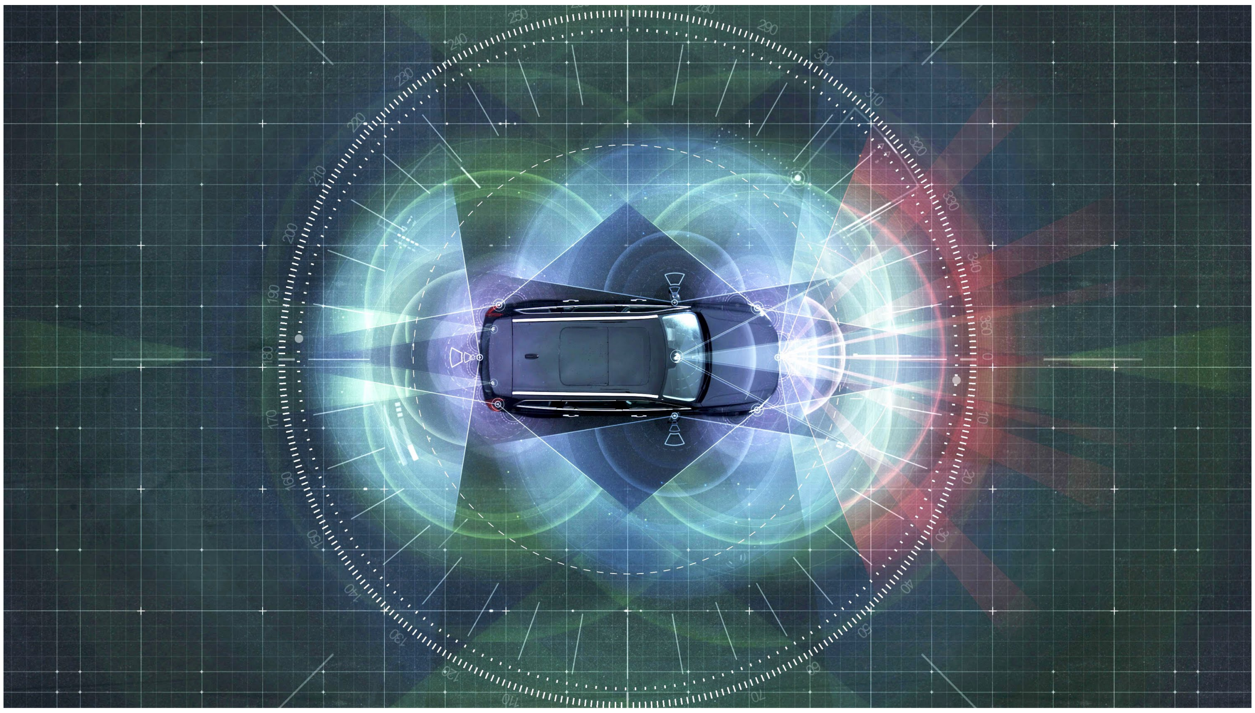


# Virtual AUTOSAR Environment on Linux

Evaluation study on performance gains from running ECU applications on POSIX threads

Master's thesis in Embedded Electronic System Design

ATHANASIOS KOLTSIDAS
OSCAR PETERSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Virtual AUTOSAR Environment on Linux
Evaluation study on performance gains from running ECU applications on POSIX threads
ATHANASIOS KOLTSIDAS, OSCAR PETERSON

Technical Supervisor: Mårten Hildell, ARCCORE AB
Academic Supervisor: Sven Knutsson, Computer Science and Engineering
Examiner: Per Larsson-Edefors, Computer Science and Engineering

Master's Thesis 2016
Department of Computer Science and Engineering
Chalmers University of Technology
and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover picture: Graphical representation of the sensor networks employed within an autonomous driven Volvo XC90. Copyright: Volvo Cars ©

Gothenburg, Sweden 2016

# Abstract

Testing an AUTOSAR application destined for an automotive Electronic Control Unit (ECU) can become an unpredictable situation, since the hardware availability of the target platform can be limited, if at all available. This thesis set out to provide developers with a virtual testing platform that would emulate the hardware behaviour in the shape of a Linux C-based application. The study was extended towards evaluating potential performance gains from running the developed application on high-end computers with similar performance levels as current powerful ECU hardware, which are able to host a UNIX-based operating system and execute multiple threads in parallel using the POSIX standard. The assessment of the generated results is realised in terms of *correctness* of a predefined execution scenario, *performance* comparison to a set of reference results and standard real-time *timing* constraints for automotive software.

Keywords: Linux, Adaptive AUTOSAR, POSIX, Operating Systems, Embedded Systems, Real-Time Systems.

# Acknowledgements

First and foremost, we would like to thank Johan Ekberg of ARCCORE AB for trusting us with this fascinating project work and providing us with all the necessary means to achieve our goals. Our deepest gratitude is due to our other technical supervisor in ARCCORE AB, Mårten Hilldel, without whose knowledge and day-to-day help this project would have not been successful. Special thanks also go to Simon Gaynor for his prior work and assistance on technical issues that troubled us during the course of our thesis. Our days in ARCCORE AB would be much less enjoyable without the support of all the employees of the company, who embraced us as their own and transformed the offices into a dream-place to work.

We would also like to thank our academic supervisor Sven Knutsson, for his valuable comments on our project documentation. Furthermore, our true-hearted appreciation goes to our examiner, Per Larsson-Edefors, for his critical feedback during the project and the motivation to produce work of high standards.

Last but not least, we thank our families and friends, for their patience, their love and uninterrupted support throughout our studies.

Athanasios Koltsidas, Oscar Peterson, Gothenburg, June 2016

# Nomenclature

| | |
|---|---|
| *API* | Application Program Interface |
| *AUTOSAR* | Automotive Open System Architecture |
| *BSP* | Board Support Package |
| *BSWM* | Basic Software Module |
| *CDD* | Complex Device Drivers |
| *CPU* | Central Processing Unit |
| *CSE* | Computer Science and Engineering |
| *E/E* | Electrical / Electronic |
| *ECU* | Electronic Control Unit |
| *EEC* | Electronic Engine Control |
| *FCFS* | First Come First Serve |
| *GPU* | Graphical Processing Unit |
| *GUI* | Graphical User Interface |
| *Hz* | Hertz |
| *I/O* | Input-Output |
| *IDE* | Integrated Development Environment |
| *IEEE* | Institute of Electrical and Electronics Engineers |
| *MCAL* | Microcontroller Abstraction Layer |
| *MCU* | Microcontroller Unit |
| *OS* | Operating System |
| *OSEK* | Open Systems and their Interfaces for Electronics in Motor Vehicles |
| *PC* | Personal Computer |
| *POSIX* | Portable Operating System Interface |
| *RR* | Round-Robin |
| *RT* | Real-Time |
| *RTE* | Runtime Environment |
| *RUP* | Rational Unified Process |
| *SJF* | Shortest Job First |
| *TBD* | To Be Determined |
| *UI* | User Interface |
| *UML* | Unified Model Language |
| *UNIX* | Uniplexed Information and Computing Service |
| *VM* | Virtual Machine |
| *s* | seconds |

# Table of Contents

# List of Figures

# List of Tables

# 1
# Introduction

This chapter is a synopsis of the content of the thesis project. It begins with a small background to Electronic Control Unit (ECU) development and why this Master's thesis is relevant, in the context of the problem definition. We then continue on with a brief description of the previous work done in the related fields, following up with the project goal. Finally we elaborate on the project limitations and potential risks that may apply to our thesis work.

## 1.1 Background

In the past, all control mechanisms in a conventional vehicle were implemented mechanically [1]. During the 1980s, there was a major shift towards electronic control, with a first joint attempt from Intel and Ford, producing the first fully electronic control unit, which they called Electronic Engine Control (EEC) (nowadays known as ECU). The basis of this ECU was a modified version of the Intel 8061 processor family. It is remarkable how 8061 and its successors were the basis of almost all ECUs produced by Ford until 2000 [2]. Early ECUs were based on analog circuitry, because analog circuits are not clock-speed-dependent. There was a brief switch to hybrid ECU systems, comprising both analog and digital logic, right before the final transition to entirely digital circuitry ECU systems around 1987.

The transition from analog to digital took place because it coincided with the time when digital electronics became fast enough to be able to process data and respond in a real-time concept [3]. Digital systems illustrated better performance and easier manipulation as the ECU technology progressed. Nevertheless, this revolution in automotive electronics led to an immense growth in ECU software applications, which were created to perform crucial vehicle operations.

Newly developed vehicle applications started to grow in size and complexity, motivated by many heterogeneous factors. According to [4] the propelling elements of this growth are indicatively the following:

- The continuous demand for lower costs, better comfort and higher security.
- A substantial rise in the number of ECUs used in a single vehicle, as well as in the functionality shared amongst the included units.
- The target ECU hardware along with the network interfaces (e.g. LIN, CAN, FlexRay and recently Ethernet) is constantly diversifying.

With the structure and shape of ECU-targeted applications becoming quite complex, the idea for a standardisation of the software architecture and the development methods started gaining increased attention within the automotive industry. To that end, different stakeholders have agreed on a standardisation of basic software functionality of automotive ECUs, known as AUTOSAR (AUTomotive Open System ARchitecture) [5]. The motivation behind this collaboration was mainly focused in containing the complexity produced from the expansion of the implemented functionality, as well as providing the products with some plasticity; enabling them to easily incorporate upgrades and modifications. Moreover, providing scalability for the developed software would improve its achieved quality and reliability.

This breakthrough has signaled the beginning of a new era in automotive application development, simplifying the process of creating additional functionality for a control unit. The impact of AUTOSAR on the automotive software development habits is becoming more significant with time and the automotive tool chain area was the first to be heavily affected. As a concept, AUTOSAR allows the development process to shift from actual implementation steps towards a series of configuration stages instead, which unchains the whole procedure from additional complexity by automatising it. Naturally, this revolutionised the development's tool world as well, since AUTOSAR enables new capabilities and features [6].

A plethora of software companies adopted the new standard and built development environments tailored to the AUTOSAR specifications. One of the software companies that embraced the AUTOSAR standard was ARCCORE AB, the host company of this project. Since September 2009 ARCCORE is an appointed Associate Member of the AUTOSAR consortium, a collaboration that signaled the beginning of a new era for the company itself [7]. Some significant players in this business area are *Vector*, *Mentor Graphics*, *QTronic Infineon*, *Artop* and *ETAS*. Each of these companies provide some sort of AUTOSAR software development solution, some also expand into hardware development, *Renesas* [8] being an example.

## 1.2 Problem Definition

Providing such a friendly environment for AUTOSAR applications' developers, led to a speed up in the process of ECU modules integration. This created the need for a more frequent and accurate testing environment that would exhibit the target hardware platform's properties. Considering that all ECU applications are so hardware-dependent, the ideal case would be to provide every designer with an actual board in order to download the application and test it while still in the development phase. Nevertheless, this is usually not the case for reasons that are explained in Section 1.4 along with the contribution of our work towards that end.

The obstacle that this project set out to lift, is to facilitate the test and verification process of AUTOSAR ECU software applications that are in the development phase. Since the intended hardware is usually a restricted resource by nature and sometimes it is even simultaneously being developed, the lack of adequate testing gives rise to a precarious situation. This will be addressed by developing a verification platform, which will help reduce the risks and costs of developing new ECU software, invoking the target hardware board for testing only in the latter stages of the process. This platform, can be classified as a Virtual Machine (VM) that incorporates the AUTOSAR OS functionality on top of a Linux distribution. Since it falls outside the scope of our thesis project to focus on the aspect of VMs, we will briefly provide a definition and a visual example (Figure 1.1) in order to clarify the intended system structure.

Our implementation can be categorised as a process VM. This type of VM translates the OS and the user-level instruction set, which compose the virtual platform, to the corresponding parts of the host platform [9]. In Figure 1.1, we illustrate the different perspective that an application realises, compared to the actual virtual architecture. Regardless, the interpretation of our platform as a VM is based on the fact that applications executed within their corresponding layer do not realise whether they are executed on a native AUTOSAR OS or through the ported OS that our application comprises.

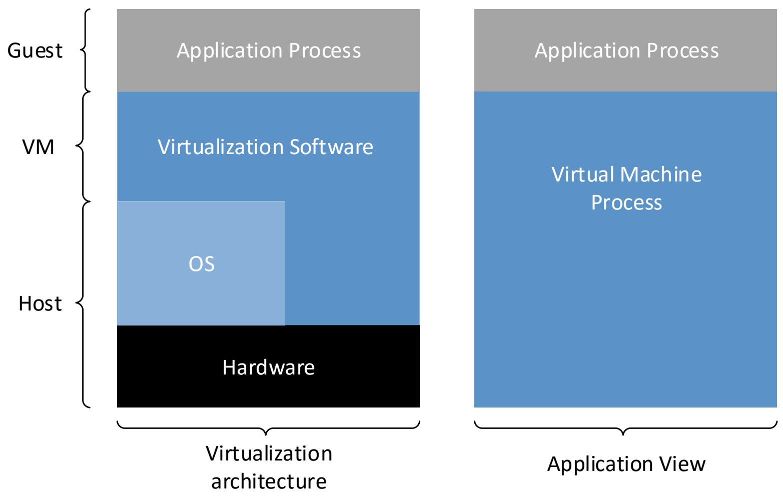


**Figure 1.1:** The right figure shows what an application sees in regards to a VM, it does not care what is underneath it. On the left we can see the actual incorporation of a VM, it is integrated together with the host OS and the hardware.

While the initial idea was to speed up the ECU software development process by replacing the target hardware with the AUTOSAR functionality in the shape of a Linux application, we will also investigate the potential benefits from shifting towards Linux-based multi-core ECU solutions, as the future path for the AUTOSAR consortium and the attempts to achieve maximum performance while maintaining the required reliability.

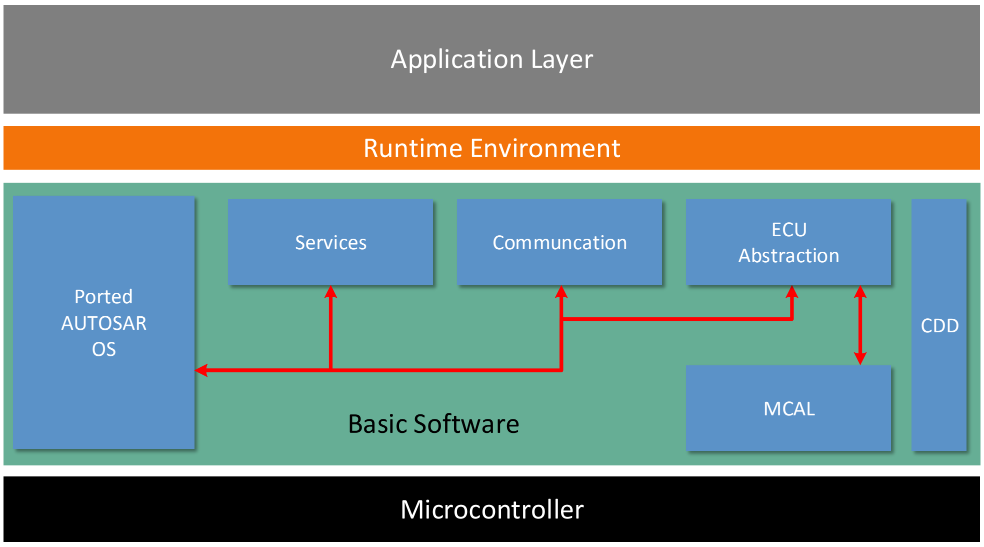


**Figure 1.2:** Simple layout figure of the AUTOSAR stack, with the OS renamed to "Ported AUTOSAR OS" since the AUTOSAR OS is being run on top of a Linux distribution instead of immediately on the hardware.

However, as we will illustrate in Chapter 2, the Basic Software Module (BSWM) as described in the AUTOSAR specifications is very complex and far too voluminous for the scope and duration of a Master's thesis work and thus our efforts are dedicated to solely port the AUTOSAR OS functionality to the test platform under development.

In order to evaluate the gains of invoking Linux and multi-core systems within our implementation, a test application provided by ARCCORE will act as a comparison framework between the Linux application and a current ECU hardware system. This evaluation shall be in regards to performance, correctness and timing. Performance is related to execution metrics, correctness to task scheduling and timing to Real-Time (RT) application constraints.

## 1.3 Motivation

Our thesis work is influenced by all the prior attempts to combine embedded OSs with mainstream platforms such as Linux or Android OS [10]. More specifically, we base much of our work on sincere efforts within ARCCORE to join the two worlds, in the context of *Adaptive AUTOSAR* [11]. Adaptive AUTOSAR is considered as the next step for the consortium and we will explain it in more detail in the next chapter. More specifically, Adaptive AUTOSAR as a concept is relevant to our project since it will be using a POSIX-thread (Portable Operating System Interface) capable OS according to the initial vision [11], with Linux distributions being the front-runner. Since this is a very promising path for the automotive software industry, it is obvious that there exist multiple attempts to port and translate the AUTOSAR Operating System (OS) functionality to a Linux-based platform, e.g. qTronic's *Silver* product [12] or Mentor's virtual platform for AUTOSAR [13]. However, developing an in-house solution provides ARCCORE with the flexibility to use the outcome of this project as the guideline and the foundation of future attempts to fully transfer the AUTOSAR BSWM functionality to Linux systems.

**Table 1.1:** Summary of the CoreMark benchmark results for processors related to our project. The first two rows include the metrics for the host machines used during the testing of our applications, noting that they are not far off from the high-end NVIDIA ECUs

| Processing Unit | Frequency | # Cores | Coremark | Coremark/Core |
|---|---|---|---|---|
| **Intel Core i7-3720QM** | 2.6GHz | 8 | 85209 | 10651 |
| **Intel Core i5-4300M** | 2.6GHz | 4 | 46085 | 11521 |
| **NVIDIA Tegra K1** | 2.3GHz | 4 | 31221 | 7805 |
| **NVIDIA Tegra X1** | 1.9GHz | 4 | 30638 | 7659 |
| **Intel Atom E3827** | 1.74GHz | 2 | 10820 | 5410 |
| **Renesas RX71M** | 240MHz | 1 | 1045 | 1045 |
| **STM STM32F756NGH6** | 200MHz | 1 | 1002 | 1002 |
| **Renesas RX64M** | 120MHz | 1 | 525 | 525 |

Driven once more by the technological advancements in electronics, invoking other parties into the automotive field as the market expands into a wider field, requiring more high end capabilities. Thus making the bridges smaller between an automotive purpose ECU and a general purpose computer. The cooperation between *NVIDIA* and *Volvo* in the*Drive Me* project [14] is a perfect example of Microcontroller Units (MCU) requiring more performance.

Table 1.1 shows performance results of various processors, including some mainstream automotive ECUs. These results were generated using the benchmark suite *CoreMark* [15], which provide a reference metric in order to quantitatively compare different types of processing units with each other. The two host machines used in this project are also included in this table, along with the Renesas and STM boards, which are very common when it comes to AUTOSAR ECUs. The RX64M is similar to the ECU used to produce our reference results (which can be seen in Appendix A). There is also an entry for an Intel Atom processor. It belongs in the same processor family as the one used in MinnowBoard MAX (Atom E38XX), which is the embedded alternative we chose to investigate in this project. Exploring the scores of the benchmark (where higher is better), we notice that the high-end *NVIDIA Tegras*, which were released in 2014 [16], are close to the performance levels of state-of-the-art Personal Computers (PC), consequently today's versions should be even faster. Thus, the performance gains from testing our application on our PCs can be considered relevant to what can be expected from platforms like the future NVIDIA ECUs.

## 1.4 Project Goal

The goal of the project is to develop a platform where AUTOSAR and Linux can coexist. The results would be a Linux C-based platform, which will be capable of running an AUTOSAR application on top of a Linux OS environment, along with an evaluation report comparing the evaluation platform with the performance of a present-day ECU board. This goal comprises a checklist of attributes for our implementation, as described below:

- The Linux platform should be based on the AUTOSAR OS properties, consequently following the OSEK specification [17].
- It should also follow the same scheduling sequence and prioritisation for the executed tasks, providing the execution framework with the required *correctness*.
- While following the requirements of AUTOSAR OS, it should simultaneously exploit all the available built-in Linux capabilities, such as *POSIX threads* [18], in order to speed-up the various OS operations.
- The platform must be based on a generic foundation; including all the necessary modules according to the AUTOSAR basic software description, However, without the need of additional adaptations for every new MCU that is going to be used as the new target hardware.
- As mentioned in Section 1.2, the prototype should respond well to real-time restrictions, fulfilling *timing* constraints of real-time applications.
- Additionally, the developed platform should operate at an acceptable (or higher) level when it comes to OS operations completion times relative to the *performance* of the target hardware.
- Moreover, we considered that it would be fascinating to use the Yocto Project [19] and transfer our platform from a standard Linux environment to a native embedded Linux OS distribution, tailored for the selected evaluation board we decided to work with, the MinnowBoard MAX [20].
- Finally, it only seems natural to contemplate on the experimental performance results from executing the same test applications on the native embedded Linux installation residing on the test board.

## 1.5 Scope Limitations

In this section we present and then elaborate on the limitations enforced by the scope of our thesis.

Since a variety of AUTOSAR applications are already available from the company, we will not develop any new test application from scratch in order to test the system. As already discussed, we will port the AUTOSAR OS functionality and thus work with a specific application implemented within ARCCORE, that tests all the fundamental OS operations, called OsSimple. As stated in Section 1.2, the AUTOSAR BSWM is too complex to port as a whole and also redundant in relation to the scope of our thesis project. Hence, we will not interface any other part of the AUTOSAR Basic Software to our platform but the OS module. To that end, we focus on the OSEK specifications, since studying the massive AUTOSAR documentation in detail does not serve the purposes of our thesis.

As discussed in Section 1.3, we will not test the platform on a state-of-the-art MCU intended for the purpose of housing adaptive AUTOSAR. Since these MCUs are approaching the performance of high-end laptops we deem it sufficient enough to test on our laptops.

Lastly, we will not be looking into implementing the RT concept on the Linux distribution intended for our host machines or for the MinnowBoard MAX, since the PREEMPT_RT mainline Linux kernel package is available to patch the standard Linux kernels and is also included in recent Yocto Project releases [19].

# 2 Theory

This chapter goes in-depth in different areas relevant to this Master's thesis. Its purpose is to set the theoretical foundation for the work that was done during the project and thoroughly explore related fundamentals in order to give a more complete picture of the subjects touched upon in this report.

We set off giving a general description about operating systems, illustrating some related categories to our project, such as embedded OSs. Next we discuss process- and threads scheduling followed by the AUTOSAR OS functionality, in the concept of presenting the OSEK standard specifications. Extending our description of the AUTOSAR standards, we elaborate on the Adaptive AUTOSAR concept and its fundamentals, wrapping up the chapter with a thorough presentation of the Yocto Project and its comprising elements.

## 2.1 Operating Systems

An operating system is a collection of software components which are designed to manage a computer's hardware. Probably the most well-known OSs historically have been Microsoft Windows, UNIX-based (Uniplexed Information and Computing Service) distributions (e.g. Linux Ubuntu) and Macintosh OS X. These three alternatives have been the basis of the majority of personal computers and handheld devices ever used, therefore users interact with at least one of them on a daily basis.

Figure 2.1 shows an architecture of the general idea behind an OS. With OSEK being no exception, it inherits this concept as its basis, as we will discuss later on in this chapter. The illustrated architecture contains an application layer comprising service programs, which control alarms and tasks, shared libraries and a kernel. The kernel is the part of the OS that handles the resource allocation (e.g. hardware access) for a computer system. As will be discussed in the upcoming section, the kernel is responsible for restricting different user and system applications from accessing restricted memory areas [21].

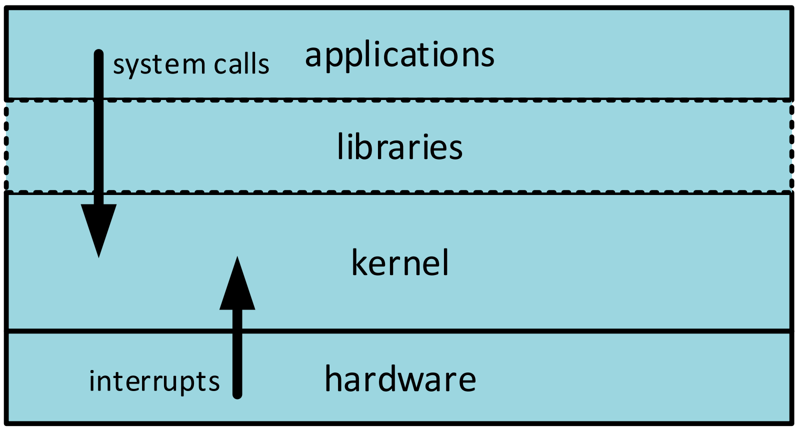


**Figure 2.1:** Generic OS software architecture layout, the kernel handles and schedules system calls from applications as well as interrupts from the hardware

### 2.1.1 Processes and Threads

Processes can be described with multiple definitions, all of which converge to the same general idea: a *process* is a program ready to be run on a Central Processing Unit (CPU) and consists of multiple elements, two of which are absolutely vital; the source code and a set of data that is required for the execution. Besides these two characteristics, a process includes some extra components that facilitate the scheduling and execution of this process, such as:

- A process identifier; the identity number of each process
- A state; as we will see later in Section 2.1.4, depending on the scheduling policy there might be from two to even more than five possible process states
- A priority; relative priority to the rest of the processes
- Memory pointers; both towards the source code and to the required set of data

These components, among others, comprise the data set that a process holds at its *process control block* [22]. The process control block is created and managed by the OS. It provides the ability to preemptively interrupt processes while executing and later resume them with no evident sign of the suspension. This concept is also known as *context* or *process switch* [23].

In a more coarse description, the program code, the necessary data structures, the process stack and the process control block compose the *process image*, the entity that contains all the necessary elements for the process to be runnable on the CPU [22].

The most important aspect of using processes, is to provide the OS with adequate control over concurrent allocation of hardware resource among programs executing on the CPU. In order to keep track of all the logistics, the OS creates a *process table* where it stashes all the vital information for each of the executing (or suspended) processes (e.g. process ID, priority etc.) [22]. However, this need for transparency and concurrency takes its toll on the performance of the system; the creation of a new process translates into creating a new memory space for it and copying the program code to that segment and allocating a temporary data stack. Moreover, a switch between processes on the CPU is quite time-consuming. Besides resetting the CPU registers, program and stack pointers, the OS usually requires cache entries as well as the translation lookaside buffer to be invalidated [23]. To make matters worse, modern CPUs can support the execution of more processes than they can fit in the main memory, hence this might lead to transferring a process image from the hard disk drive to the main memory.

In an attempt to ameliorate this situation, OSs started supporting *multithreading*. This comprises the concept of *threads* and the ability of the OS to schedule and execute, on the CPU, multiple concurrent execution paths enclosed by a single process. There can be many different alternatives on how various processes are scheduled along with their multiple instruction traces (i.e. threads).

In order to distinguish a process from its included threads, one can point out the two fundamental characteristics of a traditional process which are [22]:

- Resource allocation unit and process image owner: a process holds the virtual address space that includes the process image and occasionally acts as the owner of hardware resources such as Input/Output (I/O) devices and the main memory.
- Execution and scheduling element: A specified execution flow is followed, possibly through multiple programs. The executed process might intersperse with other processes as well. For that reason, processes keep a scheduling priority and an execution state within the scheduled entity.

In this context, the process can be treated as the container entity, the framework which provides all the necessary tools and data for the threads to execute. Threads on the other hand, are interchanged between different scheduling states, receive CPU time according to their relative priority and can follow different execution paths within the same process source code.

It is worth taking into consideration that in case that the CPU is single core, it would naturally not support parallel execution, but it may still support multithreading. This is achieved by using a sophisticated scheduling algorithm, that allocates resources to the most acute activity, masquerading the execution so that it appears to be in parallel since all threads are continuously progressing although not atomically.

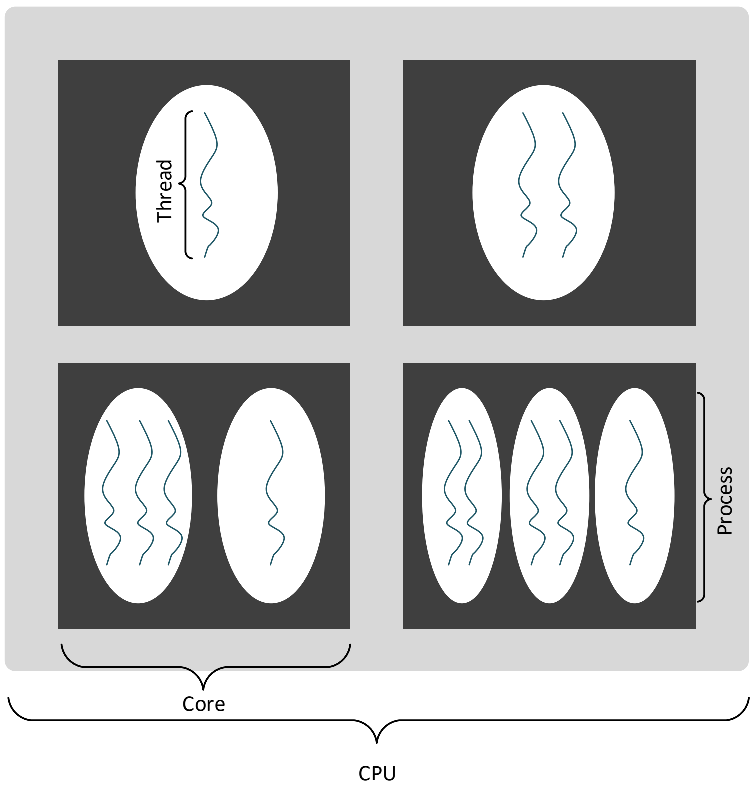


**Figure 2.2:** In this figure we provide a visualisation of the differences between the concepts of processes and threads. As it can be seen, a process (white elliptic shapes) can include one or more threads, while there can exist one or more processes executing over a single CPU core.

### 2.1.2 POSIX

As presented in Section 2.1.1, the concept of threading existed in advance of the POSIX standardisation. For more than a decade, there have existed multiple thread interfaces, not always compatible with each other and hence with limited portability between operating systems. In 1995, the Institute of Electrical and Electronics Engineers (IEEE) released a C-based thread standardisation, the IEEE 1003.1c standard also known as POSIX, in an attempt to tackle this issue and increase the portability of multi-threaded programs [18].

#### 2.1.2.1 Thread Management with POSIX Standard

After multiple revisions, the POSIX standard currently includes more than 100 POSIX threads (or Pthreads) function calls, that can be divided into the following categories [24]:

- *Thread manipulation*: operations forced on threads (e.g. creation, termination etc.) or operations that set specific attributes to threads,
- *Mutexes*: functions that apply mutual exclusion (Section 2.1.3) on the thread execution by creating / destroying and locking / unlocking mutexes,
- *Other synchronization structures*: functions that manipulate locks and barriers and
- *Condition variables*: calls that handle inter-thread communication between threads that are controlled by the same mutex. Comprises creation, waiting and signaling upon pre-specified values of a condition variable.

All POSIX-based threads have specific properties which are stored in the form of attributes within standardised data structures:

- *typedef unsigned long int pthread_t*: the individual thread identifier
- *typedef struct pthread_attr_t*: the attributes object that includes all the vital information for the creation of a thread (e.g. the stack size, stack address and the scheduling priority of the thread to be created).

Many of the provided function calls are falling outside the scope of our thesis project, since they provide very sophisticated operations. Nevertheless, we present some of the key parts of the Pthreads API that were utilised in our thesis project (also shown in Table 2.1).

The beginning of the life of a new thread is signaled with `pthread_create`. The formal syntax for this call is [25]:

```
int pthread_create (pthread_t *thread, const pthread_attr_t *attr,
void * (* start_routine) (void *), void *arg);
```

The new thread starts executing the defined `start_routine`. It can be terminated by either calling `pthread_exit`, returning from the `start_routine` or getting canceled by calling `pthread_cancel`. The argument `attr` points to the specific set of attributes comprising the structure described previously. This attributes object is initialized using the `pthread_attr_init` function. Upon successful creation, the `pthread_create` function stores the identifier of the newly-created thread in the pointed by `*thread` memory space. It should be noted that after the thread is created we are allowed to call `pthread_attr_destroy` and discard the thread attributes object.

In the case that the `pthread_exit` function is called, the calling thread is initialising all the cleanup handlers along with any destructor functions that handle thread-specific data [25]. According to its standard syntax:

```
void pthread_exit(void *retval);
```

if the terminating thread is joinable, an other thread included in the same process can utilise the `retval` and call:

```
pthread_join(pthread_t thread, void **retval).
```

As shown in Figure 2.3, a thread that calls `pthread_join` waits for the thread specified in `thread` to terminate via calling `pthread_exit`. In case the thread has already terminated, `pthread_join` returns immediately.

There are different ways described within POSIX to "kill" a thread. The two basic alternatives are `pthread_exit` and `pthread_cancel`. While the functions that we described for the former comprise `pthread_cancel` as well, the main distinction between the two calls lies in the fact that the individual steps that lead up to the actual thread termination happen *asynchronously* in the `pthread_cancel` call [25]. In spite the fact that this might seem like a minor difference, as we will showcase in Chapter 3, the use of one or the other can substantially affect the performance of an application that invokes multiple task activation and termination operations.

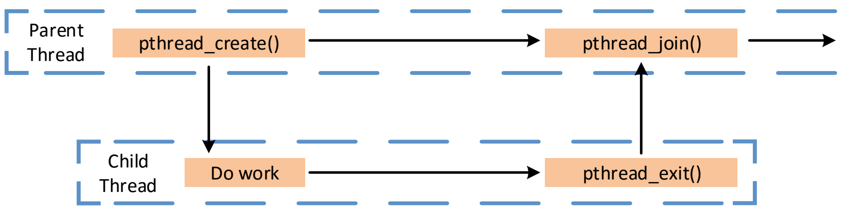


**Figure 2.3:** The parent thread creates a child thread with the help of `pthread_create` then calls `pthread_join`, thereby waiting until the child thread has done its work and called for a thread cancellation via `pthread_exit`

#### 2.1.2.2 Pthreads and Synchronisation

Besides thread management, POSIX provides a plethora of synchronisation-related standard functions and structures in order to empower the parallel execution with mutual exclusion (Section 2.1.3).

### 2.1.3 Process Interaction

As illustrated in Figure 2.2, on a current multi-core CPU, more than one process can execute in parallel. Under multiple scenarios, there might exist the need of *inter-process communication*, two or more processes may require to exchange produced results or share memory locations' content. However, as with any other newly introduced concept, there have been some notable issues in achieving safe and correct interaction between separate processes [26]:

- What would be a convenient way to establish effective communication between processes?

**Table 2.1:** Coarse description of the Pthread function calls that are related to thread management, based on the *Linux Manual Pages* [25]. The full list of POSIX functions can be found in Appendix ...

| **Function Call** | **Description** |
|---|---|
| `pthread_create` | Creates a new thread within the calling process; the new thread starts executing from a defined starting routine |
| `pthread_exit` | Terminates the calling thread and returns a value in case another thread wants to join with the thread under termination |
| `pthread_cancel` | Sends a request to cancel the specified thread; when and whether the specified thread will react to this request cannot be pre-determined |
| `pthread_join` | Waits for the specified (joinable) thread to terminate; if the thread has already terminated, the `pthread_join` call returns immediately |
| `pthread_yield` | The calling thread yields the processor in order to allow another ready thread run |
| `pthread_attr_init` | Initializes the thread attributes object specified as a parameter with default attribute values |
| `pthread_attr_destroy` | The thread attributes object is destroyed, when no longer required. The thread created using these attributes is not affected by the destruction of the object |

- How could we ensure that each process will respect other processes that attempt to access the same resource (the concept of *critical regions*)?
- How could we enforce the desired sequence of execution, by synchronising the processes that need to interact with each other?

Even though our project is focused on the thread-level, these concerns affect this type of scheduling as well, since the threads are the scheduled entities of the processes, as described thoroughly in Section 2.1.1. These three questions posed above, are the root causes for some traditional issues related to process communication, such as the concept of *race conditions.*

#### 2.1.3.1 Race Conditions and Critical Regions

In case two or more processes perform closely related functions, there might be the need to share data segments on which all of them are able to both perform read and write operations. In a case like that, all the prerequisites exist for *race conditions* to occur. The outcome of this situations is non-deterministic and lies solely on the execution scenario that takes place every time that the race conditions are repeated: depending on which of the processes runs first the produced result will vary [26].

Evidently, the requirement for a viable solution that would allow processes (and consequently threads) to access atomically shared resources (e.g. memory regions)

**Table 2.2:** Coarse description of the Pthread function calls that are related to thread synchronisation, based on [25]

| **Function Call** | **Description** |
|---|---|
| `pthread_mutex_init` | Initialises the mutex using the specified attributes |
| `pthread_mutex_destroy` | Destroys the mutex object |
| `pthread_mutex_lock` | The specified mutex object shall lock upon calling; if the mutex is already locked, the thread calling this call will block |
| `pthread_mutex_unlock` | Releases the specified mutex object |
| `pthread_cond_init` | Initialises the condition variable with pre-specified attributes |
| `pthread_cond_destroy` | Destroys the given condition variable |
| `pthread_cond_wait` | Blocks on the specified condition variable |
| `pthread_cond_signal` | In case any threads are blocked on the specified condition variable, unblocks at least one of the blocked threads |
| `pthread_barrier_init` | Initialises a barrier object based on the pre-specified attributes |
| `pthread_barrier_destroy` | Destroys a barrier attributes object |
| `pthread_barrier_wait` | Synchronises participating threads at the specified barrier; the calling thread shall block until the required number of threads have called the wait function specifying this barrier |

is crucial to allow uncomplicated inter-process communication.

This requirement led to the introduction of two key concepts in the existing communication schemes between processes (and between threads), *mutual exclusion* and *critical regions*. Mutual exclusion describes the ability to *exclude* all but one process from accessing and manipulating shared data segments, which were characterised as critical sections. However, in order to achieve faultless process communication based on the critical regions paradigm, the following fundamental conditions have to stand [26]:

1. No more than one process can be simultaneously present within the same critical section.
2. There can be no presumptions drawn for the technical specifications of the CPUs included in the system.
3. A process (or thread in our case) that is not running inside a critical section is unable to block others.
4. There can be no process *starvation* under the critical regions scheme (the situation when a process waits indefinitely to access a region by getting bypassed constantly by other processes).

