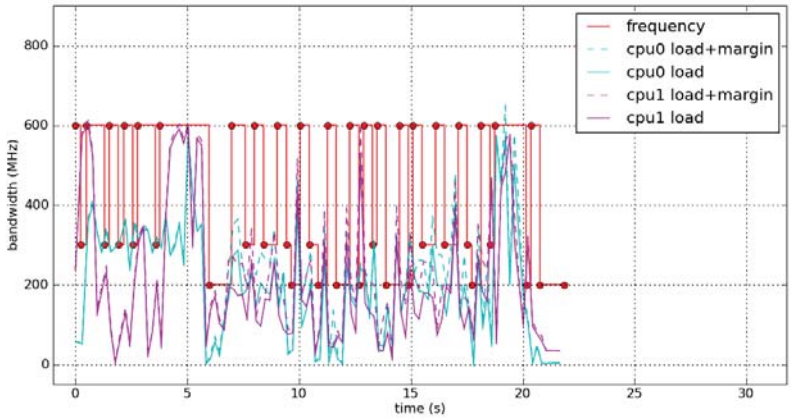

Figure B.136: System, run 5.5

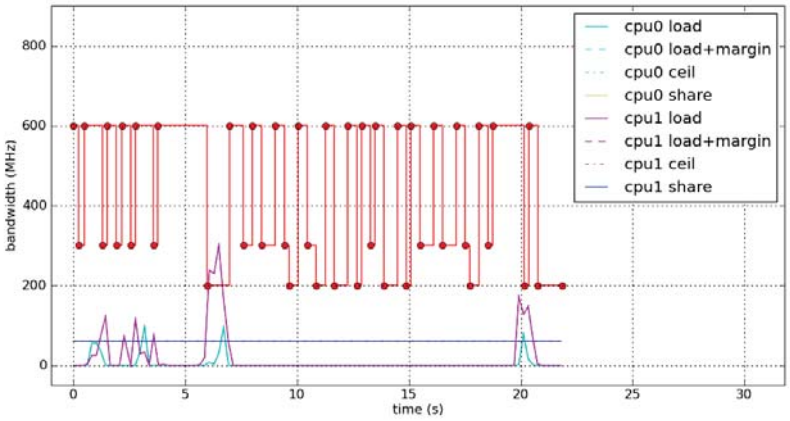

Figure B.137: Self, run 5.5

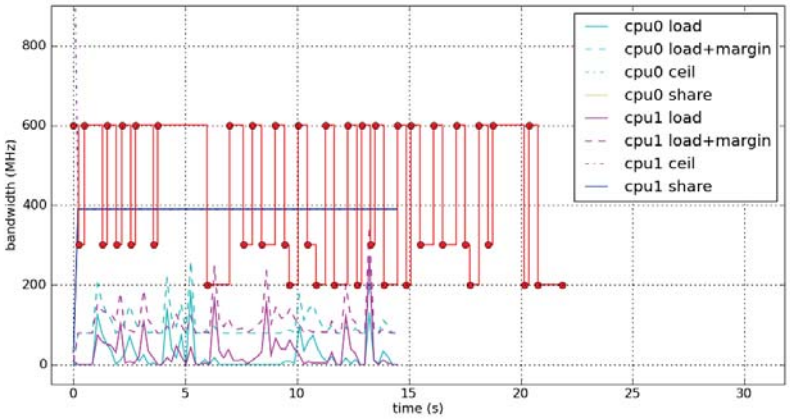

Figure B.138: Calculator, run 5.5

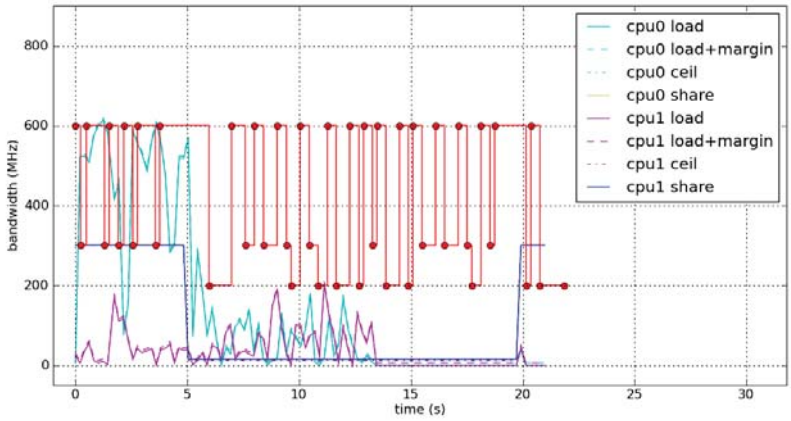

Figure B.139: SwiFTP, run 5.5

B.6 User scenario 6

See Section 7.2.6 for a description of this scenario.

