CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Time Critical Messaging Using a Real-Time Operating System

Master of Science Thesis in Computer Science – algorithms, languages and logic

ANDRÉAS HALLBERG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GOTHERBURG
Gothenburg, Sweden 2017

Time Critical Messaging Using a Real-Time Operating System

ANDRÉAS HALLBERG

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gotherburg
Gothenburg, Sweden 2017

Time Critical Messaging Using a Real-Time Operating System
ANDRÉAS HALLBERG

Supervisor: Jan Jonsson, Department of Computer Science and Engineering
Examiner: Mary Sheeran, Department of Computer Science and Engineering

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gotherburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

Time Critical Messaging Using a Real-Time Operating System
ANDRÉAS HALLBERG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gotherburg

Abstract

This thesis evaluates the possibility and the effects on performance of running an existing software, not designed for real-time operation, on a real-time operating system. The software being investigated is Unite Communication Server developed by Ascom Wireless Solutions, and is used for wireless communication within the healthcare sector.

The software, which originally runs on Red Hat Enterprise Linux 7 with stock kernel, was installed on a Red Hat Enterprise Linux 7 system (the host) running a kernel patched with the real-time patch RT-Preempt. Performance was then measured by an external computer (the client) connected directly through wired Ethernet. The host was also put under a number of different loads to further amplify the effects of the real-time runtime.

The real-time kernel is shown to give equal or better determinism under all loads, albeit only marginally if one considers how the software is used. The stock kernel is deemed good enough regarding performance and determinism while also being more stable, and migration to the real-time kernel is therefore advised against for this particular software. Furthermore, the standard Intel Ethernet driver for Linux is identified as a substantial source of nondeterminism that should preferably be avoided in networking applications with strict timing constraints.

Nonetheless, switching to the RT-Preempt based real-time kernel showed to be a simple way to increase determinism for this system, especially since no changes to the software were required.

Keywords: real-time, Linux, software porting, performance measurement, operating systems, rt-preempt

Acknowledgements

I would like to thank Ascom Wireless Solutions for providing me with the opportunity to do this master's thesis.

I would also like to thank the Unite team for all their help during the course of this project. Special thanks to my supervisor at Ascom, Mats Andreasen, for continuously providing me with useful feedback.

Finally, I want to thank my supervisor at Chalmers, Jan Jonsson, for advising me throughout the project, and my examiner, Mary Sheeran at Chalmers, for making this thesis possible.

Andréas Hallberg, Gothenburg, February, 2017

Contents

List of Figures

1
Introduction

Real-time systems can today be found almost everywhere in different shapes and forms. Be it everyday consumer electronics such as mobile phones and television sets, or control systems for industrial robots or missile defense systems, they most certainly have some aspect that can be considered a real-time constrained system: the television set must have processed the next video frame when it should be displayed or the video will stutter, and the missile defense system must detect an incoming missile, calculate its trajectory, and give the instruction to fire before destruction of property and lives is inevitable. The consequences of missing the deadline for these two examples may not be considered comparable according to many, but from the viewpoints of the systems themselves, not being able to meet the deadline means one thing only: system failure.

Although the above mentioned systems were examples of embedded systems, real-time performance is also often desired in general purpose systems, including (but not limited to) desktop computers, where functionality such as audio/video playback or Voice over IP (VoIP) communication is expected to work well enough to give the user a pleasant experience.

Ascom Wireless Solutions develops wireless mission critical on-site communication solutions for customers globally, with focus on healthcare. As a manufacturer of products involved in mission critical aspects of a company or business, one of the most important qualities of the products is reliability. As the main advantage of a real-time operating system over a non-real-time operating system is the reliability in response time, Ascom is looking for the possible gains obtainable by migrating one of the main products to a real-time operating system.

1.1 Unite Messaging Suite

The Unite Messaging Suite is a communication platform developed by Ascom that links Ascom messaging systems with mission-critical work processes and tasks. The system is designed for high performance, flexibility, and reliable communication as it is designed to be deployed in areas where such properties are of paramount importance, such as the healthcare sector. The overall structure of the Unite Messaging Suite can be seen in figure 1.1.

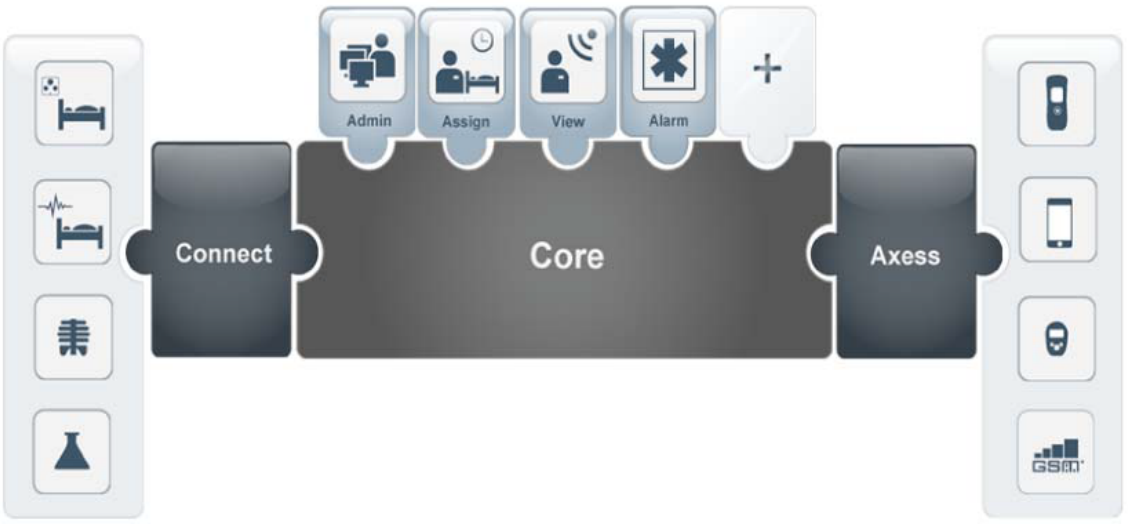

Figure 1.1: Ascom Unite Messaging Suite for healthcare

The Core component in itself consists of two components: The Unite Application Manager, where customer applications are run in a Windows environment, and a second component responsible for handling alarms and ensuring that a message is sent to the intended device. This second component is available in two varieties:

1. The Unite Connectivity Manager. An embedded Linux server platform sold as a complete package including software and hardware

2. The Unite Communication Server. A purely software based product intended as an alternative to the Unite Connectivity Manager

The focus of this project is the Unite Communication Server (from here on referred to as Unite CS, or ”the software”), which will be described further in the next section.

1.2 Unite Communication Server

The Unite CS is intended to run on a machine provided by the customer (physical or virtual). The software itself consists of multiple applications communicating with each other over the Unite protocol, which is a proprietary communication protocol built on top of UDP.

The machine on which the software is run is not limited to running the Unite CS exclusively, but could – although not recommended – be a general purpose server where other processes are running simultaneously. At the moment of writing, Unite CS requires the machine to run Red Hat Enterprise Linux 7 (or other compatible OS, such as CentOS) a general purpose operating system not intended for hard real-time operation, something that brings drawbacks regarding using it as a means to distribute mission critical messages.

Unite CS is not a medical device and therefore need not comply with regulations applicable for medical devices. It is however developed in accordance with certain regulations, for instance IEC 62304, should it become classified as a medical device in the future.

1.2.1 Deployment method and portability

Unite CS is installed through a shell script with appended compressed binary data containing the applications in the suite and dependencies in the `rpm` file format. When the install script is executed the binary data is decompressed and the applications are installed through the `yum` package manager[1]. When installed it uses the init system `systemd` for initialization and starting of the service.

The programs in the suite are written mainly in portable C and C++ and should themselves be rather portable between platforms with a compliant compiler.

1.3 Problem definition

For software such as Unite CS, being able to provide low variance in response times for messages not only makes the product seem more attractive against competing actors, shorter response times can also improve the outcome of critical situations where time is of essence, for example medical emergencies. Running Unite CS in a real-time operating system also helps by minimizing risks for problems like priority inversion, or giving too much CPU time to non-real-time tasks, such as system updates or logging functionality. As Unite CS is run on a machine provided by the customer, and Ascom not being able to control what other applications the customer is running, evaluating how well the software is performing in such environments, both with and without a real-time operating system, is of great interest.

First, a suitable RTOS must be chosen for the task and a few criteria must be accounted for in the selection of the operating system. After a set of OSes have been evaluated, one will be selected to be used for the remainder of the project. A strong contender is already suggested by Ascom since before the start of the project, namely Red Hat Enterprise 7 with a real-time kernel, as the current solution is deployed on Red Hat Enterprise (non-real-time variant), but this suggestion should be evaluated on the same criteria as the alternatives.

After an operating system has been selected, the next step is to make the necessary changes to the application and deployment procedure in order to install the software in said operating system.

Upon successful installation in the new environment, the software needs to be adjusted in order to exploit the real-time advantages now available. This will hopefully prove to bring measurable improvements compared to the system running in the non-real-time environment, which leads to the final task for this work.

Measuring the performance of the final product will provide the hard data on how well this particular system responds to being run in a RTOS. In order to benchmark

[1]The `yum` (Yellowdog Updater, Modified) package manager is the default package manager for many `rpm` based Linux systems, including Red Hat Enterprise Linux [1] and CentOS [2].

the system, a test environment needs to be developed. The measurements should preferably be done by an external computer connected to the host machine through wired Ethernet in place of WiFi as a means to achieve reasonably consistent delays.

1.4 Purpose

The main aim for this thesis is to investigate the possible advantages of migrating the Unite Communication Server software to a real-time operating system (RTOS) and optimize it to utilize the real-time constructs available in the chosen OS. Advantages and disadvantages of such a migration will be addressed, not only from a performance perspective, but also from the perspective of the software engineer, i.e. how viable such a migration would be in practice.

1.5 Context

The deployment of a software in an environment is an important step when developing software. In order to assure customer satisfaction with the product and to increase overall perception of the quality of the product, one needs to ensure that the product is adjusted according to the needs of the customer [3, 4]. Although several models have been developed to provide guidelines on how to build the deployment architecture, Unkelos-Shpigel and Hadar [3] struggled to find any specific factors affecting the general quality of the solution. Talwar et al. [5] goes on to say that no optimal deployment method exists, but that the best method of deployment is the one that matches the deployment needs the closest, be it a manual deployment for smaller systems or systems where configurations rarely change, or scripts for systems where configurations are more likely to change.

Tyrrell et al. [6] explored the possibility of migrating an off-line medical system, in this case a frame-rate vision system designed for computer-assisted laser retinal surgery, to a real-time implementation running under an operating system running the Linux kernel. They see the problem of porting large, complex code bases through rewriting of the software, a process that is very costly, not only short term, but also longer term, as two separate code bases need to be kept in sync through changes and updates over the lifespan of the application. Instead, they propose the use of the Linux kernel, and a virtual device driver as a vessel to bring the user space executable into kernel space, thus making execution of the application non-preemptible, in addition to giving it immediate access to system level services. Their solution proved to reduce the variance of observed CPU cycles for processing a single frame by two orders of magnitude compared to running in user mode, showing that it is indeed a viable method for this use case. However, with frame-rate vision systems not generally requiring any asynchronous processing, and this solution lacking a sophisticated scheduling mechanism between real-time tasks, using this technique will not be satisfactory in a real-time system where parallel execution is required. Note that the Linux kernel have been preemptible since version 2.6, although this can be

disabled if desired [7].

As for measuring the performance of a real-time operating system, it is not a simple task, and is by some said to not be possible without the use of external hardware [8]. Aroca and Caurin measured a set of real-time operating systems using a function generator to generate interrupts, and an oscilloscope to record the responses from the systems in order to measure the response times. It is concluded that the Linux kernel, and RTAI Linux, which is an extension to the Linux kernel, are good candidates for real-time operation. This conclusion is further supported by [9] where a similar technique was applied, and RTAI Linux and Xenomai (another extension to the Linux kernel) are deemed adequate for control systems used for studies on thermonuclear fusion.

Despite being able to acquire hard measurement data on a technical level, deciding which RTOS to use for mission critical tasks remains a rich field, subject also to subjective aspects, such as the developers' familiarity with the OS, and their willingness (or unwillingness) to migrate to a new OS [8].

2
Theory

2.1 Software portability

The existence of multiple computer architectures and operating systems makes portability an important aspect of software development today. Having a product that is portable enables simpler development and distribution for different computer systems, thus increasing the potential customer base while lowering development and maintenance effort.

Creating portable software essentially means designing the software according to some standard API. Consequently, the software would be compatible with each platform that implements this API [10]. The platforms themselves may be *source compatible*, where a recompilation of the source is necessary for each platform the binary should run on, or *binary compatible*, where the same compiled binary can be run on all platforms without any recompilation [11].

For instance, the free Linux distribution CentOS is designed to be binary compatible with the commercial distribution Red Hat Enterprise Linux (RHEL), meaning any application developed for RHEL can be copied in binary form to a CentOS system and be expected to run correctly.

By developing in a portable programming language, such as C, the number of source code compatible platforms for an application can increase considerably, and as long as one adheres to the C standard the code is usually[2] portable. However, venturing outside the standard is often necessary in real-world applications. Tanaka et al. [12] identifies parts of code not portable as *porting impediments*. As the porting of an application mainly consists of rewriting non-portable segments, they consider ridding the source from such sections an important step towards increased portability for applications. Examples of such porting impediments obtained through their research include (but are not limited to):

- Use of implementation-dependent system calls
- Use of assembly language
- System-specific functions (e.g. regarding memory protection)

Such porting impediments are often necessary from a performance perspective, and replacing them with portable code usually has a high performance penalty, as each implementation of the standard defining the portable code must implement the most general interpretation for each function, thus performance inevitably suffers as portability increases [10]. However, as computing power increases, the practical

impact of this portability-performance trade-off is likely to be visible mostly in more demanding applications.