Essentially, the mutual exclusion concept is visualised in Figure 2.4. With the help of the priorities concept, we can understand the sequence of events:

- Task B is the only ready-to-execute task and thus gets the CPU and starts running,
- After a short period of time, since no higher prioritised task is ready to execute, Task B enters a critical region and continues its execution process,
- Task A becomes ready while Task B is still in the critical region and hence it cannot be preempted, according to Condition 3 and
- finally, Task B exits the critical region and Task A is unblocked and allowed to enter it.

In the upcoming sections, we will elaborate more on the fundamental concept of mutual exclusion and all the concepts that were introduced throughout the years, from the more primitive solutions to the most advanced ones.

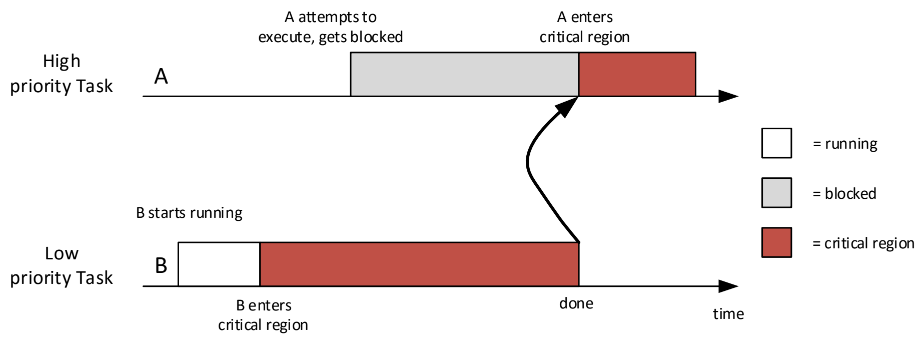


**Figure 2.4:** The lower priority task B enters its critical region since no other higher priority task is ready to execute, the higher priority task A becomes ready but is blocked from execution because task B is in a critical region. Task B eventually exits its critical region, letting A into its critical region

#### 2.1.3.2 Busy Waiting in Mutual Exclusion

In this section we will mention and describe briefly all the alternative mutual exclusion methods that invoke *busy waiting*, the situation when the processes that require to access a critical region and fail to do so, do not get suspended but keep executing a condition until they get clearance to enter the region.

The first example of this category is the method that suggests the *disabling of interrupts*. The idea behind this straightforward method is that once the process gets access to the critical region, it switches off all the interrupts and only re-enables them the moment before it leaves the critical region. This is a raw way to deny the CPU from the ability to re-schedule and possibly interrupt the current process from finishing executing the specific critical section. Nevertheless, despite being effective it is an ill-advised strategy to allow user processes to control the functionality of system interrupts. Firstly, this strategy can only be implemented within a single-core system: suspending interrupts only affect the specific core that the process is executed on, while the rest of the cores can still access the same critical section. Moreover, the kernel itself might attempt to disable the interrupts for a few instructions, in order to facilitate some of its operations and thus having a user process messing with the kernel's work might lead to precarious situations [26]. According to the aforementioned, this approach can be considered outdated and dangerous for the flawless function of the OS.

There are many more attempts that partially fulfil the four fundamental conditions of mutual exclusion, with some being more successful and sound. However, all of them include the concept of busy waiting and thus could not be considered strong candidates as the go-to solution. One of the most popular approaches is the use of *lock variables*. Essentially, a shared variable would be initialised as 0 and whenever a process attempted to enter a critical section it had to go through this lock variable: if it was 0, it would update it to 1 and enter the region. In case the lock was already 1, the process waits until the variable is turned back to 0. The basic idea is solid, however we would need atomic access to the lock variable as well, preventing the variable to be changed *simultaneously* by two or more processes. Nonetheless, this approach along with the *strict alternation*, the *Test-and-Set Lock* and a few more suggestions constitute the preludes of the *semaphores* and *monitors* solutions [26].

#### 2.1.3.3 Sleep and Wakeup Approaches

Some of the methods that include busy waiting are correct, nevertheless they may introduce unexpected phenomena, such as *priority inversion*: let us consider the case when we have two processes, $H$ with a higher priority than $L$ which at a point in time is in the critical region. Now, $H$ becomes ready and since it cannot enter the critical section it starts busy waiting. Consequently, low priority process $L$ does not leave the critical region as it can never be scheduled while $H$ is already executing. The result of this is that $H$ indefinitely waits, and starves waiting for access to the critical region which is held by a lower-priority process and this is where the name of the term originates from [26].

Let us now present the most notable methods that are based on blocking instead of busy waiting when not allowed to access a critical section, with the headline approach being the *semaphores* paradigm.

The introduction of the concept itself came in 1965 from E.W. Dijkstra, a Dutch computer scientist who proposed the use of a counter that would keep track of the number of wake-ups. This variable was named *semaphore* and it would indicate whether there were no wake-ups pending (equal to 0) or one or more wake-ups to be expected, if the value of the semaphore was greater than zero. Dijkstra also equipped his structure with two explicit operations that are equivalent to the *sleep* and *wake-up*, namely the *down* and *up* functions.

The *down* operation examines whether the semaphore value is greater than zero, in which case it decrements it and continues executing. In the case that the semaphore value is equal to zero, the calling process is suspended (put to *sleep*), with the *down* operation still pending. On the other hand, the *up* operation, upon call, increments the value of the semaphore and in case that there is one or more blocked processes, the system selects one (the method is system-dependent) which proceeds with performing the previously blocked *down* operation and continue executing [26].

The major difference from all the previous approaches described in Section 2.1.3.2 is that both the *down* and *up* operations and consequently all their comprising steps, are executed *atomically*, meaning that no other process can interleave until all the individual actions are performed. This is the fundamental upgrade to synchronisation mechanisms that solved the race conditions situations previously explained.

A subcategory of semaphores is the binary semaphores, also known as *mutexes*. This structure only requires 1 bit of representation, since its state can only be either *locked* or *unlocked* (0 or 1). Equivalent to the *down* and *up* operations, in the case of a mutex we have the *lock* and *unlock* functions that perform almost identical steps to the operations included in semaphore; the only difference is that the control performed by the *lock* function is now limited to see whether the mutex is locked, compared to the case when the semaphore's value is greater than 1 and hence could possibly allow more than one process to enter the critical region (Figure 2.5). An advantage of mutexes over semaphores is that thanks to their simplicity they can be implemented on the user-level, however this is redundant since all current OSs provide mutex support (see Section 2.1.2 for more information on the synchronisation schemes provided in UNIX systems).

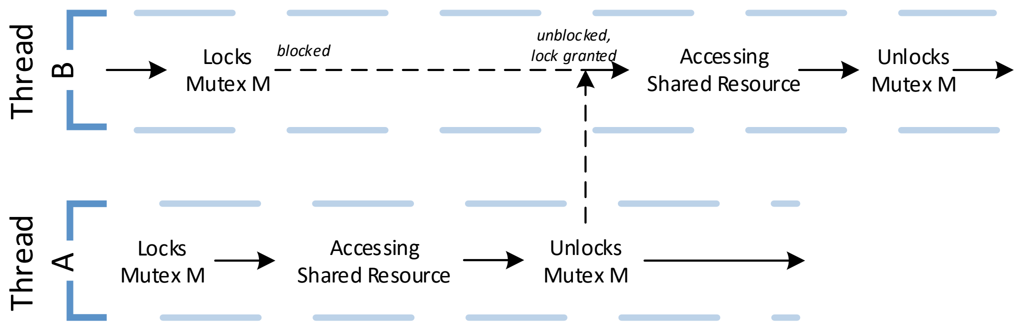


**Figure 2.5:** Illustration on how the mutexes enforce mutual exclusion between two threads that share one critical region. In relation to the busy-waiting-based mechanisms, now Thread B blocks while waiting for Thread A to unlock the semaphore, instead of occupying the CPU.

Despite the huge success that semaphores (and consequently mutexes) brought to the table, there were still issues that could occur under specific situations that would jeopardise the correct synchronisation between threads, for example *deadlocks*. As shown in Figure 2.6, there can be a situation under which the usage of mutexes can fail:

- *Task A* locks the mutex for *Resource 1*
- *Task B* locks in turn the mutex for *Resource 2*
- *Task A* attempts to lock the mutex for *Resource 2* but fails, since it is already locked by *Task B*
- *Task B* now tries to lock the mutex for *Resource 1* but also unsuccessfully as it is locked by *Task A*

Neither of the tasks are able to continue their execution, caused by both tasks requesting a resource that is occupied by the other. This is a representative example on how a system can reach the situation that is known as *deadlock* [27].

It is evident from the previous example that the use of semaphores must be proven to function correctly before implemented, otherwise it could bring the system to uncharted waters. These situations can cause issues to modern OSs and became the motivation behind the need for higher-level synchronisation schemes. This is where structures like *monitors* and *barriers* find good use.

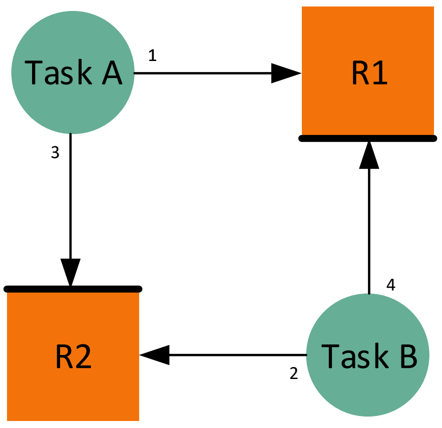


**Figure 2.6:** The numbers beside the tasks indicate the order in which they execute, resulting in a deadlock because both tasks occupy the resource the other is requesting

*Monitors* is a mean of synchronisation one level above semaphores, a set of procedures, data structures and counter variables comprise a monitor [27]. Processes can call the procedures included in a monitor at any moment in time, however they have no control over the data structures of the monitor. Allowing only one process to execute any of the procedures in a monitor, it is now the compiler's work to ensure mutual exclusion [26]. The programmer is not required to be aware of the way the compiler achieves it, however we should know how a procedure included in a monitor can communicate and inform of its status to the rest of the procedures.

This is achieved by including *condition variables* in a monitor; their purpose is to conceptualise the *block* and *unblock* paradigm used in the semaphores method by providing a pair of calls, *wait* and *signal* correspondingly. The *wait* call provides the necessary blocking point for the calling processes that are not yet allowed to execute one of the monitor's procedures, while the *signal* call is there to wake up the blocked process once the monitor procedure executed has been freed. A coarse description of what could be considered as a monitor in pseudo-code is illustrated below, the reason why we describe the monitor using pseudo-code is that it is language-related and the corresponding compiler is responsible for ensuring the mutual exclusion.

```
monitor example
    integer i;
    condition c;

    procedure do1;

    end;

    procedure do2;

    end;
end monitor;
```

Monitors might be of higher level than semaphores and take care of the mutual exclusion, however there was still a need for something more applicable to distributed systems that would also provide the ability to exchange messages between the CPUs. This was addressed by message passing mechanisms, such as the `pthread_cond_wait` and `pthread_cond_signal` as referred in Table 2.2.

The general scheme includes a pair of operations: `send(destination, &message);` and `receive(source, &message);` where the *send* provides the *destination* with a *message*, while *receive* expects a message from *source* (or anyone) and until it receives it blocks until any message or a specific one arrives. The advantage of this approach is that the primitive operations can be modelled as system calls and not be language-dependent. Despite that, there are also disadvantages related to this method, e.g. that network issues can cause a message to get lost, or the problem of identification and how the receiver can verify that the message is sent by the intended source. From another angle, the performance is also degrading when utilising message passing, especially through different CPUs or entire systems [26]. It is worth pointing out that the issue of performance forced us to explore more beneficial alternatives in handling events in our porting (more details are provided in Chapter 3).

The last and highest levelled approach that we are discussing here is the process of *barriers.* It is intended for synchronising more than two threads at a time. Barriers are in their essence roadblocks where the execution stops until all threads reach that certain point in the execution flow. The barrier set at any point needs to define the number of threads required to reach it before allowing the execution to proceed. When all threads have arrived, they start executing again at the same time. As a concept, it can be visualised with two phases, where the first phase is the arrival phase and the second the departure phase. In the arrival phase the threads are blocked until a certain amount reaches the tollgate. Once all threads have arrived, the departure phase begins; releasing the threads and allow them to continue their execution [27].

All these mechanisms and the different alternatives presented showcase the importance of realising mutual exclusion in parallel computing for one main reason, fast and fair scheduling among interacting processes.

### 2.1.4 Scheduling

Any computer with more than one process running requires scheduling, since the included threads are competing for the CPU. The decision to be made comes at the points when more than one thread are ready to execute. This is where the *scheduler* intervenes and devises the decision on which thread will receive CPU time, based on specific criteria and a scheduling policy algorithm.

#### 2.1.4.1 Process Behaviour and Scheduling Points

In order to understand the needs of each process we first have to categorise the processes according to the nature of the work they carry out. In that sense, there can be two main groups of processes [26]:

- *Compute-bound*: the process type which involves long and usual CPU bursts while they do not involve much I/O interaction and
- *I/O-bound*: vice versa, this type of processes requires short periods of time on the CPU but demands frequent I/O interaction.

This organisation among processes forces a different approach from the side of the scheduler, depending on their type. It is worth noting that as CPUs get faster and with higher throughput, the trend for general-purpose computing processes is to become I/O-bound, since the hard disk drives have not achieved equivalent speed improvements to the CPUs.

Another important aspect of scheduling is the selection of suitable re-scheduling points, namely the moments when the scheduler is required to reach a decision on which thread should execute next. There are various possible points within an execution flow when the scheduler might require to decide on that, with a few key ones being:

- A new process is created – who should get the CPU, the parent or the child process?
- A process exits
- A process blocks while performing I/O

These points of re-scheduling can differ between scheduler implementations, mainly due to whether the scheduler is categorised as *preemptive* or *non-preemptive*. The essential difference between these two types is whether the scheduler is allowed to interrupt a running process and allow another process to get the CPU instead, as it can happen with preemptive scheduling. On the other hand, non-preemptive schedulers cannot stop a process while running; in order to get the CPU back, the process itself has to yield it voluntarily in case of blocking for I/O or upon exit. As illustrated in Figures 2.7 and 2.8, the difference between the two approaches can be crucial. As it will be illustrated later on, depending on the type of system that the scheduler is destined for, there exist different requirements on the side of the scheduler.

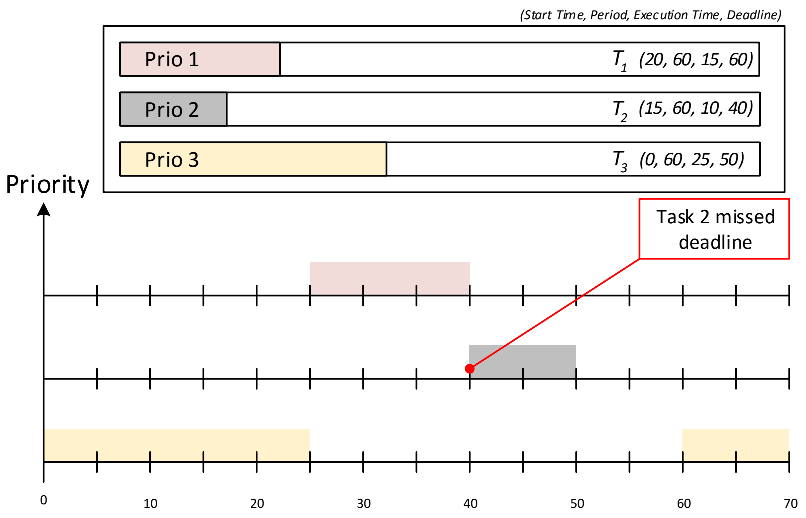


**Figure 2.7:** Non-Preemptive Scheduling resulting into a missed deadline for a Task 2, *Reproduced from [28]*

### 2.1.4.2 Scheduling Goals

Evidently, systems with different purposes have different goals in terms of their scheduling policies. A coarse classification of systems will support the process of understanding the differences between the scheduling algorithms adopted by the different system categories [26]:

- *Batch Systems*: Usually implemented as private servers of major corporations which most of the time perform non-interactive calculations and hence *non-preemptive* or *preemptive* algorithms with long CPU time slots are acceptable approaches for these types of systems. The scheduling goals of such systems are:
  - Maximum throughput (completed tasks per hour),
  - Minimum time between request and termination for each task and
  - High CPU utilisation ratings.
- *Interactive Systems*: The traditional personal computers, along with public servers whose users require interaction quite frequently. Here preemption is essential, a bug or an unexpected behaviour could deny the CPU from the rest

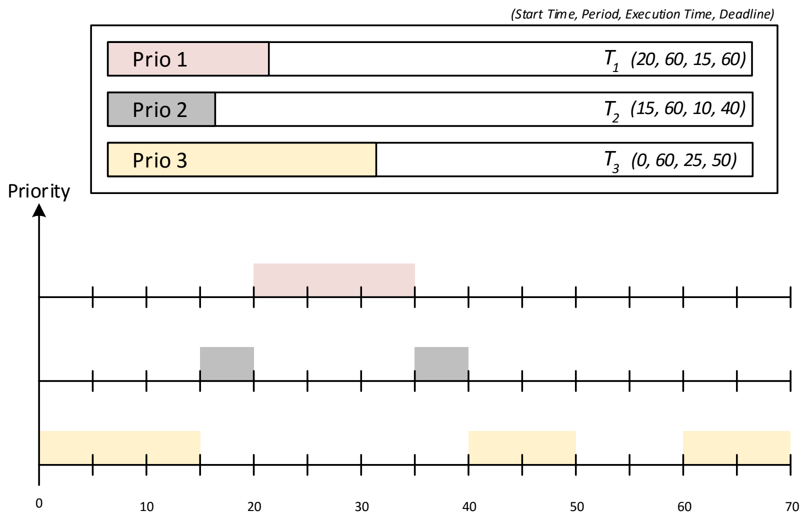


**Figure 2.8:** Preemptive Scheduling, which allows higher priority tasks to execute whenever they are ready, *Reproduced from [28]*

of the processes that require to execute, leading up to a serious user experience degradation. For interactive systems, a standard set of scheduling goals is:
  – Low response times while simultaneously
  – Meet user's expectations.
- *Real-Time Systems*: As we will elaborate quite extensively in Section 2.2.3, RT systems almost always embed preemption in their scheduler, nevertheless there can be cases when preemption is not required since all the applications are developed knowing that they are RT-intended and hence occupy the CPU for small periods of time and subsequently yield the CPU. In the case of RT-systems as we will illustrate in more details later on, the scheduling goals are the following:
  – Essentially achieving their main goal, meeting the hard deadlines, or in case of soft-RT systems
  – avoid user experience decay.

Along these lines, we will give a broad image of which are the most well-known scheduling policies, coupled with some representing examples of traditional algorithms.

#### 2.1.4.3 Popular Scheduling Policies

According to the scheduling targets for each of the systems categories, an illustration of the most significant scheduling algorithms will follow, in order to provide more practical examples of the strategies that various system types employ in order to achieve their scheduling goals.

Examining policies that comply to the requirements of batch systems, we encounter

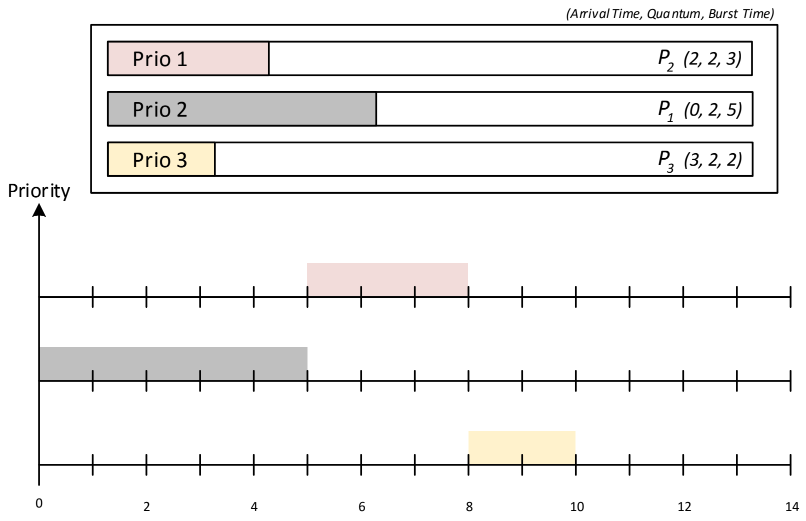


**Figure 2.9:** FCFS neglects priorities of the tasks and executes them in sequence of their arrival times

some of the basic non-preemptive scheduling algorithms, such as the *First-Come First-Served* (FCFS), seen in Figure 2.9, and the *Shortest Job First* (SJF), seen in Figure 2.10, algorithms. The former is probably the most straightforward approach when it comes to scheduling processes, since it picks the process that was at the ready state first, ignoring any other factors. The nature of FCFS allows it to be quite easy to implement but also poor in terms of performance, since an I/O-bound process would occupy the CPU while waiting for an I/O operation to complete instead of yielding and allowing others to execute.

When it comes to SJF, it infuses another aspect of fairness by calculating the amount of time that a process requires the CPU and schedule the ones that need the CPU for the smallest time period. However, it could cause *starvation* for long compute-bound processes which will get constantly bypassed by shorter processes. A preemptive version of the SJF algorithm is the *Shortest Remaining Time First*, which essentially calculates the time that every ready process has left to execute and picks the one with the least required [22].

Moving on to interactive systems, we also switch between requirements going into policies with enhanced fairness and more responsive-oriented. The first algorithm that comes in mind is shown in Figure 2.11, the *Round-Robin* (RR) algorithm. The concept behind this very wide-spread algorithm is that each process acquires a CPU time quantum in order to execute its code. After this quantum is spent and the process still has not finished executing, it acquires another quantum and gets in line behind the processes that have not executed their first quantum yet. The design of the algorithm is very simple, all the CPU needs to keep track of is the set of processes that are in ready state at each moment. Nonetheless, a really crucial aspect in order to achieve decent performance out of Round-Robin is the selection of the

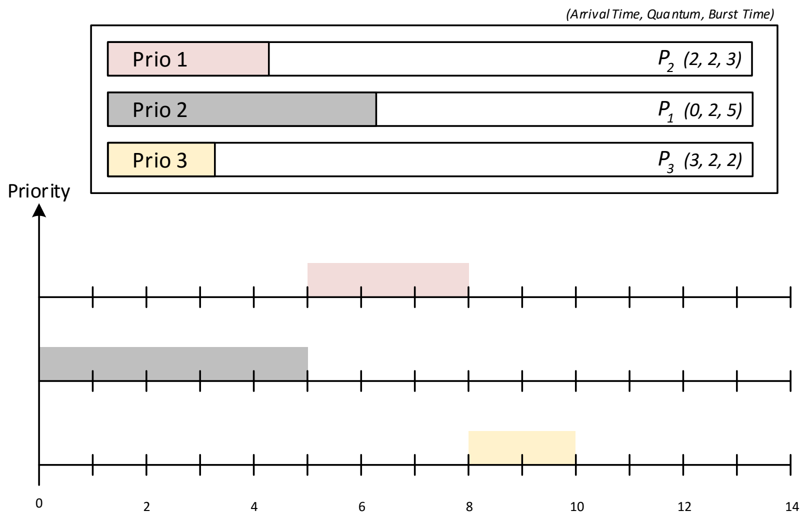


**Figure 2.10:** SJF looks at the burst time of the task and schedules them so that the shortest tasks are executed with higher priority

size for the quantum: a really small quantum would result in high context-switching overhead, while going with a longer quantum would eliminate the preemption aspect of the policy, leading to context-switching again, after the running process finishes executing and blocks [26].

All the aforementioned policies lack one vital part of practical scheduling, the concept of *priorities*. As illustrated in Figure 2.8, priorities allow the scheduler to rank the processes in order of importance; even in single-user systems there can be differences in the time-criticality of the processes and their significance in order to maintain high levels of user experience.

The idea of priorities can be combined with other existing scheduling policies and abstractions; it can also exist in the shape of a *single priorities queue* or *multiple queues*, one queue for each level of priorities whose items are scheduled with a FCFS, or any other approach [22]. Many algorithms keep the priorities in mind without making them the focal point of the implementation, e.g. the *Lottery Scheduling* approach. This scheme comprises a simple lottery tickets paradigm: each ready process receives lottery tickets for a system resource (e.g. the CPU) and at any re-scheduling point a lottery ticket is selected randomly. The owner of this ticket is allowed to use the resource, usually for a predefined quantum. In order to infuse the concept of priorities, the scheduler can provide the most important processes with more tickets and thus enhance their chances to acquire the system resource sooner. In the long run, if a process holds 20 out of 100 total tickets it will be able to run for 20% of the total CPU time [26].

Taking everything that was mentioned in this section into consideration, one can reach the conclusion that scheduling policies can be very diversifying based on the

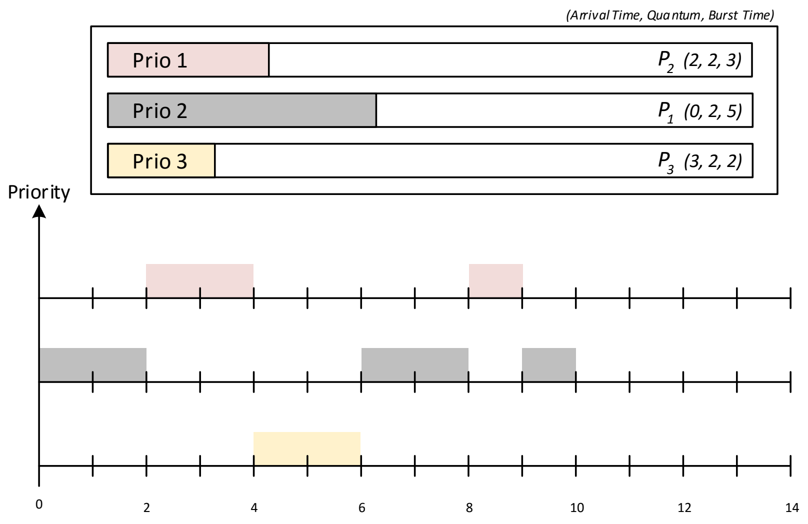


**Figure 2.11:** This algorithm, RR, makes sure that all tasks only execute a certain amount of time (the quantum) each time, going round the task ready-list

system needs and the processes profiling. Even implementations with the same basis can diverge at an extensive degree due to different decisions made on the details of the implemented policy.

## 2.2 Embedded Systems

Thanks to their immense success, embedded systems are incorporated in almost every single aspect of our every day lives. From the smart TVs in our living room, the espresso machine and the modern oven in the kitchen, to the braking systems of the cars we are driving and the traffic lights that control the vehicle flow in our streets. We will provide a solid definition later on in this chapter, along with some historic background on the origins of these types of systems and then wrap up this section with delving deeper into a major subcategory of embedded systems, the real-time systems and the OSs functioning on top of them.

### 2.2.1 Definition

The concept of an *embedded system* can be explained as a subsystem of a device or machine, which is more sophisticated than the system itself. Many supportive examples of this definition can be provided, especially from the world of automotive industry. For instance, the system that controls the windshield wipers on a vehicle is an embedded system and is obviously way more sophisticated than the mechanical parts –the wipers and the motors that perform the movement. Another example would be the air-conditioning system on a vehicle; the included embedded system is controlling multiple actuators and sensors and needless to say that it is more complex and advanced than the other comprising parts.

Additional definitions from numerous bibliography sources describe embedded systems as electronic systems that are specifically designed to execute a discrete function as a part of a larger system (not necessarily an electronic system) [21, 28]. In general, embedded systems deviate from the general-purpose computer in two fundamental ways:

- A general-purpose computer does not serve a predefined and rather specific functionality as an embedded system does. Thus the general-purpose machines require versatility when it comes to achieved performance in various use cases (e.g. graphical operations, high-resolution arithmetic computations and intensive memory accessing), in contrast to embedded devices that are designed and built to serve a narrow functionality scope.
- Conventionally, embedded systems were built simultaneously with the target software, a process that is known as hardware-software co-design [28]. However, recently there has been a shift in this habit, since concurrent embedded hardware platforms are more and more designed and implemented using the general-purpose computer paradigm: the hardware is independently developed within a predefined and standardised framework, in order for third-party embedded software developers to be able to create embedded applications or embedded OSs without having to collaborate closely with the hardware provider.

### 2.2.2 Historical Motivation

The beginning of the embedded systems era dates back to the early 1970s and coincides with the launch of the first ever microprocessor from Intel, the 4004 [29]. This particular microprocessor was used as the chipset that would control a series of new calculators that Busicom, a Japanese company, was designing. Instead of creating a bundle of different customised circuits for each of the models, Intel suggested the use of the 4004 chipset and that a set of instructions would give the different calculators the ability to perform various operations.

The imminent success of the microprocessor as a concept is known and well - documented. However, the silent and constant takeover of the computer world by the embedded systems that has been going on for almost three decades is worth mentioning. Early applications suitable for embedded devices include aircraft control systems, unmanned space shuttles and the birth of ECUs as elaborated in Chapter 1.

Since then, electronic devices with embedded systems have conquered different aspects of our lives, from medical and military operations to almost any everyday activity within every household in the modern world: from a dishwasher to cutting-edge technology products, such as smart TVs or gaming consoles.

The reasons behind the massive success of embedded systems and the shift towards them orbit around the basic characteristic that we presented already, the fact that embedded systems are designed to perform a specific operation with usually limited CPU performance, memory and power needs. On the other hand, general-purpose computer systems have to provide adequate performance on all levels (CPU, Graphics Processing Unit (GPU), memory speed and capacity to name a few) and in a way predict the demand and the workload that the end-user will exercise on the system. This process results in extra overhead in effort, energy consumption and eventually design and production costs, whereas when it comes to an embedded system it is easier to measure and determine the required specifications, in prior, for the target hardware platform.

The resulting assumption is that the embedded systems reformed the way electronic devices are used in everyday situations and due to their low complexity and production cost it will continue to revolutionise aspects of our lives. A solid argument that backs this expectation is the fact that about 98% of microprocessors have been going into embedded systems, whereas less than 2% of microprocessors are used in computers [30] and this does not seem like it would change anywhere in the near future.

### 2.2.3 Real-Time Systems

One of the most significant embedded systems subcategories is real-time embedded systems, Figure 2.12 shows a visual classification of this. The major difference from traditional embedded systems is the concept of *deadlines*, runtime limits that determine whether the application run on the real-time-capable system is executed correctly or not.

The defining point of an RT system is time criticality. This refers to a situation when meeting a deadline for a scheduled task defines whether the system functions properly or not. Depending on the nature on the system, the consequences of said situation may vary from producing erroneous results to causing bodily harm [31, 32]. Systems that can be categorised within the former situation are called soft RT systems, while the latter hard RT systems.

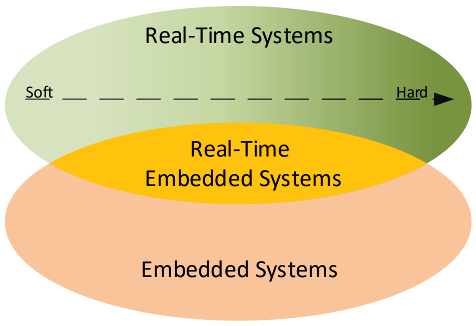


**Figure 2.12:** A representation of how one can classify different types systems, there are embedded systems, RT systems and the interweaving version: RT embedded systems, *Reproduced from [28]*

An RT system is defined as soft by simply including a deadline. If that deadline is missed, the service quality could be reduced but it would not be considered fatal to the system's operation. An example of a soft RT requirement would be keystrokes on a keyboard or playing an audio file. Humans would not notice display delays or hear the delay in a song if it is below a couple of tens of milliseconds. Missing a soft deadline can result in degradation of the provided service, if affected at all [32].

Hard RT systems on the other hand, are defined by what happens if a deadline is missed. If a deadline miss occurs the results are regarded as catastrophic for the system, since the requirements are defined as hard [32]. Most systems regarding personal safety are categorised as hard RT systems, e.g. the inputs of a pilot to an aircraft, a system controlling fuel injection for a vehicle or the joystick that leads a surgical laser.

### 2.2.4 Real-Time Scheduling

Task priority assignment can be grouped into two main classes: fixed priority and variable priority. Fixed priority is very much what it sounds like, task priorities never change during the system execution and are set at system build time. As will be presented later in this chapter, the AUTOSAR OS which is based on the OSEK specifications is a fixed priority OS. Whereas variable priority allows the priorities to change dynamically during system execution to improve the system's responsiveness or performance [31]. The two most adopted policies for fixed and variable priority will be presented below to give some examples to what priority scheduling can look like. These are the Rate Monotonic, which can be seen in Figure 2.13, and the Earliest Deadline First for fixed priority and variable priority respectively.

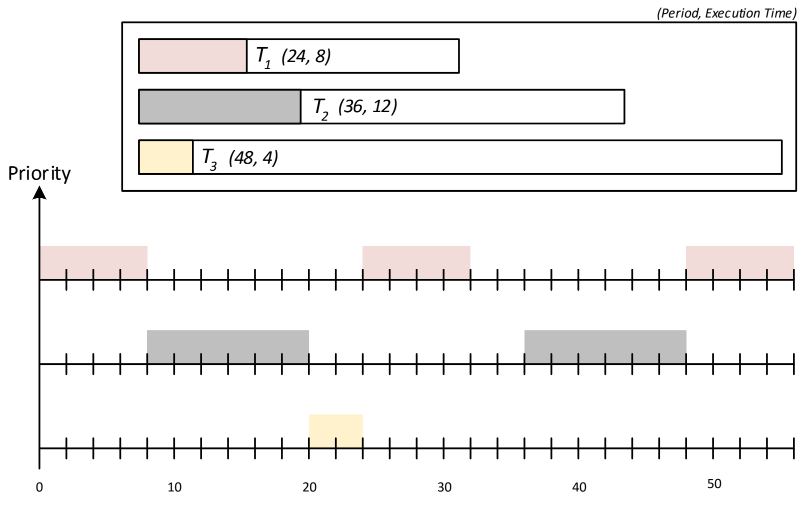


**Figure 2.13:** Scheduling based on task period with the lowest period has the highest priority, *Reproduced from [28]*

The Rate Monotonic policy bases the priority scheduling on tasks' periods. The shorter the period, the higher the priority; disregarding the computational time of the task [31]. Naturally, Rate Monotonic scheduling makes sense; a task with shorter period has less time for computation and therefore needs to have higher priority, whereas the task with a longer period have more time available and therefore can afford to wait longer. This is also the case, however this report will not dig into the mathematical proof. For the reader that wants to venture further into details of this matter can take a look at [31].

Earliest Deadline First is a policy for variable priority. It chooses task priority based on current deadlines; dynamically checking each task's absolute deadline and giving the highest priority to the task with the least amount time left [31].

### 2.2.5 Embedded Operating Systems

The desktop OSs mentioned in Section 2.1 are not adequate for embedded purposes, as such systems are in most cases designed to do specific lesser tasks, usually operating under real-time constraints. However, they are very refined, thus embedded OSs practically inherit the architecture of a general desktop OS, but are adjusted to fit a specific application. They therefore require less processing power, smaller memories, reduced power consumption and most likely also real-time support [21].

As will be described in the next section, Section 2.3, UNIX-based systems come with specific advantages over other alternatives. In [33] the authors present the reasoning behind Linux's dominance in the field of embedded OSs, formulated into a list of strong points that are favourable to picking Linux over traditional embedded OSs:

- Reliability and source code quality
- Availability of the source code
- Tools availability
- Peer Support
- Licensing status
- Low Cost

As most of the points are self-explanatory but also since it falls outside the scope of our project to explore other embedded OS alternatives, we will not elaborate more on that matter.

## 2.3 UNIX OS

UNIX (Uniplexed Information and Computing Service) is a class of OSs which provides a broad hardware compatibility. A small history flashback along with an outline of the UNIX features and structural foundation will be provided. To wrap up the section, we will present the concept of POSIX threads and the related background theory.

### 2.3.1 History of UNIX

Although the first edition of UNIX was released in 1971, its development roots date back to 1957, in Bell Labs and UNIX's forebear BESYS OS. In 1964, a collaboration between General Electric, MIT and Bell Labs produced a new OS that included some of the core characteristics of its heir, UNIX, only to be dropped as a project 5 years later [34]. This paved the way for Ken Thompson, Dennis Ritchie, Douglas Ritchie and Douglas McIlroy to work on UNIX, which appeared as mentioned before in 1971. UNIX continued to evolve in the upcoming years, leading up to the development of the C programming language from Dennis Ritchie and the rewriting of the UNIX OS [35].

### 2.3.2 Fundamental UNIX Features

Considering that the main focus of our project are embedded automotive systems, we examine the advantageous features of UNIX from that angle. Since embedded platforms became more popular, UNIX systems grew into the go-to embedded OS solution. The most distinctive attributes of the UNIX OS family are the following [34]:

- Multi-user
- Multitasking
- Portability
- Job Control
- Tools and Utilities
- Security

Even though each of these characteristics are immensely important for the UNIX popularity, in the perspective that we examine UNIX-based systems we are going to focus specifically on *multitasking*, *portability* and on *job control*.

The main idea behind *multitasking*, is that the OS takes advantage of the time that a running task waits for a resource or performs an I/O operation, to exchange it with a ready-to-execute task. This *context switch* between tasks masquerades the sequential into parallel execution [26]. This property, constitutes UNIX as a very resource efficient OS when it comes to task scheduling [34].

*Portability* is another point of advantage for UNIX-based OSs, which has much to do with the fact that UNIX's code is written in C programming language and is hence hardware independent compared to OSs that are coded in Assembly languages [34]. As illustrated in Figure 2.14, the applications are only communicating with the hardware through the kernel and thus mask any required modifications applied on the kernel in order to port the UNIX system onto the new hardware platform.

Lastly, when it comes to *job control*, UNIX provides us with the privilege to totally control the execution patterns of the tasks at hand. In that sense, UNIX provides the user with the ability to decide which applications should be executed in the background and which in the foreground, depending on their I/O dependability or their priority.