B.6.1 Run 6.1

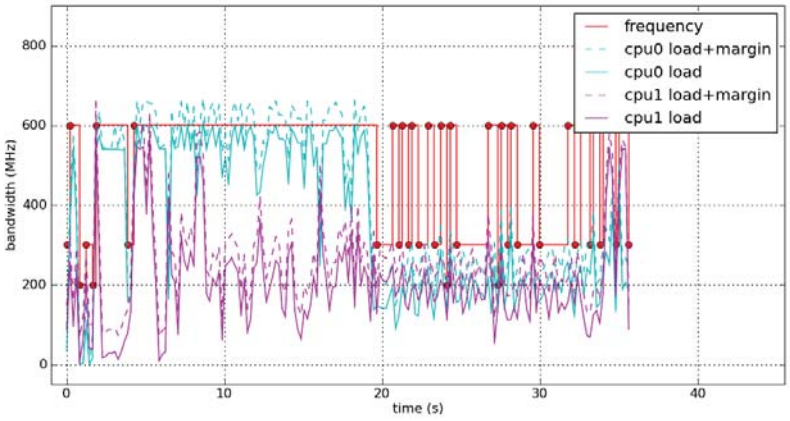

Figure B.140: System, run 6.1

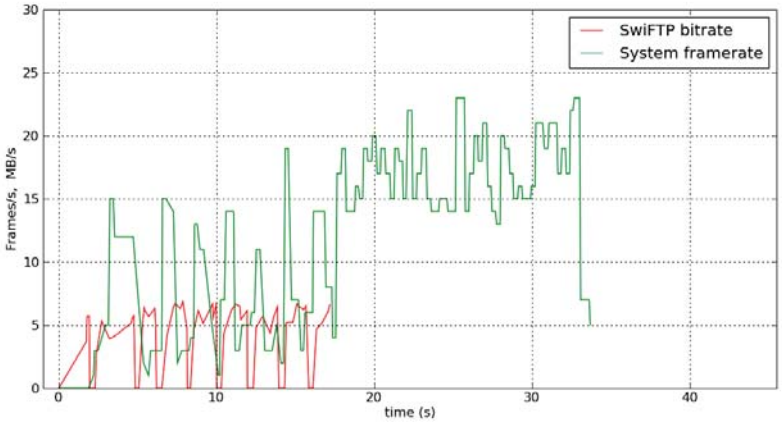

Figure B.141: FPS, BPS, run 6.1

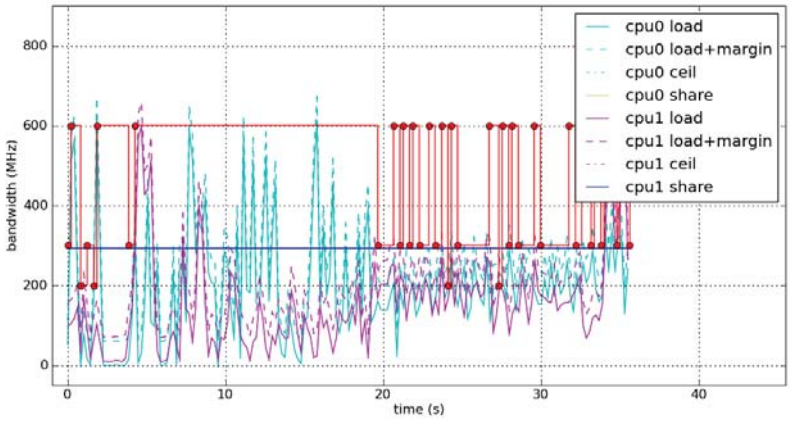

Figure B.142: Root, run 6.1

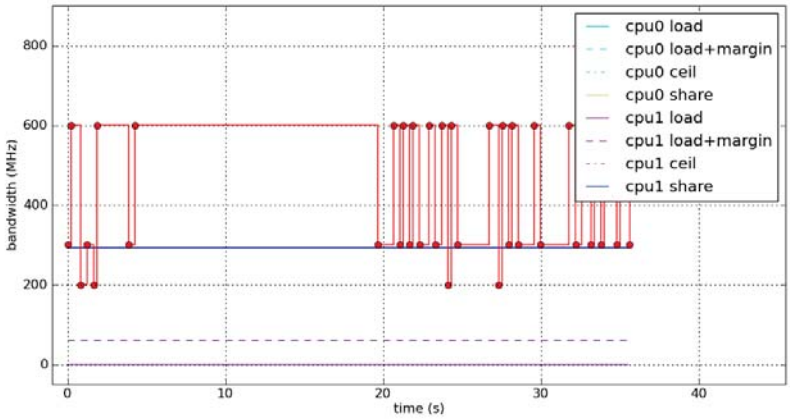

Figure B.143: fg_boost, run 6.1

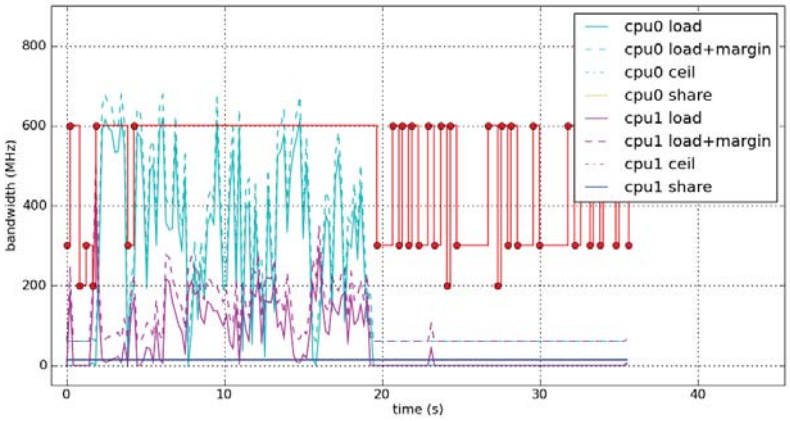