2.2 Real-Time System

Real-time systems differ from non-real-time systems in that not only the result of a computation determines the correctness of the system, but also in what time the result can be computed. If an answer cannot be computed within a set deadline, the answer loses its usefulness; more specifically, the worst case latency for the system must be guaranteed, rather than maximizing the throughput [14]. A Real-time system in itself can be classified as one of the following:

Hard real-time: For a hard real-time system, being able to perform some task before a set deadline is crucial to the system's function, and a late answer is regarded as system failure [15]. Such systems include pacemakers [16], avionics control systems, and ABS systems [17].

Soft real-time: In contrast to hard real-time systems, missing the deadline in a soft real-time system can be acceptable to a certain degree, but usually degrades the usefulness of the result, and therefore the quality of service for the system as a whole. Examples of soft real-time systems include games and vehicle comfort functionalities such as air conditioning [17].

2.3 Operating system

The operating system exists as a layer between the user and the hardware on a computer system and is responsible for providing said user with a more friendly environment in which the task of direct low-level communication with the machine is shifted from the user to the OS. Furthermore, many modern operating systems also take upon themselves the task of providing a multitasking environment, in which multiple tasks can be executed simultaneously, either through time-sharing on a single processor or real parallel execution on multiple processing units [18].

Operating systems have not always looked and functioned like they do today, but have evolved through a number of stages – what Chauhan calls generations – since the inception of the computer. In the dawn of the computer age programmers wrote programs in machine language on punch cards that talked directly with the hardware. This first generation (1940s - 1950s) was completely devoid of any kind of operating system as we know them today, and had the user responsible for even the lowest level operation [18].

The second generation (1950s - 1960s) brought higher level programming languages, as well as compilers and magnetic tapes for secondary storage. Initially, manual operation was still needed for running a user program: A compiler tape

[2]Discrepancies may exists between various implementations of the standard. For instance, the standard only defines the minimum size limits for the non-exact-width integer types, leaving the compiler to ultimately decide the sizes [13].

would be mounted on the system, compile the user program read from a card to assembly; another tape, containing the assembler would then need to be mounted for generating executable machine code from the assembly, which could then be loaded and executed. If one considers running a series of programs, possibly written in different languages necessitating different compilers, these manual interventions would render the CPU idle for long periods of time. This idle time posed a problem, as computing time was very expensive, and time spent idle is wasted time if other tasks are waiting to be executed. The users also had to, in addition to writing the programs, learn how to operate the machinery. As a means to mitigate these problems an operator could be employed that was trained for the manual tasks associated with computing. This operator could be seen as a precursor to the (automatic) operating system. Eventually, automatic job switching was introduced by utilizing a control monitor that lay resident in memory, and providing instructive information for each task on how it should be executed [18].

The third generation (1960s - 1980s) introduced concepts such as multi-programming, where the CPU would execute another job while waiting for I/O, and multi-user time-sharing systems where some fraction of each unit of CPU time was given to each user of a system, resulting in a system that felt responsive to the individual user's input. UNIX is an example of such an operating system developed during that time, and operating systems like it came to dominate computing for a long time [18].

The fourth generation (1980s - present) saw the birth of graphical user interfaces and a big shift toward user friendliness. Operating systems such as Windows, Mac OS and Linux belong to this generation [18].

For multitasking systems a certain part of the OS is of particular interest for this thesis, namely the task scheduler, which is responsible for allotting processing time to runnable tasks.

2.3.1 Task Scheduling

In a multitasking system the task scheduler's responsibility is to make sure that all tasks scheduled to run can run in a timely manner. More specifically, it needs to select at most one ready task to be run at each processing unit according to some strategy that ensures the system's function [17]. To allow the simultaneous execution of a number of tasks greater than the number of available processing units, it also needs to provide a mechanism for storing the current state of the processing unit for some task to allow it to resume execution at a later point in time on the same or on a different processing unit, a process called context switching. Through context switching, a single processing unit can be used to execute multiple tasks simultaneously by giving each task some time t where it can run uninterrupted on the CPU, after which it will be switched out for some other task and rescheduled for execution at a later point in time [18].

To aid the scheduler a task can be in a number of states [17]:

- **Ready**

The task is ready to run and is waiting for the scheduler to allow it to execute.
Possible future states:

  - Running (through promotion)
  - Terminated (through deletion)

- **Running**
  The task is executing on one of the system's processing units.
  Possible future states:

  - Ready (through preemption)
  - Blocked (while waiting for resource, or ordered to sleep)
  - Terminated (through termination)

- **Blocked**
  While the task is waiting for some resource or timer event.
  Possible future states:

  - Ready (when resource is acquired or timer event fired)
  - Terminated (through deletion)

- **Terminated** Final state for a task. From here it cannot move to any other state.

The scheduler will choose only from the tasks in the ready state which should be the next to be moved to the running state. The selection of the next task is based on the current state of the system and the states currently in the ready state. For this purpose an efficient selection algorithm must be employed, as time spent on scheduling cannot be utilized for actual work, thus such housekeeping belongs to what is called the processor overhead.

When the scheduler is allowed to interrupt a task t_1 in favor of another task t_2 without needing explicit permission from t_1 in order to do so, it is said to be a preemptive scheduler, and the act of interrupting t_1 in favor of executing t_2 is known as preemption (t_2 preempts t_1) [19].

2.3.2 Real-Time Operating System

Like the name suggests, a real-time operating system is an operating system intended for real-time operation. It differs from a non-real-time OS in that it does not emphasize flexibility or speed, but rather determinism. For an operating system this includes having a predictable interrupt latency, i.e. the delay from when an interrupt is generated, to when the associated interrupt handler is executed, and context switch latency, i.e. the delay associated with a context switch [20], and overall low processor overhead [21].

While many proprietary real-time operating systems are available, such as VxWorks and Integrity, there are also free and open source alternatives such as real-time adjusted varieties of Linux available [20].

2.3.2.1 Real-Time Linux

With the rising popularity of the free and open-source kernel Linux, there has also been an increasing interest in using it as a real-time operating system. The Linux kernel, not originally intended for hard real-time operation has seen many attempts at making it a kernel suitable as such. Different approaches have been taken in order to make this a reality [20].

RTAI and Xenomai are two Linux extensions aiming to bring real-time performance to the Linux kernel through a provided real-time API. They both employ a dual kernel approach, where a thin nanokernel is run below the Linux kernel, positioning itself between the hardware and the kernel. The nanokernel also runs the extension in question. Hardware interrupts are then intercepted by the nanokernel before being propagated further. The real-time extension is positioned before the Linux kernel in the pipeline for receiving these interrupts, giving it a chance to react to the interrupts before involving the Linux kernel, thus enabling deterministic response times for real-time tasks. Non-real-time tasks are simply propagated further to the Linux kernel where they can be scheduled according to which ever scheduler is in effect there [9].

Xenomai is also available in a single kernel configuration since version 3, where it relies on the real-time capabilities of the Linux kernel as is. This configuration is usually used in conjunction with the RT-Preempt patch [22], which will be described further in the next section.

RT-Preempt is a patch originally created by Ingo Molnar (employed by Red Hat at the time of writing) for the Linux kernel aiming to make it suitable for hard real-time operation. The patch works by making the kernel fully preemptible through some changes to the source code, such as exchanging non-preemptible spinlocks with preemptible mutexes. It also implements priority inheritance for in-kernel synchronization constructs, thus preventing priority inversion in the kernel [23]. A clear advantage of the RT-Preempt patch over the above mentioned extensions is that the RT-Preempt kernel is binary compatible with the standard Linux kernel, meaning all user applications continues to work without any recompilation; the real-time benefits propagates to the standard Linux APIs [24]. Conversely, one could also run the real-time application developed for running on the real-time kernel on the standard Linux kernel without any modification or recompilation.

Measurements by Brown and Martin [25] show that the Linux kernel with the RT-Preempt patch does not perform as well as Xenomai, although the penalty of using RT-Preempt might be more or less of a problem depending on the real-time requirements for the system [22]. The RT-Preempt patch is at the time of writing also under heavy development [23], suggesting previous measurements might lose their relevance over time.

Aside from explicit real-time extensions it should also be noted that the standard Linux kernel enables the exploitation of real-time priorities as is, giving the user better control over the responsiveness on a per-thread level for a system. Although not performant enough for being used with feedback control systems for fusion devices, Barbalace et al. [9] praised the standard Linux kernel for being highly performant and suitable for smaller dedicated real-time systems.

2.4 Certification

2.4.1 Certification for mission critical systems

Kim [26] argues that stricter government imposed requirements for certification may not lie far ahead when it comes to safety-critical software systems. As hardware has become increasingly more reliable in the last few decades, it is now only natural that the general public turns its eye towards the software systems, in particular software incorporated into systems where total reliability is imperative to the safety of the people which are under the direct control of said system. However, such certifications – what Kim refers to as Quality-of-Service (QoS) certifications – are far from the norm, something he argues stems from the difficulty of testing a system to the point where all worst case paths of execution are accounted for.

One of the most cited examples of mission critical systems is that of avionics, where a system error could compromise safety for large numbers of people. The avionics industry have therefore together with regulatory authorities defined a strict certification standard for software used in mission critical systems within the field of avionics. The standard, called DO-178 and its European counterpart ED-12 describes guidelines regarding software life-cycle processes, and puts strong emphasis on verification of the software [27]. In contrast to earlier versions, the latest version of the standard, DO-178C, also allows formal verification of programs in place of other forms of testing.

2.4.2 Certification for medical devices

Healthcare, being among industries where unsatisfactory operation can result in severe degradation in quality of life, or even death, it is an industry subject to heavy regulation. With software playing an increasingly larger role in medical devices and healthcare, the complexity of, and reliance on said software has grown accordingly [28]. To ensure the safety of patients, the development process of medical device software and its associated activities regarding regulation and certification poses many challenges for the manufacturers.

A famous example on why such regulations are necessary is the Therac-25 incidents, where between June 1985 and January 1987 six people received massive overdoses of radiation from a computerized radiation therapy machine (the Therac-25) due to faulty software, resulting in death or serious injury for the patients treated. The cause of the accidents can't be attributed solely to the coding errors, but also to poor software engineering practices, such as inadequate documentation and insufficient testing. Furthermore, excessive confidence was put in the software's correctness; that software cannot fail, leading to the dismissal of the machine (and it's software) being the reason for the individual accidents, resulting in further accidents [29].

It's important to note that only software classified as a medical device is subject to these regulations. In the EU, the European Commission describes a medical device in directive 2007/47/EC [30] (which came into force March 10 2010 [28]) as follows:

”medical device” means any instrument, apparatus, appliance, software, material or other article, whether used alone or in combination, together with any accessories, including the software intended by its manufacturer to be used specifically for diagnostic and/or therapeutic purposes and necessary for its proper application, intended by the manufacturer to be used for human beings for the purpose of:

- diagnosis, prevention, monitoring, treatment or alleviation of disease,
- diagnosis, monitoring, treatment, alleviation of or compensation for an injury or handicap,
- investigation, replacement or modification of the anatomy or of a physiological process,
- control of conception,

and which does not achieve its principal intended action in or on the human body by pharmacological, immunological or metabolic means, but which may be assisted in its function by such means.”

The directive also states [30]:

”Stand alone software is considered to be an active medical device.”

as well as [30]:

”It is necessary to clarify that software in its own right, when specifically intended by the manufacturer to be used for one or more of the medical purposes set out in the definition of a medical device, is a medical device. **Software for general purposes when used in a healthcare setting is not a medical device**.”

For software classified as a medical device, the directive continues to explain under what circumstances such software should be developed [30]:

”For devices which incorporate software or which are medical software in themselves, the software must be validated according to the state of the art taking into account the principles of development lifecycle, risk management, validation and verification.”

Where ”state of the art” is considered development in accordance with the standard IEC 62304 along with its aligned standards [28]. The IEC 62304 [31] itself describes software life cycle processes for medical device software, and has been recognized as consensus standard also by the United States Food and Drug Administration (FDA). Without developing the software in compliance with these standards, one cann<ot obtain the CE mark required for legal marketing and distribution in the EU [32].

As can be concluded from the above, there are no explicit certifications regarding the real-time performance for medical devices. Rather, the performance of the systems needs to be assured in rigorous testing of the device, which is in itself a regulated activity. A certified real-time operating system might help with reaching the desired determinism of the system if it so requires thus simplifying the development, but if satisfactory performance can be obtained from a non-certified real-time system, or even a non-real-time system, the usage of such a system will pose no regulatory or legal problems.

3
Methodology

3.1 Selection of real-time operating system

3.1.1 Evaluation of operating systems

First, a rough list of available real-time operating systems was compiled, from which the obviously unfit for the project were filtered out, leaving a manageable list of operating systems on which a deeper evaluation was performed. The OSes were evaluated primarily on the following criteria:

- Performance (or rather response time)
  What guarantees can be given regarding deterministic response for the system, and how performant is the run time and scheduler? It is possible that a less strict or less performant RTOS could be chosen if it satisfies the other criteria with high marks.
- Ease of porting
  The chosen system needs a rather high degree of compatibility with the current codebase in order for the project not to turn into a full-scale porting job. Time should preferably be spent on optimizing the current code (i.e. adding real-time related constructs and hints), not translating the majority of it to some DSL, or worse yet, some other programming language.
- Ease of deployment
  Having to spend as little time as possible dealing with the installation on the new system allows for more time to be spent on the main problem.
- Licensing
  The monetary aspect is of course of importance when selecting the operating system. A very expensive or overly restrictive license may disqualify a contender before even evaluating the other criteria.
- Certification
  The OS having certain certifications could benefit the product in several ways: It can be used purely as a sales argument, but it could also allow the system to be used in areas not before possible due to the lack of certification. Of course, the software would also need to be certified for this to be considered relevant, and this lies beyond the scope of this project.