### 2.3.3 UNIX Structure

As illustrated in Figure 2.14, the fundamental idea behind UNIX-based OSs is that all the system layers are structured as concentric circles; no system layer can be bypassed and allow communication between non-adjacent layers. This goes back to what was mentioned in the previous section and how this concentric structure conceals the hardware-specific requirements. Without going into great detail, we will now introduce the different UNIX components starting with the fundamental part of any computer system, hardware.

The hardware devices that comprise a computer system diversify. The OS-related components however are essentially included in one of the following categories:

- *CPU*: The core component of a computer machine, responsible for all the I/O, logical and arithmetic operations; it coordinates the rest of the hardware devices, such as memory hierarchy modules.
- *Memory*: The memory hierarchy devices vary from on-chip registers and cache to remote storage servers, it is their role to perform the basic memory operations, storing and making data chunks available to the CPU upon request.
- *I/O*: The devices that allow user interaction with the rest of the system.

Placed right above the hardware layer, we examine the *kernel*. Besides camouflaging the hardware-specific adaptations, the kernel is responsible to perform the fundamental OS operations, such as task scheduling, and process, resource and file management. *Process management* determines the assignment of the system devices, *resource management* defines how the system resources should be distributed across the pool of tasks e.g. CPU time and memory space, while *file management* is responsible for different file-related functions, such as allotting files to the different permission groups [34].

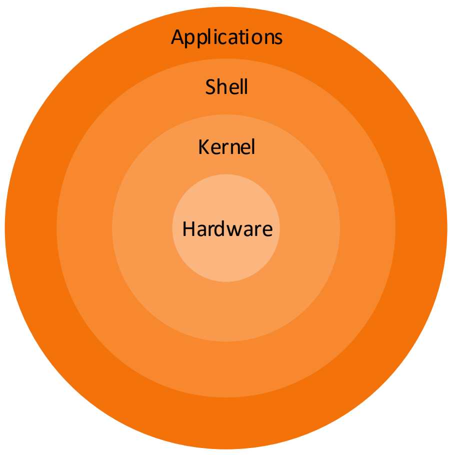


**Figure 2.14:** Concentric structure of UNIX system layers, the hardware is in the centre and is followed by the kernel, the shell and lastly the applications when moving outward in the layers

The other adjacent layer to the kernel is the *shell*. It provides the user and the kernel with the ability to communicate with each other in an efficient, secure and user-friendly way through a set of shell commands. There have been several alternatives of UNIX shells that share the same core functionality but illustrate distinctive characteristics as well [36].

The two main parts of any shell type are the *interpreter* and the concept of *scripts*. Scripts are defined by the ability to execute a set of shell commands, each of them performing a specific task [34]. The interpreter on the other hand, collaborates with the UNIX kernel in order to translate the shell commands into machine code and execute them.

Lastly, there is the *Applications* layer. This part of the UNIX OS comprises built-in functionality and tools that provide additional capability to the OS. The tools that an OS administers are classified according to the function that they perform, e.g. file operations and content searching tools [34].

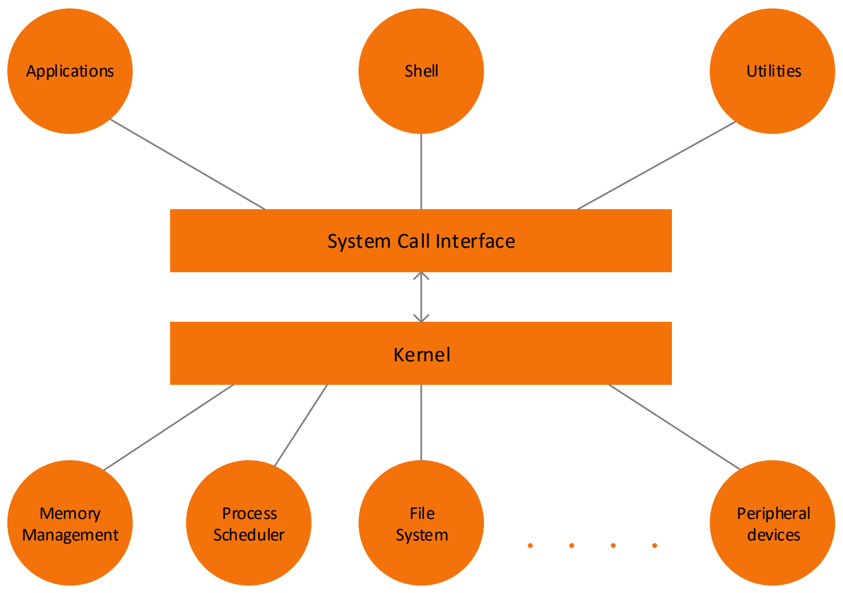


**Figure 2.15:** The figure shows some of the operations the kernel does, apart from handling more low level operations such as memory management and process scheduling, it also handles the system calls from applications, the shell and other various utilities

## 2.4 AUTOSAR Specifications

In this part of the Theory chapter we present the core concept of this project, the AUTOSAR standard. Since the motivation and the goals of the consortium have been presented in Chapter 1, we will focus on its technical specifications along with the standard software architecture destined for AUTOSAR operating ECUs.

### 2.4.1 Software Architecture

Based on the layered layout of the AUTOSAR Basic Software Module [37], it is divided into the following levels:

- AUTOSAR Applications Layer
- AUTOSAR Runtime Environment (RTE)
- Basic Software Module
- Microcontroller Unit (MCU)

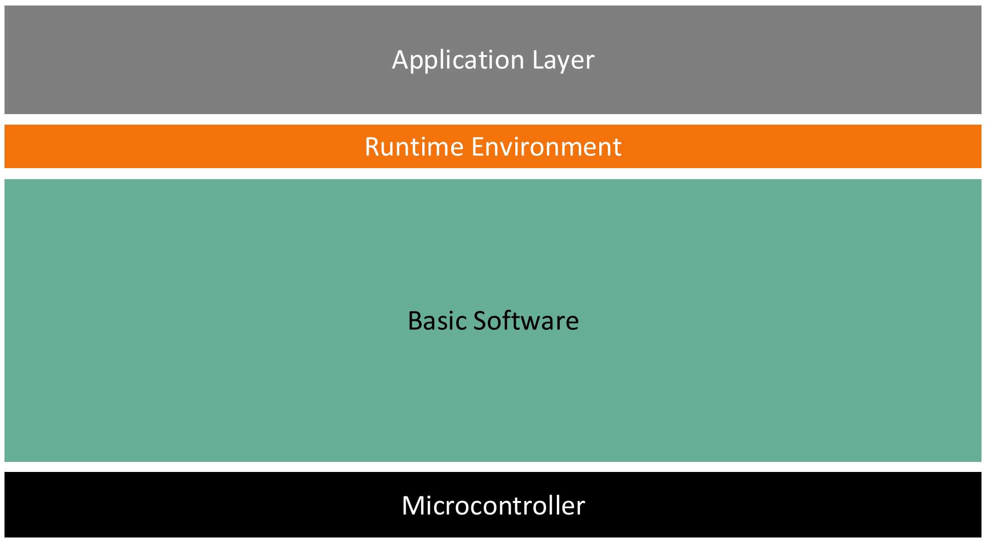


**Figure 2.16:** A coarse representation of an AUTOSAR ECU system which illustrates the layered architecture. The Application layer is at the top housing software components and, sensors and actuators. The Runtime Environment is below, handling the signal communication between the applications and the basic software modules. These modules comprise the Basic Software layer, which is the last software layer in the stack as the layer underneath is the ECU hardware

Each of these subsystems that are presented in the following segment, consist of a number of discrete components. An extensive description of their functionality falls outside the scope of this project and hence we will not go into more detail, with the exception of the operating system which will be explained in Section 2.5. But note that in order to build a proper working ECU image several generic modules need to be used.

1. *AUTOSAR Applications Layer*: The level where the created software components for a specific functionality are placed. Existing in the same layer, are the sensor/actuator software components, according to AUTOSAR standards [38]. It should be noted that test applications exists in this layer.
2. *AUTOSAR RTE*: The Runtime Environment is basically the middle-ware, an interface between the *Applications Layer* and the *Basic Software* that provides communication services to the applications existing in the top layer. Moreover, the RTE allows each AUTOSAR software component to interact with other components, both in the same or in different ECUs [38]. This is done by switching the software architecture style from layered to component-oriented and hence allow the applications to be completely independent of the underlying ECU [39].
3. *Basic Software*: This layer is divided further into sub-layers, including a *services Layer*, an *ECU Abstraction Layer*, a *Complex Device Drivers* (CDD) *layer* and a *Microcontroller Abstraction Layer* (MCAL). Between these layers, there exist standardised interface modules to facilitate the communication between

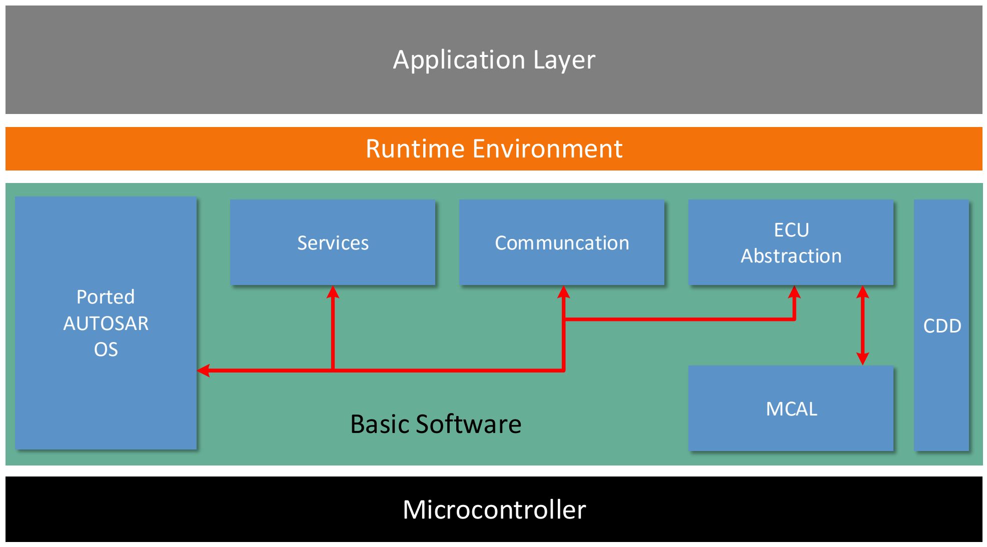


**Figure 2.17:** A more detailed illustration of the AUTOSAR software architecture and its major components, showing some of the modules mentioned in Figure 2.16

them. Each of the layers is subsequently partitioned into smaller functional entities, such as *Memory Services* and *I/O drivers* [38]. Without going into excessive detail, every entity of this type consists of smaller software modules that perform distinct operations, predefined from the AUTOSAR specifications.

4. *MCU*: This subsystem is self-explanatory. It is the hardware of the ECU and communicates with the AUTOSAR Basic Software layers through the MCAL and the CDD.

### 2.4.2 Adaptive AUTOSAR

Adaptive AUTOSAR systems will act as a hybrid that could deploy both the classic AUTOSAR applications and the new adaptive alternative but at the same time communicate with non-AUTOSAR systems included in a vehicle; e.g. a GENIVI infotainment system that runs on Linux [40] or an external non-AUTOSAR system, such as a server or a home terminal. In Figure 2.18 we can see how this is visualised, allowing the car to communicate both with other vehicles as well as an external system depicted by the house like structure, even though the applications are not connected via the same platform.

The motivation behind the new adaptive platform was touched upon in the introduction section, vehicle applications becoming more computational heavy, with the main contributor being advances in autonomous driving. However, the generic automotive MCUs of today are not focused on high-performance and are therefore not capable of handling such tasks. A trend towards having a larger, central and more computational heavy core can be seen, making high-performance hardware an interesting option. This compels AUTOSAR to adopt the idea of utilising either VMs

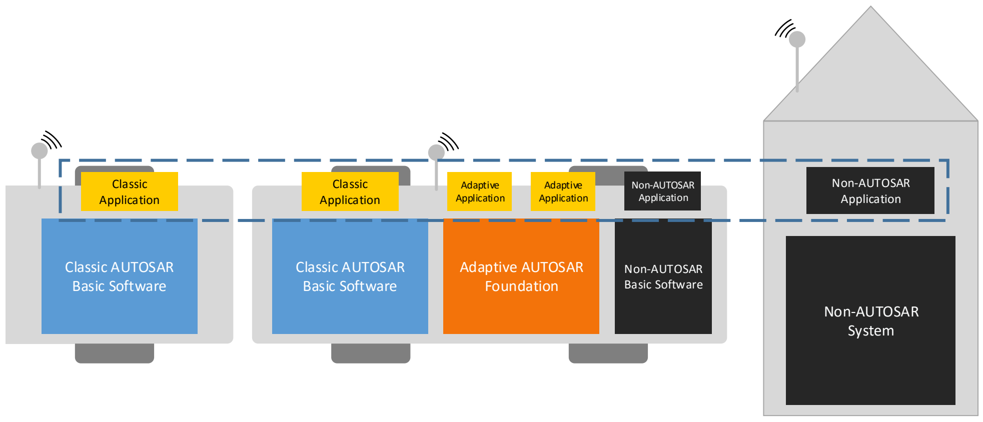


**Figure 2.18:** The figure shows how Adaptive AUTOSAR would integrate non-AUTOSAR systems with classic AUTOSAR systems, communicating with applications from other vehicles or external non-AUTOSAR systems

or more powerful hardware. Figure 2.19 depicts the gap between the classic AUTOSAR stack and a more general purpose infotainment system, e.g. a gps system or autonomous driving, which requires more performance. The adaptive AUTOSAR platform is the solution which will bridge the gap, allowing the incorporation of more computational heavy MCUs in vehicles.

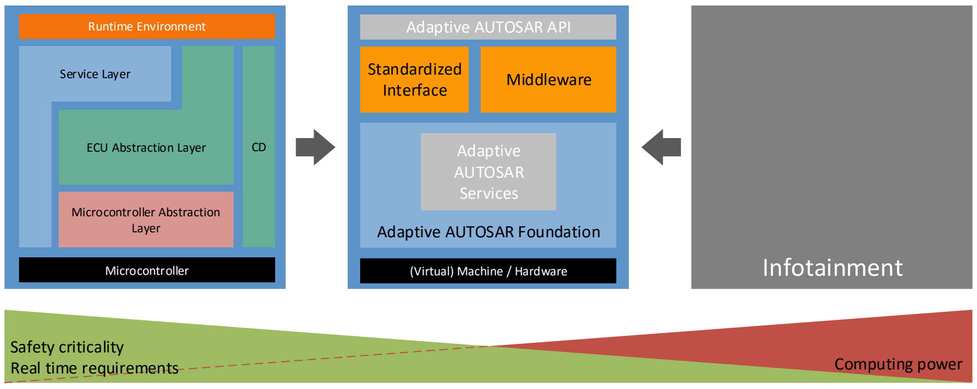


**Figure 2.19:** Adaptive AUTOSAR will fill the hole in between infotainment systems and classic AUTOSAR for applications that require less computational power compared to an infotainment system but also has fewer safety requirements

## 2.5 OSEK Specifications

This section is based on the relevant parts, to this project, of the official OSEK OS specification [41]. OSEK is a specification which is specifically designed for automotive applications. The AUTOSAR OS has OSEK as a base and therefore work very similar in most areas. We limit ourselves to study the OSEK specification as it facilitates the learning process compared to reading up on the vast AUTOSAR specifications.

The OSEK specification is designed for single core processors and with the intent to be used for ECUs. Naturally, it does not support running tasks in parallel, however it does support multithreading. It also supports event driven control units, since the majority of automotive ECUs require RT dependencies.

OSEK supports portability and re-usability of application software, translating into providing the ability to transfer an application from one ECU to another without any major changes, following the core idea of AUTOSAR.

### 2.5.1 Architecture

There are two types of entities competing for the hardware resources, interrupt services and tasks. The resources are managed by OS services which are called by a unique interface and are subsequently called by either the application or internally within the OS. The resources are divided between three processing levels; starting with the lowest priority level the *task level*, then the *logical level* and lastly the highest priority level the *interrupt level.* The task level is responsible for the task scheduling, which can be either non, fully or mixed-preemptive. The logical level handles scheduling between interrupts and tasks, and the interrupt level manages the interrupts.

### 2.5.2 Task Management

In order to understand the task scheduling two different task types must be introduced. Namely, basic tasks and extended tasks. The key difference between extended tasks and basic tasks is that extended tasks can be put into a waiting state. Whereas the basic tasks can only be terminated or interrupted by a higher priority task/interrupt in order to release the resource. Figure 2.20 shows the task state model for OSEK, it can be seen that tasks that are suspended are first activated, set as ready and then set to running. In the running state tasks can either terminate, get preempted by a higher priority service or be set into a waiting state, which applies to extended tasks only. From the waiting state the extended tasks can only be released by either an alarm or an event. Task termination is only achieved via self-termination, meaning that a parent task cannot call for a child task (or any other task) to be terminated.

### 2.5.3 Events and alarms

In the previous paragraph it was mentioned that there are two types of tasks. The extended task type has an extra mechanism called events. Events are a means of

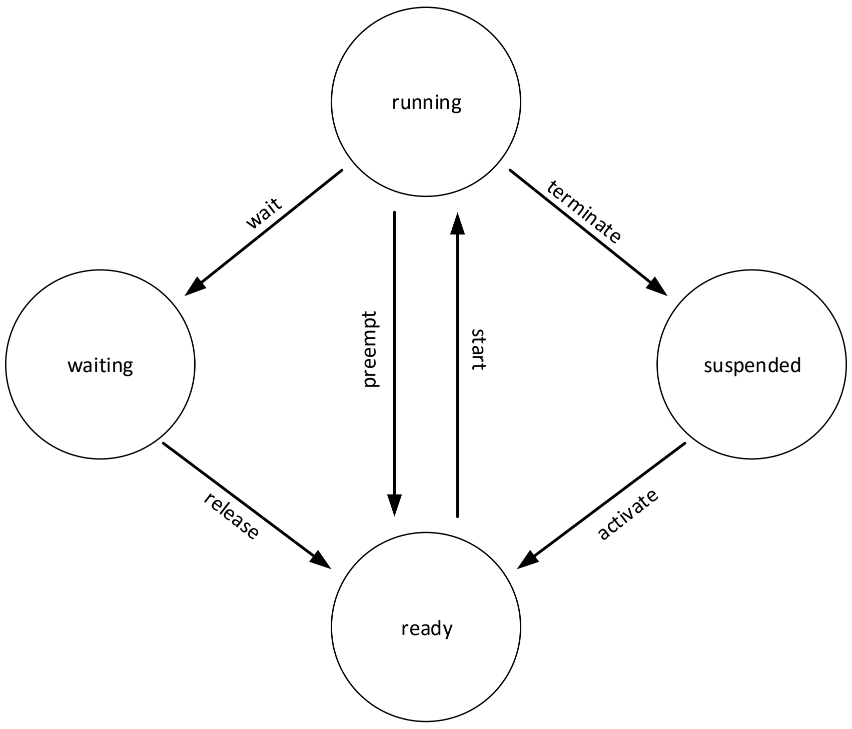


**Figure 2.20:** Task state model of how OSEK manages tasks. A task that is suspended gets activated into the ready state, from where it starts running. While running it can get preempted, terminated or be put into a waiting state (this is only for extended tasks), from which it is released with the help of events. *Reproduced from [41]*

synchronisation and are the only way an extended task can get from the running state to the waiting state, referring to Figure 2.20. Events are defined by the application and can for example be signalling of an expiring timer or availability of a resource. All tasks can set any event that is not contained in a suspended task. However, only the owner of an event can clear it (checking if the occurring event is the event the task is waiting for), which is needed if said task wants to wait for the setting of a new event.

Alarms act as the setting of an event at expiration but they can do it recursively, which a single event cannot. They are statically assigned at the time of system generation and are tied together with a specific counter and a task-callback routine. Both single and cyclic alarms exist, where a single alarm is only executed once while a cyclic is executed at a certain time interval.

The counter mentioned in the previous paragraph inclines that OSEK uses a two part concept for alarms: an implementation specific counter and an alarm call-back. When a counter expires it signals the alarm(s) tied to it, which in turn executes an alarm-callback. If the counter was managing a cyclic alarm it would reset and thereby invoking the alarm again after a certain interval, while a counter directing a single alarm would continue and most likely invoke other single alarms.

#### 2.5.3.1 Resource management

The resource management is an important topic for a single core CPU specification because of its nature; a new core cannot be allocated to a task if it is occupied. Therefore, issues like deadlock or priority inversion might occur more easily. OSEK inhibits this by ensuring single use of resources, which also prevents that accessing a resource never result in a waiting state. The same concept is extended to the interrupt level to ensure that no two tasks or interrupts occupy the same resource.

In order to ensure single use of resources, OSEK statically assigns a resource with its own ceiling priority. This priority should be set to the same level as the task with the highest priority accessing the resource but to a lower level than the lowest priority of all tasks not accessing the resource, that have higher priority levels than the highest accessing the resource. This is more easily understood via visualisation as depicted in Figure 2.21.

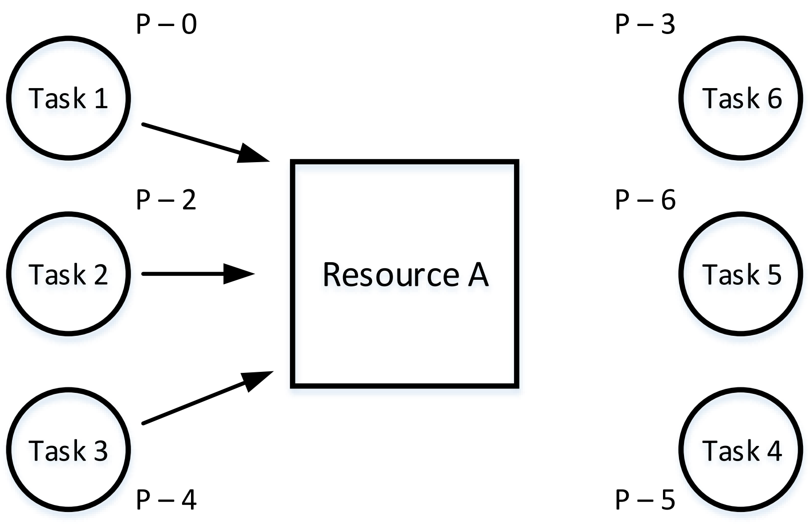


**Figure 2.21:** Tasks 1, 2 and 3 are requesting access from resource A and tasks 4 through 6 are tasks that does not request access from resource A. In order to ensure single use of resource A for this simple example, its ceiling priority must be set to the same priority of Task 3 (since it has the highest priority of all accessing tasks) but lower than the lowest priority of all non-accessing tasks with higher priority than Task 3. In this case resource A's priority level would be set to 4.

## 2.6 Yocto Project

This section will be explaining what the *Yocto Project* is and in some parts how it is relevant to our project. Starting with the background to why it exist and the major outlines following with the build system and its components. This is explained more in detail on their homepage [42] and in their reference manual [19].

### 2.6.1 Background

The Yocto Project is a work-group within the *Linux foundation*, which is an open collaboration between different corporate parties that aim to advance and promote Linux development [43]. The work-group seek to develop tools and processes that allows creation of customised Linux distributions for embedded systems.

### 2.6.2 Poky

The build system they provide is called *Poky* and allows almost endless customisation. It lets you choose which platform you are building your distribution for, what packages to include in said distribution and even down to how the kernel should operate. In order to enable this it uses something called *Bitbake* and *Metadata*.

#### 2.6.2.1 Bitbake

Bitbake can be most easily explained as the task executor and scheduler during the build time of the Linux distribution. It uses the Metadata with a concept called *Layers*; a layer can be seen as a design mechanism, allowing the user to easily change the software stack. In Figure 2.23 we can see the file system tree of the Poky build system, where each folder beginning with "meta" is one layer.

One of the more important features of using layers is the Board Support Package (BSP), which enables hardware features. They include specific drivers for a particular hardware device, e.g. containing a Linux kernel configuration and graphic drivers for the intended hardware target. This allows simple exchange of hardware platforms by downloading a BSP for supported hardware and telling Bitbake to use that layer for the build image. The image now works on the hardware specified in the BSP.

The layers can also be used for developing software, appending a developer specific layer during production. Once it is completed the layer can be removed from the Bitbake configurations and it will no longer be part of the build image. An example layer stack can be seen in Figure 2.22, where each layer has a different priority, starting with the highest priority at the bottom of the stack. The lower priority layers can be removed without problems whereas the higher priority layers are mandatory. The lowest layer is the core metadata, including low level configurations such as kernel and hardware drivers. Moving upward we see the BSP layer which defines what type of hardware architecture the image is for, a User Interface (UI) specific

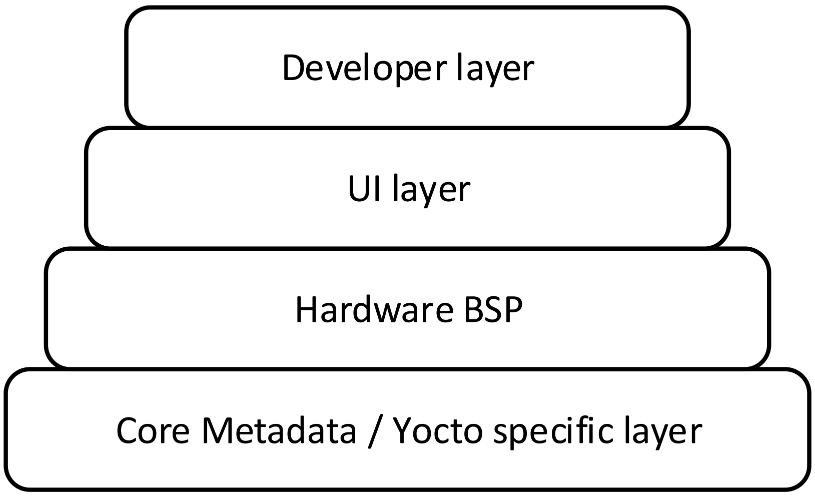


**Figure 2.22:** Visual representation of how the bitbake layer structure may look. Starting from the bottom with the fundamental layers that are included in all yocto supported builds, moving a step up into the BSP layer which defines the target hardware. Above that may be a UI layer and a developer layer, *Reproduced from [42]*

layer and last the developer layer mentioned earlier.

#### 2.6.2.2 Metadata

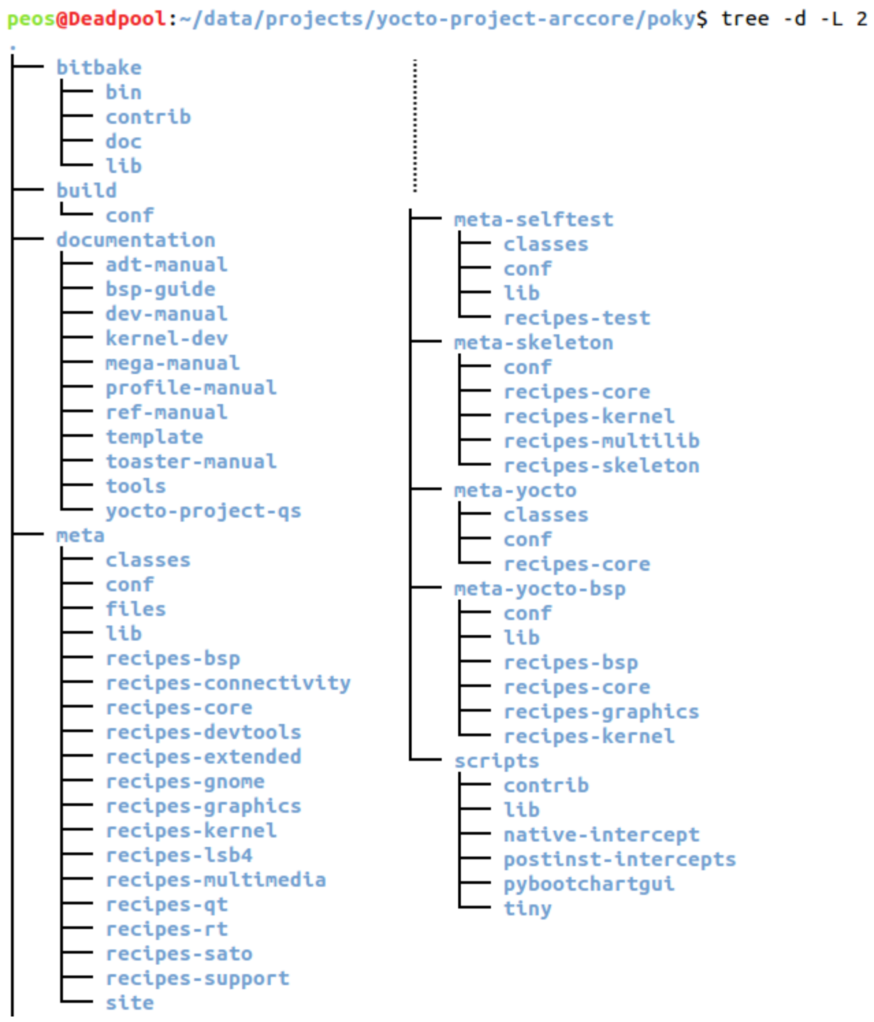
As touched upon in the previous section the Metadata is organised in layers, where each layer include information that tells Bitbake what gets built and how. The Metadata comprise three main types: *Configuration*, *Classes* and *Recipes*.

The configuration (.conf files) define global variables for the build system, e.g. the standard file system paths as in where the build system is supposed to be placed. It also defines general compiler flags, telling Bitbake how many threads of the processor to utilise or which machine architecture to build for.

The heavy lifters of the build system are the classes (.bbclass). The Yocto Project provide classes that define how to build an autotools based piece of software, how you build a Linux kernel and how to generate a root file system image; autotools is a build system for Linux programs.

Recipes (.bb) tell Bitbake what packages, an individual piece of software that needs to be built, to include in the file system image e.g. the gnu tar command (command for extracting and compressing files) or the gtk library (toolkit for creating graphical UIs). These recipes differ from other build systems as they are not defining the packages to be built but rather how to build them and where to fetch the source.

The standard recipe build steps follow the sequence of fetching, unpacking, patching, configuring, compiling, installing and lastly packaging the installed software into binary formats. Most commonly these commands are inherited versions from the autotools .bbclass, to correctly build the software using the desired build system. Bitbake allows these commands to be overwritten so they can be customised for each recipe if needed.



```
peos@Deadpool:~/data/projects/yocto-project-arccore/poky$ tree -d -L 2
.
├── bitbake
│   ├── bin
│   ├── contrib
│   ├── doc
│   └── lib
├── build
│   └── conf
├── documentation
│   ├── adt-manual
│   ├── bsp-guide
│   ├── dev-manual
│   ├── kernel-dev
│   ├── mega-manual
│   ├── profile-manual
│   ├── ref-manual
│   ├── template
│   ├── toaster-manual
│   ├── tools
│   └── yocto-project-qs
├── meta
│   ├── classes
│   ├── conf
│   ├── files
│   ├── lib
│   ├── recipes-bsp
│   ├── recipes-connectivity
│   ├── recipes-core
│   ├── recipes-devtools
│   ├── recipes-extended
│   ├── recipes-gnome
│   ├── recipes-graphics
│   ├── recipes-kernel
│   ├── recipes-lsb4
│   ├── recipes-multimedia
│   ├── recipes-qt
│   ├── recipes-rt
│   ├── recipes-sato
│   ├── recipes-support
│   └── site
│
:
├── meta-selftest
│   ├── classes
│   ├── conf
│   ├── lib
│   └── recipes-test
├── meta-skeleton
│   ├── conf
│   ├── recipes-core
│   ├── recipes-kernel
│   ├── recipes-multilib
│   └── recipes-skeleton
├── meta-yocto
│   ├── classes
│   ├── conf
│   └── recipes-core
├── meta-yocto-bsp
│   ├── conf
│   ├── lib
│   ├── recipes-bsp
│   ├── recipes-core
│   ├── recipes-graphics
│   └── recipes-kernel
└── scripts
    ├── contrib
    ├── lib
    ├── native-intercept
    ├── postinst-intercepts
    ├── pybootchartgui
    └── tiny
```

**Figure 2.23:** Screen capture of the poky file system, displayed with help from the *tree* package in Linux, it shows only the directories of the highest levels. The bitbake layers can be seen by the folders starting with "meta", where this screen capture shows a newly downloaded poky directory with no specific layers added as it only includes the yocto layers: *meta*, *meta-yocto* and *meta-yocto-bsp*. The *meta-selftest* layer is used for testing and the *meta-skeleton* layer is a template which one can use to build an own layer. The "bitbake" directory contain all configurations for bitbake and the "build" directory is the output path of the build, e.g. it will contain the image

# 3 Methods

This chapter is a detailed story of how the project team has been working, explaining and describing the work structure overview. It starts with a description of the Rational Unified Process (RUP), which is the working methodology of the project, including its structure and practices, ending that part by explaining the concept of working with iterations. The section following will present the tools used to achieve the results after which we will go through our implementation architecture. The last segment discusses what we did throughout the project, structured in accordance with the iterations.

## 3.1 Rational Unified Process

The RUP is designed to be a software engineering process to provide structure and discipline for a project organisation. The goal is to safeguard a high-quality production so that the software meets end-users' needs effectively and on time.

In order to achieve good productivity the RUP utilises the creation and maintenance of models instead of focusing on production of documentation. It employs the Unified modelling Language (UML) which allows a project team to communicate requirements, architectures and designs more easily [44].

Arguably the best part of the RUP is that it is configurable. It lets us modify the process to fit our project and our software development, as the team only consists of two people which, compared to other projects, is a quite small team.

### 3.1.1 Effective Practices

The RUP deploys commercially proven approaches to software development for development teams [45]. These approaches are divided into six practices which are called "*best practices*"; mostly because they are commonly used in industry by successful organisations and not so much in regards to their perceptible value. The practices are listed below:

- Iterative development
- Manage requirements
- Component-based architectures
- Model software visually
- Verify software quality
- Control changes to software

An iterative development approach ensures an increasing understanding of the problem by consecutively assessing the project from a new perspective and each time enhance it further. The risk level of the project is greatly reduced since the process encourages the use of a development method that addresses the highest risk items every cycle [45].

The RUP supports component-based software development, which spurs early baselining of a sturdy executable architecture. Consequently, a more well comprehended life-cycle is achieved because each component has one primary function which executes without interfering with other operations [45]. This in turn yields a very good basis to use UML [46], as the primary modelling blocks are part of the architecture itself [44].

For better understanding and comprehension of the problem, the RUP utilises visual modelling. By representing software in a visual way you capture the structure and behaviour of the architecture and components much quicker and more easily. Hence, it allows you to hide the details and "display" the architecture as graphical building blocks. It also helps to maintain consistency between a design and its implementation, as it forces you to see how your building blocks fit together [44, 45].

Verifying the software quality is a key practice, as it helps in dealing with common issues such as performance and reliability. These are central in concurrent applications quality and should be meticulously revised to match the requirements [44]. The RUP integrates the quality verification into the process so that it is present every step of the way and not regarded as an afterthought.

Since software applications rarely follow the planning in every detail, it is extremely important to be prepared for changes when they occur. This is especially true in a process that utilises an iterative approach as the application is subject to change because the functionality takes shape throughout the process.

This project will not venture very deep in terms of managing requirements as this is more relevant to high-level projects and from a business point of view. This practice is therefore not included.

### 3.1.2 Process Overview

The RUP can most simply be described in two dimensions [45]. Referring to Figure 3.1, the horizontal axis displays the time where the project enters different phases with iterations in each phase, while the vertical axis presents the process work-flows, where the workload in each phase is depicted by the colourised slopes.

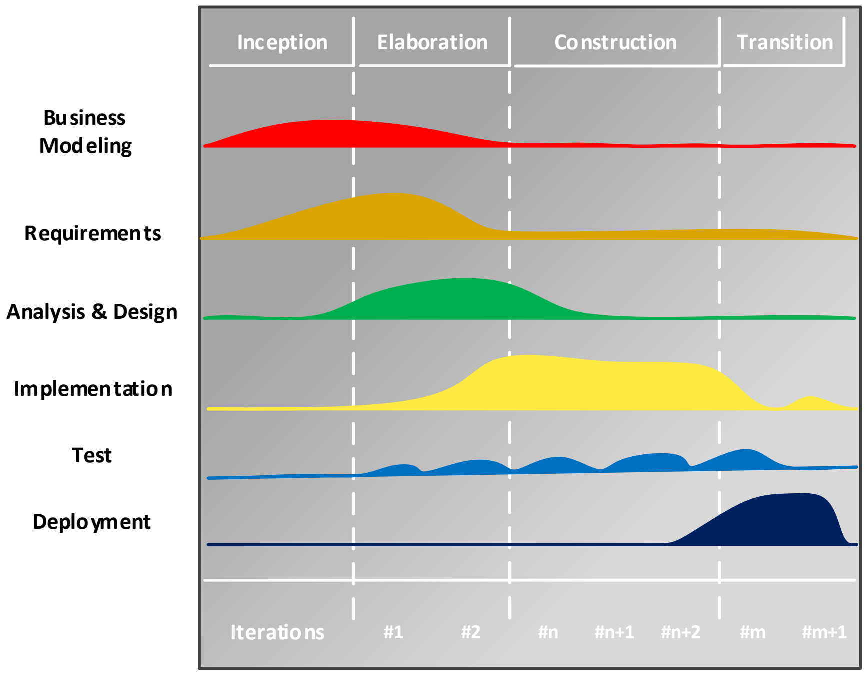


**Figure 3.1:** An example of what a project's process could look like. The horizontal axis displays time, shaping different phases and iterations. Work-flows of the process are distributed along the vertical axis, where the workload of each phase is indicated by the slopes.

### 3.1.3 Static Structure of the Process

A project process is something that describes *who* does *what*, *how* and *when*. The RUP describes these as *workers*, *activities*, *artifacts* and *work-flows* respectively.

A worker should be distinguished from an individual of the team and could instead be seen as a "hat" which can be worn by a person. An individual may wear different "hats" throughout the project. The worker is a role which is defined by how an individual taking that position should carry out the required work [45].

A worker is tied to a set of activities. Instead of having a specific person doing a specific task, the task is linked to the worker. This means that whoever is in the position of a certain worker will do the set of activities tied to that worker and not tasks assigned to them personally.