Figure B.144: bg_non_interactive, run 6.1

B.6.2 Run 6.2

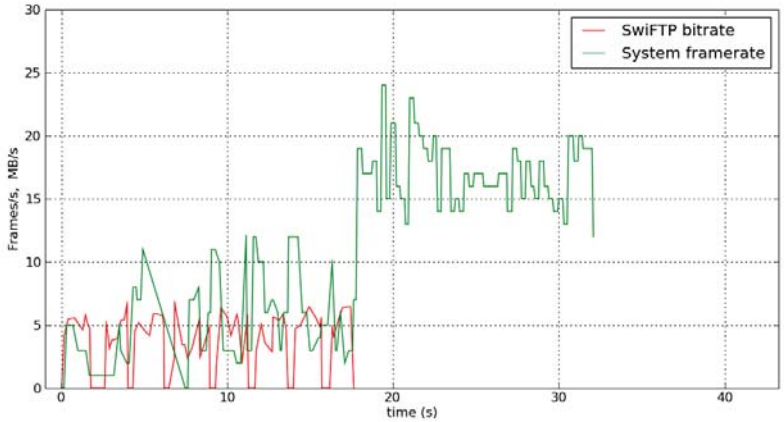

Figure B.145: FPS, BPS, run 6.2

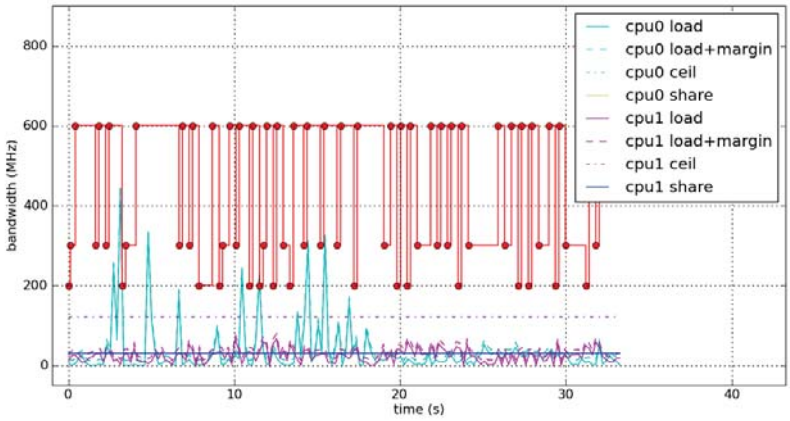

Figure B.146: Root, run 6.2

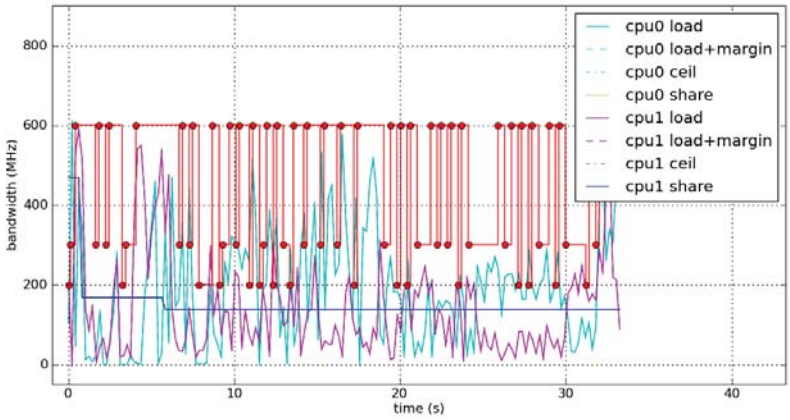

Figure B.147: Slacker, run 6.2

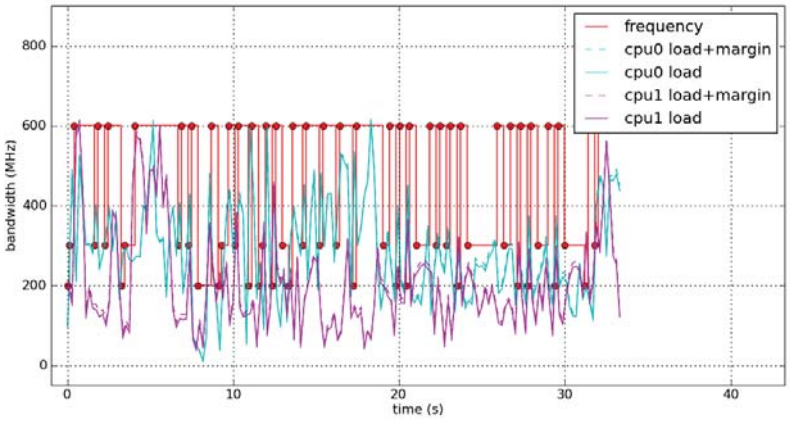

Figure B.148: System, run 6.2

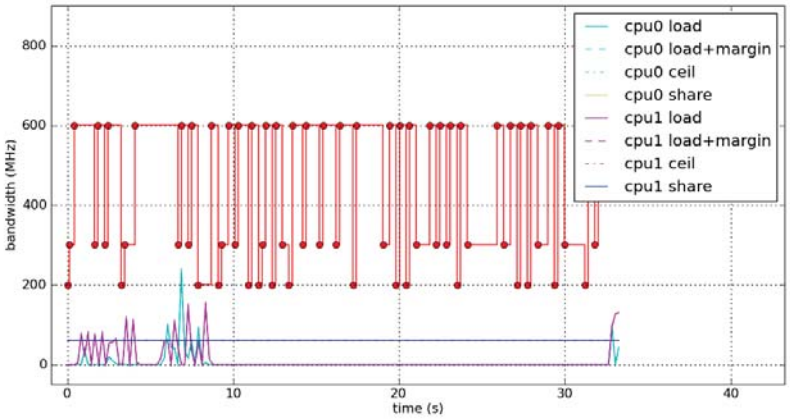