3.1.1.1 Red Hat Enterprise Linux 7 with Real-Time Kernel

Red Hat Enterprise 7 with real-time kernel was the most obvious choice, since RHEL7 (without the real-time kernel) is the officially supported operating system for Unite CS at the time of writing. It has several attractive qualities making it a strong candidate, one of them being premium support from Red Hat [33]. It is running a kernel with the RT-Preempt patch.

Performance: Based on RT-Preempt patch. Should give a clear advantage in real-time performance over the standard kernel [25].

Ease of porting: No changes needed. Running the RT-Preempt patch requires no code changes, nor recompilation of software.

Ease of deployment: No changes needed. Installation procedure is identical to that of the current product.

License: Commercial. Free developer license available.

Certification: None

3.1.1.2 CentOS 7 with Real-Time Kernel

CentOS 7 with real-time kernel is another very strong contender, as it is an operating system binary compatible with Red Hat, meaning deployment and porting don't require any extra work. It also runs a kernel with the RT-Preempt patch.

Performance: Based on RT-Preempt patch. Should give a clear advantage in real-time performance over the standard kernel [25].

Ease of porting: No changes needed. Running the RT-Preempt patch requires no code changes, nor recompilation of software.

Ease of deployment: No changes needed. Installation procedure is identical to that of the current product.

License: Free software

Certification: None

3.1.1.3 Red Hat Enterprise Linux 7 or CentOS 7 with Xenomai dual-kernel extension

By using the Xenomai extension the customer is able to run time critical processes on the co-kernel while letting the Linux kernel deal with non-real-time tasks.

Performance: Should perform better than RT-Preempt [25], but not quite on

the same level as VxWorks [9].

Ease of porting: Provides POSIX ”skin”, simplifying compiling POSIX compliant code for the co-kernel. Recompilation according to the Xenomai build procedure needed for all modules. At the moment of writing, Unite CS consists of over 1700 `Makefile`s that would need to be modified, making this a daunting endeavor.

Ease of deployment: Provided the software is successfully ported, new packages have to be built for the recompiled modules, which could then be installed through the same install script used in the current solution.

License: Free software

Certification: None

3.1.1.4 Green Hill Software INTEGRITY-178 RTOS

INTEGRITY-178 is a commercial real-time operating system heavily focused on reliability and performance. For instance, it is used in over 20 different aircraft models, both military and civilian [34].

Performance: Expected very good

Ease of porting: Complicated. Although POSIX compliant, not running a Linux system necessitates recompilation of all modules. In addition, the certified part of the INTEGRITY-178 OS only supports ANSI C and Embedded C++ and not the full C++ runtime rendering the source code incompatible [35]. A major rewrite of the whole software suite would be necessary to comply with the available runtimes. However, one has access to a full C and C++ runtime in the non-certified part of the operating system.

Ease of deployment: Complicated. Not an `rpm` based system, thus requiring rewrite of deployment script and packages

License: Commercial

Certification: DO-178B Level A, EAL 6+

3.1.1.5 Wind River VxWorks

VxWorks is another widely used high-performance real-time operating system with customers such as NASA, Boeing, Airbus, and Northrop Grumman [36].

Performance: Expected very good

Ease of porting: Complicated. Although POSIX compliant, not running a Linux system necessitates recompilation of all modules.

Ease of deployment: Complicated. Not an `rpm` based system, thus requiring rewrite of deployment script and packages

License: Commercial

Certification: None for the operating system itself, but Wind River supplies tools for simplifying obtaining certification for applications developed for the OS

3.2 Testing the real-time performance of the operating system

After evaluating a number of operating systems, Red Hat Enterprise Linux (RHEL) 7 with a real-time kernel was selected, as it is the only operating system officially supported by Ascom for running the Unite CS software suite. In addition RHEL has a free evaluation license, making it a good candidate for the project. Furthermore, the real-time kernel supplied by Red Hat is based on the RT-Preempt patch, meaning no changes to the code, or recompilation of the software is necessary.

The real-time kernel was installed according to the installation guide provided by Red Hat [37] through the `yum` package manager. Although no major problems arose during install, a few dependencies could not be resolved automatically as according to the installation guide, and had to be installed manually.

In order to test the interrupt latency for the real-time kernel the program `cyclictest`, which is the most frequently cited metric for real-time Linux performance [38], was used in conjunction with the `dohell` script, which is a script designed to generate load on a Linux machine through the use of commonly available commands and is used as part of a latency test shipped with Xenomai for measuring the real-time performance [39].

Simplified, `cyclictest` measures latency according to the following pseudocode, presented by Rowand [38]:

```
clock_gettime((&now))
next = now + par->interval
while (!shutdown) {
   clock_nanosleep((&next))
   clock_gettime((&now))
   diff = calcdiff(now, next)
   # update stat-> min, max, total latency, cycles
   # update the histogram data
   next += interval
}
```

In short, it measures the difference between the time when a thread should have been woken up and the time it actually woke up.

The same test was run under the same conditions when running the standard

Linux kernel, as well as with the real-time kernel as verification that the real-time kernel was installed correctly and functioning as to be expected.

`cyclictest` was run with the following command:

```
cyclictest -p 30 -t -n -l 100000 -H 1000 -histfile hist
```

The above command starts one thread for each CPU core on the system, running 100000 loops each, using `clock_nanosleep` for measuring time.

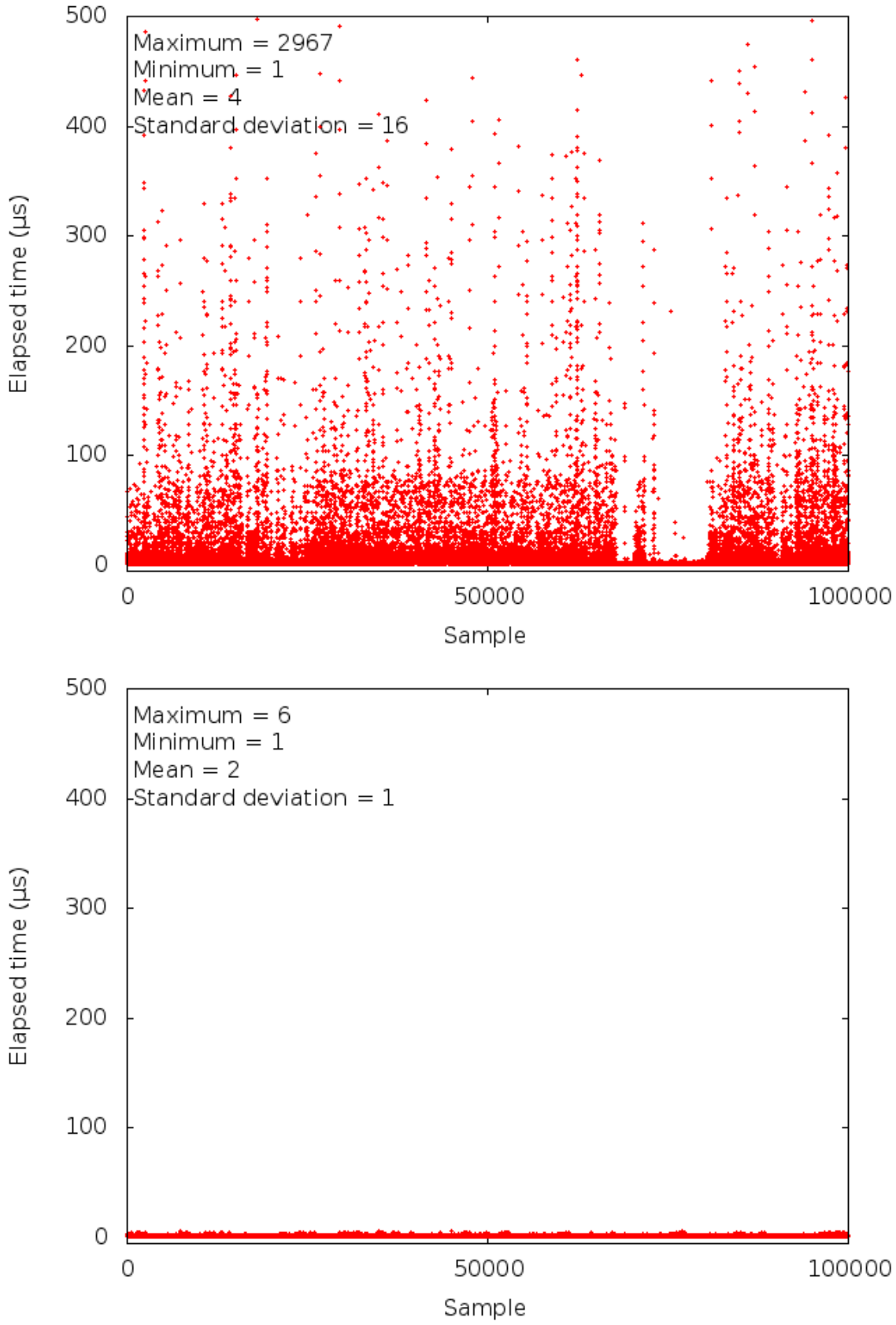

Figure 3.1: Above: Cyclictest with standard kernel
Below: Cyclictest with real-time kernel

As can be seen in figure 3.1 the interrupt latency was far lower and more consistent when running the real-time kernel, which suggests that the kernel was indeed correctly installed and running.

3.3 Setup

The tests and measurements were run on a remote machine connected to the host computer directly via wired Ethernet in order to minimize network delay. To improve accuracy of the measurements[3], the remote machine itself was running a real-time operating system (CentOS with RT-Preempt patch). During the measurements, care was taken not to interfere with either the remote or host computer.

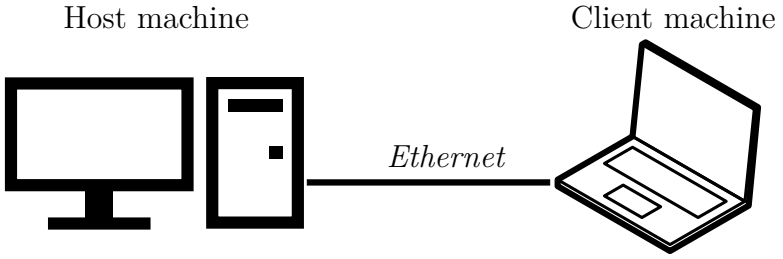

Figure 3.2: Setup used for measuring Unite CS

3.3.1 Host machine

The following system was used to run Unite CS:

Hardware:
Processor: Intel Core i5-2500
Memory: 12GiB DDR3 1333Mhz
Hard drive: 500GB Hitachi HDS72105
Network interface controller: Intel 82579V Gigabit Ethernet

Software:
Operating system: Red Hat Enterprise Linux 7.2 Maipo
Kernel: 3.10.0-327
Real-time kernel: 3.10.0-493.rt56

3.3.2 Client machine

The following system was used for measuring the performance of the host computer:

Hardware:
Processor: Intel Core i5-3210M

[3]When a reply is obtained, the thread responsible for measuring the delay needs to wake up fast in order to not to skew the results with its own scheduling latency.

Memory: 8GiB DDR3 1600Mhz
Hard drive: 128GB SAMSUNG MZ7PC128
Network interface controller: Intel 82579M Gigabit Ethernet

Software:
Operating system: CentOS 7.2.1511
Kernel: 3.10.0-327
Real-time kernel: 3.10.0-514.rt56

3.4 Testing the software

A test suite was developed in order to automate the measurements of the response times for the software. The test script is itself written in bash script, and internally uses a modified C++ application provided by Ascom that communicates with the Unite CS. The modified software then uses the tsc[4] clock to acquire accurate measurements for the elapsed time between sending a message to the Unite CS, and receiving a reply. In addition to merely measuring the response time, the test suite is also responsible for putting the computer under load i.e. starting up processes with the purpose of slowing the machine down so that the advantages of running the real-time operating system become more prominent. This is necessary as a system usually is more predictable when not overloaded [8].

During transmission of a single message, multiple round trips are made over the network due to how the applications and Unite protocol work. The *elapsed time* therefore does not describe the time elapsed until the first response is received from Unite CS, but rather when the last response is received. Using Wireshark, it was determined that four round trips (eight packets in total) were done over the network for every message.

The test procedure was structured as follows:

1. Start processes generating load on the computer running Unite CS
2. Every 30 milliseconds send a message to the Unite CS and record the time elapsed before an answer is received.
3. Repeat step 2 until 10000 messages have been sent.
4. Repeat the test for every operating system and load we are interested in measuring

For each operating system tested, measurements was performed with and without the software being adjusted for real-time operation. Due to the fact that the real-time kernel's obvious advantage is when running processes with a real-time priority, the remainder of the report mainly discusses differences when running processes with real-time priority, regardless of kernel.

[4]The tsc (Time Stamp Counter) clock is a high resolution clock that measures the number of clock cycles since some point in time, e.g. a reboot

3.5 Adjusting the software for real-time operation

To adjust the software for real-time operation, certain processes need to be promoted to a real-time priority. By inspecting the source code of Unite CS and monitoring system calls, the Unite CS processes involved in the critical information path were identified. When setting a real-time priority only these processes were promoted.

Three alternatives on how to perform the promotion are presented below.

3.5.1 Using `chrt`

Under Linux it is possible to set the real-time priority for a running process through the `chrt` command. As a first step towards reaping the benefits of the real-time kernel, this command was used to give certain processes belonging to the Unite CS suite a real-time priority. It can be easily done through one command, making it a very simple procedure not requiring substantial intervention from neither the customer nor Ascom:

```
pgrep -w "($Program1|$Program2|$ProgramX)" | xargs -L1 chrt -p -f $priority
```

The command takes a list of program names as part of the regular expression passed to the `pgrep` command, which returns the process IDs for these programs. These are then passed into the `chrt` command via the `xargs` command, where they are given a `SCHED_FIFO` (denoted by the `-f` flag) scheduling policy with priority `$priority`. This line of code could easily be put in a startup script for the target computer, although it needs to be run every time one of the included applications are started or restarted.