How to carry out the work is defined by the worker. But in order to carry out the work some prerequisites may be required; tools is such an example. These are described as artifacts in the RUP. An artifact is defined as a piece of information that is modified, produced or used in a process [45]. So an artifact is both something used by a worker in an activity and it can also be the output of said activity. Examples of artifacts may be use-case models, model elements, source code and executables. A use-case model is something that describes the behaviour of a system, or part of a system, by defining the interactions between an actor and the system. The actor can be either a human, a subsystem, an external system or even time.

### 3.1.4 Iterations

In the concept of the RUP, each project phase can be additionally parsed into iterations. An iteration is a complete development loop, resulting in a new product release (either internal or external) of an executable version of the product. Usually a subset of the final product requirements are implemented during each cycle, which eventually evolves into the final system.

Comparing the iterative approach towards the traditional waterfall process [47], the former has the following advantages presented below [44], [45]:

- Mitigate potential risks earlier on
- Change is manageable
- Higher level of reuse
- The project team is invoked in a learning process throughout the project
- Higher overall quality

## 3.2 Development Tools and Code Versioning

The following section is a short discussion and description of the tools used in the project. It will first discuss how the group has used UML and with the help of what tools, it will then introduce the online TeX platform we have worked with. After that the employed IDE will be presented, finishing with the code versioning system used within ARCCORE.

### 3.2.1 Modelling and Visualisation Tools

Software can be complex to visualise and therefore very hard to work with. It can be even harder to attempt to present some of its functionality to someone that has not been involved in the development procedure. In order to facilitate this process the project group has used, in accordance with the RUP, the UML. It should be noted that it has not been followed religiously, but rather as an idea where the actual modelling has been done with “pseudocode”.

There is a vast amount of software solutions available which can handle this type of modelling process, the graph editor *yEd* is such an application. We chose to use it since it is a freeware and easily understood according to our supervisors’ suggestion.

### 3.2.2 Integrated Development Environments

Working with software programming languages today can be very much facilitated by using an IDE. Arguably one of the most powerful and resourceful Integrated Development Environments (IDE) on the market is *Eclipse* [48]. It was our choice to go with that, for the aforementioned reasons but also since the company's own tool suite *Arctic Studio* is an extension to Eclipse. More specifically, we have been using the C/C++ version of Eclipse and not the company's own tools suite, since it was not readily available for Linux at the start of the project.

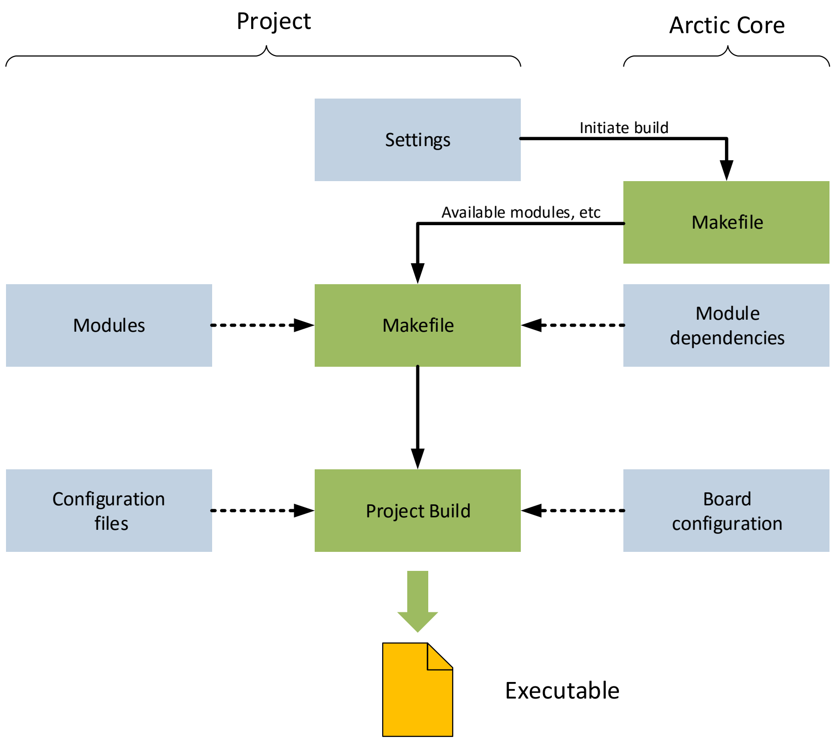


**Figure 3.2:** ARCCORE's makefile system is handled in the background when working inside *Arctic Studio*. Creating a project automatically sets the settings that allows you to build the project. What happens is that the project settings are directed to *Arctic Core* which in turn responds with information useful to the created project, e.g. which available modules exist for the target board. The user specifies what modules they want to use and modify their variables to fir their purpose, the makesystem then handles the module dependencies and the board configuration, *Reproduced from [7]*

#### 3.2.2.1 Makefile System and Arctic Core

This section will briefly discuss and describe the build system, which is based on the standard Linux make package [49], used in the company's products. The system comes with the company's product *Arctic Core* (essentially a set of configurations for hardware used regularly, by ARCCORE), letting the user create their own project in the company's tool-suite *Arctic Studio*. Together with the user's inputs and the configurations, provided by Arctic Core, the makesystem builds the project into an executable. These executables can be downloaded onto an embedded board creating an ECU or a part of an ECU system. The process can be most easily explained visually in Figure 3.2.

The left side shows what happens on the user side of things (Project) and the right side shows how Arctic Core operates. The user inputs settings regarding what modules they want, what hardware architecture it is for and for what hardware board it is intended. The makefile system reads the settings and returns information needed to complete the configuration (e.g. what modules are available). The user selects what modules they want in the build, where the makefile system automatically checks what dependencies these modules have and incorporates them. Finishing the configuration on the project side together with the board configuration –defaults are used unless otherwise specified– from Arctic Core lets the makefile system build the project into executable files. This ties back to Section 1.1, explaining how the implementation is realised in a series of configuration steps.

### 3.2.3 Code Versioning and Data Analysis

The company uses the version management system *Git*. More specifically, an Atlassian BitBucket server which is tailored for professional teams using Git, since it employs extra administrative features. Hence, this system is also used by the project group, working on separate branches in order to be able to experiment on different approaches without interfering with the company's ongoing projects. Git can be managed via terminal commands but is very much facilitated by using available software applications such as GitEye. GitEye is one of many Git clients that provide a Graphical User Interface (GUI) for simpler code management. GitEye can handle all the standard operations for repositories and was selected among other Git clients since it collaborates well with our main IDE, Eclipse CDT, as well as being a freeware.

We used MATLAB in order to produce a good visualisation of our results. Not only is it easier to look at, compared to a plethora of tables, it also provides a simpler view as you can understand what you are looking at more easily.

### 3.2.4 Documentation

Besides bitBucket, the other online platform we used was ShareLatex. ShareLatex is a collaborative editorial platform which allows users to write simultaneously on the same project, much like how Google Docs is used for standard documents but for TeX documents. Google Docs was also used as a draft writing tool as sharelatex can sometimes be slow since the compilation is done online.

## 3.3 Implementation Architecture

As touched upon in Chapter 1, the starting point of our thesis was a related company project. They have been working with AUTOSAR to Linux porting, in an attempt to establish communication through the Ethernet module from the host OS. This effort was the foundation on which we projected our own future work and a guideline for our application's source code.

This section will present the different levels within our implementation beginning with the system level which is based on the OS port done by a previous company project. This is followed by the application level which includes the applications we have tested the system with.

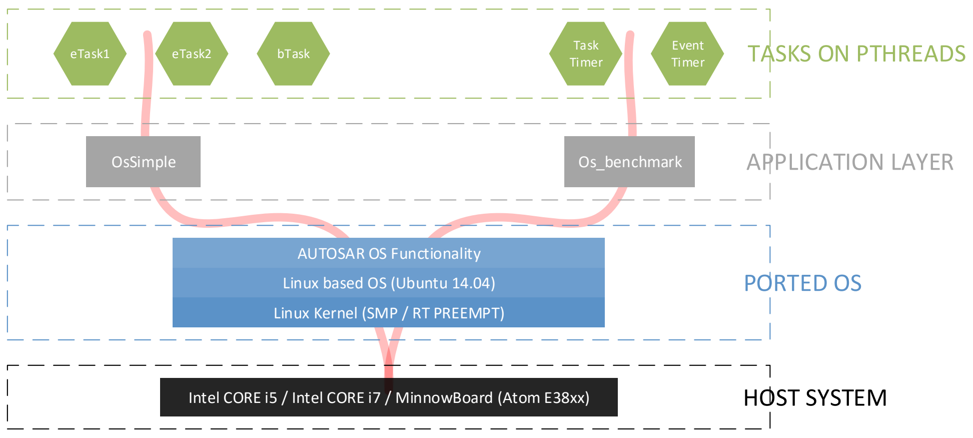


**Figure 3.3:** Following the initial idea described in Chapter 1 we illustrate the final shape of the developed system, divided in layers. Everything from the *PORTED OS* layer and above is on the software side, with all the three hardware alternatives housing a UNIX-based OS to execute the application on.

### 3.3.1 System Level

Our implementation is divided into different layers as depicted in Figure 3.3. The system level, based on a standard Linux distribution kernel, is the core of our platform. Essentially, all the OS functionality is implemented in this layer according to the OSEK specifications, much of which has been taken from the already developed AUTOSAR OS. Nevertheless, we have introduced key features regarding multi-threading support, allowing the OS to utilise multiple core CPUs and multiple threads execution on each of them. Figure 3.4 visualises in deeper detail the structure of one part of the system layer.

The system level is defined by its nature, being run in the background, handling processes such as task activation and termination, events and resource management. The system level for this project however, is a bit different from other standards

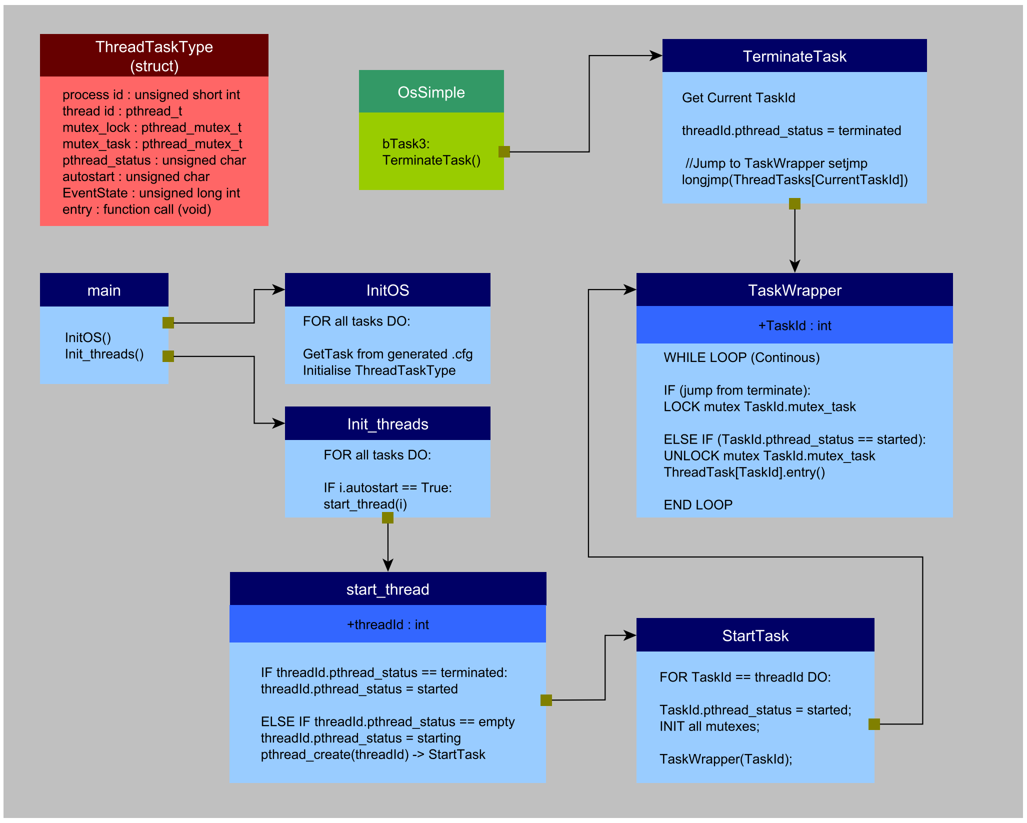

ThreadTaskType
(struct)
process id : unsigned short int
thread id : pthread_t
mutex_lock : pthread_mutex_t
mutex_task : pthread_mutex_t
pthread_status : unsigned char
autostart : unsigned char
EventState : unsigned long int
entry : function call (void)
OsSimple
bTask3:
TerminateTask()
TerminateTask
Get Current TaskId
threadId.pthread_status = terminated
//Jump to TaskWrapper setjmp
longjmp(ThreadTasks[CurrentTaskId])
main
InitOS()
Init_threads()
InitOS
FOR all tasks DO:
GetTask from generated .cfg
Initialise ThreadTaskType
Init_threads
FOR all tasks DO:
IF i.autostart == True:
start_thread(i)
TaskWrapper
+TaskId : int
WHILE LOOP (Continous)
IF (jump from terminate):
LOCK mutex TaskId.mutex_task
ELSE IF (TaskId.pthread_status == started):
UNLOCK mutex TaskId.mutex_task
ThreadTask[TaskId].entry()
END LOOP
start_thread
+threadId : int
IF threadId.pthread_status == terminated:
threadId.pthread_status = started
ELSE IF threadId.pthread_status == empty
threadId.pthread_status = starting
pthread_create(threadId) -> StartTask
StartTask
FOR TaskId == threadId DO:
TaskId.pthread_status = started;
INIT all mutexes;
TaskWrapper(TaskId);

**Figure 3.4:** Illustration of the execution path and its flow through different functions when a task activation or termination is called. Inside the red frame, we provide the description of the structure that is used throughout our implementation and describes the concept of a thread executing a specific task, including all the necessary fields to hold the relevant information.

since it is not as low-level as anticipated: instead of relying on its own scheduler, it is running on top of a Linux OS, utilising the UNIX kernel and its scheduler, leaving less control over task priorities. This ties back to the project goal (Section 1.4) to investigate how an AUTOSAR defined OS would operate on top of Linux.

The transformation of the AUTOSAR OS functionality to a Linux-based, C-written application was never expected to be a straightforward task; the main issue, as anticipated, was the adaptation of the different OSEK-specified operations to function within the multi-core environment of a standard Linux distribution. As it will become more evident in the upcoming sections, this required experimentation with different synchronisation schemes and paradigms in order to achieve a fail-safe parallel execution scenario. We achieved that by utilising the POSIX standard (Section 2.1.2) to create schedulable entities in the shape of Pthreads.

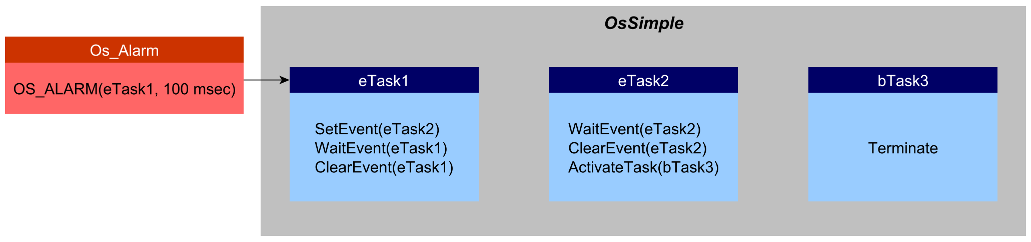


**Figure 3.5:** The execution scenario of the test application *OsSimple*

### 3.3.2 Application Level

The application level resides on top of the system level. As hinted by the name, it houses applications: *OsSimple* and *OsBenchmark* are two examples which we have been working on during this thesis. *OsSimple* has been the main test application we used to test *correctness*, *performance* and compliance to real-time *timing* constraints of the ported OS. The example is very much what it sounds like: a simple application that contains most of the fundamental OS functionality. This includes operations like terminating and activating a task, setting and clearing events and setting alarms and their corresponding counters. According to the execution scenario of *OsSimple*, which can also be seen in Figure 3.5:

1. An alarm sets an event (`Event1`) for extended task `eTask1`
2. `eTask1` sets an event (`Event2`) for extended task `eTask2` and waits for a new event, `Event1`
3. `eTask2` activates basic task `bTask3` and waits for a new `Event2`
4. `bTask3` does nothing and self-terminates
5. Loop through step (1)

*OsBenchmark* would essentially function as a stopwatch to time the performance of the different operations carried out in the same fashion as in *OsSimple*. Before terminating, a function would calculate the mean times over the iterations and return the results to the execution environment.

However, we did not manage to get the *OsBenchmark* to perform correctly, since it is targeted to run on single-core hardware and thus assume sequential execution. Essentially, the thread calling e.g. for a task activation was not suspending afterwards, continuously calling for activation by "firing" multiple requests. This behaviour is making timing intervals between calls irrelevant (Figure 3.6).

The second part of Figure 3.6 shows the required behaviour and also the resulted one after we applied a synchronisation scheme to force the calling thread to wait for the called task to return, before sending another activation request. Apart from this fix, the *OsBenchmark* continued producing dubious results and we decided to proceed with testing utilising our initial test application, *OsSimple*.

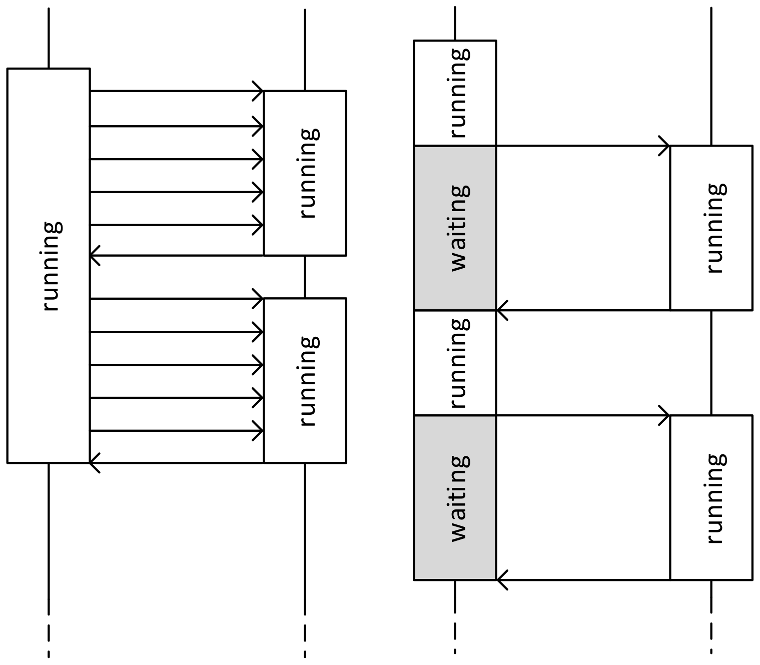


**Figure 3.6:** The left figure represents the benchmarking application, without any synchronisation applied. Skewing the results since the application would continue making function calls without letting the child thread finish. The right figure shows the wanted behaviour, the parent thread waiting while the child thread is executing.

Updating it with checkpoints between the desired operations, we managed to construct a solid test platform for our performance testing and postponed working on fixing the *OsBenchmark* for later in the course of the project.

The opportunity to do so appeared during April, when the company organised a 2-day event, which was focused on team building by allowing employees to work with others than their usual project partners on small fixes and issues that had been postponed as they were characterised non-urgent.

As we were not affected by said issues, which were mostly revolving around the company's AUTOSAR development tool, *Arctic Studio*, we decided to take part in the event by putting our efforts on something that would benefit both our project and future use of the company's testing suite. We therefore chose to rework the test framework of *OsBenchmark* so that it would function for parallel execution as well. The main idea behind the test was kept unsullied, but in order to synchronise each test iteration we chose to use barriers (as introduced in Section 2.1.2), allowing each task to finish before continuing with the next iteration.

## 3.4 Development Process

This section will delve deeper into the practical work that was carried out during the thesis, the modelling process of the application under development, the shape of our solution and the course of actions that was followed in order to solve problems and exploit optimal performance from our platform. Our development path was based on the RUP and thus divided into iterations (Section 3.1). Before proceeding with describing each iteration and the work that was carried out during, we will highlight the key points of our testing approach and method.

### 3.4.1 Test Strategy

In order to come up with a solid testing strategy our supervisors provided us with a set of reference results (Appendix A). According to our supervisors' request, we focused our efforts to the set of operations shown below, since they are considered the most critical when it comes to predicting and placing a ceiling on their completion times (*Timing* aspect):

- Task Activation
- Task Termination
- Set Event
- Clear Event
- Task Start-up

As one can notice by simply comparing the reference results to the list above, the *Task Start-up* metric is not included in the reference. This is due to a difference in the timing methods between our own and the reference approach: we count the *Set Event* time as strictly the interval between calling and returning from the function that handles the event setting, while the reference method *Set Event* time includes the *Task Start-up* time. This newly introduced metric holds the amount of time it takes for a task to be activated after its parent task has blocked waiting for an event to be signaled.

As mentioned already in Chapter 1, our intention was to investigate the developed application in terms of *correctness*, *performance* and *timing*. These three axes acted as the foundation of our testing strategy throughout the course of this project and provided with valuable inspiration on how to approach and interpret the generated arithmetic results. As it is presented more clearly in the upcoming chapters, we reflect on all three of these criteria and elaborate on the effect they have on the requirement specification for each of the upcoming iterations.

### 3.4.2 Implementation Iteration 1

The first iteration started quite early in the project, immediately after the planning phase, while simultaneously we were researching on the most important theoretical aspects of the project. The majority of the time spent was related to code reviewing in order to understand the previous work done by ARCCORE employees.

#### 3.4.2.1 Requirements

The initial iteration was focused on building a runnable application on which we would base the upcoming versions of our implementation. Consequently, the basic requirements were set towards that goal, as the outcome of the first iteration. Our attempts were based on the prior work done within ARCCORE AB, but diverge the point of interest from Ethernet communication establishment to following correctly the *OsSimple* execution scenario.

#### 3.4.2.2 Development Steps

Following the proposed structure according to the prior work carried out in ARCCORE, the basic version of our application was handling the task termination in an unorthodox way: instead of properly terminating a running thread, upon call, the `TerminateTask` function was performing a non-local jump, utilising the C standard library `setjmp.h` [50], to the `TaskWrapper` function. This header allows deviations from the standard execution flow by enabling functions to "jump" and continue executing from a predefined point in another function [25]. To visualise the behaviour of this pair of calls, we provide a straightforward example of code along with its resulted output:

```c
#include <setjmp.h>

main() {
 jmp_buf env;
 int i;

 i = setjmp(env);
 printf("i = %d\n", i);

 if (i != 0) exit(0);

 longjmp(env, 2);
 printf("Does this line get printed?\n");
}

******************************************

Output:

i = 0
i = 2
```

As it can be seen in the output section, there is no print after the long jump is performed. This is due to the fact that after the long jump back to assigning the value of $i$, its new value is printed and then, since $i$ != 0, the program exits.

The purpose of that jump was to put the tasks to sleep when `TerminateTask` is called, without destroying the corresponding Pthread. If a task activation was called for one of the sleeping tasks, this task would be reassigned to its Pthread and thus a new thread did not have to be created. However, there were multiple activation attempts before a successful one, since the duration of the sleep was much longer compared to the amount of time that an activation request would need to be executed.

When it comes to the task activation sequence of events, the flow diverges between two different paths, depending on whether this is the first attempt to activate (and thus create) a task or just a re-activation. If it is the first case, the execution flow goes through the *Start_Thread* where the threads are initially created and assigned to the intended tasks by calling the corresponding application function. In case the Pthread was already created, the activation takes place in the `TaskWrapper`, where the status flag of the task is changed to *STARTED* and its corresponding function is invoked.

#### 3.4.2.3 Testing Phase

Wrapping up the first development phase, the application was in a running state and fulfilled the basic requirement of *correctness.* Our goal was to confirm that the scheduling sequence described in the *OsSimple* application would be executed accordingly in our implementation. By using a quite detailed logging system, we were able to verify that the implicit operations of task activation and task termination were carried out properly. The same method was followed to verify the proper function of the chain of alarms and events. Issues such as the multiple activation attempts that we mentioned previously were postponed to be examined during the next iteration, since the fundamental requirement for this phase was to reach a point where our application would perform all the basic operations correctly.

It is worth pointing out that through the first test and evaluation stage we got the opportunity to familiarise with the debugging process of the company's Makefile system and the source tree structure. We also reached a better understanding on how AUTOSAR OS should operate and we were able to transfer the theoretical knowledge of OSEK to the actual implementation of the OS porting.

### 3.4.3 Implementation Iteration 2

Iteration 2 was the step of the project during which we started considering the aspect of *performance* in our implementation process. After this iteration's testing phase we constructed the requirements for the upcoming iterations in regards to achieving better performance without jeopardising the *correctness.*

#### 3.4.3.1 Requirements

The outcome of the first evaluation stage left us searching for a more viable solution to the tactic employed for task activation and termination in the OS porting. We did not have concrete performance results from the previous iteration, however it was obvious by the structure of the code and the amount of time that the threads were sleeping each time, that we were far off the target reference results. Our intention was to achieve improved performance, while simultaneously devising a task termination mechanism that would allow the OS to remain in an idle state, in case no tasks were running.

#### 3.4.3.2 Development Steps

Shifting our efforts to improve the *performance*, we realised that the extensive logging system that provided us with invaluable feedback during the previous iteration was causing too much overhead. Moreover, a minor quality-of-life change was done by disabling the `Daemonise` function, which upon call would execute the *OsSimple* application in the background. Disabling the this function facilitated the development process, since it became easier to keep track of the application being run in the foreground, printing information messages in the process and not directly to the system log.

Studying the code structure more extensively, helped us realise that ordering the threads to sleep (especially for one millisecond) instead of actually terminating them was not just ineffective time-wise, but was also deviating from the OSEK specification, considering that our initial implementation is not supporting an idle state as the AUTOSAR OS should do.

Instead of using the non-local jump in `TerminateTask` to `TaskWrapper`, we utilised a pair of standard POSIX functions, `pthread_exit` and `pthread_join`. These functions as explained thoroughly in Section 2.1.2, allow a POSIX thread to properly terminate, while the parent thread waits for the termination of its child Pthread, providing with the necessary synchronisation to achieve scheduling correctness. This rework however, resulted in too much work overhead. Significant restructure of the code had to be done before we would be able to have the new version of our application working correctly. This was due to the fact that the whole functionality was implemented based on the sleeping tasks paradigm.

After meeting with our technical supervisors regarding the idea of terminating the threads instead of putting them to sleep, we were advised to go forward with the testing phase of this iteration, since we had a runnable application that was behaving as expected, however they predicted that this approach would not achieve similar performance to the reference results (Appendix A) and hence we would have to come up with something even more efficient.

Following their advice, we briefly explored alternative ways to fulfil the iteration's requirements but also achieve performance gains. The solution we came up with was to switch to a more suitable POSIX standard function, the `pthread_cancel`. The difference between `pthread_exit/pthread_join` and `pthread_cancel` is essentially that the latter does not wait until the target Pthread is terminated, rather it returns immediately upon call (Section 2.1.2). This evidently provides an advantage to our implementation, as the use of `pthread_cancel` ensures both the termination of the calling Pthread and the correct chain of events when it comes to task scheduling. It also achieves better performance compared to the blocking paradigm of `pthread_exit`, since it performs these operations asynchronously.

After both versions of the application were executing correctly, we proceeded to the testing and evaluation phase in order to produce constructive results and elaborate on the findings.

#### 3.4.3.3 Testing Phase

We began this testing phase with benchmarking the first alternative: the `pthread_exit/pthread_join` implementation. To that end, we focused on the task activation and termination operations, since the functions related to events were not affected by this shift in the implementation. The results, as foretold by our supervisors, were not close enough to our target timings. Subsequently, we turned our efforts to generate performance results for the second alternative, the `pthread_cancel` version of our application. As anticipated, we received substantial improvements compared to the first alternative.

Besides the very constructive results obtained by evaluating the second iteration version of our application, we also performed the performance testing for the implementation of the first iteration. This only verified our initial assumption: the sleeping approach was performance-shredding.

### 3.4.4 Implementation Iteration 3

The mechanism through which we handled termination during Iteration 2 has shown signs of significant improvement but not at the desired level to be able to compare them with the set of reference results that was our main goal. Hence, during this iteration we had to revise our approach to that end and verify the results not only in terms of *correctness* and *performance* but also include RT deadlines, relative to real-world automotive application deadlines. This required also to equip the Linux kernels of our systems with RT capabilities, which was achieved using the *ChronOS* RT Linux patch [51].

#### 3.4.4.1 Requirements

The `pthread_cancel` undoubtedly gave us better results. However, we were still quite far from achieving similar timings to the reference results (Appendix A). That convinced us that we should try going back to using the `TaskWrapper` but not in the previous fashion: now we would neither actually terminate the tasks and kill their corresponding Pthreads nor putting them to sleep. Once again, reaching a faster solution for task termination was the main focus but keeping the correctness intact while exploring different solutions proved to be the main challenge.

#### 3.4.4.2 Development Steps

The problem with the `TaskWrapper` was the sleep involvement. By just removing the sleep command we would not solve the issue of performance, since we would leave the application execution with no synchronisation point and thus leaving a function running indefinitely without locking it or suspending it (in the OS world this is called *busy waiting*). Inspired by the fact that the events are also handled by mutexes, it only seemed natural to try and synchronise the tasks using mutexes in a similar way.

Essentially, each time the `TaskWrapper` is called via the `TerminateTask` function, it puts that thread into a suspension state by locking it with a mutex. To unlock it we use the `pthread_mutex_unlock` function call in task activation to ensure that the task is restarted on the same Pthread after it has been "terminated".

Apart from this major update in the application's structure, we also concluded that a big contributor to bad performance was the logging functionality, that proved to be vital for the purposes of debugging during the first iteration, however since it had already served its purpose we decided to remove it in an attempt to ameliorate the performance results.

#### 3.4.4.3 Testing Phase

During this test phase, we carried out the usual performance testing, witnessing finally a breakthrough: the results for *Task Activation*, *Task Termination* and *Clear Event* were significantly better than the reference results (Table 4.3a). Nevertheless, the results for *Task Start-up* and *Set Event* were still worse than anticipated. After careful consideration we came to the conclusion that this was caused by the poor performance that the `pthread_cond` was showcasing.

Another important aspect of this testing phase was the performance evaluation under load: our supervisors suggested to experiment with average (around 50%) and full (100%) load applied on the host systems' CPUs. This way we managed to extract valuable data for the performance metrics we examine our developed system for and the difference in behaviour from one host system to the other. It is worth mentioning that in order to increase the CPU load and manage to control it percentage-wise, we used two Linux programs, *stress* and *cpulimit*. *Stress* is enforcing load onto the CPU while *cpulimit* can be used to keep the CPU load up to the required percentage.

As mentioned in the introduction of this iteration, we also experimented with RT patched kernels as the test environment for our application. We performed exactly the same set of test scenarios as we did with the standard SMP kernel and all the load alternatives. The generated results are included in Appendix C and discussed in Chapter 5.

### 3.4.5 Implementation iteration 4

After completing Iteration 3, the only metrics that were still not close to or better than the reference results were the SetEvent and Task StartUp. In order to achieve this, we focused on the method that events were handled in our current implementation version and came up with a more efficient way. Based on the selected approach, we carried out the same testing scenarios for the updated implementation version, along with tests on the selected embedded alternative, the *MinnowBoard MAX* [52].

#### 3.4.5.1 Requirements

The requirements for our final iteration were very specific: replace the synchronisation mechanism that controlled the events up until Iteration 3 with something more efficient in terms of performance, in order to achieve the desired results. After finishing with the modifications on our application, we planned to run it over an embedded Linux distribution hosted in our test board, the *MinnowBoard MAX* in order to get a taste on how our VM would perform running over a slower host machine. In terms of testing, we would follow the same approach we did in Iteration 3 with the addition of running the same test scenarios on the test board, with a variation in the kernels utilised.

#### 3.4.5.2 Development Steps

The major issue that still persisted in the end of Iteration 3 was the high *SetEvent* and *StartUp* times, which should sum up to a relative number to the reference results. The reason behind this was that the synchronisation scheme which was managing the events included a message passing approach, namely the `pthread_cond_wait` and `pthread_cond_signal` pair of function calls (Section 2.1.2).

As explained in Section 2.1.3.3, these types of methods provide the system with high level communication channels between processes, with the ability to exchange messages with each other, however in the expense of performance. The fact that the additional functionality was redundant for the purposes of synchronising the events operations steered our efforts to implement an equivalent scheme while using a simpler mechanism. It only seemed natural to employ again the usage of mutexes, as we did with managing the tasks operations in Iteration 3. It was also sensible, since the only part of the pair of function calls used before that was vital for our implementation was the blocking/unblocking paradigm, attainable also through a pair of mutex calls.

After finalising the updates of the synchronisation scheme that controlled the events, we turned our attention to transferring our emulation platform to the embedded alternative we were advised to use for the purposes of testing, an Intel Atom-based *MinnowBoard MAX*. For that purpose, we used the *Yocto Project* [19] to create customisable embedded Linux images, tailored for our Intel platform. We built two different images in order to observe any potential deviations on the results, since they included distinct versions of Linux kernels:

- A *Sato* image, which is mainly purposed for mobile platforms and is equipped with a full Graphical User Interface (GUI) – compared to the RT distribution that is command-line-based.
- An RT version, which essentially is a minimal Linux distribution armed with extra RT capabilities, a preemptive scheduling policy and a higher resolution system clock.

It is worth mentioning that we run our application *on top* of the Linux distributions and did not append it as a layer within the images created. It would be quite time-consuming to create our own customised distribution, since we would have to perform multiple extra tasks in order to achieve that, without any insurance that we would get any performance gains from this approach.

#### 3.4.5.3 Testing Phase

For the testing phase of Iteration 4 we followed the exact same path as we did in the previous iteration, testing our application in regards to all three aspects, *correctness*, *performance* and *timing* both on the standard SMP Linux kernel but also on the RT-patched kernel, as well as including the concept of load in our tests. The only thing that was added to this iteration was naturally the tests for our embedded alternative, the results of which can be found in Appendix D.

After finalising this phase, we presented the generated results to our technical supervisors, who concurred that the progress achieved during the course of this project was quite impressive and gave us the green light to proceed with documenting our work and elaborating on our results.

# 4

# Results

This chapter contain results from the testing phases, carried out throughout the course of this project. The first section will explain how the tables and graphs are going to be presented. The results will start off by showing the average values for each metric in each iteration, followed by a more detailed view with the test points from Iterations 3 and 4 in the shape of graphs. The chapter wraps up with the correctness and timing evaluation results.

Before all the results are displayed, a small version of Table 1.1 is shown in Table 4.1 below to reiterate the technical specifications of each host machine used during testing. Observe that despite the difference in the amount of cores between the two CPUs, the Core i5 yields higher benchmark score per core compared to the Core i7.

**Table 4.1:** The first two rows of Table 1.1 showing the host machines used during the testing of our applications

| Processing Unit | Frequency | # Cores | Coremark | Coremark/Core |
|---|---|---|---|---|
| **Intel Core i7-3720QM** | 2.6GHz | 8 | 85209 | 10651 |
| **Intel Core i5-4300M** | 2.6GHz | 4 | 46085 | 11521 |

## 4.1 Performance Results

The performance results will be displayed both in the shape of tables and graphs, in order to be easily interpreted. The generated results from Iterations 1 and 2 are presented only with tables (Tables 4.2a and 4.2b) since a visual representation is not considered vital for this stage of the project; the reader can grasp the difference in magnitude by simply contemplating on the numerical results. They will be displayed along with the corresponding tables from Iteration 3 and 4 (Tables 4.3a and 4.3b). The averages shown in these tables are generated from tests carried out on the generic SMP kernel.

The performance results from the latter iterations will also be presented with graphs, one graph for each metric and host machine. The metrics that are highlighted and shown in this chapter are the *SetEvent* and *Startup* metrics. Each graph shows one metric with two sets of data points, one set for each kernel type (SMP and RT-patched). The remaining graphs from the testing are available in Appendix C. The results for the host machine *peos* (i5) will be displayed first, then the results from host machine *ethan* (i7) will follow. This order stays the same through both iterations.

The captions underneath each graph contain vital information, mainly the points of interest for the illustrated metric and how its performance correlates to the previous averages. It should be noted that some graphs that included high "spikes" have been scaled up, so that the more interesting parts are highlighted. This might lead to some data points not being shown in the graphs, however the general idea is preserved and the numerical results in the tables are kept intact despite this skewing.

**Table 4.2:** The average execution times for Iteration 1 (a) and 2 (b). Iteration 2 shows quite a large increase in performance due to the remove of sleep in `TaskWrapper`

(a) Iteration 1

| Metric [ns] | Core i5 | Core i7 |
|---|---:|---:|
| Task Activation | 541780 | 534029 |
| Task Termination | 20382 | 39873 |
| Set Event | 42522 | 19591 |
| Clear Event | 19874 | 21342 |
| Task Start-up | 59328 | 63780 |

(b) Iteration 2

| Metric [ns] | Core i5 | Core i7 |
|---|---:|---:|
| Task Activation | 20218 | 10852 |
| Task Termination | 14618 | 7936 |
| Set Event | 13784 | 8244 |
| Clear Event | 5318 | 10827 |
| Task Start-up | 22102 | 27203 |

**Table 4.3:** The geometric mean for each metric on the two different host machines for the implementation in Iteration 3 (a) and 4 (b). Iteration 4 gave better performance for both host machines, but is much more noticeable for the i7

(a) Iteration 3

| Metric [ns] | Core i5 | Core i7 |
|---|---:|---:|
| Task Activation | 238 | 1332 |
| Task Termination | 148 | 374 |
| Set Event | 2645 | 4971 |
| Clear Event | 193 | 1189 |
| Task Start-up | 3309 | 24981 |

(b) Iteration 4

| Metric [ns] | Core i5 | Core i7 |
|---|---:|---:|
| Task Activation | 198 | 232 |
| Task Termination | 154 | 136 |
| Set Event | 1086 | 133 |
| Clear Event | 174 | 519 |
| Task Start-up | 1365 | 213 |

With the magnitudes of the average execution metrics in mind, let us delve deeper into the graphs below which focus on the *SetEvent* and *Startup* metrics.