Figure B.149: Self, run 6.2

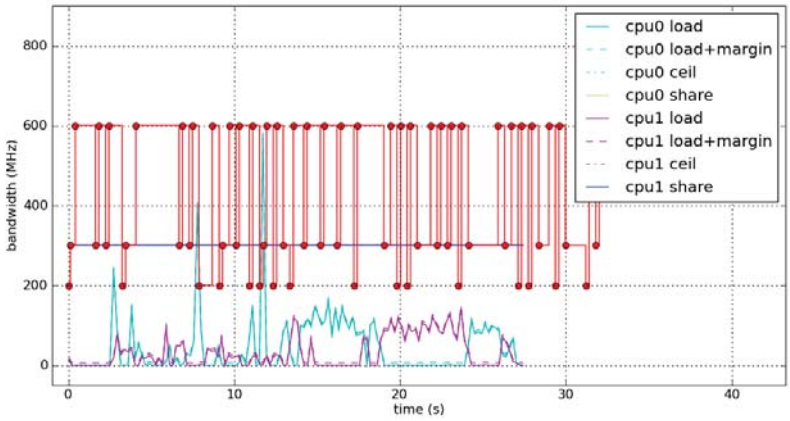

Figure B.150: Missile Intercept, run 6.2

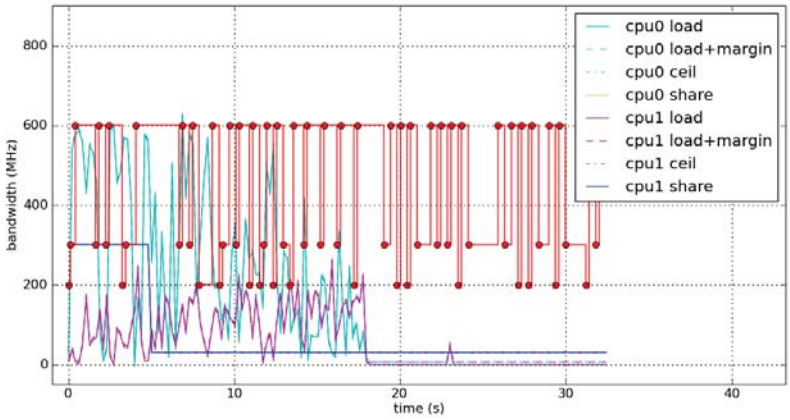

Figure B.151: SwiFTP, run 6.2

B.6.3 Run 6.3

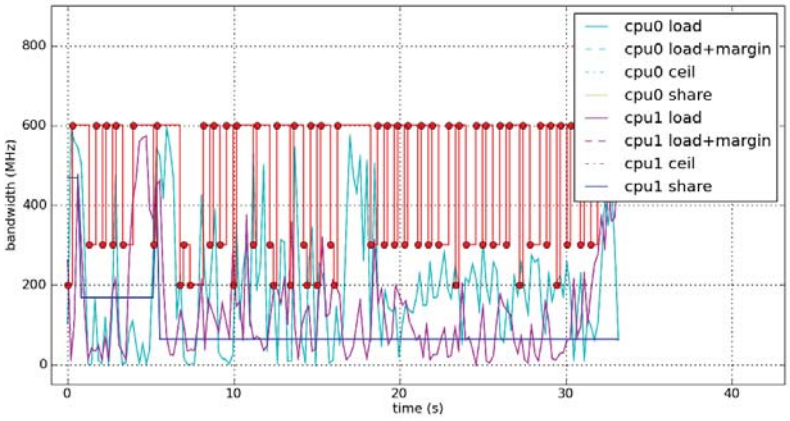

Figure B.152: Slacker, run 6.3

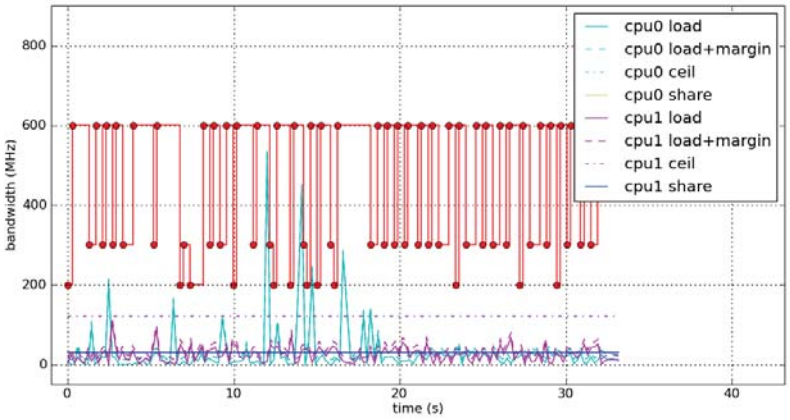

Figure B.153: Root, run 6.3