3.5.2 Create scripts replacing the binaries

Another approach is to replace the executables for the programs with scripts pointing to and running the real executable with certain scheduling settings. The creation of these scripts would then be managed by a central application where one can view and set the priorities for each application. Such a solution would not require any manual intervention between restarts, but would pollute the file system structure, rendering management of the installation more complicated. Moreover, having the system depend on both the script and the original executable to function properly would add an additional point of failure, something that should be minimized in a mission critical system such as Unite CS. That said, this method has the advantage of being deployable on an existing installation without updating the Unite CS.[5]

[5] Applications setting real-time priorities for processes must have the rights to do so on the target system. An application running under the root user normally has this right, however, if the application is started as a `systemd` service, which is the case with Unite CS, a real-time priority limit must be specified in the `.service` file in the `[Service]` section, like so:

```
LimitRTPRIO=50
```

This will give the process the rights to promote processes to a real-time priority up to 50.

3.5.3 Adjusting the program code

The most attractive long term solution is to set the desired priority in the application responsible for starting each individual program in the Unite CS suite. Such a change would necessitate changes to the source code and recompilation of the particular program, but would ultimately produce the same result as with the `chrt` command, but without the need for any housekeeping between restarts. This solution should preferably be accompanied by a per-application user adjustable priority setting available from some settings console for the Unite CS.[6]

A proof of concept was created for this solution, where a real-time priority attribute was added to the applications' configuration files which are stored in the XML format. The application responsible for starting the applications was then modified to parse this attribute, as well as setting the priority for each application through the POSIX conforming command `sched_setscheduler` available through the Linux scheduling API.

An application was also developed in the Python programming language for administering the real-time priorities for each application. It reads and writes to the configuration XML files directly, as well as setting the priorities for the currently running applications through the `chrt` command. This tool is required to be run as the root user on the target system as the configuration files for the applications are owned by the root user in a standard installation of Unite CS.

[6] See footnote 5

4
Results

The following results were obtained when benchmarking Unite CS according to section 3.4, with and without a real-time kernel. The results are categorized according to which load the machine running the Unite CS was subject to at the time of measurement.

It should be noted that running with the real-time kernel sometimes stalled the CPU when closing applications in the Unite CS suite, resulting in a frozen system. It happened rarely, but as the system freezes during a stall – requiring a reboot – it needs to be addressed. Applying step 2 in [40], giving the timer softirq threads priorities higher than those given to the Unite CS threads, reduced the frequency of such stalls greatly, although they still occurred.

Results for non-promoted processes are provided in Appendix A.

4.1 `dohell`

The following measurement shows the results obtained when stressing the system with the `dohell` script.

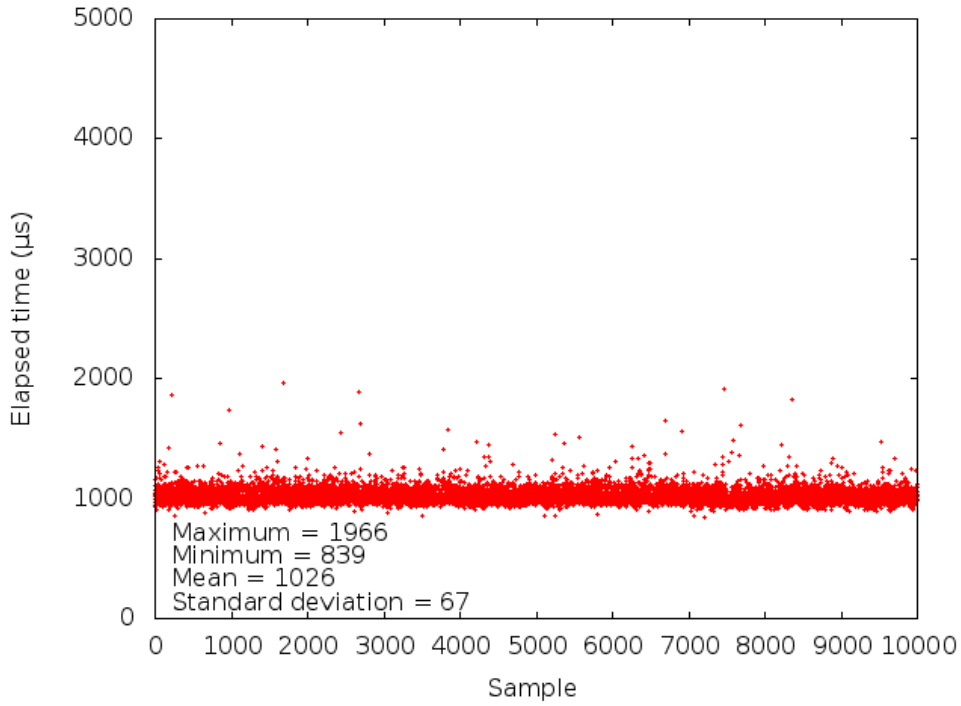

Figure 4.1: Response times for Unite CS on RHEL7 not running the real-time kernel with Unite CS processes promoted to a real-time priority. Load: `dohell`

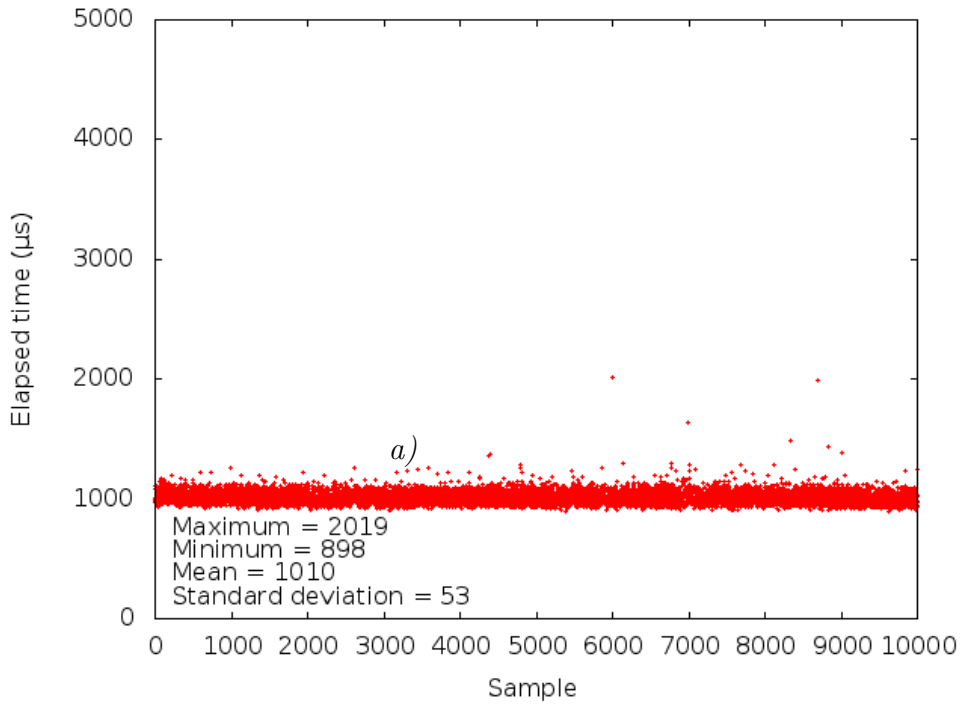

Figure 4.2: Response times for Unite CS on RHEL7 running the real-time kernel with Unite CS processes promoted to a real-time priority. Load: `dohell`

4.2 Concurrent memory allocations

During these measurements the application `stress` was used to perform repeated memory allocations while measuring the system. The following command was run:

```
stress --vm 40 --vm-bytes 256M
```

This command starts 40 worker threads, each continuously allocating and freeing 256 megabyte chunks of memory.

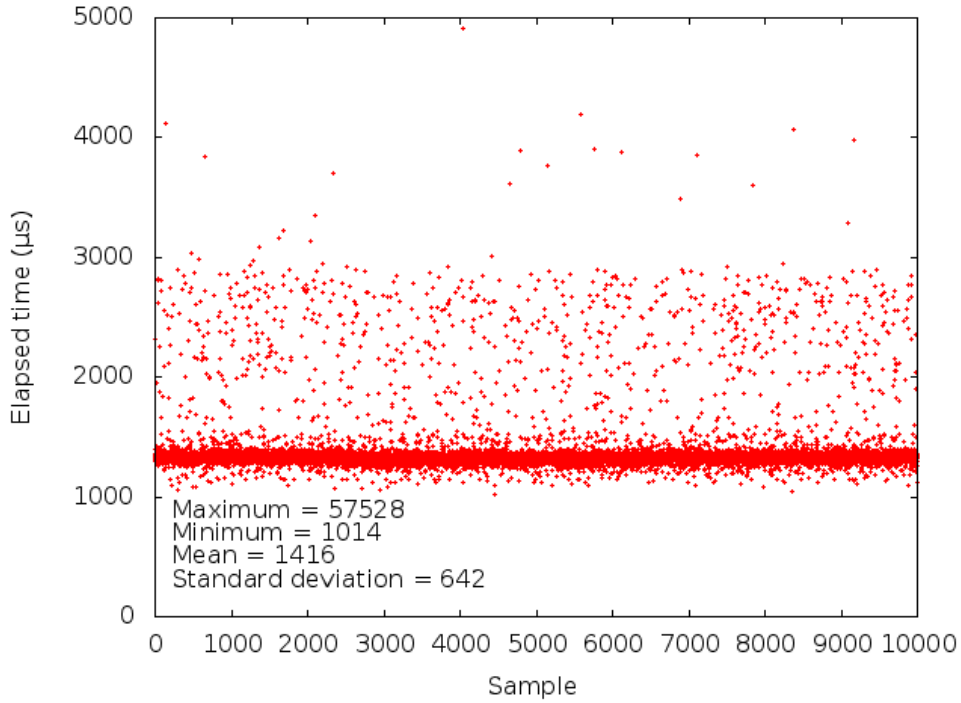

Figure 4.3: Response times for Unite CS on RHEL7 not running the real-time kernel with Unite CS processes promoted to a real-time priority. Load: `stress --vm 40 --vm-bytes 256M`

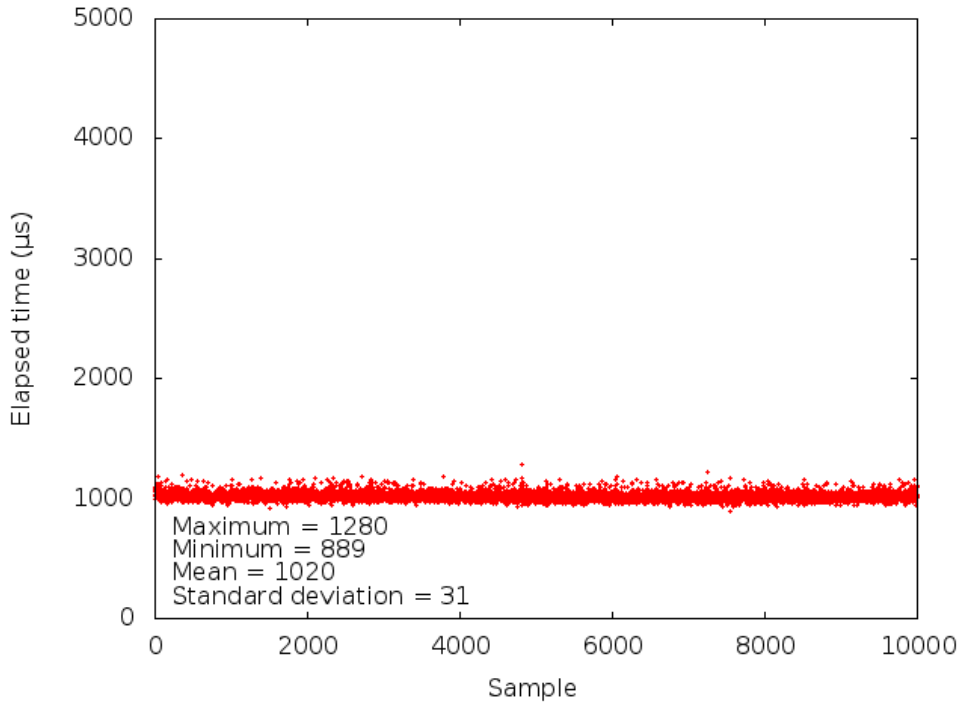

Figure 4.4: Response times for Unite CS on RHEL7 running the real-time kernel with Unite CS processes promoted to a real-time priority. Load: `stress --vm 40 --vm-bytes 256M`

Note that for figure 4.3 the second largest latency was 5451 µs.

4.3 Stressing CPU

During these measurements the application `stress` was used to utilize the CPU while measuring the system. The following command was run:

```
stress --cpu 100
```

This command starts 100 worker threads, each continuously calculating the square root of a random number.