When comparing the iterations we can see how the *SetEvent* metric for the *peos* machine (in Figures 4.1 and 4.2) becomes noticeably faster, but drops in correctness. However, looking at Figures 4.3 and 4.4 for the *ethan* machine we see that it keeps a high correctness with the generic SMP kernel as well as getting more concentrated points together with the faster execution. It dropped the execution times from ∼5,000 ns to ∼500 ns.

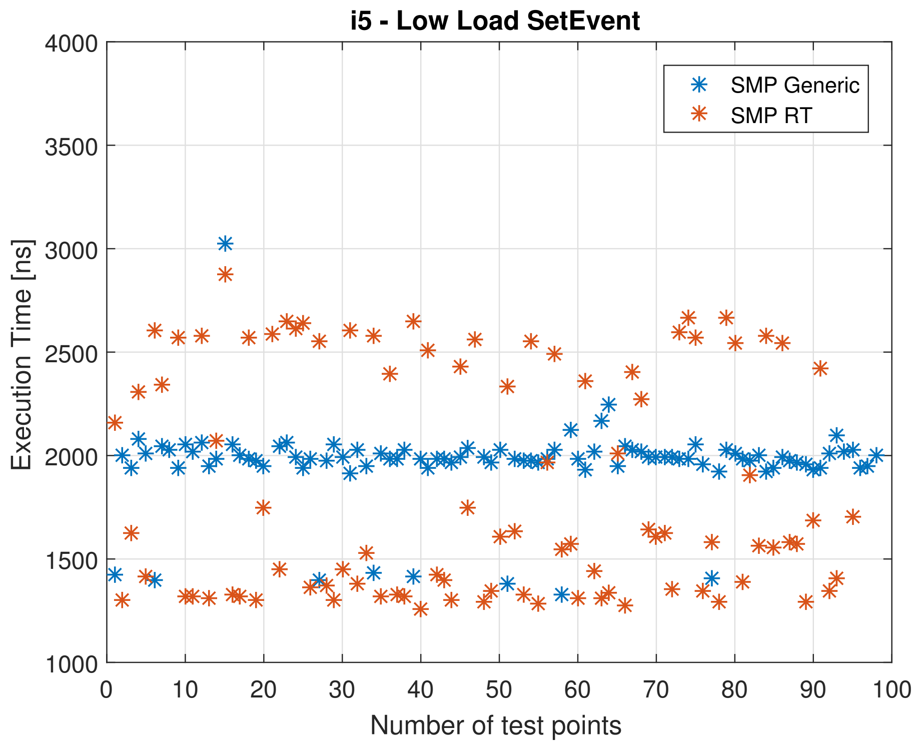


**Figure 4.1:** iteration 3: Some correctness loss as the points do not reach to 100 test points, the generic kernel displays better concentration of the test points, averaging about 2,500 ns

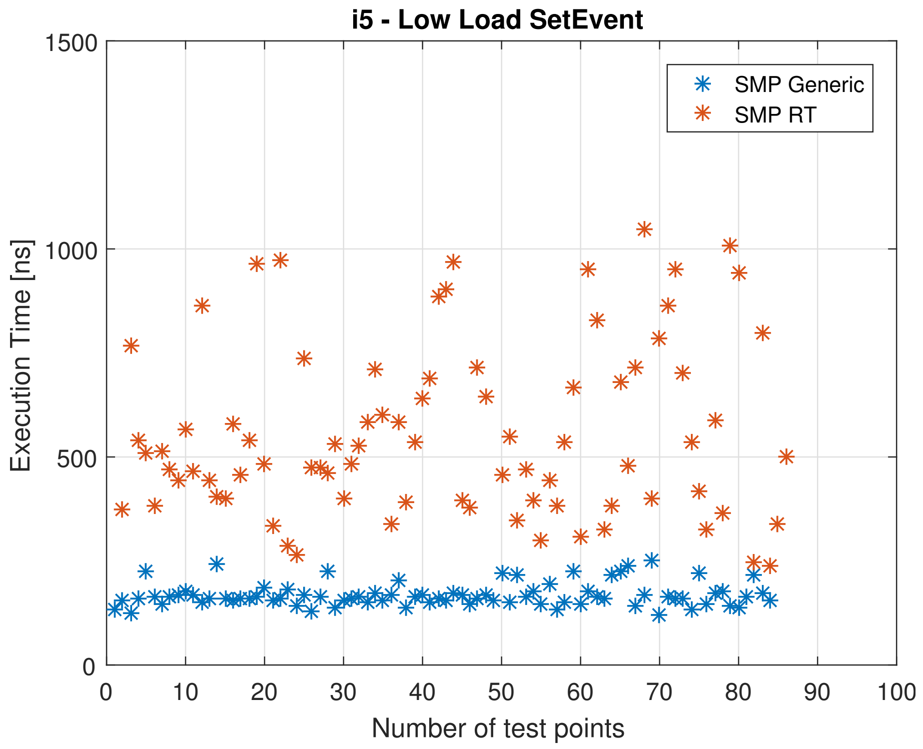


**Figure 4.2:** Iteration 4: The generic kernel keeps the points concentrated around 200 ns which is a major increase in performance. However, the correctness is down to 85%

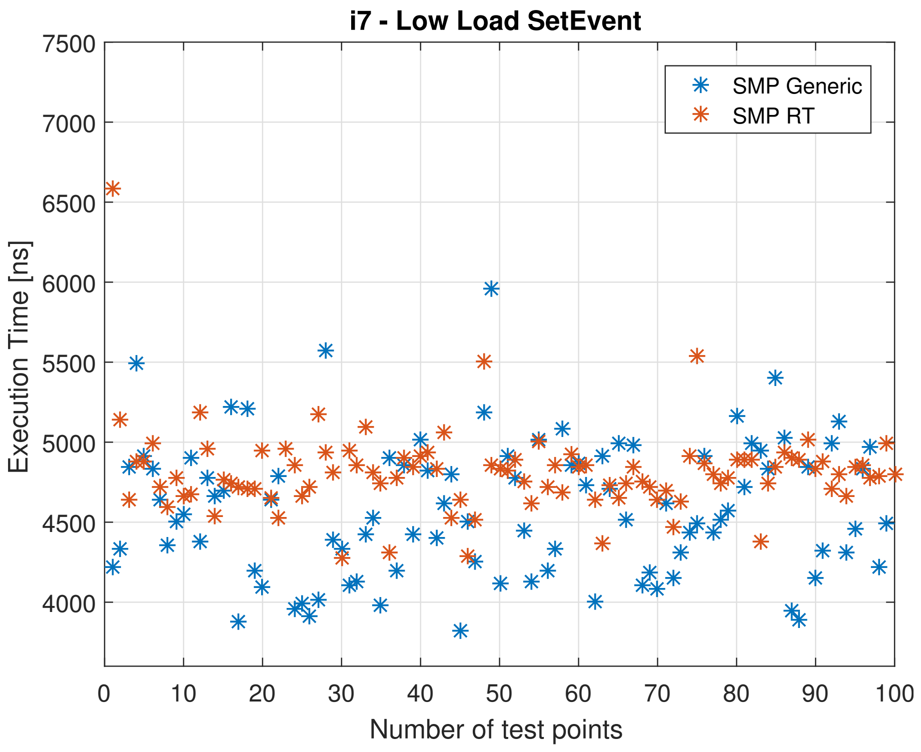


**Figure 4.3:** Iteration 3: Following the trend of high correctness but with a higher average than the i5 counterpart at around 5,000 ns

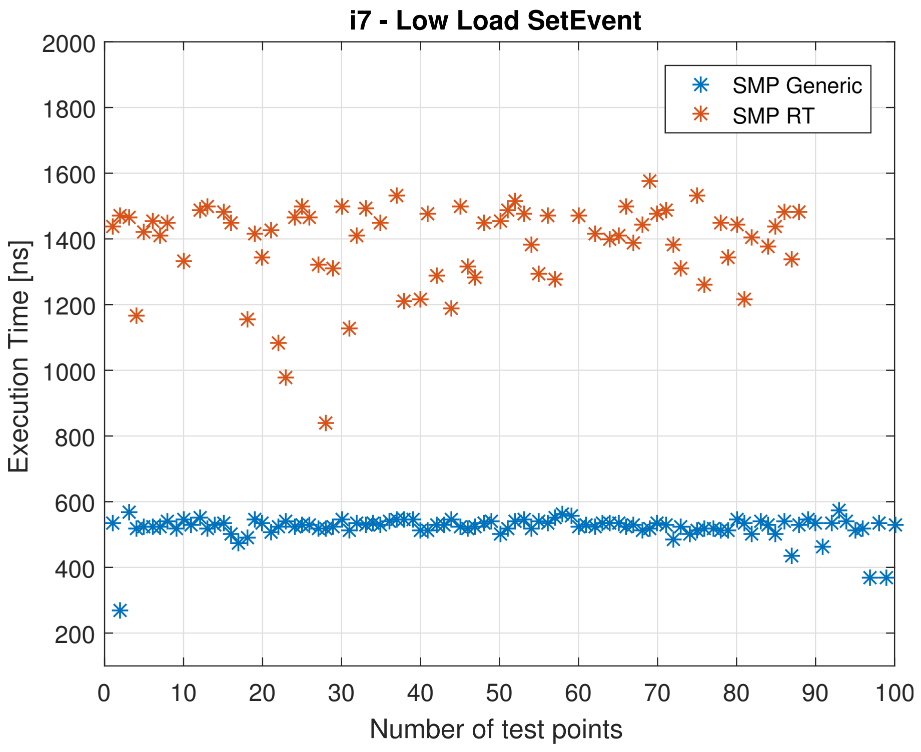


**Figure 4.4:** Iteration 4: Still very good correctness, much better than the i5 counterpart. Execution is also remarkably quicker, down by almost 4,000 ns

The *Startup* metric shows a similar story. The *peos* machine gains in performance for both kernels as it can be seen in Figures 4.5 and 4.6. However, similar to the *SetEvent* metric, it looses some correctness. The performance gains are not as outstanding as with the *ethan* machine, but are still worth mentioning.

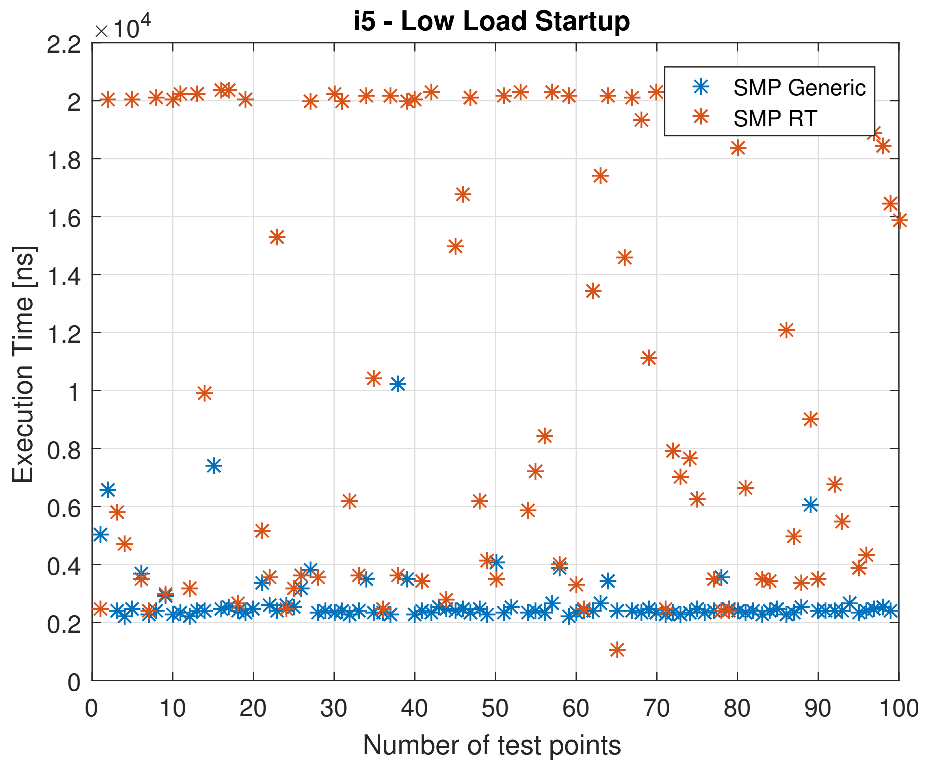


**Figure 4.5:** Iteration 3: The RT kernel executes around 20 $\mu s$, whereas the generic executes around 3,000 ns

In a similar manner as with the other metric, the *ethan* machine gains performance while keeping high correctness and concentration of points, as it can be seen in Figures 4.7 and 4.8. The execution times drop from ∼25,000 ns to below 200 ns. This is all in regards to the generic SMP kernel, the RT kernel illustrates different results; much like the SMP kernel for the *peos* machine, the RT kernel achieves performance gains but declines roughly 10% in terms of correctness.

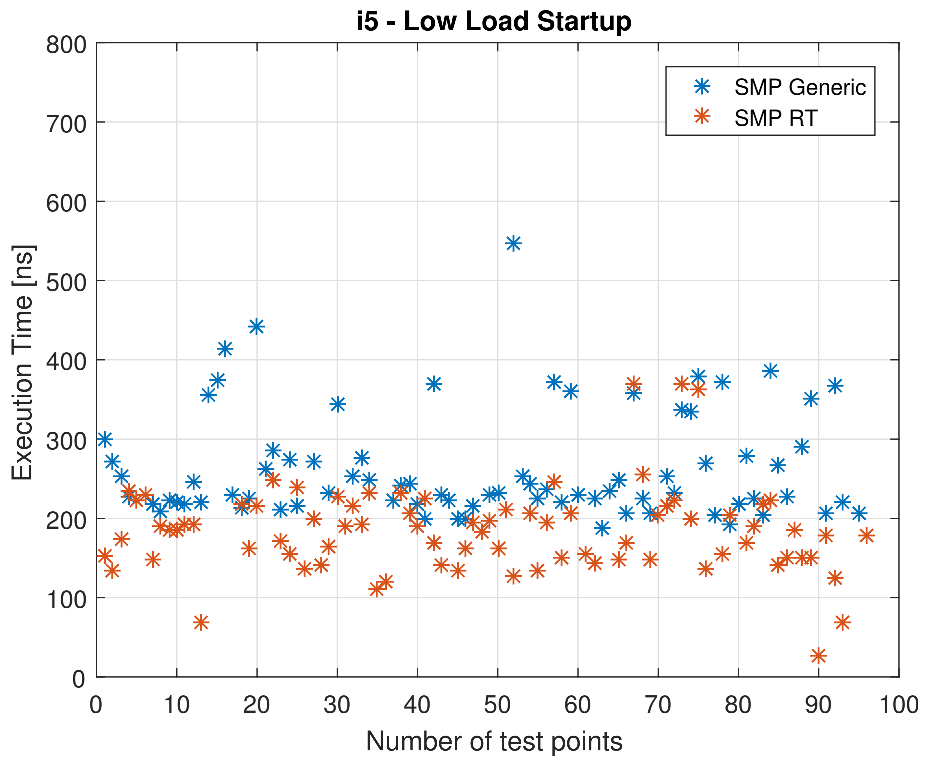


**Figure 4.6:** Iteration 4: Start-up has suffered in correctness, now around 95%. Nevertheless, great performance increase compared to Iteration 3, now down to $\sim 200$ ns

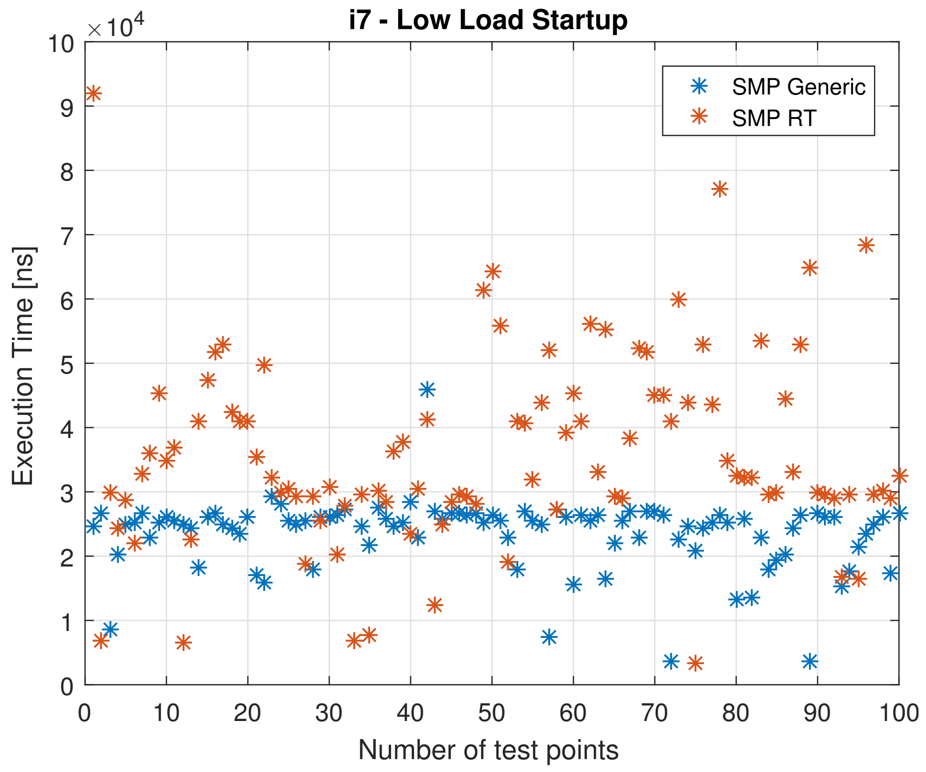


**Figure 4.7:** Iteration 3: Concentrated points with high correctness. Average execution time around 25,000 ns which is significantly worse than its i5 counterpart

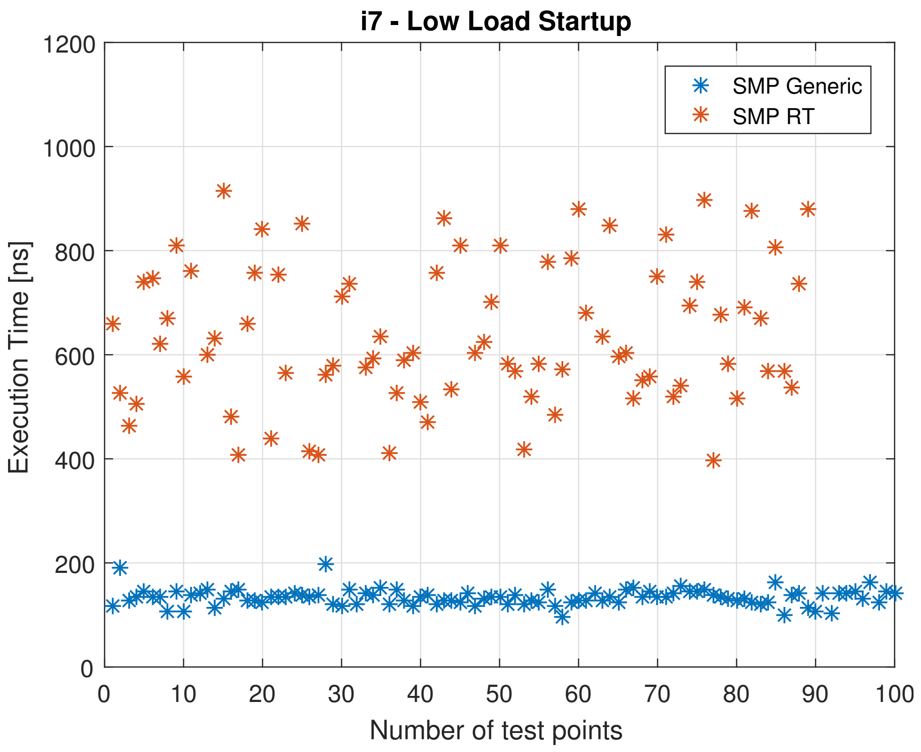


**Figure 4.8:** Iteration 4: Major performance gain, down to 200 ns from 25,000 ns. The RT kernel drops in correctness while the generic SMP preserves its high levels of correctness

## 4.2 Correctness and Timing Results

The *correctness* of iterations 1 and 2 are regarded in terms of correct execution, related to the expected execution scenario explained in Section 3.3.2.

The correctness results for Iterations 3 and 4 are on the other hand based on the amount of valid points we got during testing. A data point is seen as valid if it is below a certain threshold, making sure it would finish before its intended deadline. The threshold is application-specific, therefore we based it on the average use case of an ECU destined for an automotive purpose. The threshold for *SetEvent* and *Start-up* was set to 5 ms while for all other metrics it was set to 1 ms. All testing scenarios include 100 test nodes, in order to use the same amount of points as the reference results, a number that provides a clear notion of the achieved correctness in terms of percentage (Tables 4.4 to 4.7).

**Table 4.4:** Iteration 3: High correctness for all metrics in general, with the SMP kernel being better than the RT

| **Peos (i5)** | **Low Load** | | **Average Load** | | **Full Load** | |
|---|---|---|---|---|---|---|
| **Metrics [%]** | **SMP** | **RT** | **SMP** | **RT** | **SMP** | **RT** |
| Task Activation | 100 | 100 | 100 | 100 | 100 | 100 |
| Task Termination | 100 | 100 | 100 | 100 | 100 | 100 |
| Set Event | 98 | 95 | 99 | 97 | 98 | 86 |
| Clear Event | 100 | 100 | 100 | 100 | 100 | 100 |
| Task Start-up | 99 | 100 | 99 | 95 | 99 | 87 |

**Table 4.5:** Iteration 3: High correctness, similar to the i5. However, the RT kernel shows better correctness than the SMP kernel

| **Ethan (i7)** | **Low Load** | | **Average Load** | | **Full Load** | |
|---|---|---|---|---|---|---|
| **Metrics [%]** | **SMP** | **RT** | **SMP** | **RT** | **SMP** | **RT** |
| Task Activation | 100 | 100 | 100 | 99 | 100 | 100 |
| Task Termination | 100 | 100 | 100 | 100 | 100 | 100 |
| Set Event | 99 | 100 | 91 | 91 | 77 | 93 |
| Clear Event | 100 | 100 | 100 | 100 | 100 | 100 |
| Task Start-up | 100 | 100 | 97 | 96 | 85 | 97 |

**Table 4.6:** Iteration 4: Worse correctness compared to Iteration 3, most noticeable under high load, as the correctness degrades quite rapidly. The RT kernel showcases better correctness for higher load compared to the SMP

| **Peos (i5)** | **Low Load** | | **Average Load** | | **Full Load** | |
|---|---|---|---|---|---|---|
| **Metrics [%]** | **SMP** | **RT** | **SMP** | **RT** | **SMP** | **RT** |
| Task Activation | 100 | 100 | 99 | 100 | 99 | 99 |
| Task Termination | 100 | 100 | 100 | 100 | 100 | 100 |
| Set Event | 84 | 86 | 28 | 73 | 34 | 39 |
| Clear Event | 100 | 100 | 100 | 100 | 100 | 100 |
| Task Start-up | 95 | 96 | 22 | 57 | 30 | 25 |

**Table 4.7:** Iteration 4: Correctness takes a large hit compared to Iteration 3 when considering the RT kernel. The SMP kernel is still performing quite well for three of the metrics, *Task Activation*, *Task Termination* and *Clear Event*.

| **Ethan (i7)** | **Low Load** | | **Average Load** | | **Full Load** | |
|---|---|---|---|---|---|---|
| **Metrics [%]** | **SMP** | **RT** | **SMP** | **RT** | **SMP** | **RT** |
| Task Activation | 100 | 95 | 100 | 87 | 100 | 89 |
| Task Termination | 100 | 77 | 100 | 66 | 100 | 47 |
| Set Event | 100 | 88 | 25 | 19 | 37 | 32 |
| Clear Event | 100 | 94 | 100 | 82 | 100 | 90 |
| Task Start-up | 100 | 89 | 25 | 18 | 67 | 41 |

# 5

# Discussion

The discussion chapter will start off by discussing all results followed by an explanation and reasoning behind our decisions made on how to display our results, why we omitted some parts and why it does not affect our conclusions. We will also discuss why we chose to test the way we did, in regards to CPU load and the duration of our testing sessions.

## 5.1 Results Discussion

The scheme selected for the testing scenarios had to do with the fact that the reference results also include the same number of testing points (100 points), while when it comes to time intervals each testing loop was decided to execute for 20 s after experimenting with different values on that matter. Above 20 s, we were not yielding higher accuracy from each iteration: running a 10 s loop compared to the 20 s gave a deviation of 50% on the results, while the 20 s to 100 s loop only diverged at most 10%. Hence, the decision made was based on the trade-off of spending unnecessarily extensive amounts of time for each test scenario, when there would be no realistic gains in terms of accuracy. Moreover, considering that this deviation might even be caused by the Pthreads non-deterministic execution every time, we considered the 20 s an adequate duration for each test loop.

Between each test loop there is a sleeping interval of 1 second, in order to allow the host machines to bring down the load of the CPU before restarting the application and not to accumulate load from one iteration to the next. Taking into account all the time quantum, one test session takes roughly 35 minutes to complete, which was manageable in comparison to have to deal with triple that time in case we went for the 100 s duration loop.

It became clearer after the first iteration and the shift from the *sleep* strategy, that the dominant metrics would be the *SetEvent* and task *StartUp*. This can be explained through the definition of the *StartUp* metric (Section 3.4.1), since it includes the time it takes to clear an event set for a specific task along with the activation time required, while the *SetEvent* included a complicated synchronisation and communication scheme that managed the events, up until Iteration 4 (Section 3.4.5).

During Iterations 1 and 2 the concept of *correctness* was limited to verifying that the intended execution scenario was followed, specifically in our case that the basic task *bTask3* was terminated and re-activated properly (Sections 3.4.2 and 3.4.3). However, there was a shift in our approach in Iteration 3, since it was our intention to invoke the aspect of RT constraints (the *timing* concept we talk about throughout our work).

For that purpose, we had to determine a test-independent time constant that would represent the deadline for our metrics, in order to decide whether or not each of the testing points performs within an acceptable time frame and thus can be considered *correct*. This is visualised in Tables 4.4 - 4.7, where we included percentages of correct operation for each of the functions performed during each test session. It is quite clear from the illustrated results that the our assumption for the *SetEvent* and *StartUp* metrics is verified, since they showcase the lowest correctness levels throughout Iterations 3 and 4, degrading vastly as the CPU load increases. Moreover, from the same tables results, it can be noticed that in general Iteration 3 is *more reliable* than Iteration 4.

Since we did not have the background to make a decision on our own, we turned to our technical supervisors for consultation and they advised us to use a 5 ms deadline for the *SetEvent*/*StartUp* metrics and 1 ms for the faster operations, such as *ClearEvent* or *TerminateTask*. These numbers are considered fast deadlines in the world of automotive software, with 10 and 100 ms being also relevant. However, we decided to go with the fastest requirement, since our host machines are better equipped than the traditional ECU hardware and also we intended to reach the highest performance for our application, in order to examine the strictest constraints, considering the possibility that with ECU hardware going faster, the "traditional" real-time deadlines will diminish.

The performance testing, which was the focal point throughout our project development, was carried out in a similar way independently of which iteration was addressing. With the exception of course of the expansion of the testing scenarios pool (different kernel versions, load variance) the core concept of the performance testing remained unchanged, in order to enable us to compare the generated results metric for metric.

In the course of Iterations 1 and 2 (Tables 4.2a and 4.2b), we calculated the mean times for each of the metrics using a simple *arithmetic mean* (average) operation on the test points. Besides the ease which comes with it, there was no need to take extra care of the average, since the deviation between the numerical results was low.

The mean values for Iterations 3 and 4 metrics are displayed in Tables 5.1 and 5.2. At first, after calculating the means for all the metrics as the arithmetic means of the testing points, we were discouraged that due to high peaks in some samples, our average values were so heavily affected. We realised that the *geometric mean* here would be the suitable way to calculate the average values for our metrics, since the

**Table 5.1:** The geometric mean and the arithmetic mean for each metric on the two different host machines for Iteration 3

| | **Core i5** | | **Core i7** | |
|---|---|---|---|---|
| **Metrics [ns]** | **AM** | **GM** | **AM** | **GM** |
| Task Activation | 155 | 148 | 408 | 374 |
| Task Termination | 240 | 238 | 1436 | 1332 |
| Set Event | 488394 | 2645 | 224182 | 4971 |
| Clear Event | 194 | 193 | 1294 | 1189 |
| Task Start-up | 190972 | 3309 | 28676 | 24981 |

arithmetic mean is very heavily influenced by the atypically large values that occur. On the other hand, the advantage of the geometric mean is that it is not affected by such incidents and provides more accurate average results.

**Table 5.2:** The geometric mean and the arithmetic mean for each metric on the two different host machines for Iteration 4

| | **Core i5** | | **Core i7** | |
|---|---|---|---|---|
| **Metrics [ns]** | **AM** | **GM** | **AM** | **GM** |
| Task Activation | 162 | 154 | 138 | 136 |
| Task Termination | 208 | 198 | 233 | 232 |
| Set Event | 3302516 | 1086 | 134 | 133 |
| Clear Event | 176 | 174 | 521 | 519 |
| Task Start-up | 1045226 | 1365 | 213 | 213 |

Diving deeper into the performance results and how they can be interpreted, the massive drop documented from Iteration 1 to Iteration 2 especially in the Task Activation times is related to the change in our approach, replacing the sleep invocation with a standard initialisation/termination paradigm that made good use of the POSIX standard. It has also benefited all the metrics across the tables, since the sleeping was not only affecting the activation but all the operations performed by a task. We chose to use the `pthread_cancel - pthread_join` pair of calls (instead of the `pthread_exit - pthread_join`) with good reasoning: as it can be seen in Table 5.3 the result for a thread termination using the `pthread_cancel` call is much improved. The deviation recorded for the task activation, where the `pthread_exit` alternative is faster for 1.5 $\mu s$ can be credited to the nature of POSIX calls and the non-deterministic behaviour between separate runs.

A similar vast decrease in all the performance metrics is recorded from Iteration 2 to Iteration 3, where we have another major shift to the way we handled task-related operations. This is where these colossal gains in performance stop to be generated, except the *SetEvent* and *StartUp* metrics from Iteration 3 to 4 and specifically for the *ethan* (i7) host machine. However, achieving to surpass the threshold of the reference results by a large margin in Iteration 4 (and in some cases in Iteration 3 as well) left us satisfied and with the opportunity to consider the development process

**Table 5.3:** Pthread test results utilising `pthread_join` together with `pthread_cancel` or `pthread_exit`, showing the geometric mean of roughly 350 test-points measured in nanos.

| | **Thread Activation** | **Thread Termination** |
|---|---|---|
| `pthread_cancel` | 17371 [ns] | 12832 [ns] |
| `pthread_exit` | 15858 [ns] | 18167 [ns] |

complete. As mentioned before, we treat the results generated through both these iterations as final, since despite the major improvement in results we have deteriorated correctness levels in Iteration 4.

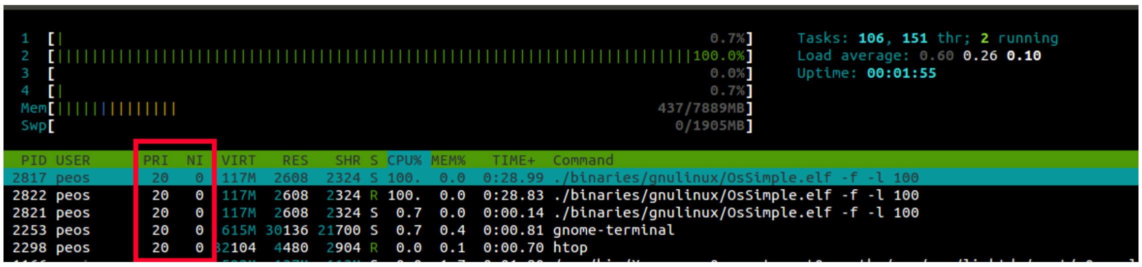
Besides the standard SMP Linux kernels we showcase the results of execution over an RT-patched kernel, the *ChronOS* RT Linux kernel [51]. This is an academic adaptation of the previously officially-supported *CONFIG_PREEMPT_RT* patch, implemented by a research group within Virginia Tech. Besides providing preemption, it also supports lower-latency, "sharper" interrupts. Despite these properties, the usage of the RT patch did not always yield better results and usually with deteriorated correctness levels, mainly due to the fact that our test application's (*OsSimple*) generated priorities were equal for all the tasks and hence the ability to preemptively execute them did not provide us with performance gains. Another reason is that we did not force higher priorities for our application threads (Figure 5.1) and thus the Linux scheduler treated our application equally to any other system process. This had a major impact on the loaded execution results, as illustrated with the help of the corresponding graphs residing in Appendix C.

A distinct mention has to be made concerning the results generated under load. The reason we only consider the low-load results relevant to real-life situations is that this would be the usual execution scenario for an automotive application; even in a system that would be responsible for more than one tasks, there is practically no scenario under which all the available CPU cores would be fully loaded. All the results generated from loaded execution are included in Appendix C. Someone can note a few inconsistencies, most of which are caused by the Linux scheduler's inability to spread Pthreads load evenly. Even under a low-load test scenario, our application is assigned solely to one core, which is constantly fully loaded, sporadically spreading some low-load threads to run on other cores as well (Figure 5.1). After these types of observations, we researched for related experiences from other developers, only to verify that the Linux scheduler has been treated like a "black-box" from the Linux community and with good reason: as described in [53], there are many instances when the Linux scheduler fails to perform as expected: there is a situation where the scheduler treats an eight-core processor as two-groups of four cores, which in times can lead to one group being overloaded with the other not contributing at all.

Similar occurrences of scientific interest continue to take place during testing under load: during Iterations 3 and 4, we observe that the average-load results were worse than the corresponding full-load timings. This can be due to either the Linux scheduler inefficacy to handle load effectively, or because of the scheme that we employed in order to force a specific percentage of load onto the CPU cores (Section 3.4.4.3), which is achieved through two Linux programs – additional overhead enforced on each core.

Another aspect of the testing phase that has been omitted from the Results section is the *MinnowBoard MAX* performance outcome. As discussed in Chapter 1, the Intel Atom CPU of the MinnowBoard is a middle-ware solution between the high-end host machines and the traditional, single core ECU hardware. The purpose of testing our application on an embedded alternative was mainly to verify that it performs well on a system other than the two personal-computer environments that we utilised to develop it. We also intended to explore the process of creating a customised Linux distribution, tailored for the intended hardware, and implant our application as an extra system layer on it.

However due to the complexity of this objective, combined with the lack of previous experience and adequate time, we ended up running the application on top of a variety of generic embedded Linux images with different kernel versions (a mobile and a terminal-based RT image). This translated into more complexity and at least slightly below the performance levels compared to what we would expect if we implemented our own bitbake layer. The results produced with the two generic kernel images can be seen in Appendix D.



```
  1  [|                                                                          0.7%]     Tasks: 106, 151 thr; 2 running
  2  [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]     Load average: 0.60 0.26 0.10
  3  [                                                                           0.0%]     Uptime: 00:01:55
  4  [|                                                                          0.7%]
  Mem[|||||||||||||                                                      437/7889MB]
  Swp[                                                                     0/1905MB]

  PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 2817 peos        20   0  117M  2608  2324 S 100.  0.0  0:28.99 ./binaries/gnulinux/OsSimple.elf -f -l 100
 2822 peos        20   0  117M  2608  2324 R 100.  0.0  0:28.83 ./binaries/gnulinux/OsSimple.elf -f -l 100
 2821 peos        20   0  117M  2608  2324 S  0.7  0.0  0:00.14 ./binaries/gnulinux/OsSimple.elf -f -l 100
 2253 peos        20   0  615M 30136 21700 S  0.7  0.4  0:00.81 gnome-terminal
 2298 peos        20   0 32104  4480  2904 R  0.0  0.1  0:00.70 htop
```

**Figure 5.1:** Application is occupying one core 100%, inside the red square it can be seen how Linux gives all processes equal "niceness" (Linux Priority)

# 6 Conclusion

This chapter intends to provide the reader with our views on issues we have encountered and assess whether we achieved to fulfil our project goals in the shape of the final outcome of our project. The chapter will end with our thoughts about future work, what directions we feel this project could be expanded towards.

Our initial goal of joining the concepts of AUTOSAR and Linux was achieved by implementing the OSEK specifications in a Linux environment. The major obstacle in order to achieve correctness, in terms of the execution scenario, was the uni-core nature of the OSEK, which was contradicting to the exploitation of our multi-core CPUs. In order to surpass that, the POSIX library provided the necessary tools to get the correct synchronisation scheme for our application.

Ensuring the correct behaviour, allowed us to focus on performance. We managed to generate significantly improved results, especially during our final iteration in comparison to the set of reference results. Even without utilising the concept of priorities, the contribution of parallel execution with Pthreads to the progress achieved compared to our initial implementation version was monumental.

Despite this fact, it can be noticed that we were not able to achieve the same magnitude of performance gains, always in comparison to the upgrade on the hardware we used for our purposes; going from a 210 MHz single-core embedded system to a 2.6 GHz multi-core computer does not translate into equivalent performance rewards. The concluding point behind this is that our test systems, besides our application, are loaded by multiple sources (i.e. complex network interfaces, fancy GUIs, OS operations, software updates) that, no matter how hard we tried to exclude while testing, are considered vital for a standard Linux distribution and hence the OS treats them accordingly. The low-end ECU hardware performs solely a dedicated function, executing a predefined sequence of tasks with simpler network interfacing and higher priority over other non-critical processes.
Another detrimental factor towards achieving greater performance levels, is the existence of multiple layers of software components that interact with each other in order to achieve the necessary virtualisation between the hardware and the Pthread level (3.3). Evidently, this fragmentation into layers causes extra overhead and latencies that are non-existent in the concept of the classic AUTOSAR software architecture. In other words, the high-end platform cannot exploit all the computing power, since it is burdened with additional and more complex software layers to execute.