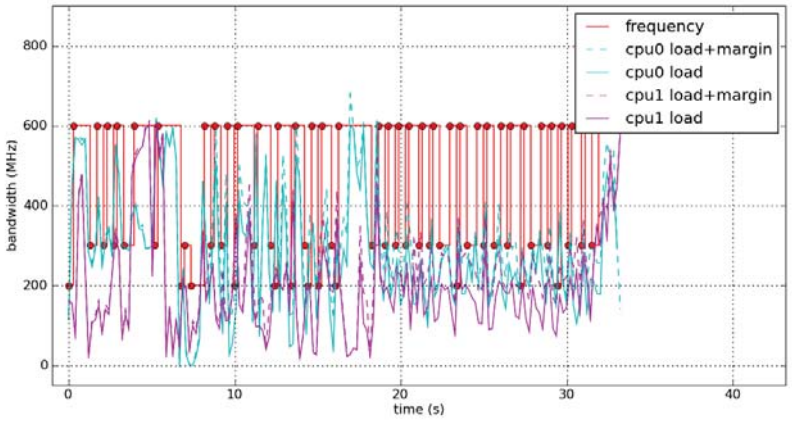

Figure B.154: System, run 6.3

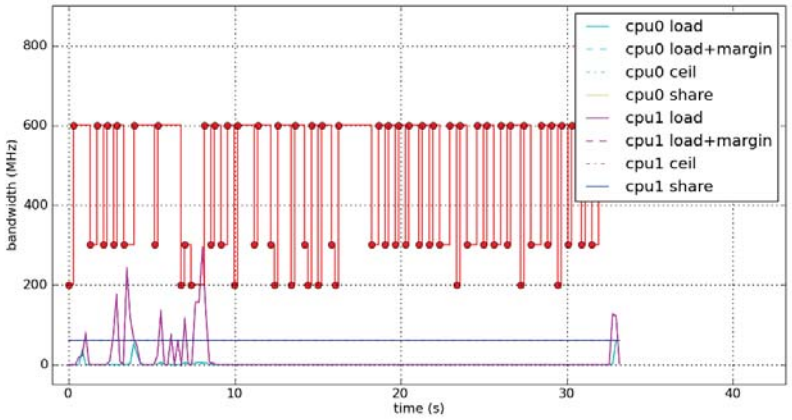

Figure B.155: Self, run 6.3

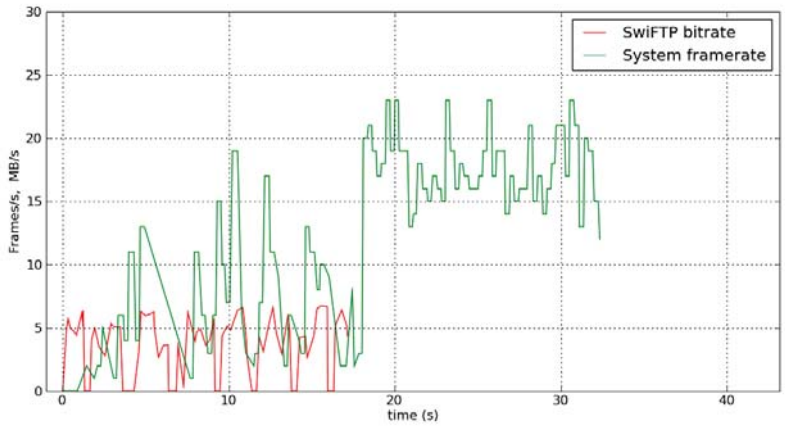

Figure B.156: FPS, BPS, run 6.3

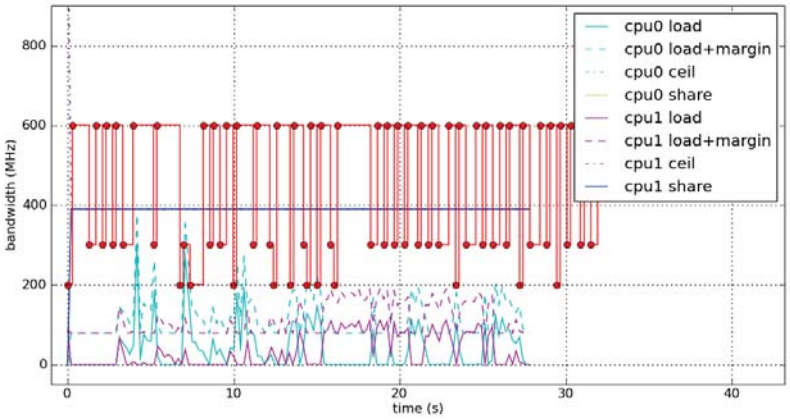

Figure B.157: Missile Intercept, run 6.3

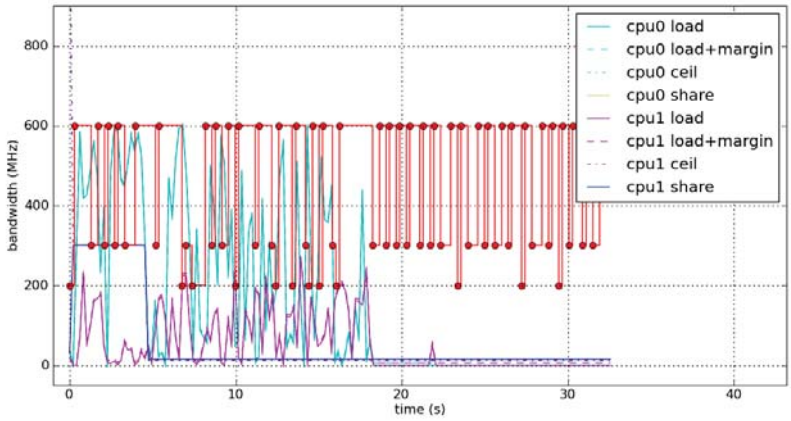