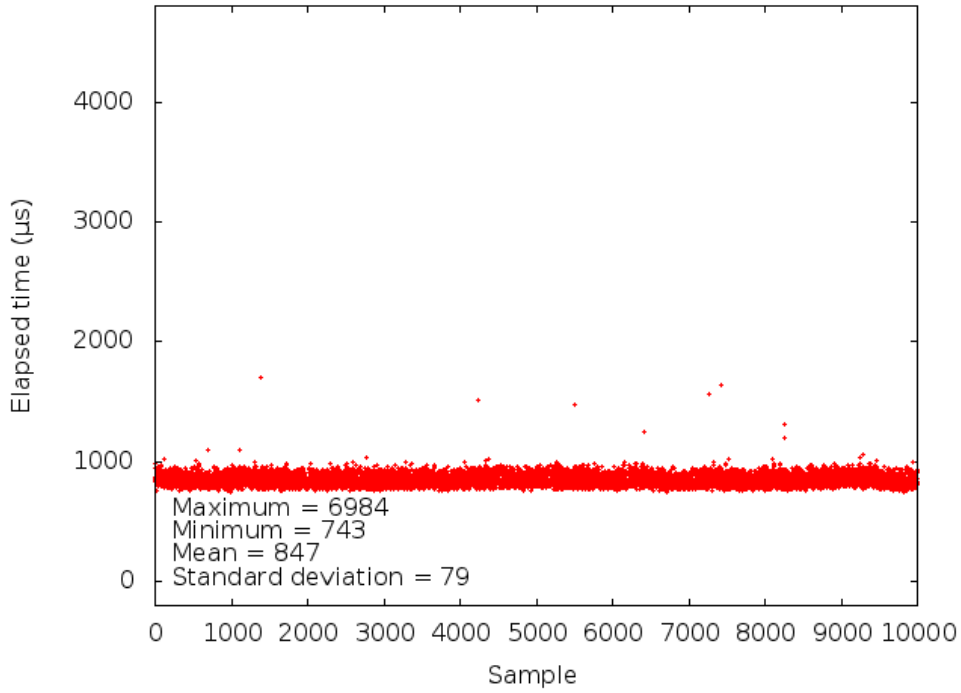

Figure 4.5: Response times for Unite CS on RHEL7 not running the real-time kernel with Unite CS processes promoted to a real-time priority. Load: `stress --cpu 100`

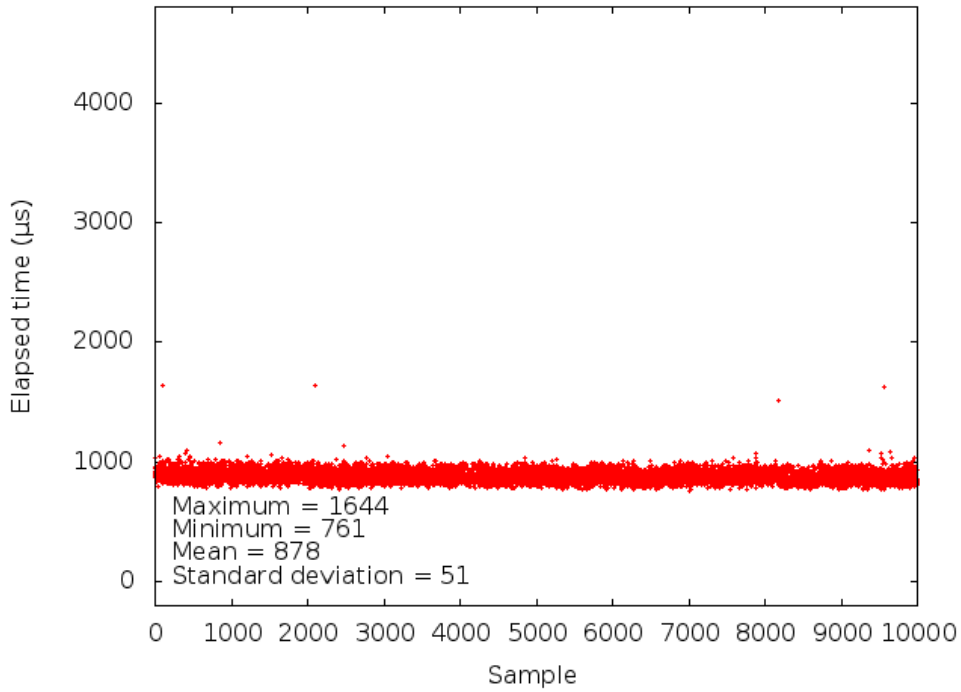

Figure 4.6: Response times for Unite CS on RHEL7 running the real-time kernel with Unite CS processes promoted to a real-time priority. Load: `stress --cpu 100`

5
Investigation

By looking at the figures in chapter 4 and comparing them with those in appendix A it is clear that running the vital Unite CS processes with a real-time priority improves determinism for the system greatly regardless of load, something that is true running both with and without the real-time kernel.

This chapter aims to discuss and comment on the differences in the results between the real-time and non-real-time kernel for the different loads. It also discusses how network latency affects the measurements.

5.1 Load: `dohell`

By observing figures 4.1 and 4.2 it can be concluded that running a real-time kernel does give some improvements regarding determinism over running the non-real-time kernel when promoting the Unite CS processes to real-time priorities. Additionally, marginally higher throughput was also observed when running the real-time kernel. However, the maximum values, which are ultimately the metric of importance, did not see any improvement.

Looking at figure 4.2 (real-time kernel), a pattern can be observed. The samples exhibiting a higher latency to the extent that it is visible on the scatter plot seem to occur at rather regular intervals. This is especially clear beneath the *a)* mark. To understand why such behavior was observable further insight in the `dohell` script was needed.

The `dohell` script is rather simple, and basically stresses the system through four commands:

```
cat /proc/interrupts

ps w

dd if=/dev/zero of=/dev/null

ls -lR /
```

By testing each command alone, it was found that running `ls -lR /` produced these spikes in latency. The command itself lists all files on the system recursively, starting from the root folder. Trying to produce a minimal working example, the following command was found to produce the same spikes

```
ls -R -U -i /
```

Where `-R` enables recursive listing, `-U` disables sorting (removing any excessive CPU usage), and `-i` enables printing of inode number. The flag of interest here is the `-i` flag. Without it these spikes did not occur. Running both commands under the `strace`[7] tool gave the following output on the machine running Unite CS:

% time	seconds	usecs/call	calls	errors	syscall
75.97	0.254045	1	207979	1	lstat
11.60	0.038779	1	50632		getdents
6.05	0.020232	1	25313	3	openat
2.96	0.009901	0	25325		close
2.02	0.006769	0	25324		fstat
1.32	0.004420	3	1332		write
0.02	0.000075	2	33		mmap
0.02	0.000056	3	20		mprotect
0.02	0.000055	2	25	12	open
0.00	0.000014	7	2		statfs
0.00	0.000013	1	11		read
0.00	0.000011	0	25		brk
0.00	0.000007	7	1	1	access
0.00	0.000005	3	2		stat
0.00	0.000005	2	3		munmap
0.00	0.000003	2	2	2	ioctl
0.00	0.000002	1	2		rt_sigaction
0.00	0.000002	2	1		futex
0.00	0.000001	1	1		rt_sigprocmask
0.00	0.000001	1	1		execve
0.00	0.000001	1	1		getrlimit
0.00	0.000001	1	1		arch_prctl
0.00	0.000001	1	1		set_tid_address
0.00	0.000001	1	1		set_robust_list
0.00	0.000000	0	2		mremap
100.00	0.334400		336040	19	total

Figure 5.1: `strace-cf ls -RUi /`

[7] `strace` is an program used for monitoring which system calls are used by some program running on a system.

% time	seconds	usecs/call	calls	errors	syscall
47.67	0.027230	1	50632		getdents
27.24	0.015559	1	25313	3	openat
12.53	0.007157	0	25325		close
9.16	0.005231	0	25324		fstat
2.91	0.001664	2	975		write
0.14	0.000079	2	33		mmap
0.11	0.000063	3	20		mprotect
0.09	0.000053	2	25	12	open
0.04	0.000023	1	25		brk
0.04	0.000022	11	2		statfs
0.02	0.000010	1	11		read
0.01	0.000006	3	2		stat
0.01	0.000006	2	3		munmap
0.01	0.000003	2	2	2	ioctl
0.01	0.000003	3	1	1	access
0.00	0.000002	1	2		rt_sigaction
0.00	0.000001	1	1		rt_sigprocmask
0.00	0.000001	1	1		execve
0.00	0.000001	1	1		getrlimit
0.00	0.000001	1	1		arch_prctl
0.00	0.000001	1	1		futex
0.00	0.000001	1	1		set_tid_address
0.00	0.000001	1	1		set_robust_list
0.00	0.000000	0	2		mremap
100.00	0.057118		127704	18	total

Figure 5.2: `strace-cf ls -RU /`

Figure 5.1 and 5.2 reveals that when the `-i` flag is enabled, the `ls` command uses the `lstat`[8] system call for every file on the system. The command also spends most of its time executing this system call.

Compiling the `ls` program from source, and eliminating the `lstat` call (thus breaking the printing of inodes) further verified that repeatedly using this system call indeed was responsible for the spikes in latency.

Redoing the measurements with a modified `dohell` script, where the `ls` command is changed to `ls -RU /` produced the following results:

[8] `lstat` is a system call defined by the POSIX standard used for querying information about a file.

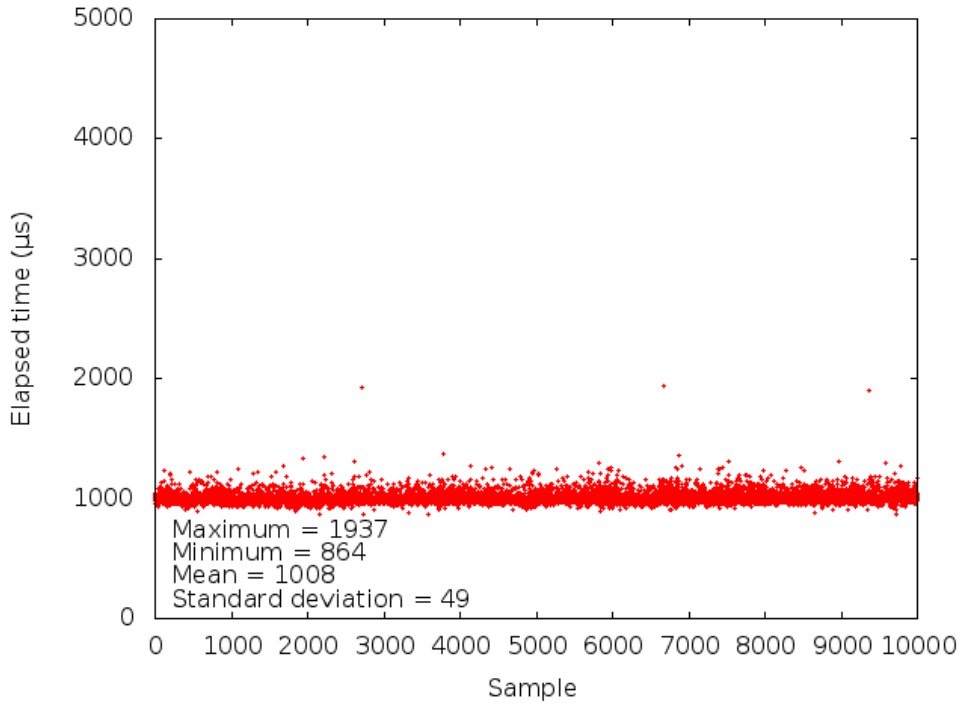

Figure 5.3: Response times for Unite CS on RHEL7 not running the real-time kernel with Unite CS processes promoted to a real-time priority. Load: Modified `dohell`

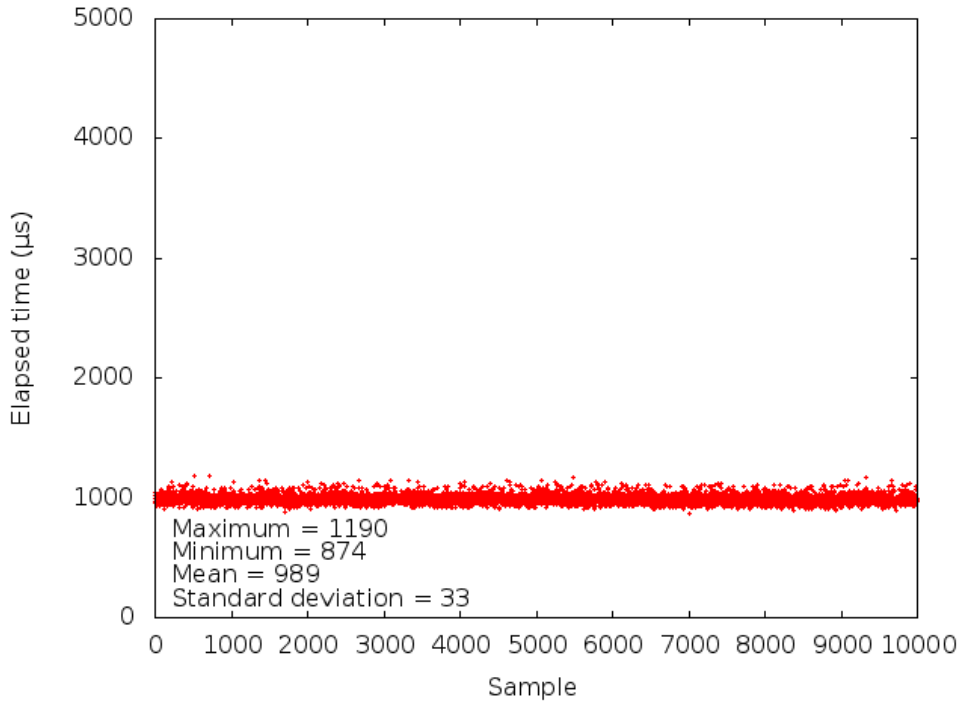

Figure 5.4: Response times for Unite CS on RHEL7 running the real-time kernel with Unite CS processes promoted to a real-time priority. Load: Modified `dohell`

It can be seen that determinism improved for both kernels when eliminating the `lstat` call. This might lead one to believe that Unite CS accesses the disk within the critical execution path. However, monitoring the system calls for Unite CS processes showed that not to be the case. In conclusion, the `lstat` call is considered a heavy system call where the real-time kernel shows only minor improvement over the standard kernel.

5.2 Load: Concurrent memory allocations

This load produced the most obvious improvements for the real-time kernel. It is likely that this is due to the Linux kernel shipped with the tested version of RHEL7 being compiled with the preemption mode `PREEMPT_VOLUNTARY`. This preemption mode does allow for some preemption within the kernel, but only at certain points [41]. Consequently, the processes wanting to wake up may not be able to do so due to preemption being disabled during the memory allocations.

The `ftrace` tool can be used in conjunction with the wakeup_rt tracer to monitor the wakeup latency for real-time threads. Indeed, using it while measuring under this load showed a very high wakeup latency when running the normal kernel, on the order of several thousand microseconds due to the `stress` application. On the other hand, when using the real-time kernel the latency rarely rose above 30 microseconds.