Seeing how the solution of Iteration 4 improved the performance, allows further exploration of the possibilities to shift towards a more centralised structure of the system network in vehicles, instead of the one-ECU-per-function paradigm. Utilising hardware with multi-core capabilities, shows how the execution times work their way down toward the microsecond range (adding several operations together), resulting in a tremendous increase in performance as today's standard signal intervals are most commonly 5 ms, 10 ms or 100 ms.

Our platform categorises as a soft RT system and thus missing the set deadlines does not result into any serious harm. However, since this is the first step of a developing project, we addressed missed deadlines more attentively as this allows this project to develop into a hard RT system (which is very common when it comes to automotive features), once the company decides to use our porting as a real-life ECU software module.

The Yocto Project shows to be a great prospect for embedded Linux developers. Our embedded alternative was able to perform relatively close to the reference processor even though we only used generic embedded Linux images with no extra customisation, due to the lack of time. Presumably, working groups focused on the Adaptive AUTOSAR development will have the opportunity to customise the basic distributions provided by Poky to create tailored embedded Linux images to the needs of the target hardware. Extra attention should be paid to scheduling-related issues since, according to our test results, the Linux scheduler is the biggest contributor to unpredictable situations.

All these questions marks around the generic Linux scheduler and its ability to handle Pthreads consistently, along with the RT behaviour of our application and how this could be affected by external factors, the interpretation of our generated results and the concept of the Yocto Project, which is a great prospect for embedded Linux developers, constitute an enticing foundation for future researchers, both in academia and in the industry. Undoubtedly, these are very exciting times for the field of automotive, since the shift on the hardware side will eventually be followed by a transition to a multi-threaded era for ECU software applications.

# References

[1] K. Reif, Ed., *Automotive Mechatronics: Automotive Networking, Driving Stability Systems, Electronics.* Springer Vieweg, 2015.

[2] T. Cloud, *Technical Notes on The EEC-IV MCU*, Nov. 1997.

[3] “Engine control unit,” Hyundai Wiki, Feb. 2016. [Online]. Available: http://hyundai.wikia.com/wiki/Engine_Control_Unit

[4] B. S. Stefan Voget, Michael Golm and F. Stappert, *Automotive Embedded Systems Handbook.* CRC Press, 2008.

[5] “AUTOSAR development partnership,” Feb. 2016. [Online]. Available: http://www.autosar.org/

[6] S. Voget, “Autosar and the automotive tool chain,” in *Proceedings of Design, Automation & Test in Europe 2010*, Mar 2010, pp. 259 – 262.

[7] (2015) Arccore. [Online]. Available: http://www.arccore.com/

[8] (2016) Renesas - development tools. [Online]. Available: http://am.renesas.com/products/tools/index.jsp

[9] J. E. Smith and R. Nair, “The architecture of virtual machines,” *Computer*, vol. 38, no. 5, pp. 32–38, May 2005.

[10] I. Kalkov, D. Franke, J. F. Schommer, and S. Kowalewski, “A real-time extension to the android platform,” in *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, ser. JTRES ’12. ACM, 2012, pp. 105–114. [Online]. Available: http://doi.acm.org/10.1145/2388936.2388955

[11] S. Furst. (2015) Autosar the next generation – the adaptive platform. [Online]. Available: http://conf.laas.fr/cars2015/CARS-EDCC2015/CARS@EDCC2015_files/AUTOSAR_CARS@EDCC%202015.pdf

[12] (2016) Silver 3.2: Virtual ecus for rapid control development. Alt-Moabit 92, D-10559 Berlin. [Online]. Available: https://www.qtronic.de/en/silver.html

[13] (2013) Virtual autosar platform. [Online]. Available: http://www.mentorkr.com/products/automotive/event/07_Korea_Auto_Event_14-GM_v2.1.pdf

[14] (2016, Jan.) Nvidia’s deep learning car computer selected by volvo on journey toward a crash-free future. Nvidia. [Online]. Available: http://nvidianews.nvidia.com/news/nvidia-s-deep-learning-car-computer-selected-by-volvo-on-journey-toward-a-\crash-free-future

[15] (2016) Coremark - an eembc benchmark. [Online]. Available: http://www.eembc.org/coremark/index.php

[16] (2016) Nvidia tegra. [Online]. Available: http://www.nvidia.com/object/tegra-x1-processor.html

[17] P. H. Feiler. (2003, November) Real-time application development with osek a review of the osek standards. [Online]. Available: http://www.sei.cmu.edu/reports/03tn004.pdf

[18] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook.* William Pollock, Oct. 2010.

[19] S. Rifenbark, *Yocto Project Mega-Manual*, March 2016. [Online]. Available: http://www.yoctoproject.org/docs/2.0.1/mega-manual/mega-manual.html

[20] (2016) Minnowboard max - minnowboard wiki. [Online]. Available: http://wiki.minnowboard.org/MinnowBoard_MAX

[21] A. Holt and C.-Y. Huang, *Embedded operating systems.* Springer, 2014.

[22] W. Stallings, *Operating Systems - Internals and Design Principles*, 7th ed. Prentice Hall, 2011.

[23] M. v. S. Andrew S. Tanenbaum, *Distributed Systems: Principles and Paradigms*, 2nd ed. Pearson Education, 2014.

[24] B. Barney. POSIX Threads Programming. Lawrence Livermore National Laboratory. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads/

[25] M. Kerrisk, *The Linux man-pages project*, Linux Foundation. [Online]. Available: https://www.kernel.org/doc/man-pages/

[26] A. S. Tanenbaum, *Modern Operating System*, 3rd ed. Pearson Prentice Hall, 2009.

[27] M. L. Scott, *Encyclopedia of Parallel Computing*, D. Padua, Ed. Boston, MA: Springer US, 2011. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-09766-4_252

[28] X. Fan, *Real-Time Embedded Systems - Design Principles and Engineering Practices.* Oxford: Newnes, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128015070000018

[29] M. Barr and A. Massa, *Programming Embedded Systems: With C and GNU Development Tools*, 2nd ed. O'Reilly Media, Inc., 2006.

[30] F. Krief, *Communicating Embedded Systems: Networks Applications*, ser. ISTE. Wiley, 2013.

[31] I. C. Bertolotti and G. Manduchi, *Real-Time Embedded systems*, R. Zurawski, Ed. CRC Press - Taylor & Francis Group, 2012.

[32] C. Hallinan, *Embedded Linux Primer*, ser. Prentice Hall Open Source Software Development Series. Pearson Education Inc., 2007.

[33] K. Yaghmour, J. Masters, G. Ben-Yossef, and P. Gerum, *Building Embedded Linux Systems.* O'Reilly Media, 2008.

[34] B. Harwani, *Unix and Shell Programming.* Oxford University Press, 2013. [Online]. Available: http://app.knovel.com/hotlink/toc/id:kpUSP00005/unix-shell-programming/unix-shell-programming

[35] P. H. Salus, *A Quarter Century of UNIX.* Addison-Wesley Professional, 1994.

[36] R. K. Michael, *Mastering UNIX Shell Scripting.* Wiley Publishing, 2008.

[37] (2015) Autosar layered software architecture. [Online]. Available: http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/general/auxiliary/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf

[38] Autosar: Layered software architecture. [Online]. Available: http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/general/auxiliary/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf

[39] (2014) AUTOSAR layered software architecture. [Online]. Available: http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/

[40] G. Andersson, *GENIVI Alliance Reference Architecture*, October 2015. [Online]. Available: http://www.genivi.org/sites/default/files/resource_documents/GENIVI_Reference_Architecture_29Oct2015.pdf

[41] T. Wollstadt and et al., *OSEK/VDX - Operating System Specification*, Feb. 2005. [Online]. Available: portal.osek-vdx.org/files/pdf/specs/os223.pdf

[42] (2016) Yocto project - about. [Online]. Available: https://www.yoctoproject.org/about

[43] (2016) The linux foundation. [Online]. Available: http://www.linuxfoundation.org/

[44] Rational unified process. [Online]. Available: https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf

[45] P. Kruchten, *Rational Unified Process, The: An Introduction*, 3rd ed. Addison Wesley, Dec. 2003.

[46] G. B. James Rumbaugh, Ivar Jacobson, *The Unified Modeling Language Reference Manual*, 2nd ed. Addison Wesley, Jul. 2004.

[47] M. McCormick, "Waterfall vs. agile methodology," 2012. [Online]. Available: http://www.mccormickpcs.com/images/Waterfall_vs_Agile_Methodology.pdf

[48] Eclipse cdt. [Online]. Available: https://eclipse.org/cdt/

[49] *make(1) - Linux man page.* [Online]. Available: http://linux.die.net/man/1/make

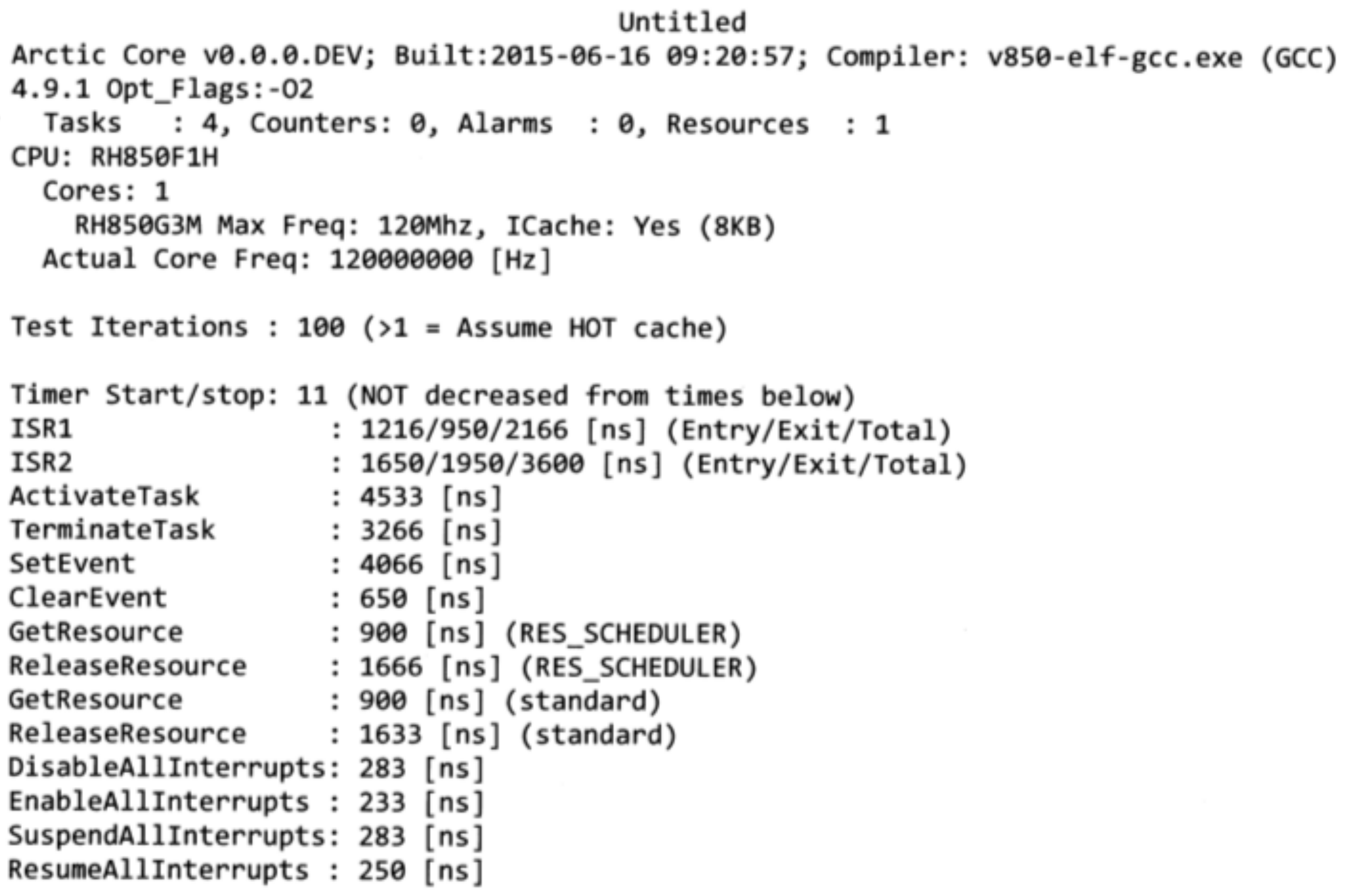
[50] D. M. R. Brian W. Kernighan, *The C Programming Language*, 2nd ed. PRENTICE HALL, 1988.

[51] (2011) Chronos real-time linux – about. [Online]. Available: http://chronoslinux.org/wiki/About_ChronOS_Linux

[52] Minnow board. [Online]. Available: http://wiki.minnowboard.org/MinnowBoard_Wiki_Home

[53] J.-P. L. et. al. (2016, April) The linux scheduler: a decade of wasted cores. [Online]. Available: https://www.ece.ubc.ca/~sasha/papers/eurosys16-final29.pdf

# A
# Reference Performance Results



```
                                 Untitled
Arctic Core v0.0.0.DEV; Built:2015-06-16 09:20:57; Compiler: v850-elf-gcc.exe (GCC)
4.9.1 Opt_Flags:-O2
  Tasks   : 4, Counters: 0, Alarms  : 0, Resources  : 1
CPU: RH850F1H
  Cores: 1
    RH850G3M Max Freq: 120Mhz, ICache: Yes (8KB)
  Actual Core Freq: 120000000 [Hz]

Test Iterations : 100 (>1 = Assume HOT cache)

Timer Start/stop: 11 (NOT decreased from times below)
ISR1                : 1216/950/2166 [ns] (Entry/Exit/Total)
ISR2                : 1650/1950/3600 [ns] (Entry/Exit/Total)
ActivateTask        : 4533 [ns]
TerminateTask       : 3266 [ns]
SetEvent            : 4066 [ns]
ClearEvent          : 650 [ns]
GetResource         : 900 [ns] (RES_SCHEDULER)
ReleaseResource     : 1666 [ns] (RES_SCHEDULER)
GetResource         : 900 [ns] (standard)
ReleaseResource     : 1633 [ns] (standard)
DisableAllInterrupts: 283 [ns]
EnableAllInterrupts : 233 [ns]
SuspendAllInterrupts: 283 [ns]
ResumeAllInterrupts : 250 [ns]
```

# B
# CoreMark Benchmark Results

Intel Core i7

```
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 28166
Total time (secs): 28.166000
Iterations/Sec   : 85209.117376
Iterations       : 2400000
Compiler version : GCC4.8.4
Compiler flags   : -O2 -DMULTITHREAD=8 -DUSE_PTHREAD -DPERFORMANCE_RUN=1
-lrt -lpthread-2.19
Parallel PThreads : 8
Correct operation validated. See readme.txt for run and reporting rules.
CoreMark 1.0 : 85209.117376 / GCC4.8.4 -O2 -DMULTITHREAD=8 -DUSE_PTHREAD
-DPERFORMANCE_RUN=1  -lrt -lpthread-2.19 / Heap / 8:PThreads
```

Intel Core i5

```
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 17359
Total time (secs): 17.359000
Iterations/Sec   : 46085.604009
Iterations       : 800000
Compiler version : GCC4.8.4
Compiler flags   : -O2 -DMULTITHREAD=4 -DUSE_PTHREAD -DPERFORMANCE_RUN=1
-lrt -lpthread-2.19
Parallel PThreads : 4
Correct operation validated. See readme.txt for run and reporting rules.
CoreMark 1.0 : 46085.604009 / GCC4.8.4 -O2 -DMULTITHREAD=4 -DUSE_PTHREAD
-DPERFORMANCE_RUN=1  -lrt -lpthread-2.19 / Heap / 4:PThreads
```

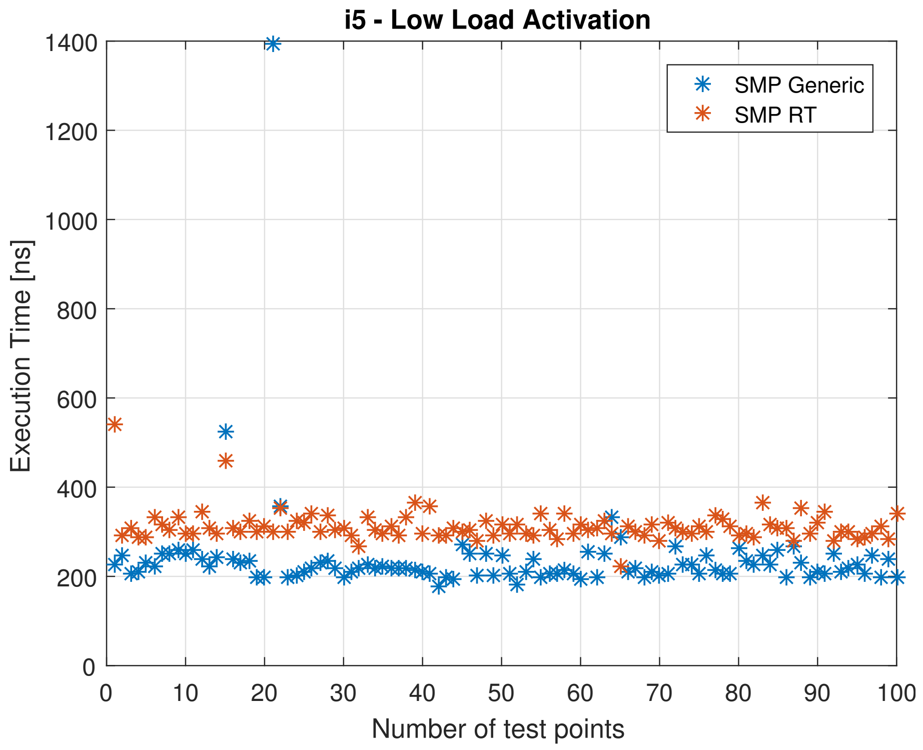
# C

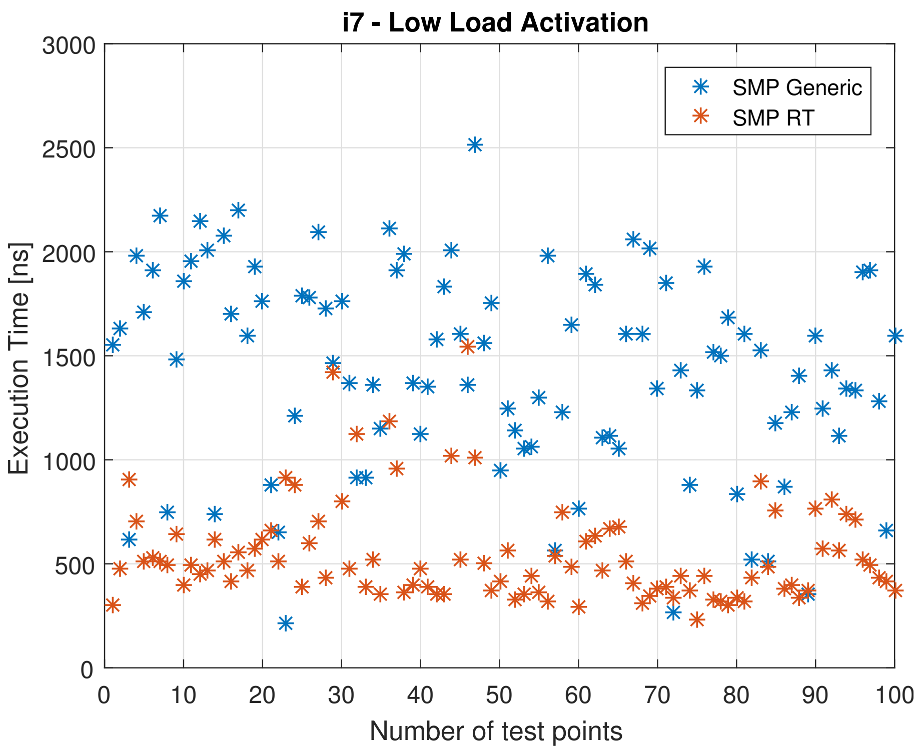
# Performance Results under Load

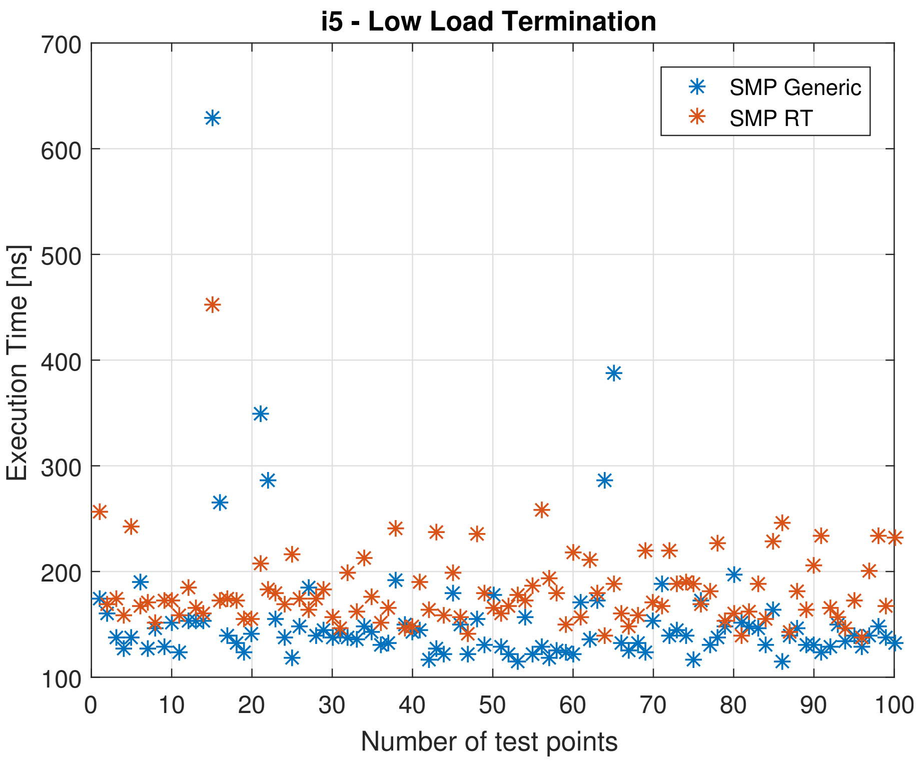
The remaining graphs from the test results. The CPU was loaded both with roughly 50% load and 100% load.

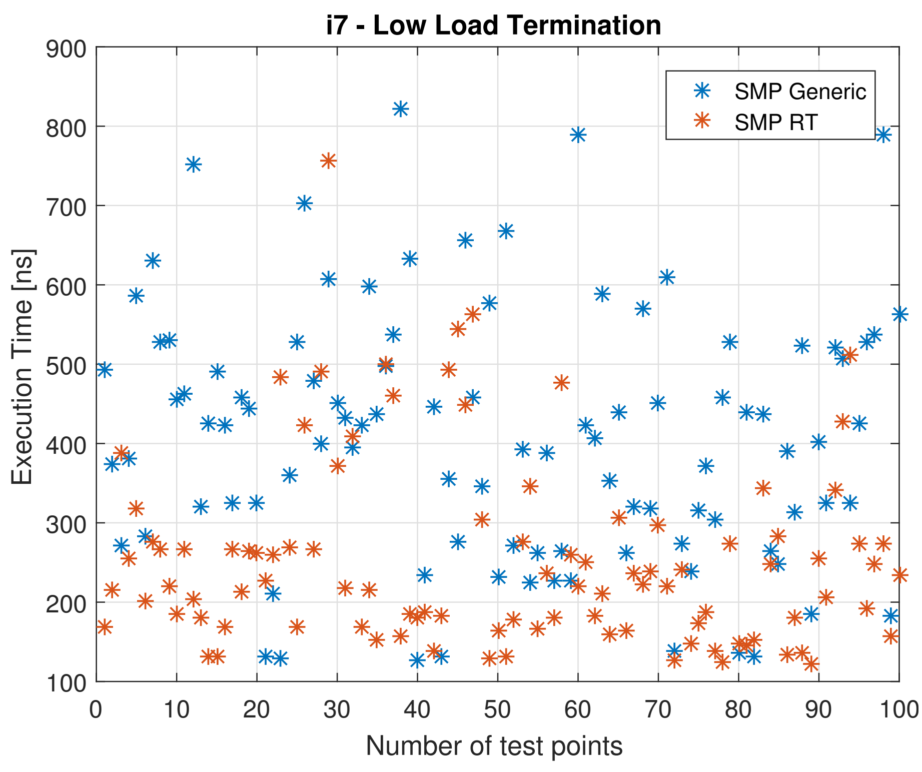
Iteration 3 will be displayed first, then iteration 4. They are ordered so that each metric can be compared between the two host machines.

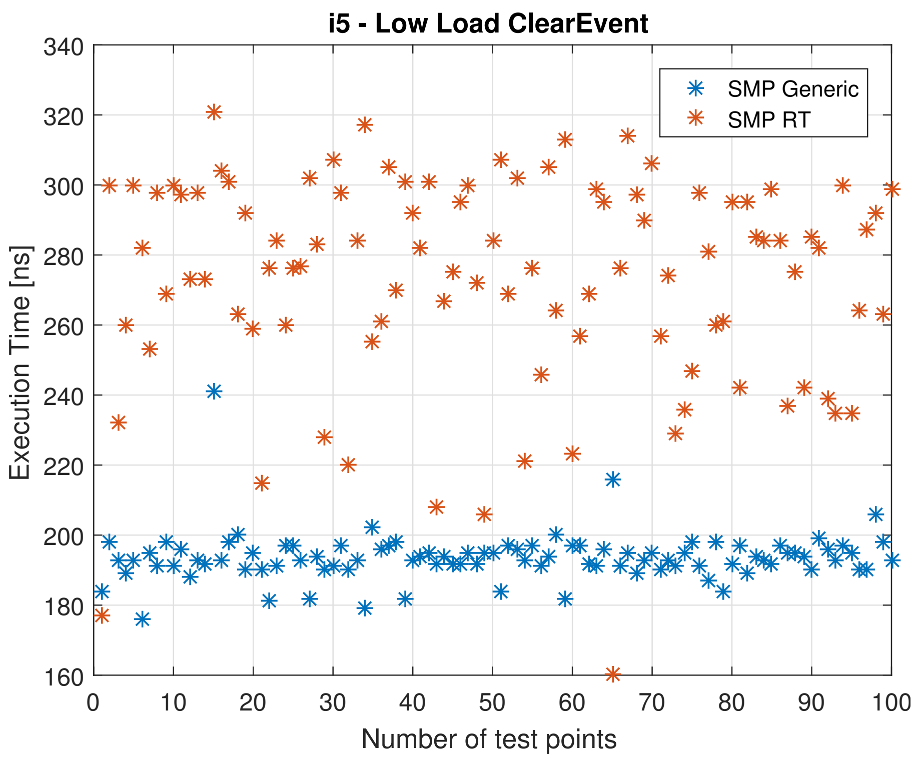
## C.1 Iteration 3

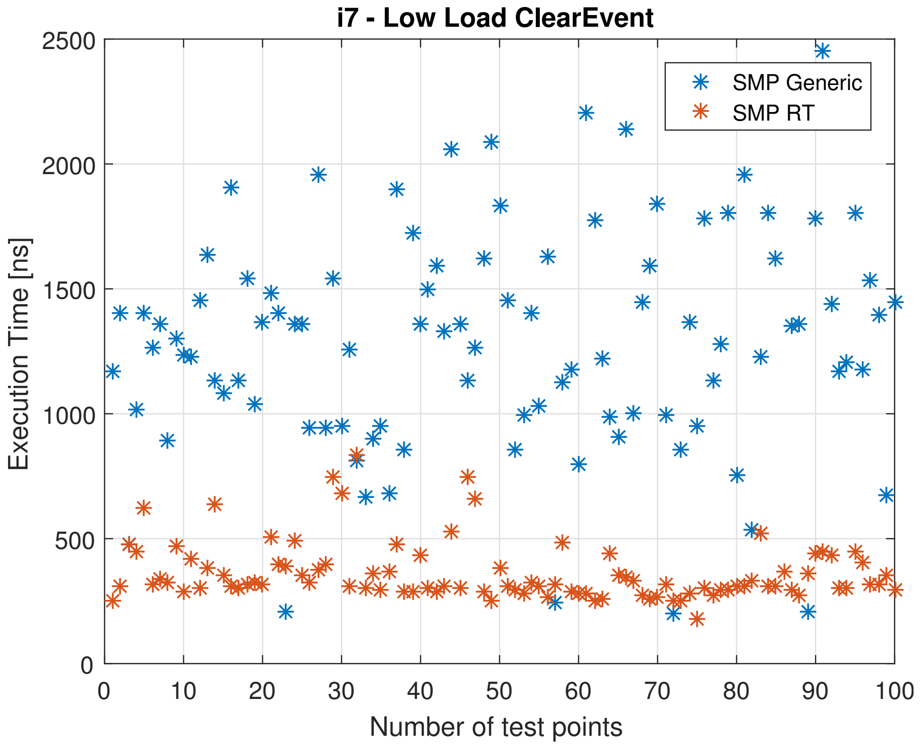


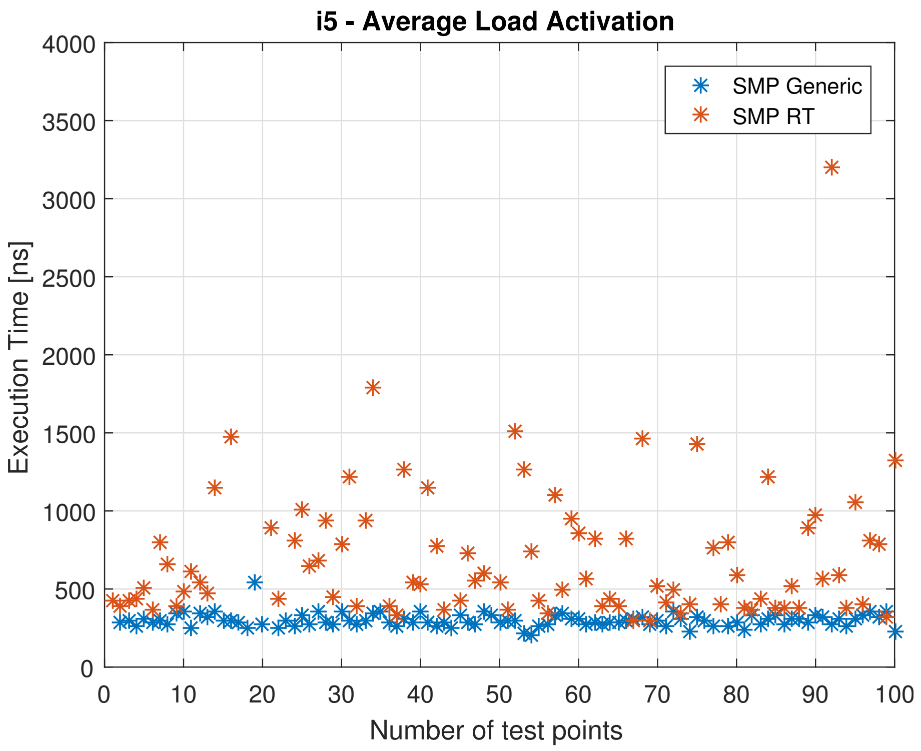

i7 - Low Load Activation
SMP Generic
SMP RT
Execution Time [ns]
Number of test points
3000
2500
2000
1500
1000
500
0
0
10
20
30
40
50
60
70
80
90
100

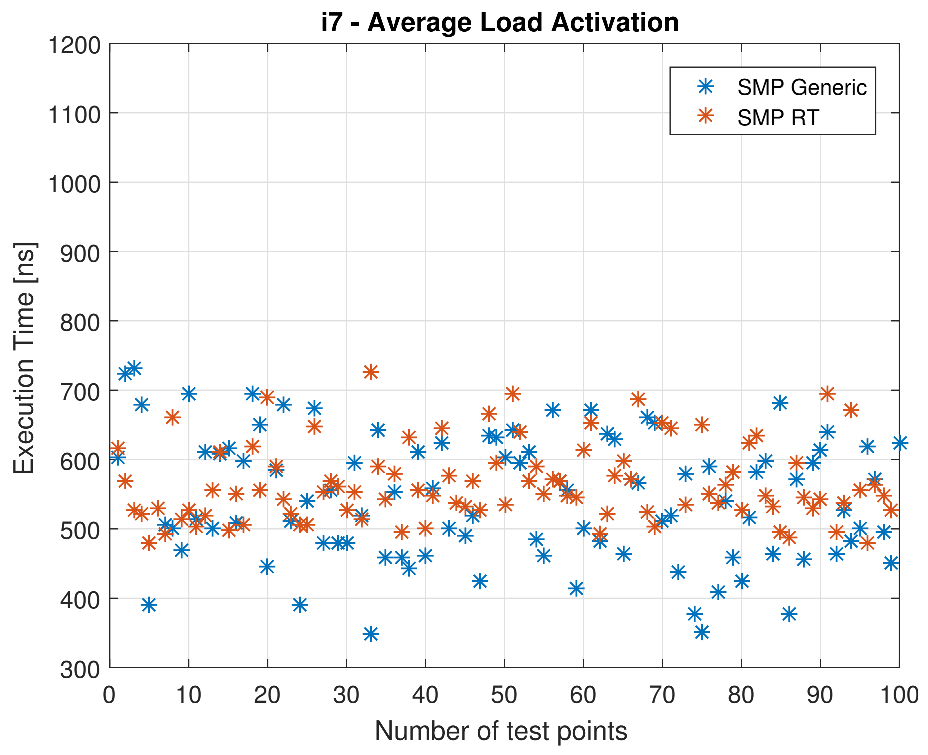

i5 - Low Load Termination
SMP Generic
SMP RT
Execution Time [ns]
Number of test points
700
600
500
400
300
200
100
0
10
20
30
40
50
60
70
80
90
100

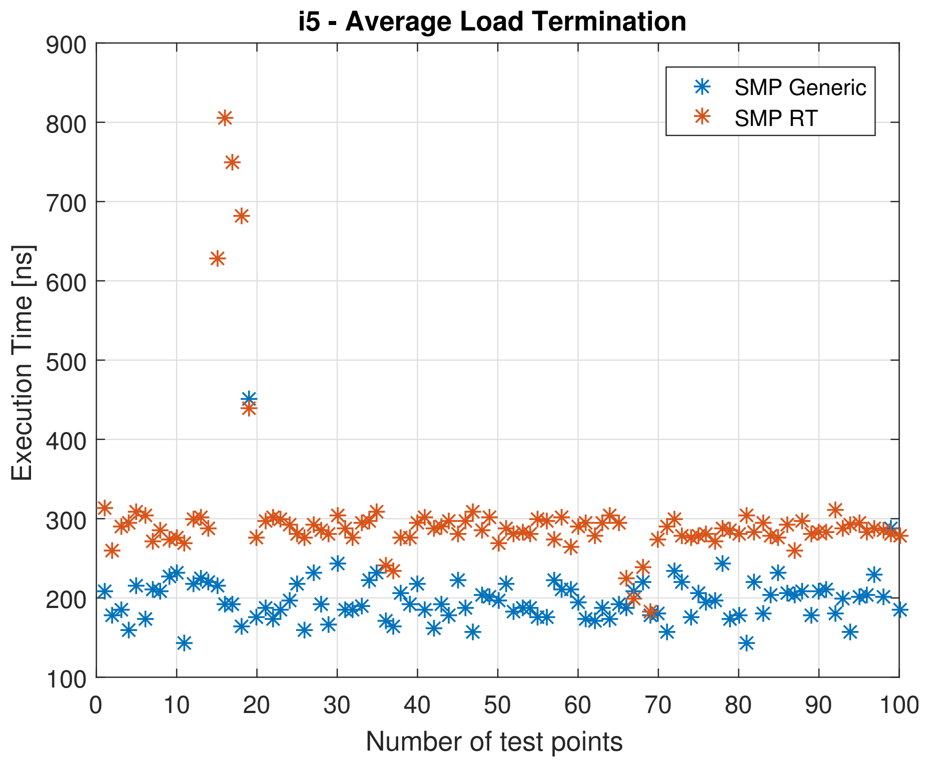

i7 - Low Load Termination
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
100
200
300
400
500
600
700
800
900
0
10
20
30
40
50
60
70
80
90
100

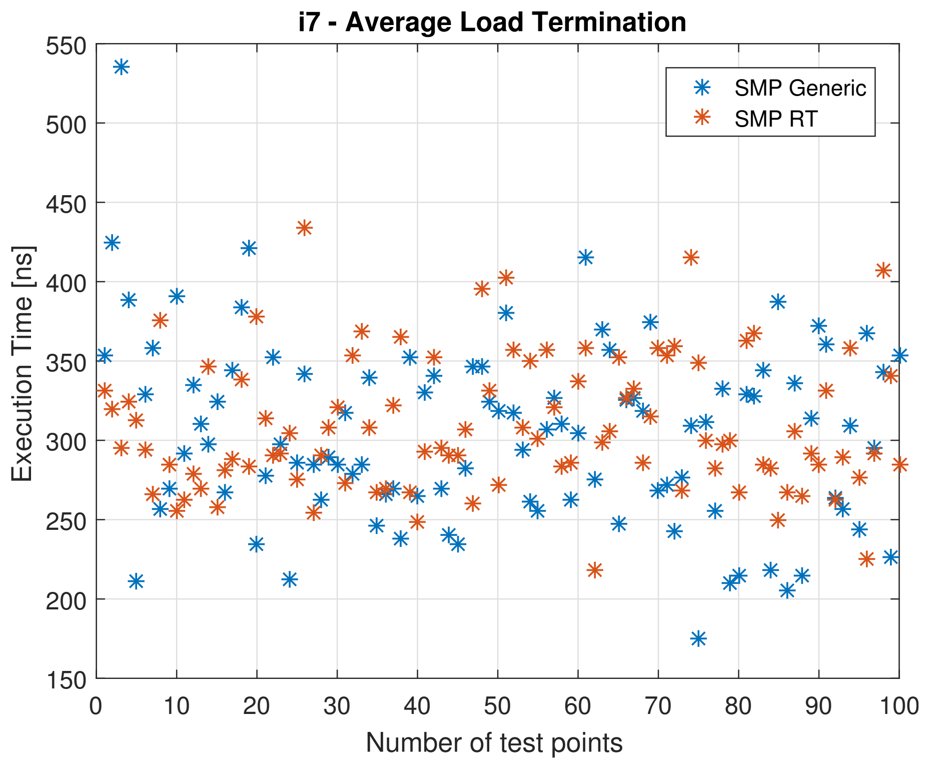

i5 - Low Load ClearEvent
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
160
180
200
220
240
260
280
300
320
340
0
10
20
30
40
50
60
70
80
90
100