Figure B.158: SwiFTP, run 6.3

B.6.4 Run 6.4

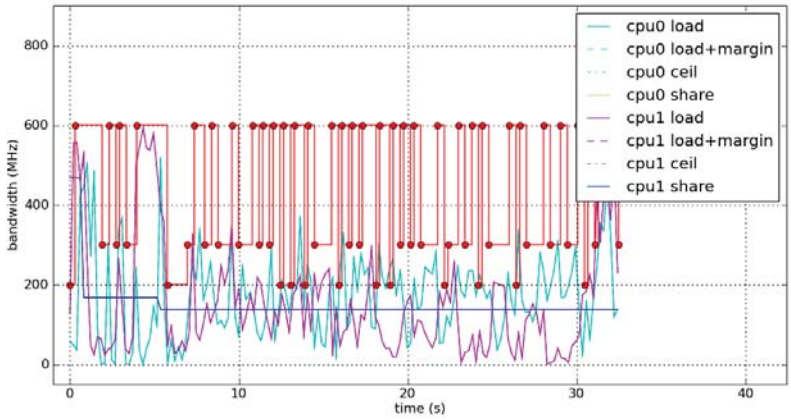

Figure B.159: Slacker, run 6.4

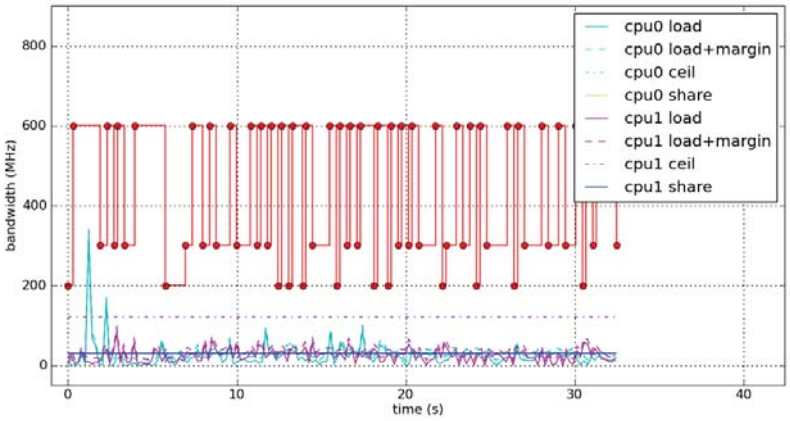

Figure B.160: Root, run 6.4

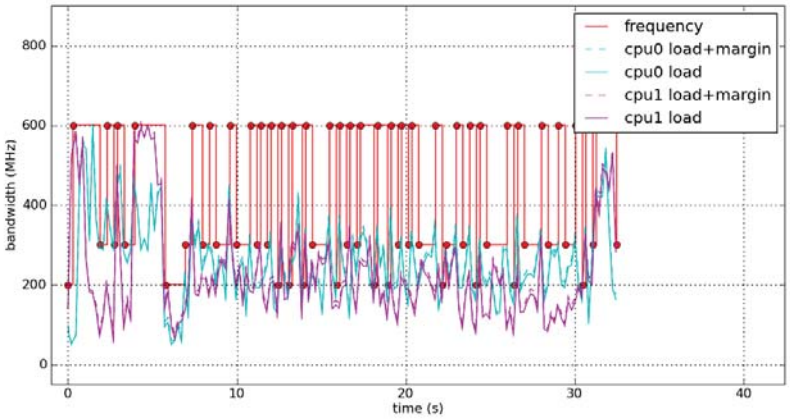

Figure B.161: System, run 6.4

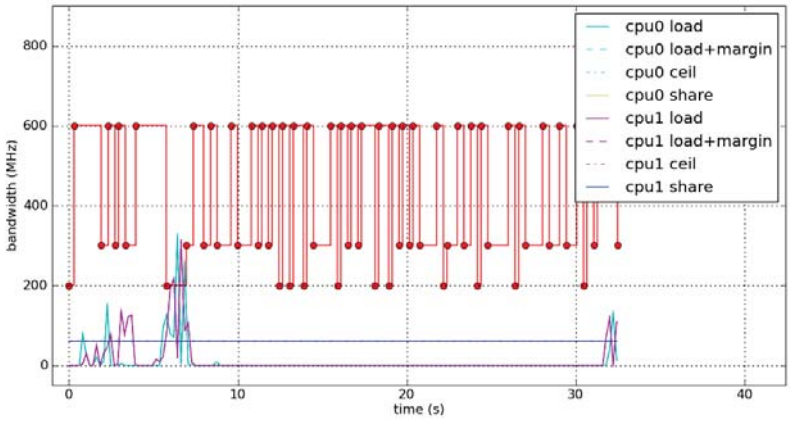

Figure B.162: Self, run 6.4

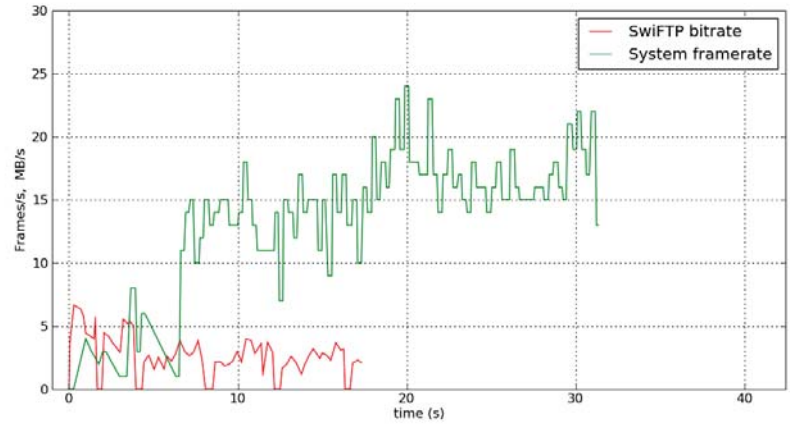