Keeping this in mind, the Linux kernel offers another low latency preemption mode which is included in the mainline Linux kernel called `CONFIG_PREEMPT`. Compiling the kernel with this preemption mode allows for further preemption in the kernel, although not to the same extent as with the RT-Preempt patch. Repeating the same measurement with a kernel compiled with `CONFIG_PREEMPT` enabled yielded the following result:

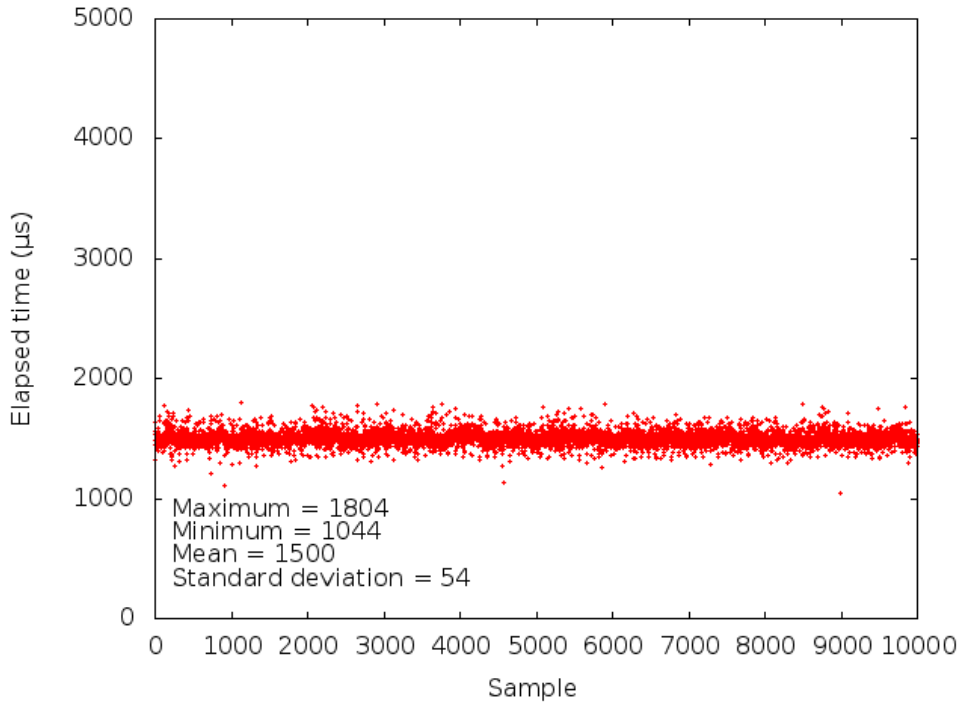

Figure 5.5: Response times for Unite CS on RHEL7 running kernel compiled with `CONFIG_PREEMPT` with Unite CS processes promoted to a real-time priority. Load: `stress --vm 40 --vm-bytes 256M`

It can be seen that the `CONFIG_PREEMPT` enabled kernel gives much better results than the standard kernel shipped with RHEL7 regarding latency for this particular case. However, a kernel configured as such is at the time of writing not provided by Red Hat but has to be compiled from source or acquired from a third party, which is not supported by Red Hat [42].

5.3 Load: Stressing CPU

For this load no clear improvements were seen, and the high maximum seen in figure 4.5 was a lone outlier. This result was expected, as the load itself (calculating square roots of random numbers) does not utilize any system calls, i.e. no non-preemptible kernel code paths are executed; Unite CS processes can always preempt the lower priority tasks when needed without any significant delay.

5.4 The impact of network communication

As mentioned in chapter 3, the two computers used in the measurements were connected directly via wired Ethernet in order to minimize delay caused by network communication. Even doing so, it can be observed that even the results with the lowest standard deviation (figure 4.4) exhibit a span of 391 microseconds between minimum and maximum. The culprit for such values is unlikely to be the inter-

rupt latency of the system, which was measured to be on the order of single digit microseconds.

To measure the latency of the network, two simple test applications were created. A server which resides on the machine that is normally running Unite CS, and a client which resides on the machine carrying out the measurements. The client measures the latency by sending a series of messages over the UDP protocol, waiting for an acknowledge message from the server after each message. The latency is then measured as the time that passed between the first message sent, and the last message acknowledged. This procedure, with four round trips (eight messages in total), resembles the execution of the application used for the original measurements while keeping the focus on network latency only. Both applications are run with a real-time priority.

Measuring 10 000 runs while loading the host computer with the command
`stress --vm 40 --vm-bytes 256M`
yielded the following result:

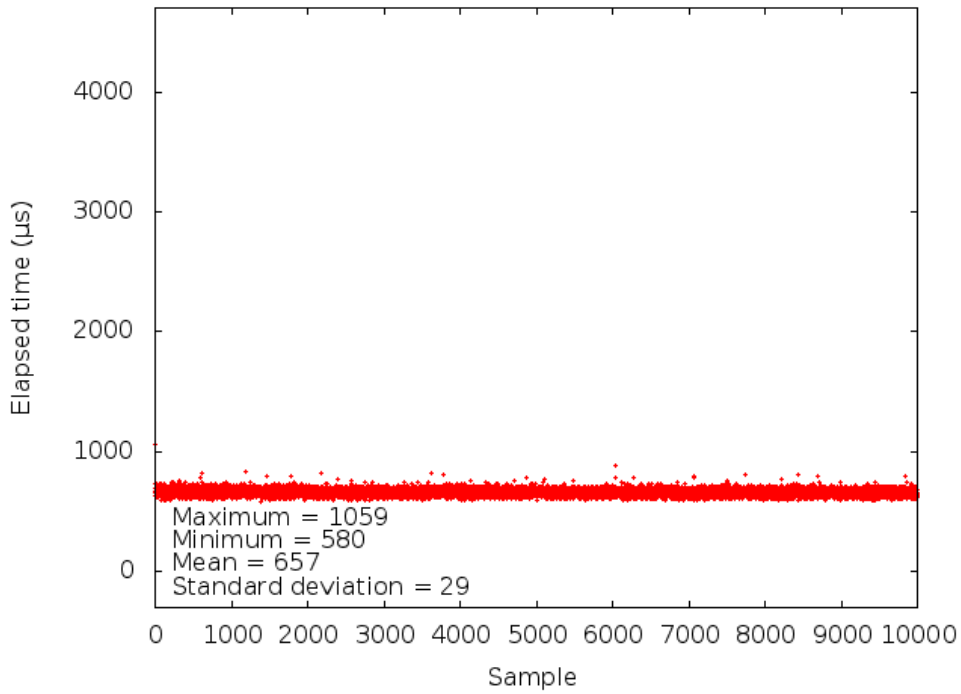

Figure 5.6: Response times for mock server on RHEL7 running the real-time kernel. Load: `stress --vm 40 --vm-bytes 256M`

These results show similar numbers regarding determinism as figure 4.4, suggesting that the use of wired Ethernet and associated drivers might introduce a fair amount of nondeterminism to the system.

5.4.1 Improving network determinism using Xenomai with RTnet

While porting Unite CS to the Xenomai extension requires too big changes to the build procedure to be viable for this project, porting the mock server and client still provided insight in how a real-time adjusted Ethernet driver can improve the network determinism of the system.

First, the latest (at the point of writing) Linux kernel with a suitable Xenomai patch, 4.1.18, was patched with the Xenomai dual-kernel extension with RTnet[9] enabled. The kernel was then compiled and installed on the host RHEL7 installation.

After recompiling the mock server for Xenomai and RTnet, the measurements from section 5.4 were repeated, giving the following result:

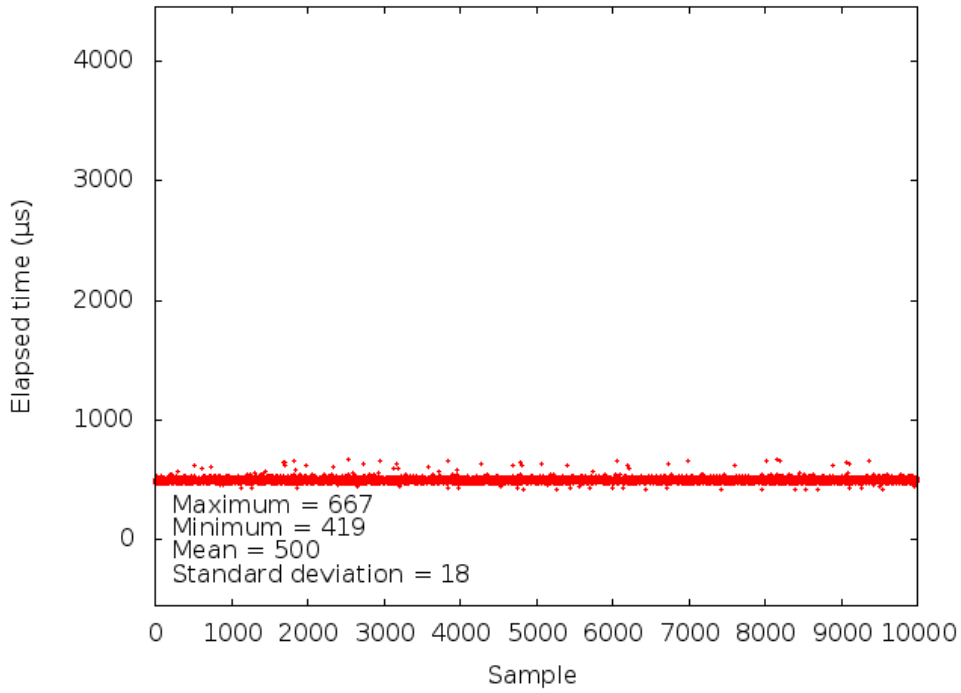

Figure 5.7: Response times for mock server on RHEL7 running the Xenomai extension kernel with RTnet. Load: `stress --vm 40 --vm-bytes 256M`

Installing the Xenomai patched kernel also on the machine performing the measurements gave the following result:

[9]RTnet is an open source hard real-time network protocol stack available for Xenomai and RTAI [43]. It supports many common network cards, including the ones available in the machines used for this thesis.

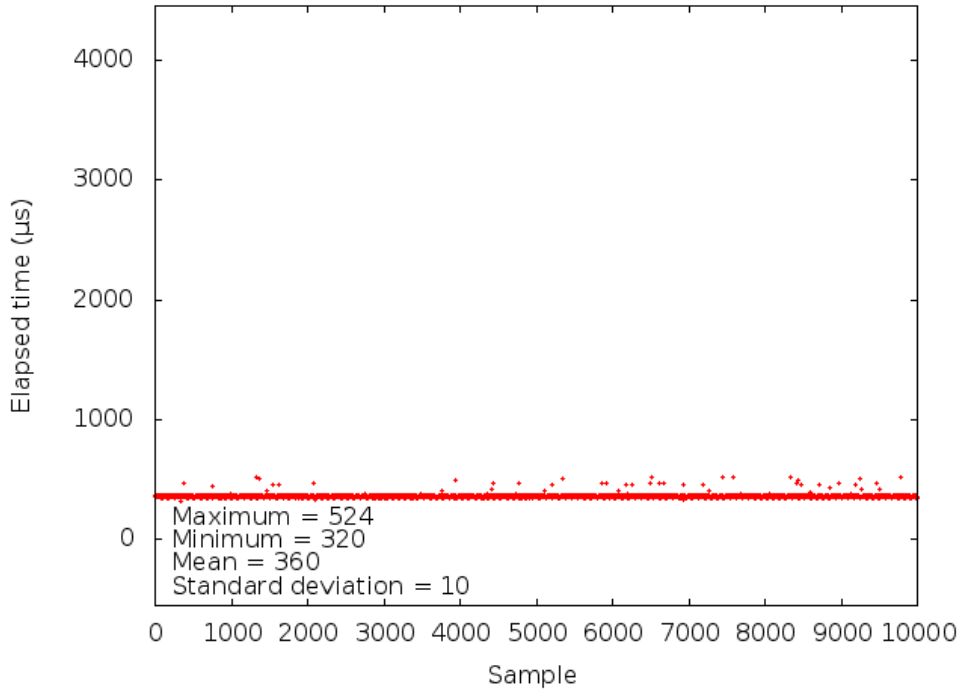

Figure 5.8: Response times for mock server on RHEL7 running the Xenomai extension kernel with RTnet driver *and* running the Xenomai extension with RTnet on the computer doing the measurement. Server load: `stress --vm 40 --vm-bytes 256M`

As can be seen in the above figures, determinism is greatly improved when using the RTnet driver under Xenomai, especially when used on both network nodes, confirming that the standard Intel Ethernet driver (in this case the kernel module e1000e) in conjunction with the Linux kernel considerably impairs determinism for this particular setup.

However, even assuming Unite CS can be successfully ported to utilize RTnet, the solution is difficult to adapt due to a number of reasons: Firstly, while running the RTnet driver, all applications that utilize the network need to be compiled to use RTnet themselves, which is not really an alternative for the machines running Unite CS[10]. Secondly, RTnet was not completely stable on the host machine and sometimes needed to be restarted. Thirdly, just as with `CONFIG_PREEMPT`, this solution would not be supported by Red Hat, should problems with the operating system arise.

[10]Other applications may use the network normally with some configuration if an additional Ethernet line is available on the host computer.

6
Discussion

From the results in chapter 4 one can conclude that the real-time kernel was more deterministic under every load tested. Yet, seeing the improvements as a motivation to require, or even recommend customers to migrate to a real-time environment is debatable:

Firstly, the performance of the system was very good even when running the standard kernel. With a measured maximum response time of ~58 ms, a system where the the final actor is a person, such as Unite CS, could already be seen as performant enough for its purpose. This maximum was also a lone outlier, with the second largest latency being ~7 ms. Moreover, Unite CS – being a soft real-time system – might not suffer from such delays occasionally.