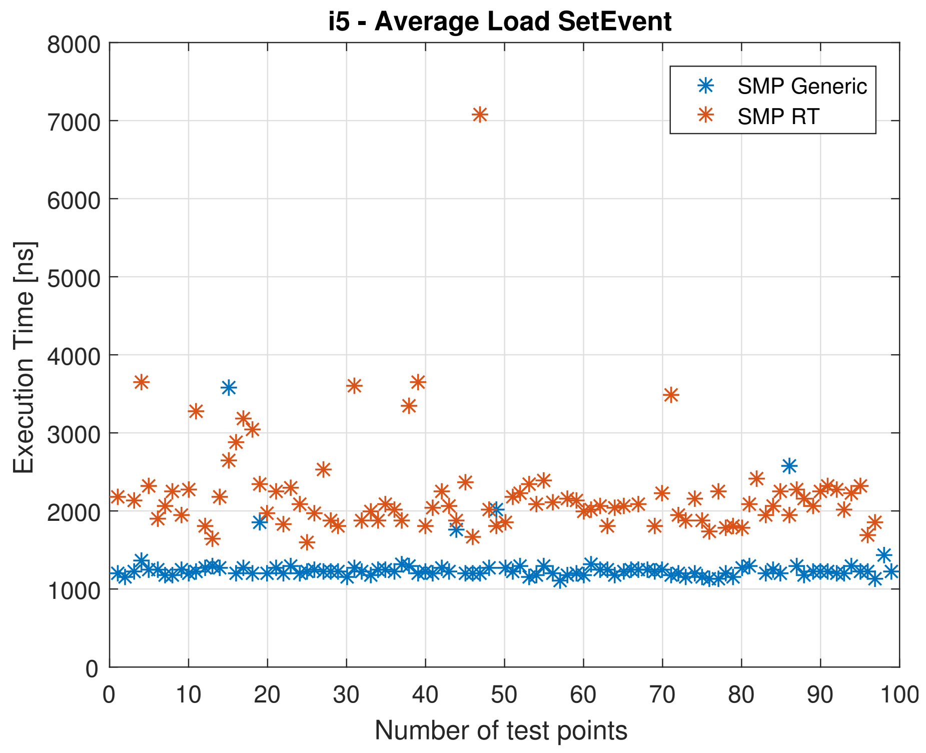

i7 - Low Load ClearEvent
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
0
500
1000
1500
2000
2500
0
10
20
30
40
50
60
70
80
90
100

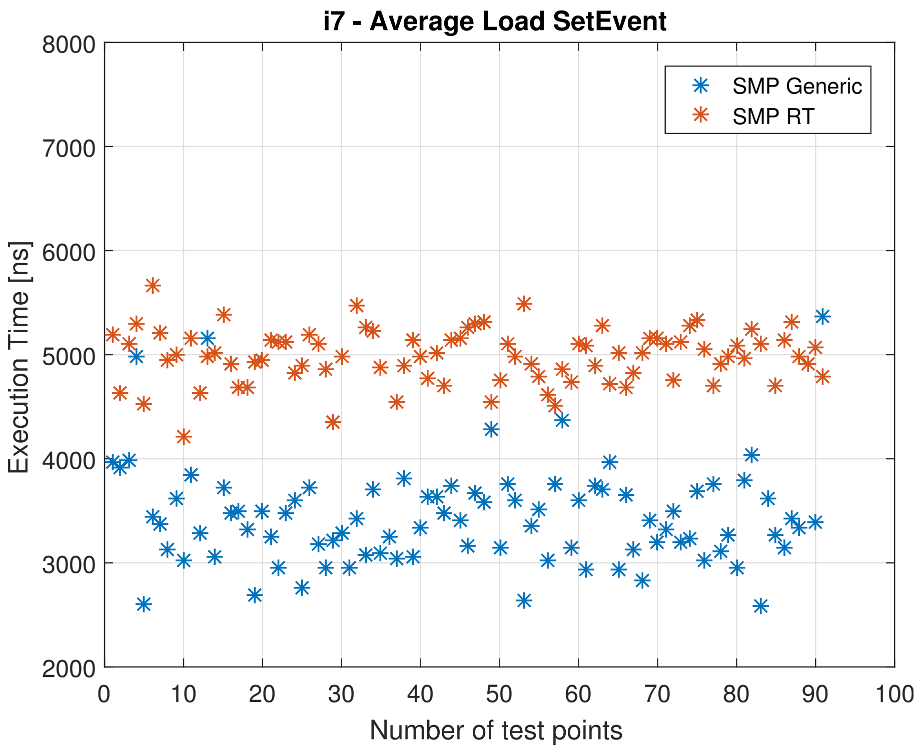

i5 - Average Load Activation
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
0
500
1000
1500
2000
2500
3000
3500
4000
0
10
20
30
40
50
60
70
80
90
100

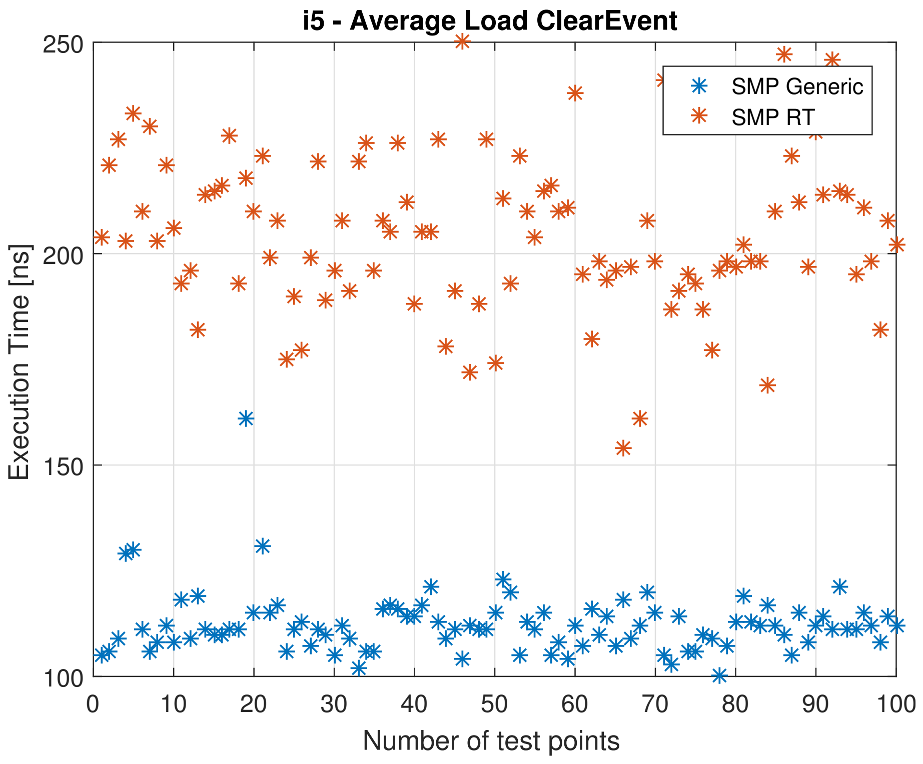

i7 - Average Load Activation
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
1200
1100
1000
900
800
700
600
500
400
300
0
10
20
30
40
50
60
70
80
90
100

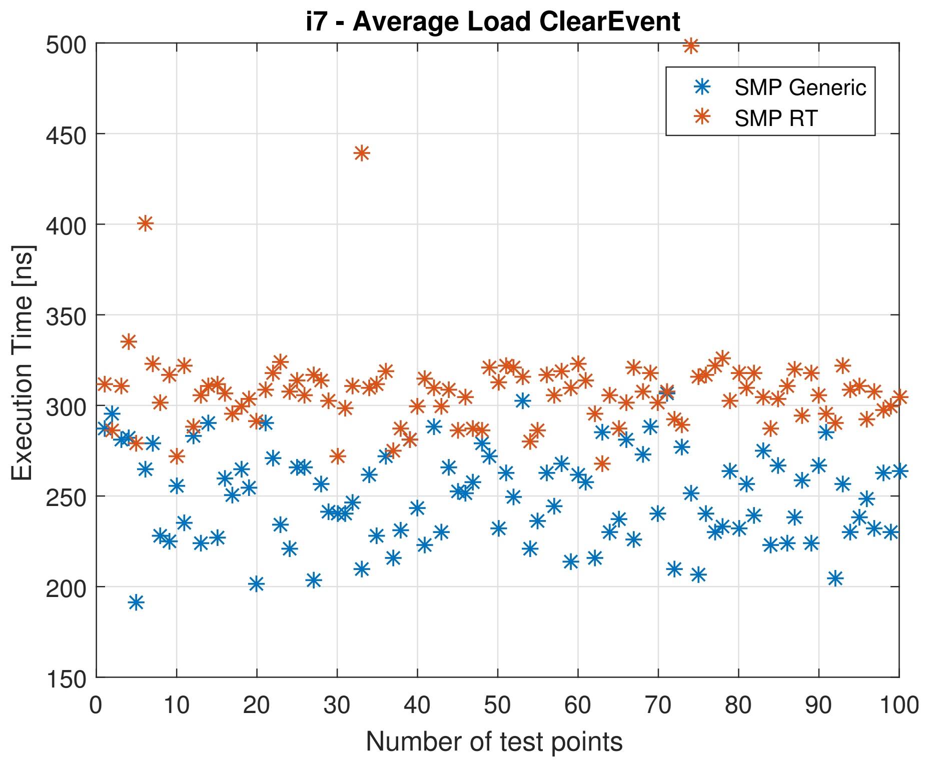

i5 - Average Load Termination
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
900
800
700
600
500
400
300
200
100
0
10
20
30
40
50
60
70
80
90
100

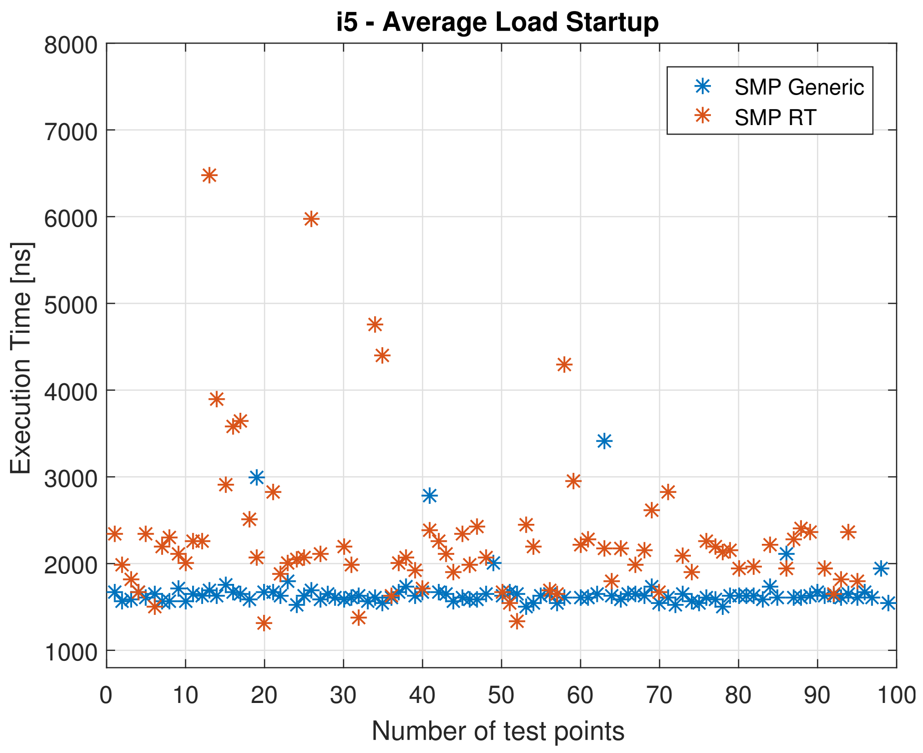

i7 - Average Load Termination
Execution Time [ns]
550
500
450
400
350
300
250
200
150
0
10
20
30
40
50
60
70
80
90
100
Number of test points
SMP Generic
SMP RT

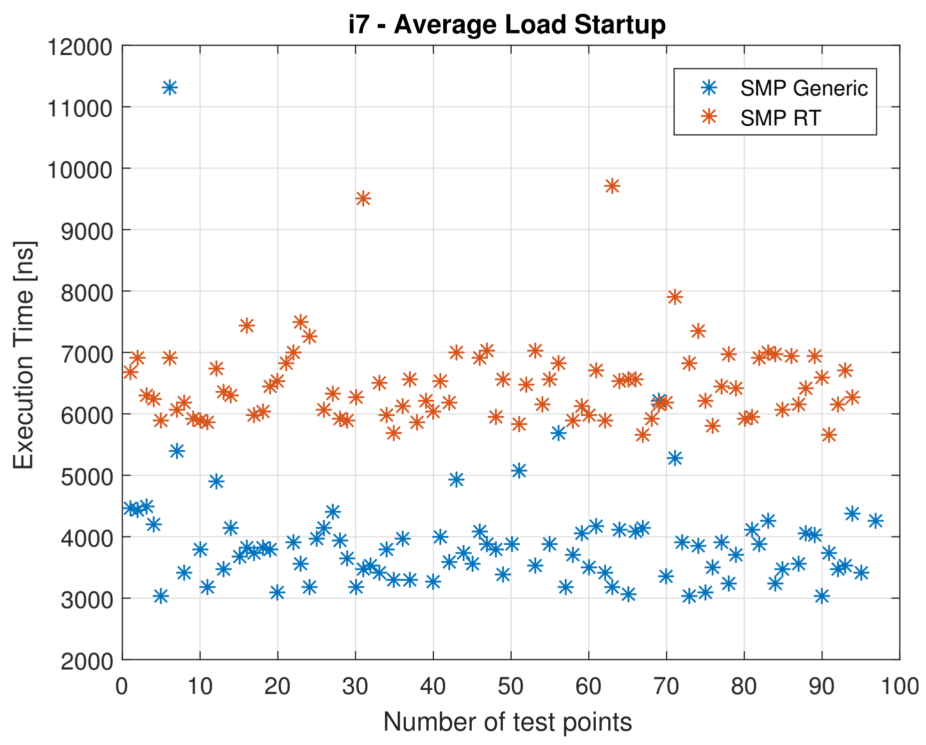

i5 - Average Load SetEvent
Execution Time [ns]
8000
7000
6000
5000
4000
3000
2000
1000
0
0
10
20
30
40
50
60
70
80
90
100
Number of test points
SMP Generic
SMP RT

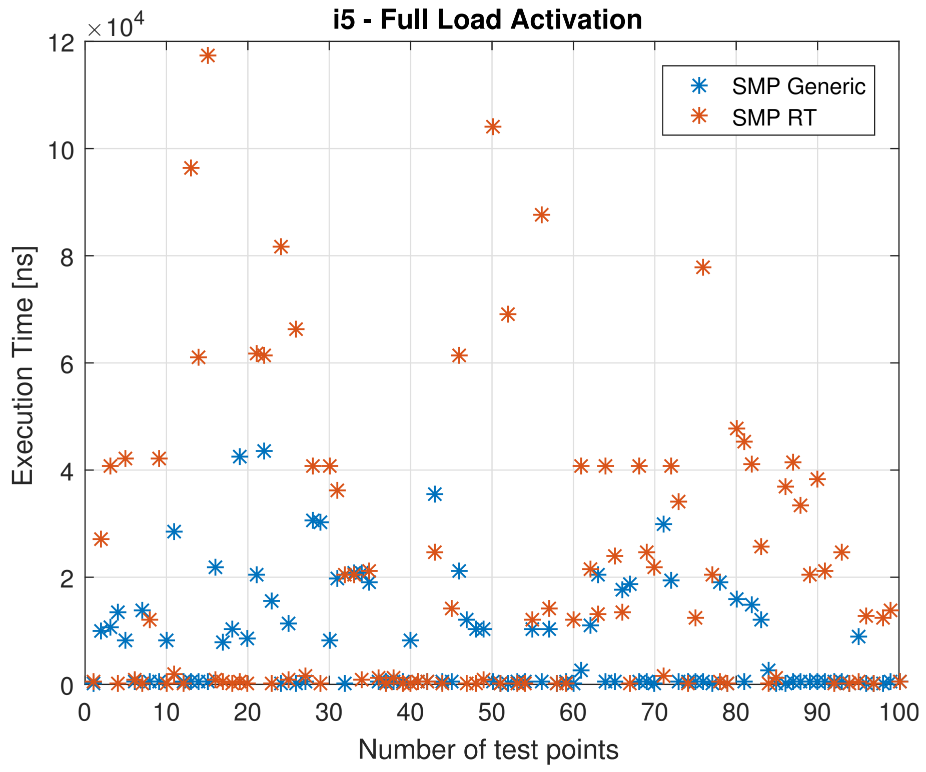

i7 - Average Load SetEvent
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
8000
7000
6000
5000
4000
3000
2000
0
10
20
30
40
50
60
70
80
90
100

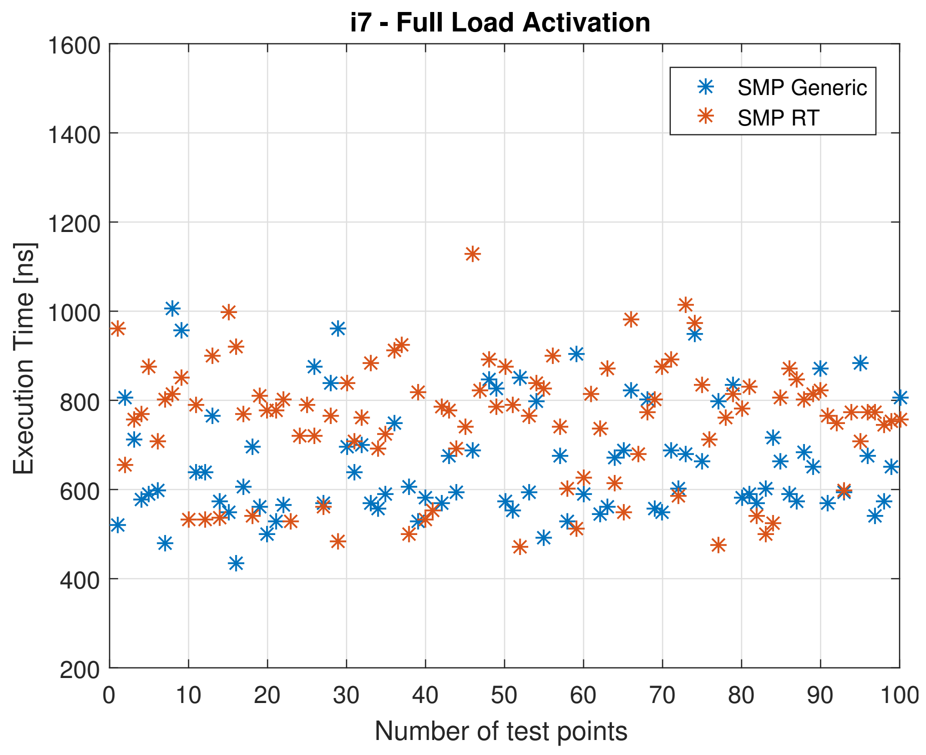

i5 - Average Load ClearEvent
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
250
200
150
100
0
10
20
30
40
50
60
70
80
90
100

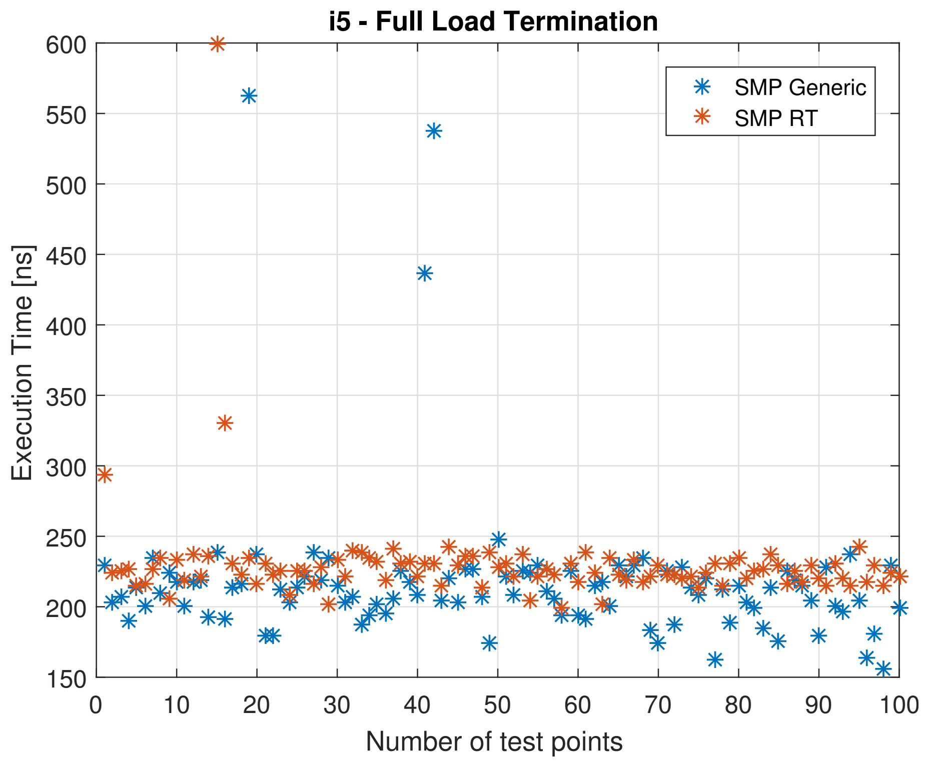

i7 - Average Load ClearEvent
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
500
450
400
350
300
250
200
150
0
10
20
30
40
50
60
70
80
90
100

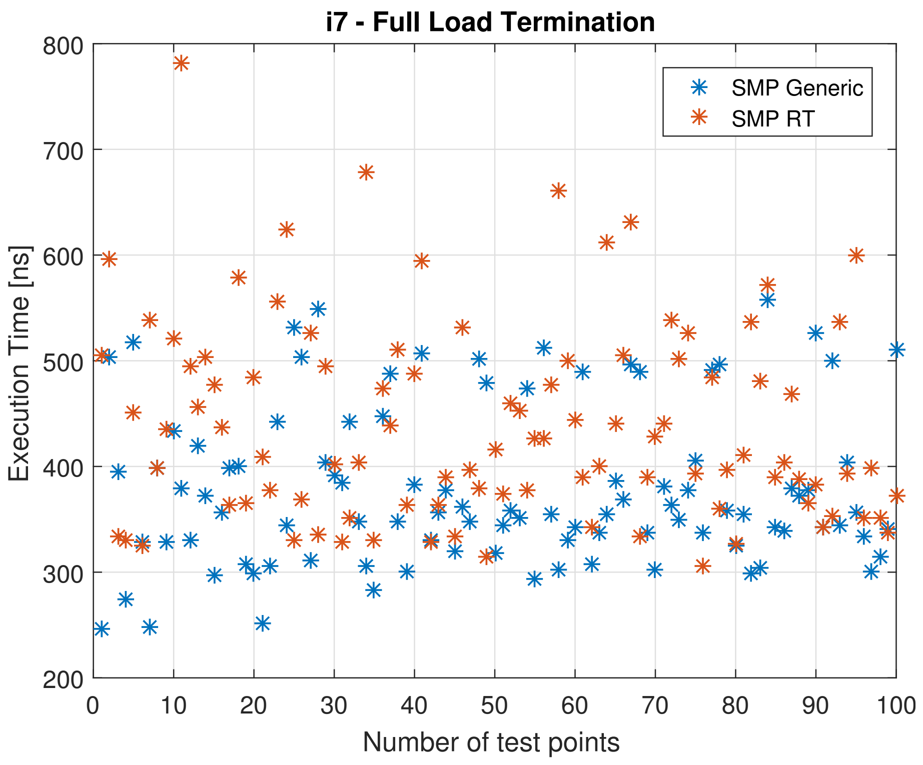

i5 - Average Load Startup
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
8000
7000
6000
5000
4000
3000
2000
1000
0
10
20
30
40
50
60
70
80
90
100

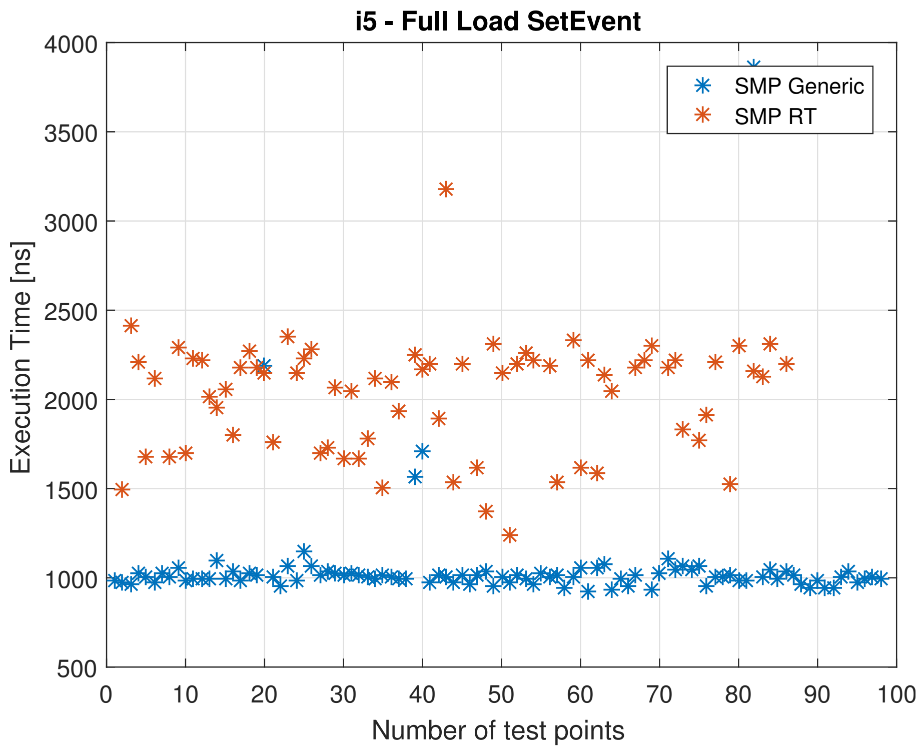

i7 - Average Load Startup
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
12000
11000
10000
9000
8000
7000
6000
5000
4000
3000
2000
0
10
20
30
40
50
60
70
80
90
100

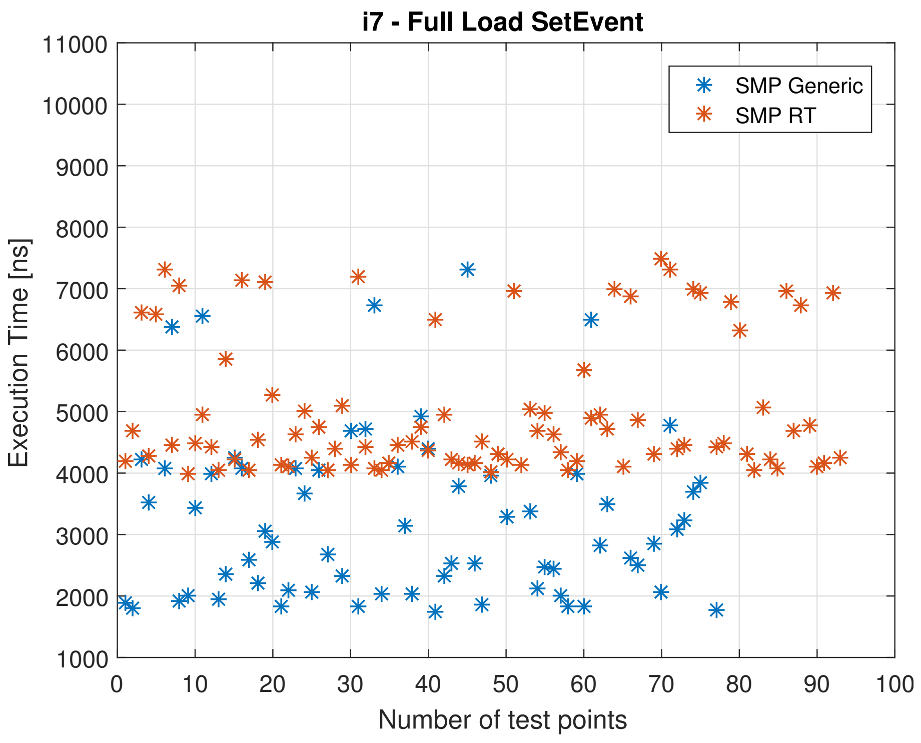

i5 - Full Load Activation
$\times 10^4$
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
12
10
8
6
4
2
0
0
10
20
30
40
50
60
70
80
90
100

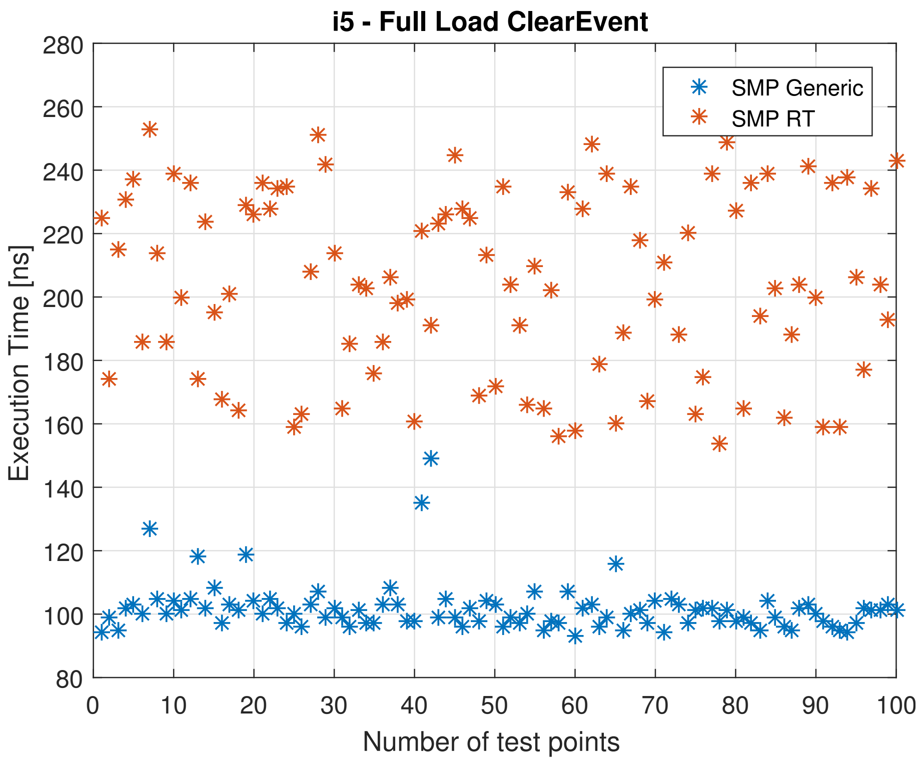

i7 - Full Load Activation
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
1600
1400
1200
1000
800
600
400
200
0
10
20
30
40
50
60
70
80
90
100

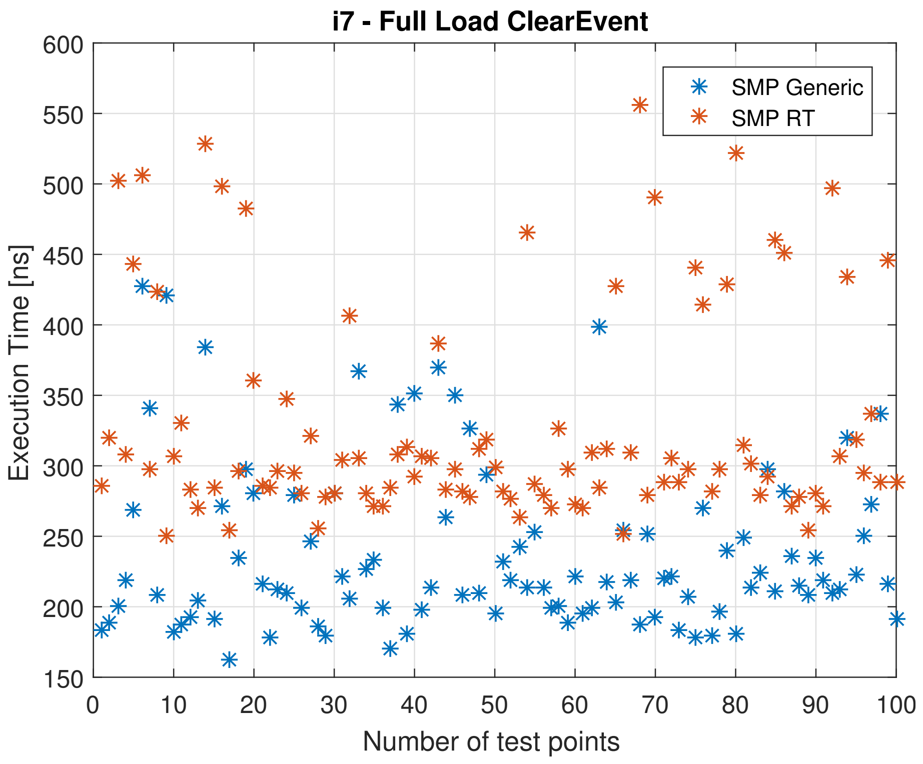

i5 - Full Load Termination
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
600
550
500
450
400
350
300
250
200
150
0
10
20
30
40
50
60
70
80
90
100

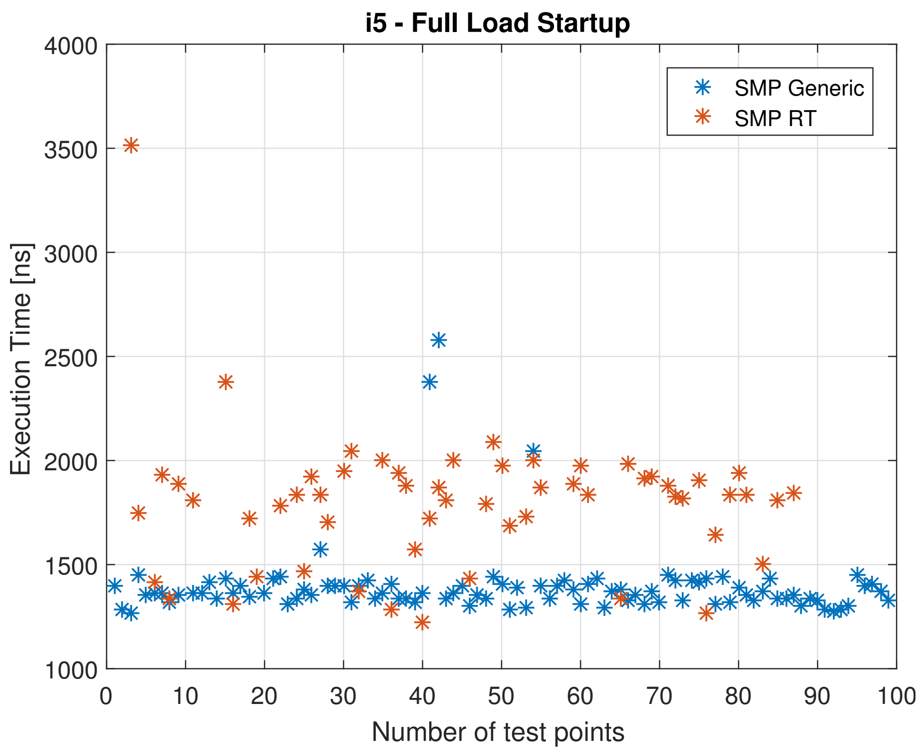

i7 - Full Load Termination
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
800
700
600
500
400
300
200
0
10
20
30
40
50
60
70
80
90
100

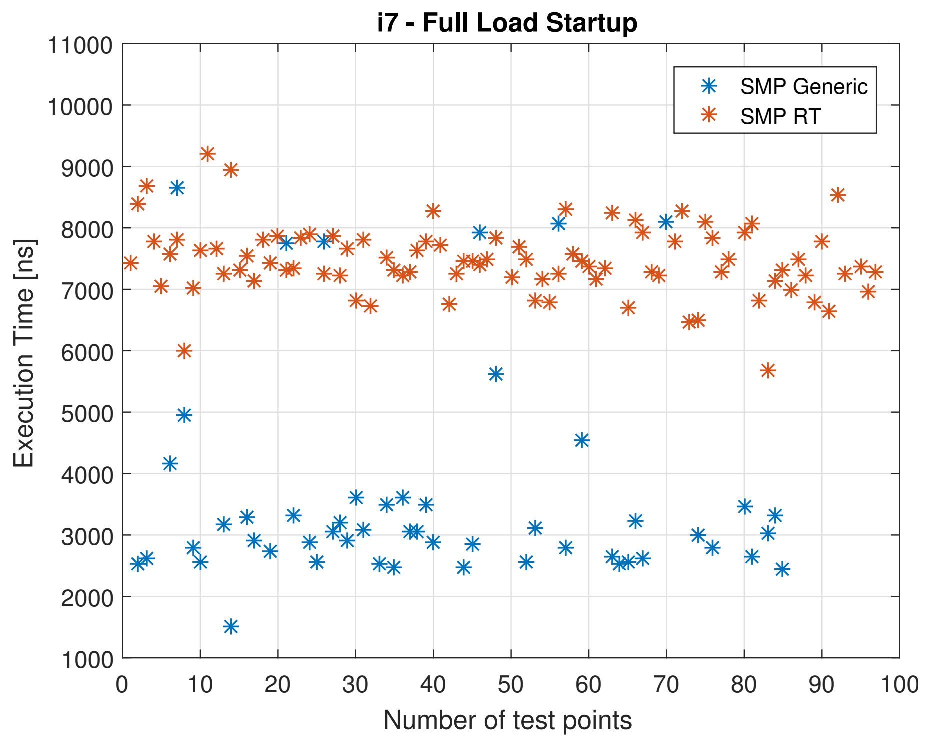

i5 - Full Load SetEvent
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
4000
3500
3000
2500
2000
1500
1000
500
0
10
20
30
40
50
60
70
80
90
100


i7 - Full Load SetEvent
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
11000
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
10
20
30
40
50
60
70
80
90
100