Figure B.163: FPS, BPS, run 6.4

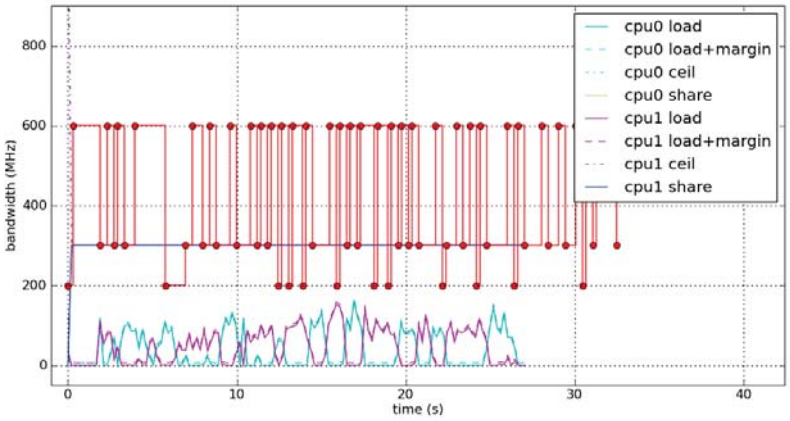

Figure B.164: Missile Intercept, run 6.4

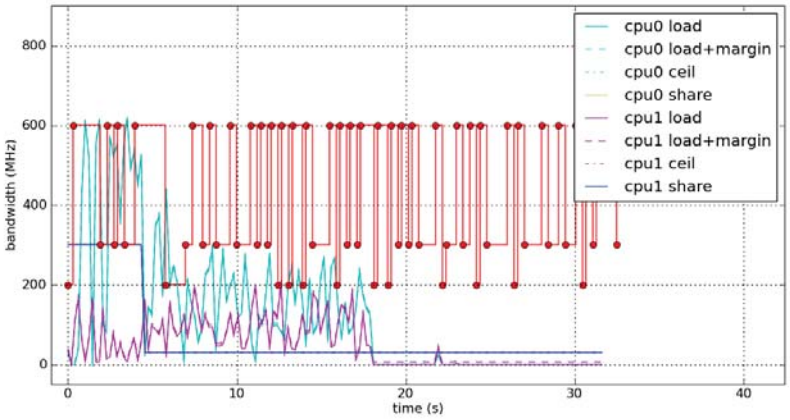

Figure B.165: SwiFTP, run 6.4

B.6.5 Run 6.5

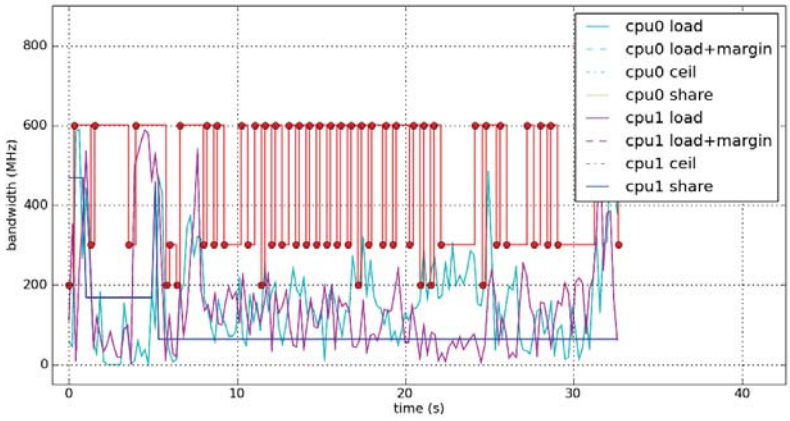

Figure B.166: Slacker, run 6.5

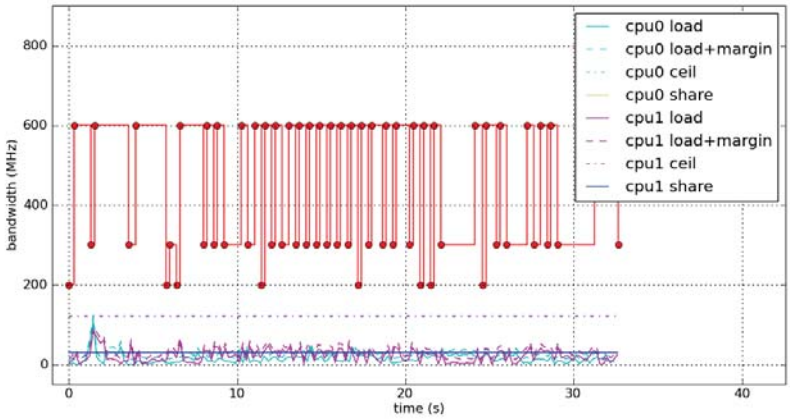

Figure B.167: Root, run 6.5