Secondly, the improvements observed are rather minor considering how the system is used in practice. Keeping in mind that the Unite CS will deliver messages to devices operated by people, improvements on the order of single milliseconds may not bring any noticeable improvements when put to practice, as the recipient of the message needs to react to the notification signal, produce their device, and read the message before any action can be taken regarding the content. If the duration of such a procedure was measured as rigorously as the Unite CS during this thesis, the observed deviation in durations would surely mask the improvements gained by running the real-time kernel. The same is also true for the networking involved when running in a production system. As the network communication itself has an associated latency that is not constant, it is rather possible that the network latency would also render the improvements completely unnoticeable, especially over Wi-Fi.

Finally, with Unite CS not being a medical device or subject to any other regulations regarding response time, providing a system that is well tested and *good enough* is perfectly adequate. Until there is a definitive need for a more responsive system, say, should it incorporate some machine-to-machine aspect (such as controlling other medical devices), there is not enough motivation based on the results obtained in this thesis to migrate to a real-time operating system.

Furthermore, running the software on the real-time kernel required modifications to the system in order for it not to hang, namely raising the priority of the timer softirq threads. Such an operation is rather invasive, and affects not only Unite CS, but the system as a whole, and could negatively impact other applications running on the customer's machine. If not absolutely necessary, such a modification should be avoided. Additionally, even after this adjustment the system hung, albeit rarely. Ceasing to function is of course unacceptable for a software of this nature. Conversely, the non-real-time kernel did not once, under any circumstance, hang or cease to function. Note that this paragraph only discourages using the tested

real-time kernel and version. Other kernels, operating systems, or hardware, might not suffer from the same problem.

However, using real-time priorities for crucial Unite CS processes can improve determinism greatly on an existing system at a very low cost. Naturally, although no issues were encountered during the measurements, further testing should be carried out to ensure the stability of the solution.

It is also important to keep in mind that the metric of interest for a real-time system is the maximum response time. Not performing thorough complexity analysis on the code, one can only obtain such a number by repeated testing to the point where the gathered maximum can be deemed an approximation of the true maximum.

6.1 Reflection

Looking back on the project, a few things can be mentioned that might be helpful to know in case a similar evaluation is to be performed in the future.

For one, Unite CS was not subject to particular constraints regarding determinism at the time of writing. It was also not reported to perform sub-par to expectations prior to this thesis. Considering this, even if the results might be of interest, evaluating how such a system could be improved upon in terms of performance by using a real-time operating system could be seen as a case of premature optimization. If the research itself is not the goal of such an evaluation, it is very possible that the effort will give very small practical gains for the system.

Furthermore, using the built in real-time capabilities of the stock Linux kernel resulted in much bigger improvements than going from the stock kernel to the real-time kernel. This is consistent with what is said by Bíba et al. [37], that standard tuning will give 90% of the latency gains, and using a real-time kernel will provide the remaining 10%. Keeping this in mind, tuning the current system should, if not already done, be the first step towards increasing determinism; a real-time operating system is not a silver bullet.

It may also be true that the term *"Real-time operating system"* still carries the misconception that it will increase performance of a system. While this is generally true for *real-time performance*, it is not true for *throughput*, which is what many think about when they hear the word "performance". Rather the contrary, throughput will often suffer in favor of increased determinism, something that could be observed in section 4.3. This misconception is not a recent phenomenon, but has been around for decades, and was highlighted by Stankovic in the paper "Misconceptions about real-time computing: a serious problem for next-generation systems" [44].

Another point worth noting is that throughout this thesis, it is assumed that the customer is able to run unrelated software on the same machine as Unite CS (as this is not prohibited by Ascom). The obtained results, and also the results obtained by Barbalace et al. [9], show that the improvements regarding determinism that can be expected when moving to a real-time operating system are often sub-millisecond. If improvements on that scale did prove to have major significance regarding the safety of patients, surely it would be more reasonable to force the customer to dedicate a

machine solely to most critical parts of Unite CS in order to ensure as low response times as possible.

7
Conclusion

Achieving major gains regarding determinism for some software by simply running it on a real-time operating system is not something that should be expected unless the software itself is developed from the bottom and up with real-time operation in mind. Many otherwise indispensable tools in the programmers toolbox are not viable if determinism is a top priority, and care must be taken in order to make sure the implementation is free from unbounded operations.

That aside, one should also ask the question whether or not the system *needs* real-time guarantees. As excluding tools such as dynamic memory allocation and disk access greatly inhibits flexibility and introduces complexity for the implementation, doing so only to save some fractions of a millisecond when it is not really needed might be a bad business decision not only in short, but also long term.

That said, improvements were indeed observed when running the real-time kernel, especially when the host computer was subject to heavy memory allocations in the background. For a Linux system relying on sub-millisecond determinism while simultaneously requiring the ability to dynamically allocate memory in non-real-time processes, merely switching the real-time kernel might improve the system enough for it to pass the performance requirements. Considering its binary compatibility with the standard kernel and open source nature, a kernel patched with RT-Preempt can easily be evaluated if so desired.

However, for Unite CS the performance is already deemed adequate *without* the real-time kernel if running the critical programs with a real-time priority, and doing so should preferably be adopted in the current product, as it is a very simple solution with clear benefits. It is an easily deployable solution, not requiring any modification to the customer's systems. It can be shipped as an update to Unite CS, and be activated and disabled on demand on a per-application basis with the help of the real-time administrative tool developed as part of this project.

Furthermore, UDP over Ethernet was also found to introduce quite a bit of non-determinism to the system, which was shown to be due to the Linux Ethernet driver. Using the RTnet network stack under Xenomai improved network determinism considerably, suggesting that the standard Ethernet driver might not be preferable in real-time systems with stringent timing constraints.

Bibliography

[1] Red Hat. *What is yum and how do I use it?* URL: https://access.redhat.com/solutions/9934 (visited on Nov. 18, 2016).

[2] CentOS. *Managing Software with yum.* URL: https://www.centos.org/docs/5/html/yum/ (visited on Nov. 18, 2016).

[3] N. Unkelos-Shpigel and I. Hadar. "A multitude of requirements and yet sole deployment architecture: Predictors of successful software deployment". In: *Twin Peaks of Requirements and Architecture (TwinPeaks), 2013 2nd International Workshop on the.* May 2013, pp. 19–23.

[4] M. V. Mantyla and J. Vanhanen. "Software Deployment Activities and Challenges - A Case Study of Four Software Product Companies". In: *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on.* Mar. 2011, pp. 131–140.

[5] V. Talwar et al. "Comparison of Approaches to Service Deployment". In: *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05).* June 2005, pp. 543–552.

[6] J. A. Tyrrell et al. "Efficient migration of complex off-line computer vision software to real-time system implementation on generic computer hardware". In: *IEEE Transactions on Information Technology in Biomedicine* 8.2 (June 2004), pp. 142–153.

[7] A. Santhanam. *Towards Linux 2.6, A look into the workings of the next new kernel.* IBM Global Services. 2003. URL: http://www.informatica.co.cr/linux-scalability/research/2003/0923.html (visited on Oct. 31, 2016).

[8] R. V. Aroca and G. Caurin. "A real time operating systems (RTOS) comparison". In: *Sao Carlos–SP–Brasil* (2009).

[9] A. Barbalace et al. "Performance Comparison of VxWorks, Linux, RTAI, and Xenomai in a Hard Real-Time Application". In: *IEEE Transactions on Nuclear Science* 55.1 (Feb. 2008), pp. 435–439.

[10] Y. Shinjo and C. Pu. "Achieving efficiency and portability in systems software: a case study on POSIX-compliant multithreaded programs". In: *IEEE Transactions on Software Engineering* 31.9 (Sept. 2005), pp. 785–800.

[11] T. Macieira. *Binary compatibility for library developers.* Conference presentation slides from LinuxCon North America, New Orleans, Sept. 2013. URL: https://events.linuxfoundation.org/sites/events/files/slides/Binary_Compatibility_for_library_devs.pdf (visited on Nov. 17, 2016).

[12] T. Tanaka et al. “Approaches to making software porting more productive”. In: *TRON Project International Symposium, 1995., Proceedings of the 12th*. Nov. 1995, pp. 73–85.

[13] *ISO/IEC 9899:201x Programming languages – C*. International Organization for Standardization. 2011. URL: http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf (visited on Nov. 17, 2016).

[14] K. S. Reddy. *New real-time operating systems for embedded systems*. 1st ed. Laxmi Publications, 2014.

[15] S. Siewert and J. Pratt. *Real-Time Embedded Components and Systems with Linux and RTOS*. Mercury Learning and Information, 2016.

[16] A. M. K. Cheng. “Cyber-Physical Medical and Medication Systems”. In: *2008 The 28th International Conference on Distributed Computing Systems Workshops*. June 2008, pp. 529–532.

[17] M. Chetto. *Real-time Systems Scheduling*. 1st ed. Iste, US: Wiley, 2014.

[18] N. Chauhan. *Principles of operating systems*. New Dehli: Oxford University Press, 2014.

[19] J. M. Garrido. *Principles of modern operating systems*. 2nd ed. Burlington, MA: Jones & Bartlett Learning, 2013.

[20] J. A. Stankovic and R. Rajkumar. “Real-Time Operating Systems”. In: *Real-Time Systems* 28.2 (2004), pp. 237–253.

[21] P. Hambarde, R. Varma, and S. Jha. “The Survey of Real Time Operating System: RTOS”. In: *Electronic Systems, Signal Processing and Computing Technologies (ICESC), 2014 International Conference on*. Jan. 2014, pp. 34–39.

[22] Xenomai. *How does Xenomai deliver real-time?* 2016. URL: https://xenomai.org/start-here/#How_does_Xenomai_deliver_real-time (visited on Oct. 19, 2016).

[23] L. Fu and R. Schwebel. *RT-PREEMPT HOWTO, About the RT-Preempt Patch*. 2016. URL: https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO#About_the_RT-Preempt_Patch (visited on Oct. 19, 2016).

[24] J. Singh et al. *RT-PREEMPT WIKI, Frequently Asked Questions*. 2016. URL: https://rt.wiki.kernel.org/index.php/Frequently_Asked_Questions (visited on Oct. 19, 2016).

[25] J. H. Brown and B. Martin. *How Fast Is Fast Enough? Choosing between Xenomai and Linux for Real-time Applications. Technical Report*. Tech. rep. Invariant Systems, 2010. URL: https://www.osadl.org/fileadmin/dam/rtlws/12/Brown.pdf (visited on Oct. 19, 2016).

[26] K. H. Kim. “Toward QoS certification of real-time distributed computing systems”. In: *High Assurance Systems Engineering, 2002. Proceedings. 7th IEEE International Symposium on*. 2002, pp. 177–186.

[27] Y. Moy et al. “Testing or Formal Verification: DO-178C Alternatives and Industrial Experience”. In: *IEEE Software* 30.3 (May 2013), pp. 50–57.

[28] M. Mchugh, F. Mccaffery, and V. Casey. “Software process improvement to assist medical device software development organisations to comply with the amendments to the medical device directive”. In: *IET Software* 6.5 (Oct. 2012), pp. 431–437.

[29] N. G. Leveson and C. S. Turner. “An investigation of the Therac-25 accidents”. In: *Computer* 26.7 (July 1993), pp. 18–41.

[30] *Directive 2007/47/EC*. European Commission. 2007. URL: http://eur-lex.europa.eu/legal-content/en/ALL/?uri=CELEX:32007L0047 (visited on Nov. 3, 2016).

[31] *Medical device software – Software life cycle processes*. International Organization for Standardization. 2006. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=38421 (visited on Nov. 3, 2016).

[32] M. Zema et al. “Developing medical device software in compliance with regulations”. In: *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. Aug. 2015, pp. 1331–1334.

[33] Red Hat. *Red Hat Enterprise Linux For Real Time*. 2015. URL: https://www.redhat.com/en/files/resources/en-rhel-real-time-datasheet-INC0223687.pdf (visited on Jan. 16, 2017).

[34] Green Hills Software. *Achieve Total Reliability for your Electronic Products*. URL: http://www.ghs.com/MaximizeReliability.html (visited on Nov. 4, 2016).

[35] Green Hills Software. *INTEGRITY-178 EAL 6+ certified, safety-critical RTOS*. URL: http://www.ghs.com/products/safety_critical/integrity-do-178b.html (visited on Nov. 4, 2016).

[36] VxWorks. *VxWORKS Product Overview*. URL: https://www.windriver.com/products/product-overviews/2691-VxWorks-Product-Overview.pdf (visited on Feb. 3, 2017).

[37] R. Bíba et al. *Red Hat Enterprise Linux for Real Time 7 Installation Guide*. 2015. URL: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux_for_Real_Time/7/pdf/Installation_Guide/Red_Hat_Enterprise_Linux_for_Real_Time-7-Installation_Guide-en-US.pdf (visited on Oct. 31, 2016).

[38] F. Rowand. *Using and Understanding the Real-Time Cyclictest Benchmark*. Conference presentation slides. 2013. URL: http://events.linuxfoundation.org/sites/events/files/slides/cyclictest.pdf (visited on Oct. 19, 2016).

[39] Xenomai. *DOHELL(1) Manual Page*. 2014. URL: http://www.xenomai.org/documentation/xenomai-head/html/dohell (visited on Oct. 24, 2016).

[40] Red Hat. *Avoiding RCU Stalls in the real-time kernel*. 2016. URL: https://access.redhat.com/solutions/2260151 (visited on Oct. 19, 2016).

[41] Ingo Molnar. *Voluntary Kernel Preemption*. URL: https://lwn.net/Articles/137259/ (visited on Dec. 21, 2016).

[42] Red Hat. *Production Support Scope of Coverage*. URL: https://access.redhat.com/support/offerings/production/soc (visited on Dec. 22, 2016).

[43] RTnet. *Hard Real-Time Networking for Real-Time Linux*. URL: http://www.rtnet.org (visited on Jan. 3, 2017).

[44] J. A. Stankovic. "Misconceptions about real-time computing: a serious problem for next-generation systems". In: *Computer* 21.10 (Oct. 1988), pp. 10–19.

A

Appendix 1

A.1 `dohell` with no real-time promotion

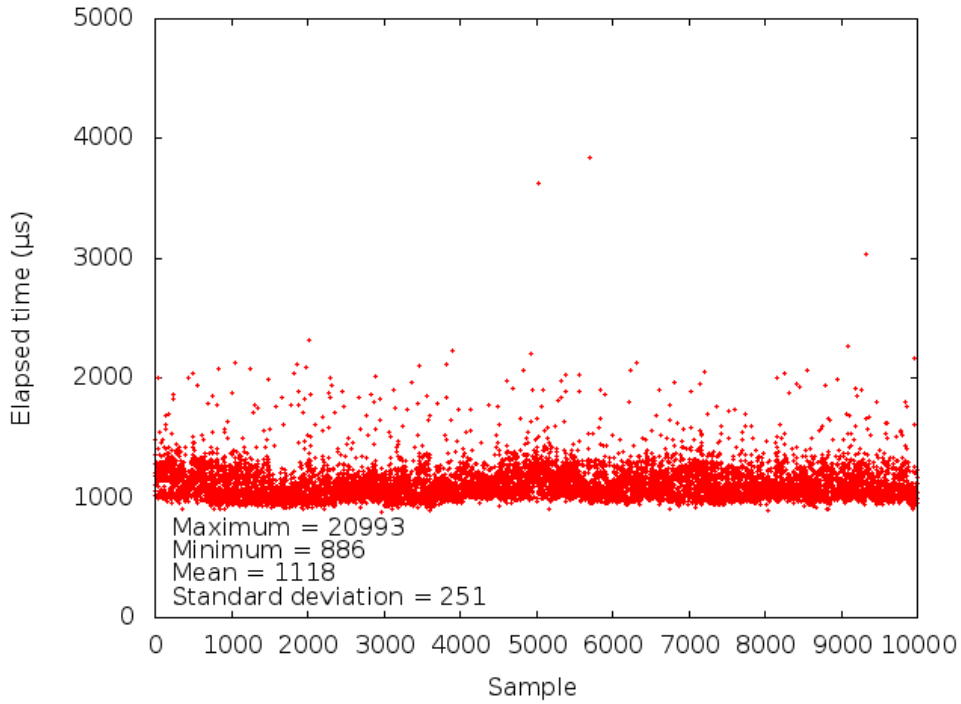

Figure A.1: Response times for Unite CS on RHEL7 not running the real-time kernel. Load: `dohell`

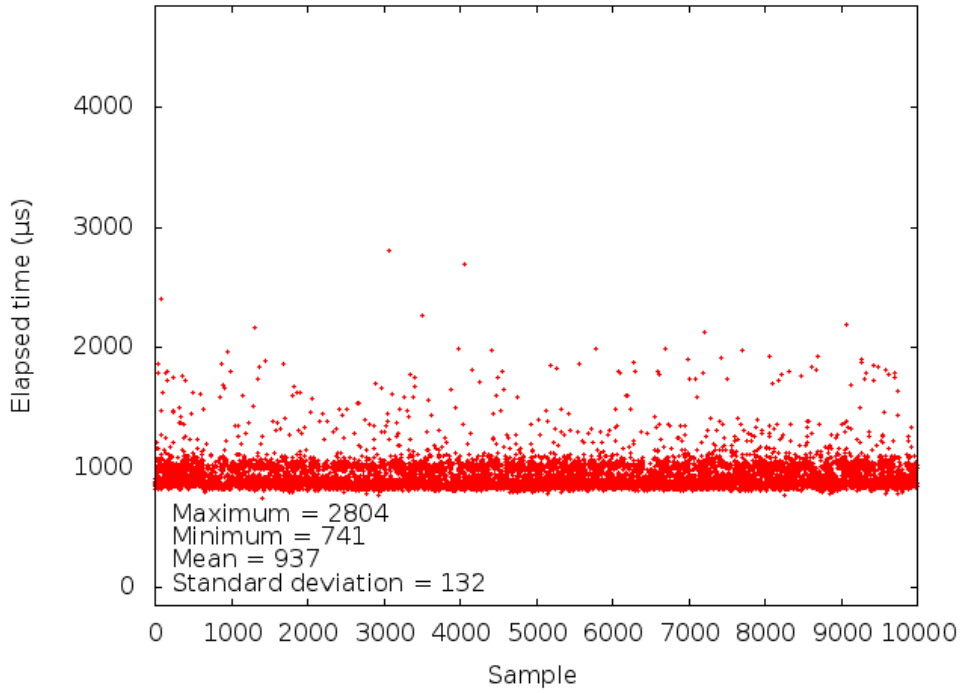

Figure A.2: Response times for Unite CS on RHEL7 running the real-time kernel. Load: `dohell`

A.2 Concurrent memory allocations with no real-time promotion

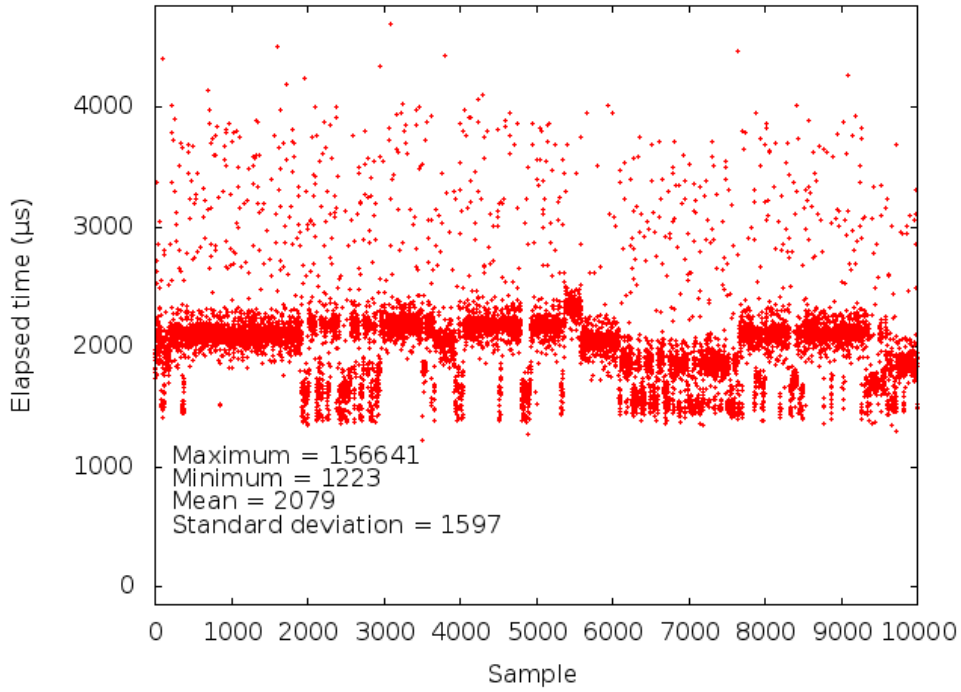

Figure A.3: Response times for Unite CS on RHEL7 not running the real-time kernel. Load: `stress --vm 40 --vm-bytes 256M`

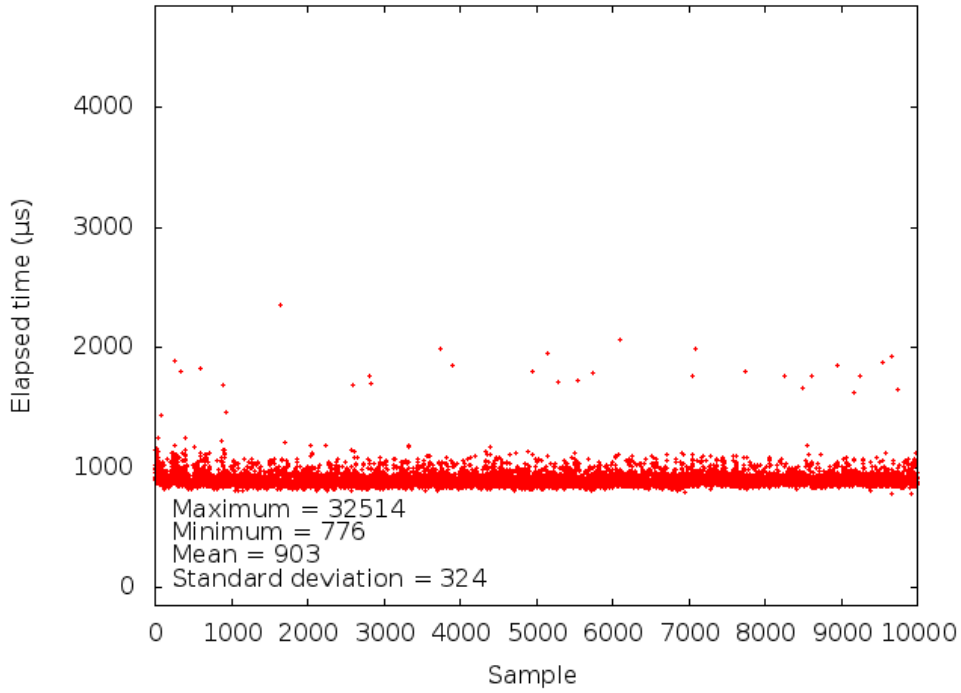

Figure A.4: Response times for Unite CS on RHEL7 running the real-time kernel. Load: `stress --vm 40 --vm-bytes 256M`

A.3 Stressing CPU with no real-time promotion

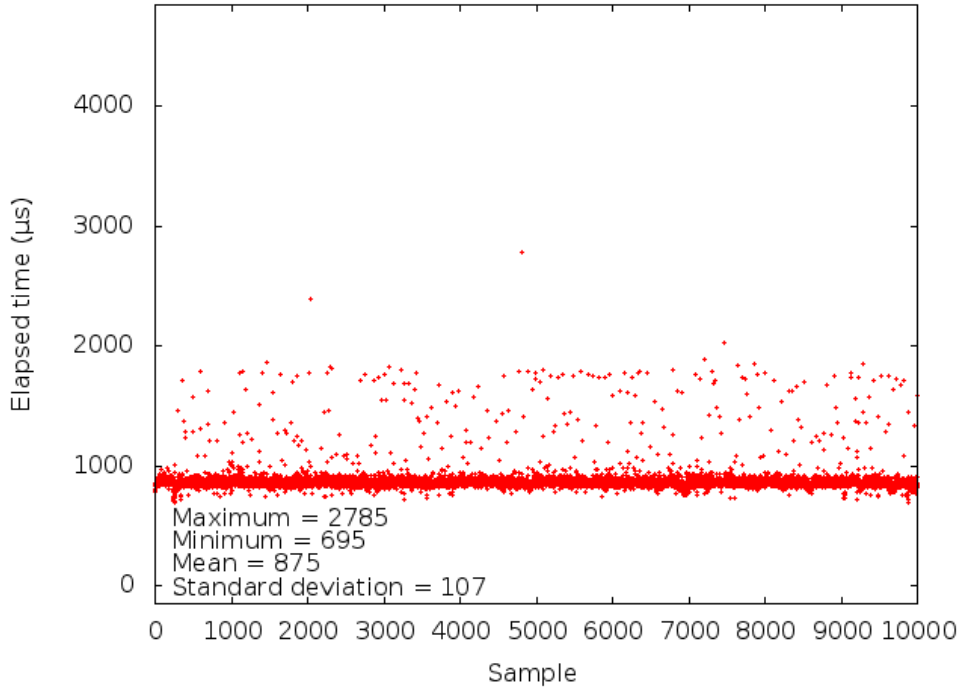

Figure A.5: Response times for Unite CS on RHEL7 not running the real-time kernel. Load: `stress --cpu 100`

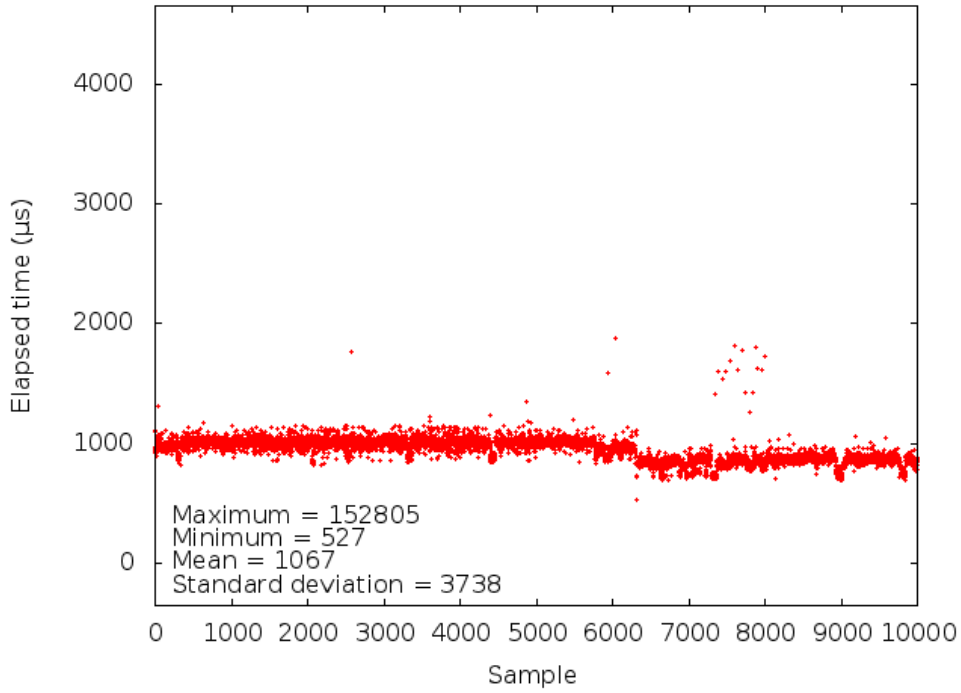

Figure A.6: Response times for Unite CS on RHEL7 running the real-time kernel. Load: `stress --cpu 100`