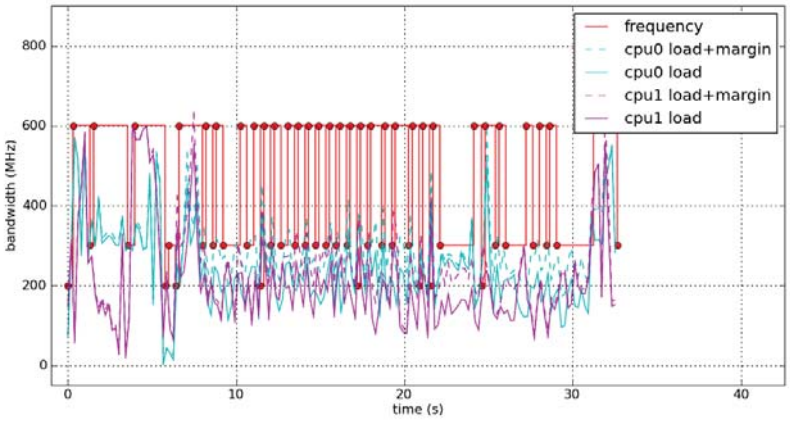

Figure B.168: System, run 6.5

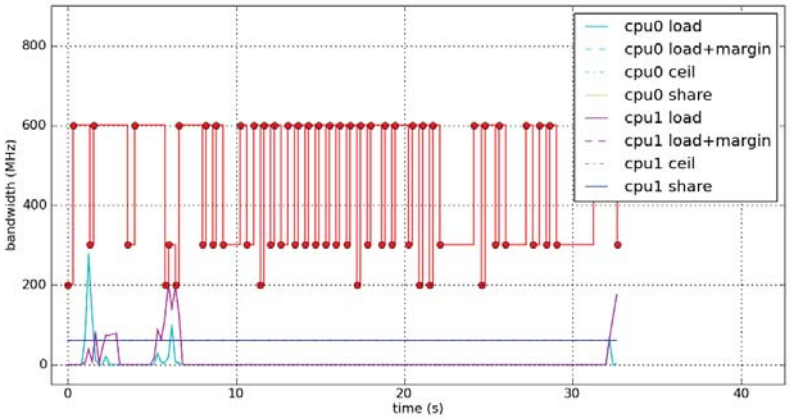

Figure B.169: Self, run 6.5

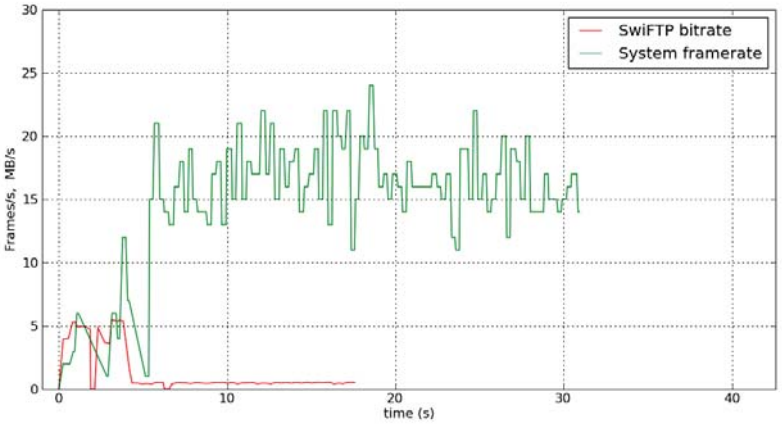

Figure B.170: FPS, BPS, run 6.5

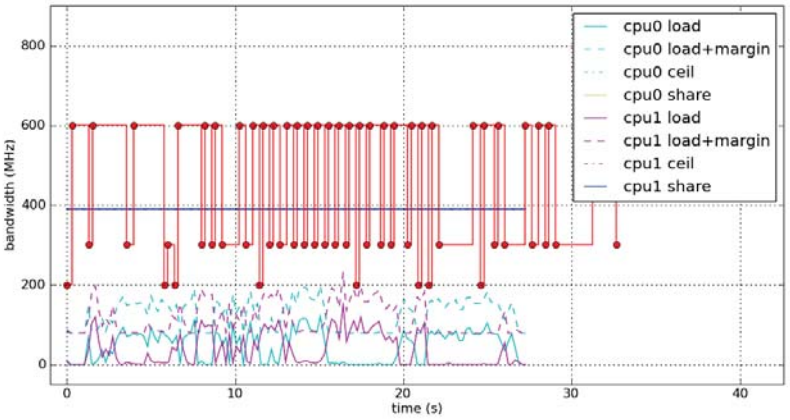

Figure B.171: Missile Intercept, run 6.5

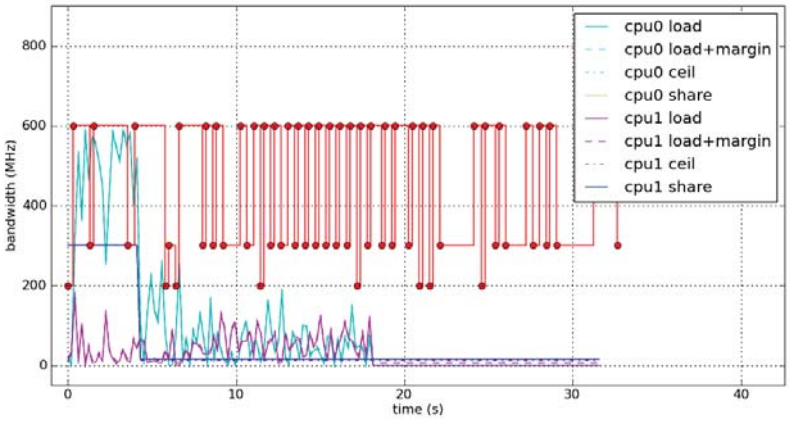

Figure B.172: SwiFTP, run 6.5