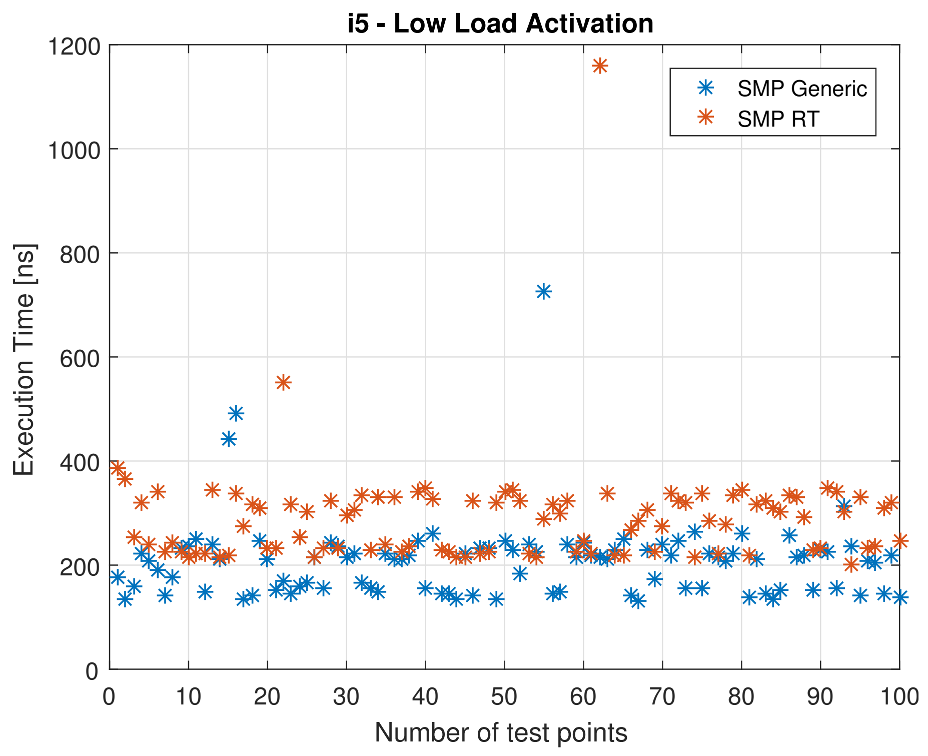

i5 - Full Load ClearEvent
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
280
260
240
220
200
180
160
140
120
100
80
0
10
20
30
40
50
60
70
80
90
100

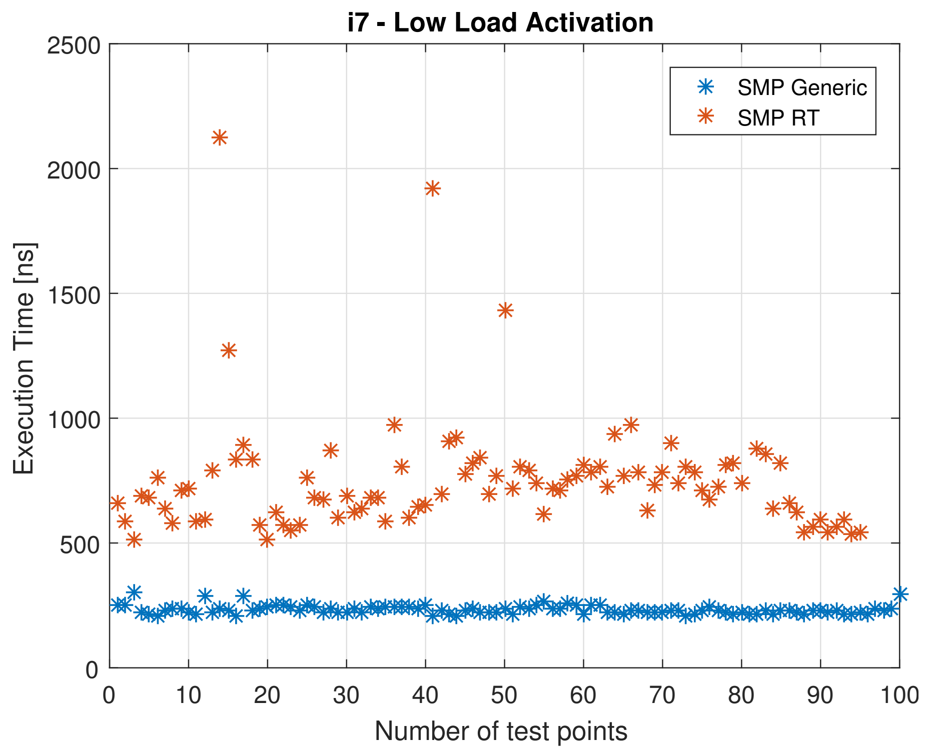

i7 - Full Load ClearEvent
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
600
550
500
450
400
350
300
250
200
150
0
10
20
30
40
50
60
70
80
90
100

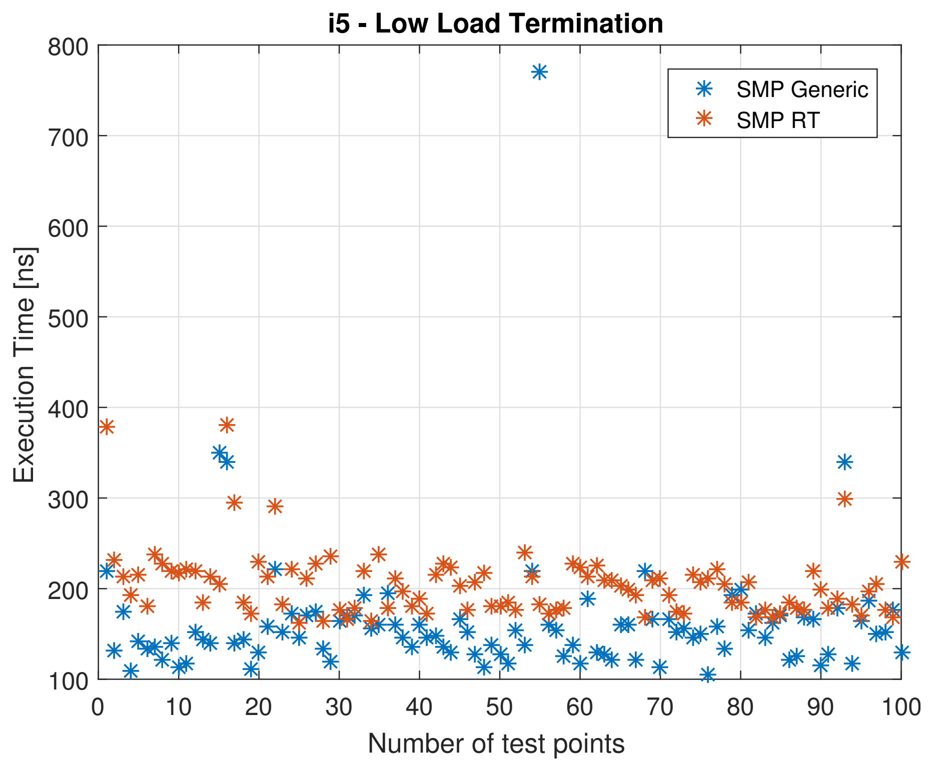

i5 - Full Load Startup
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
4000
3500
3000
2500
2000
1500
1000
0
10
20
30
40
50
60
70
80
90
100

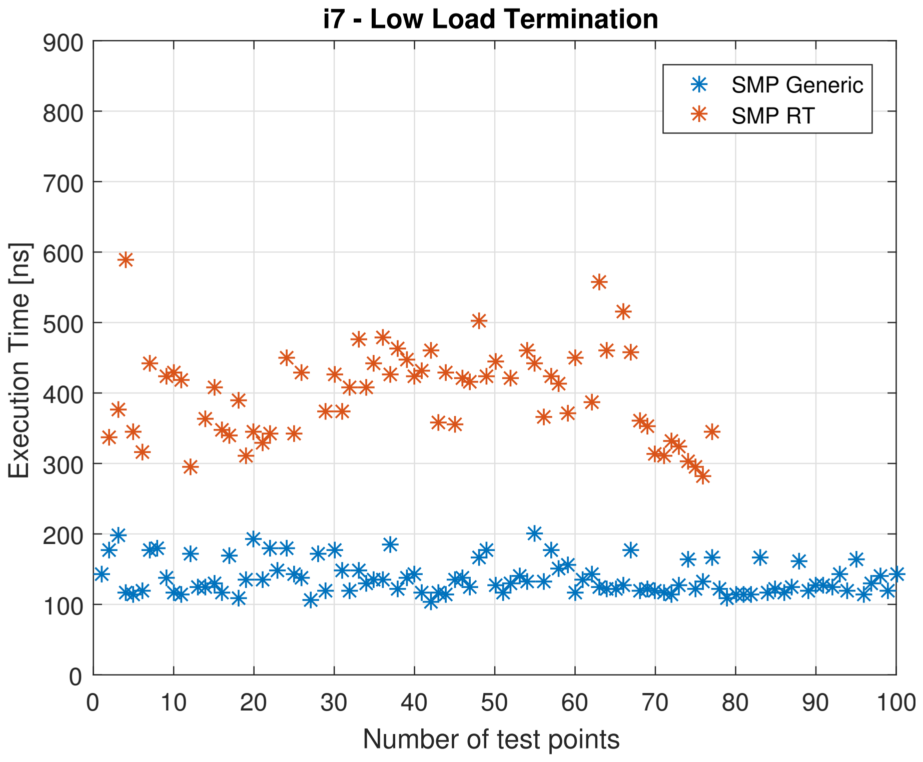

i7 - Full Load Startup
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
11000
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
10
20
30
40
50
60
70
80
90
100

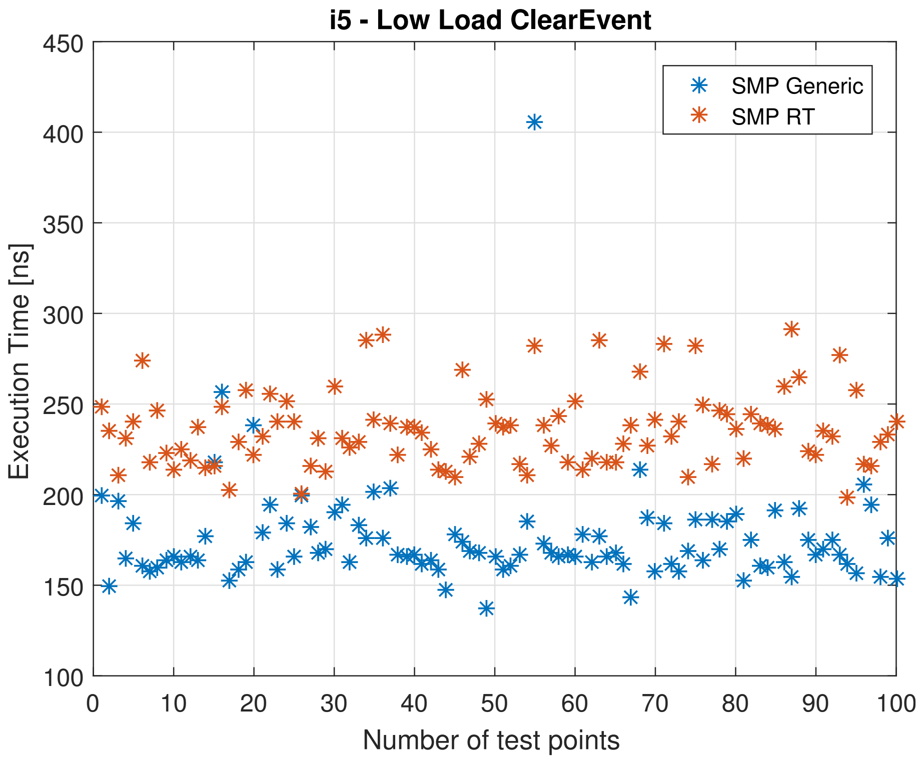
## C.2 Iteration 4

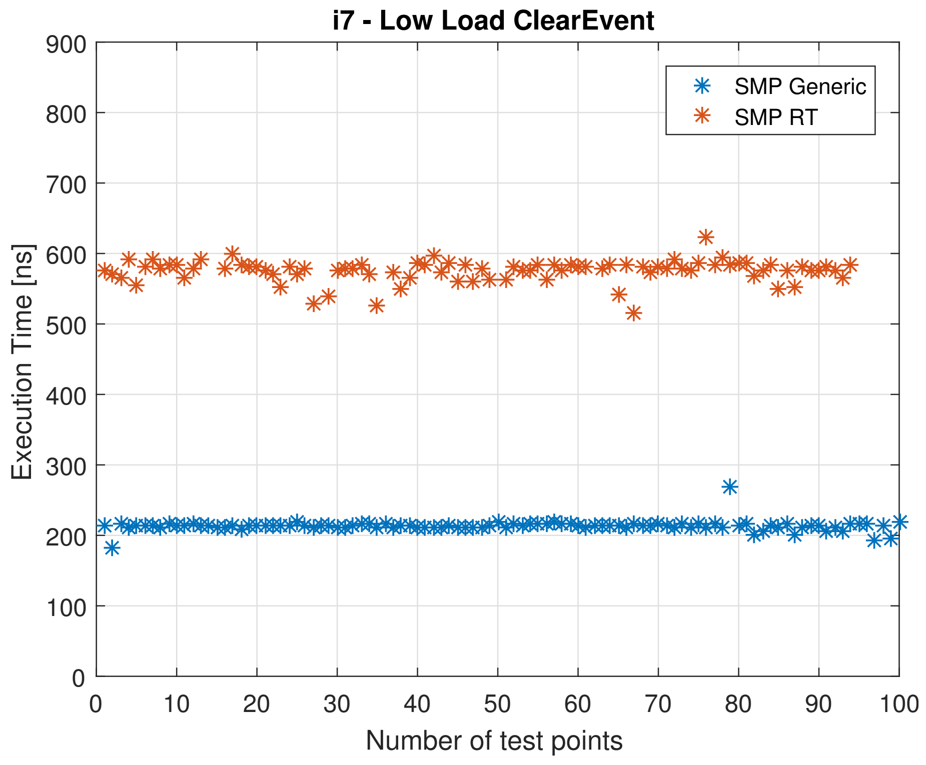


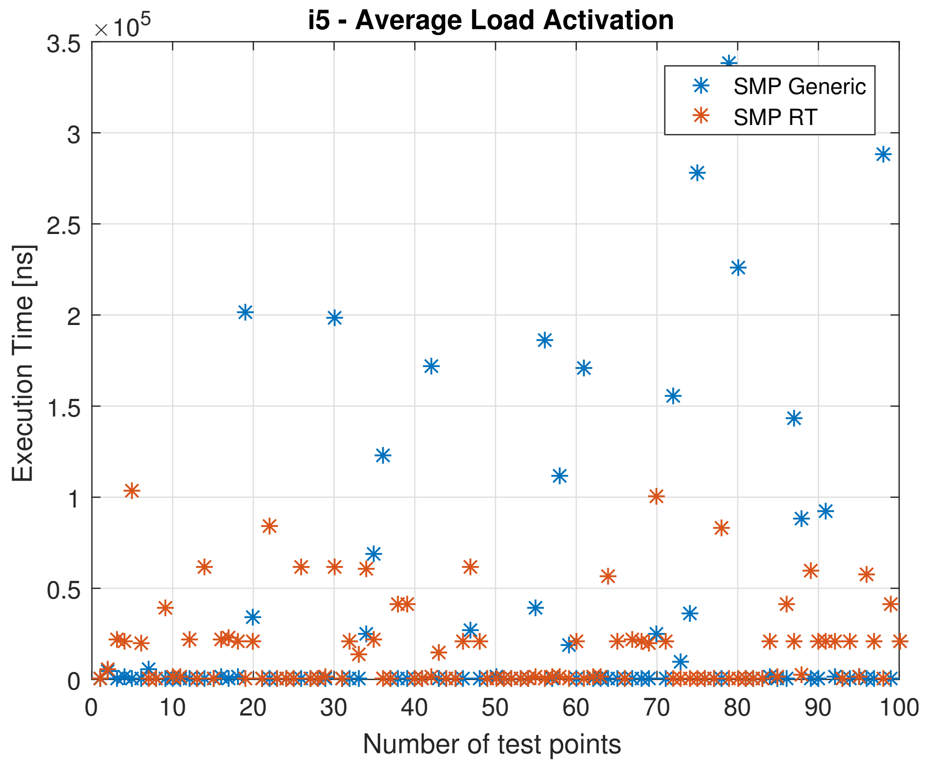

i7 - Low Load Activation
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
0
500
1000
1500
2000
2500
0
10
20
30
40
50
60
70
80
90
100

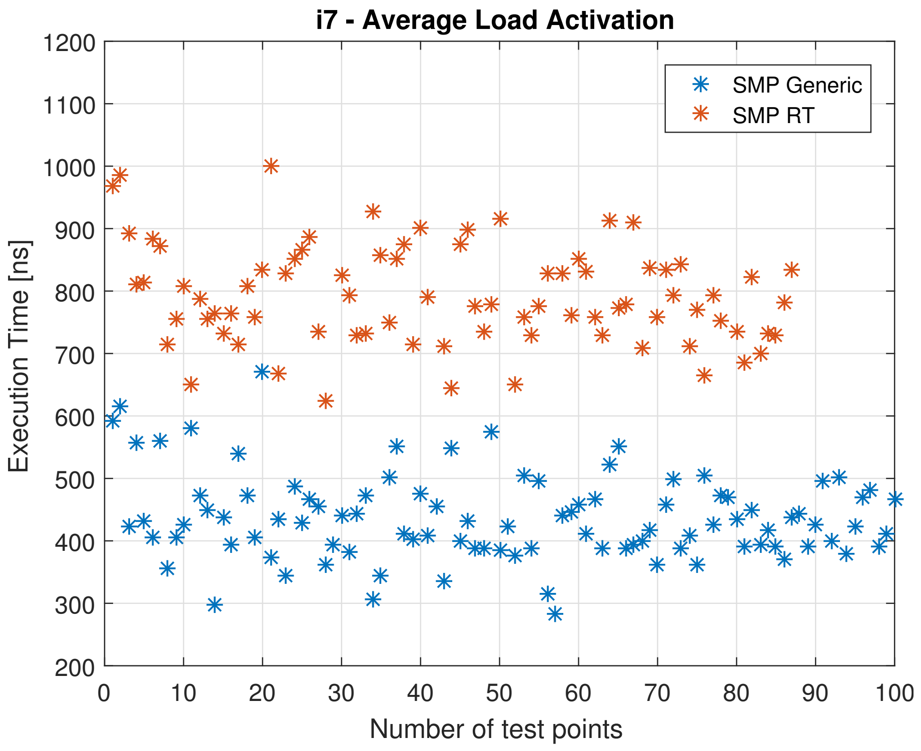

i5 - Low Load Termination
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
100
200
300
400
500
600
700
800
0
10
20
30
40
50
60
70
80
90
100

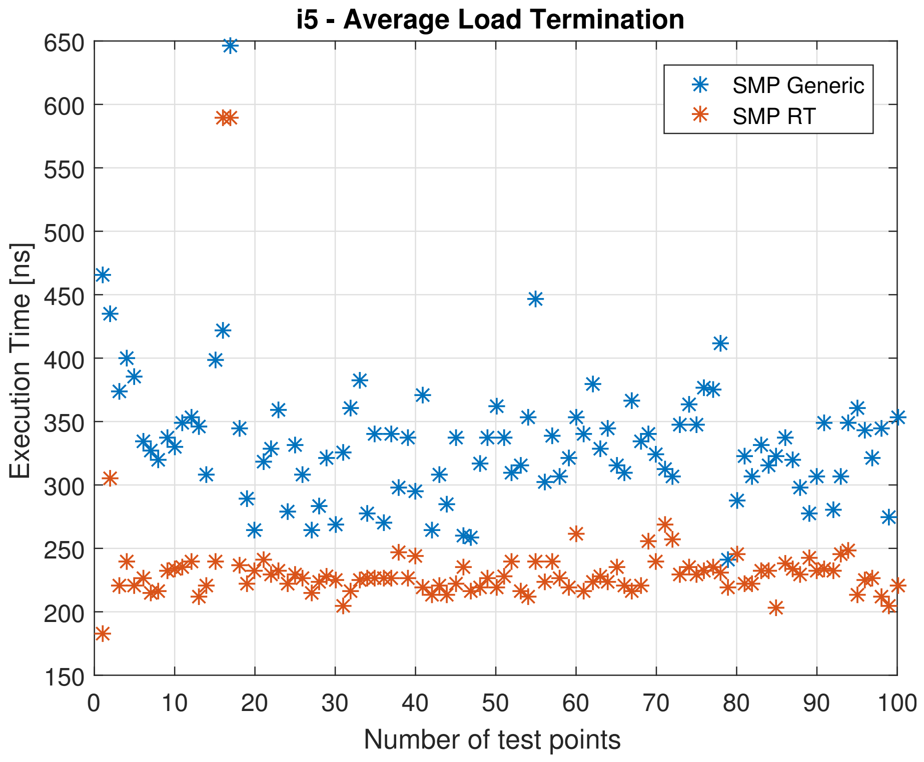

i7 - Low Load Termination
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
0
100
200
300
400
500
600
700
800
900
0
10
20
30
40
50
60
70
80
90
100

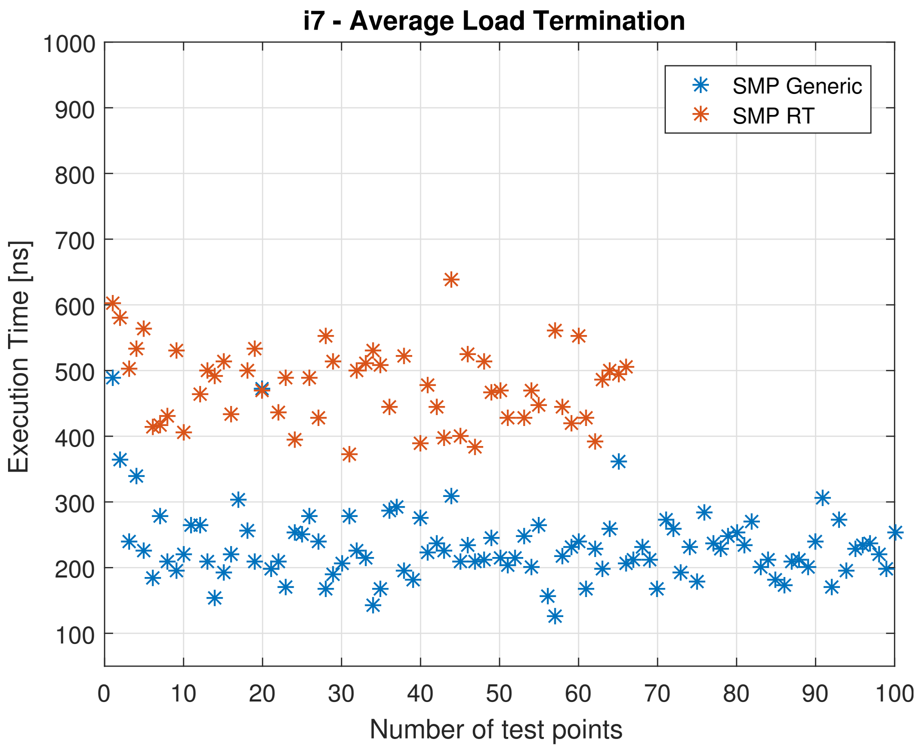

i5 - Low Load ClearEvent
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
100
150
200
250
300
350
400
450
0
10
20
30
40
50
60
70
80
90
100

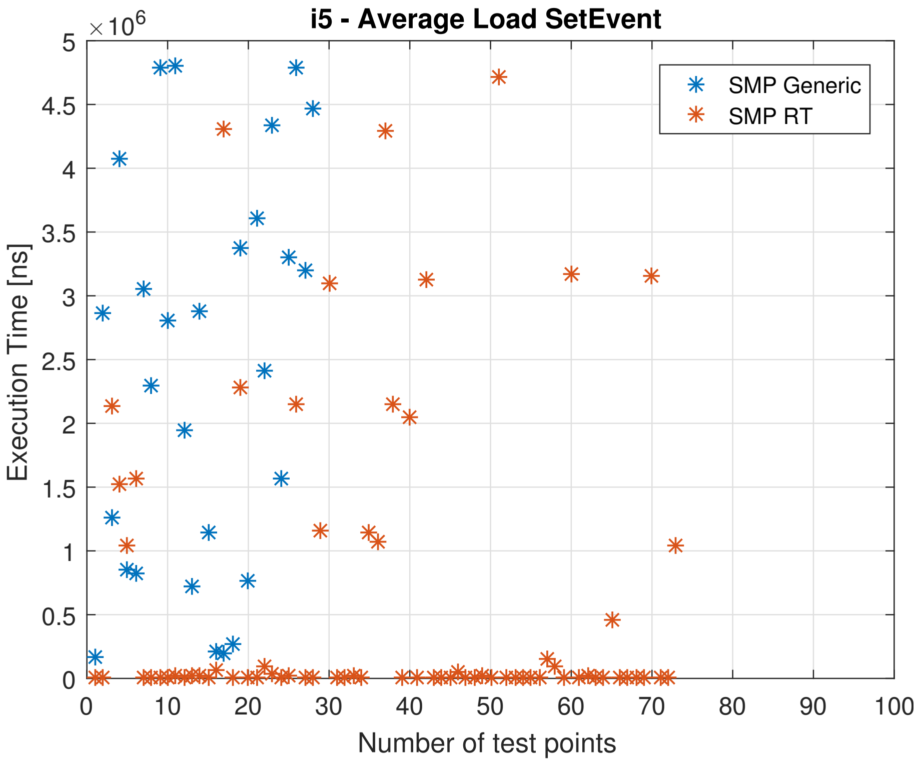

i7 - Low Load ClearEvent
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
0
100
200
300
400
500
600
700
800
900
0
10
20
30
40
50
60
70
80
90
100

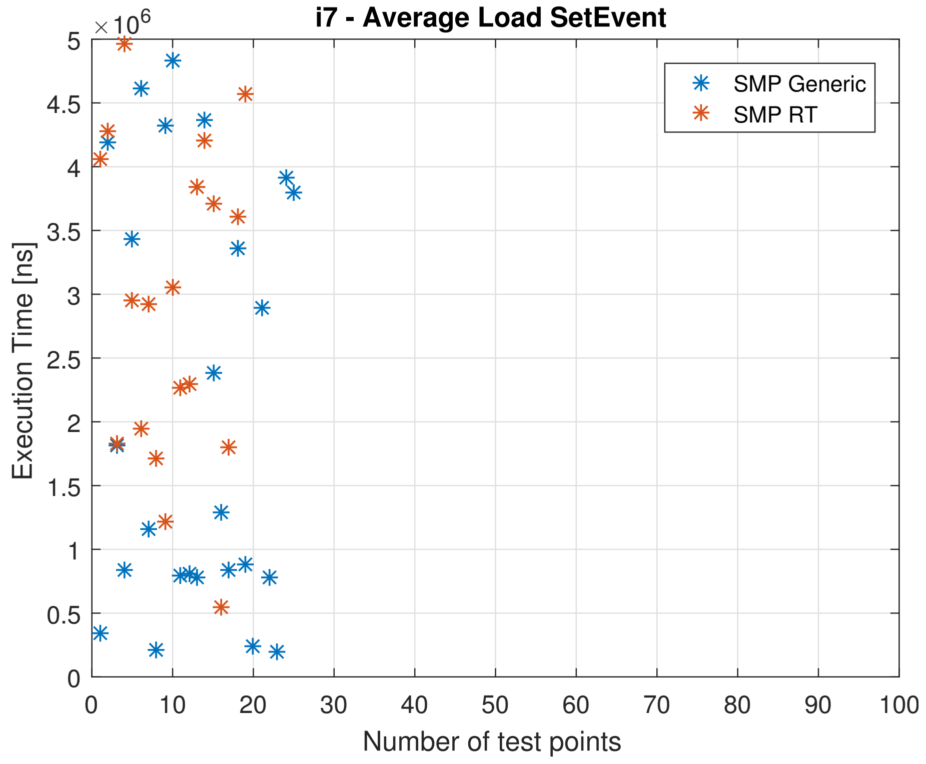

i5 - Average Load Activation
×10^5
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
0
0.5
1
1.5
2
2.5
3
3.5
0
10
20
30
40
50
60
70
80
90
100

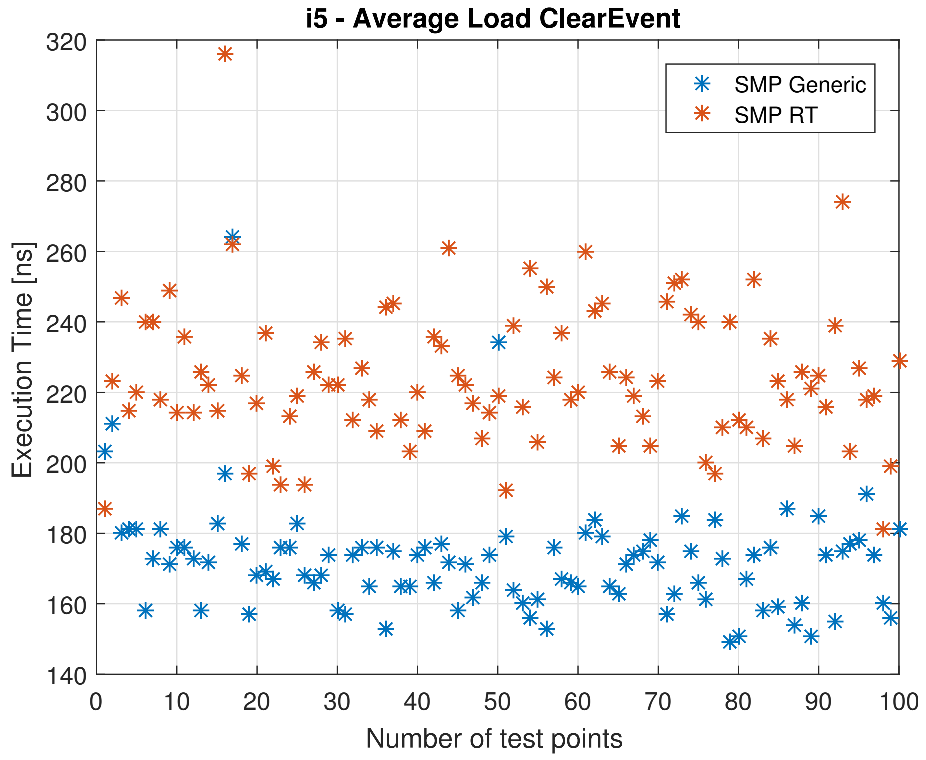

i7 - Average Load Activation
SMP Generic
SMP RT
Execution Time [ns]
Number of test points
1200
1100
1000
900
800
700
600
500
400
300
200
0
10
20
30
40
50
60
70
80
90
100

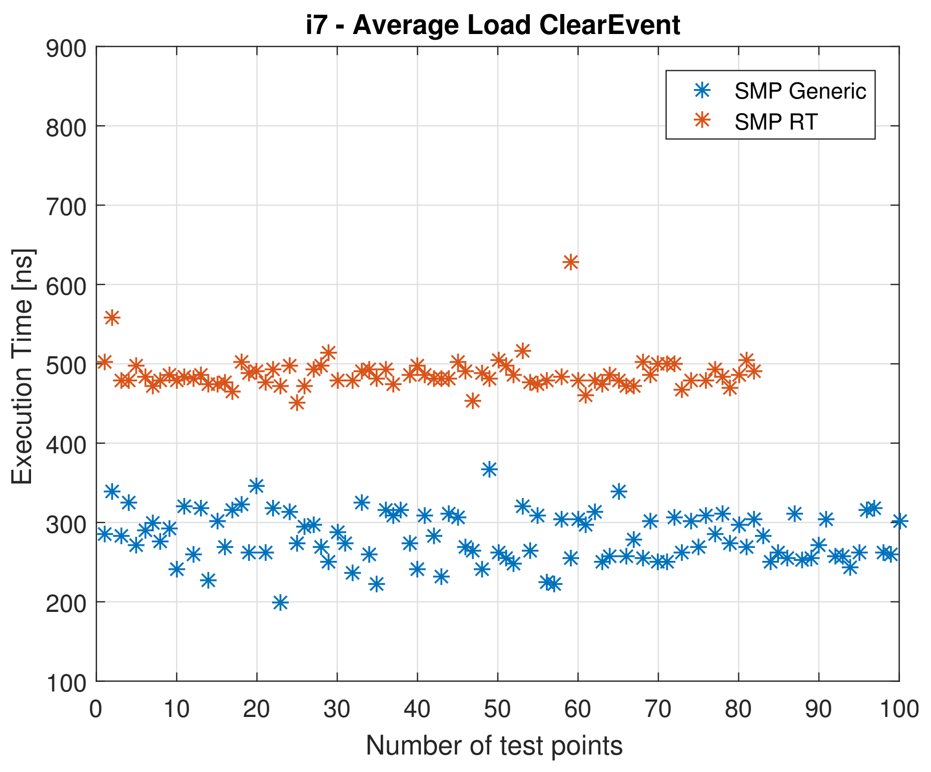

i5 - Average Load Termination
SMP Generic
SMP RT
Execution Time [ns]
Number of test points
650
600
550
500
450
400
350
300
250
200
150
0
10
20
30
40
50
60
70
80
90
100

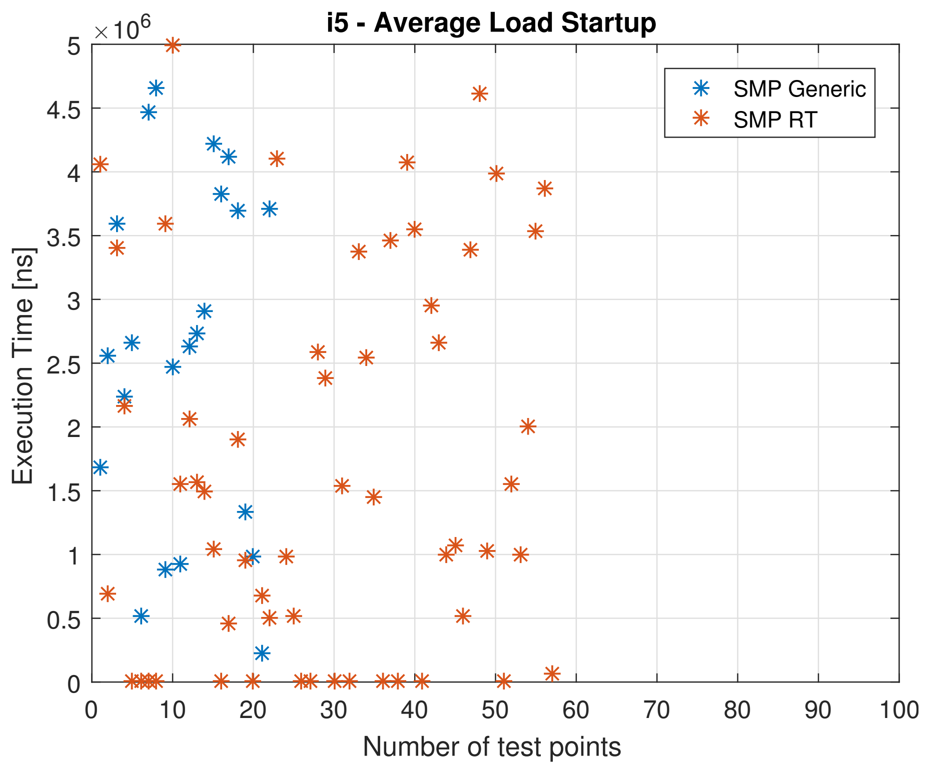

i7 - Average Load Termination
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
1000
900
800
700
600
500
400
300
200
100
0
10
20
30
40
50
60
70
80
90
100

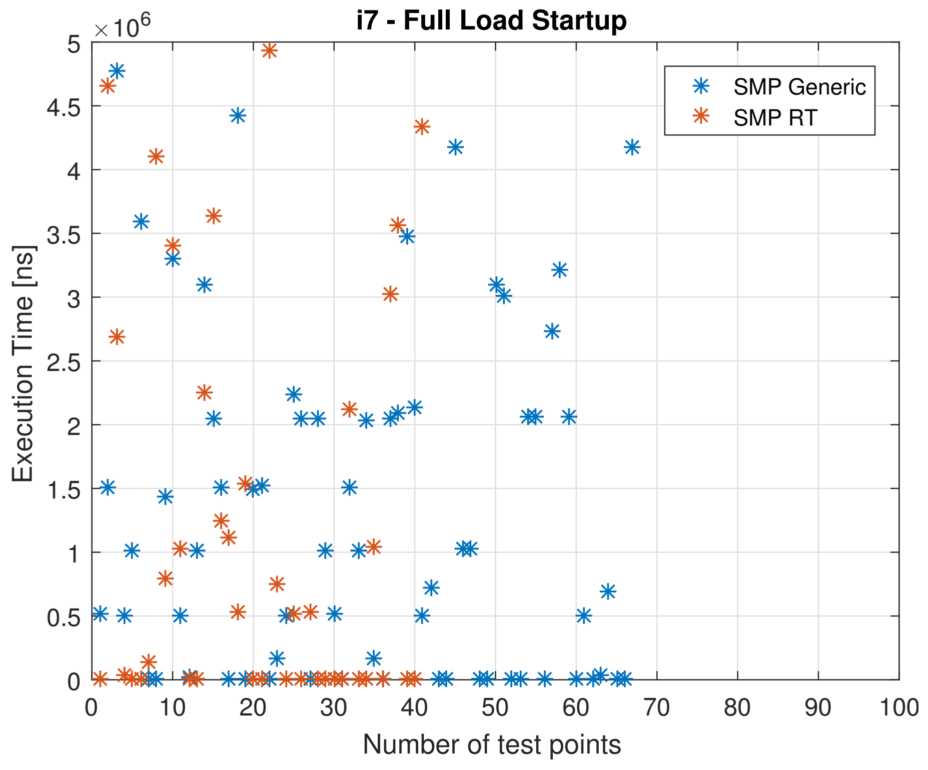

i5 - Average Load SetEvent
$\times 10^6$
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
10
20
30
40
50
60
70
80
90
100

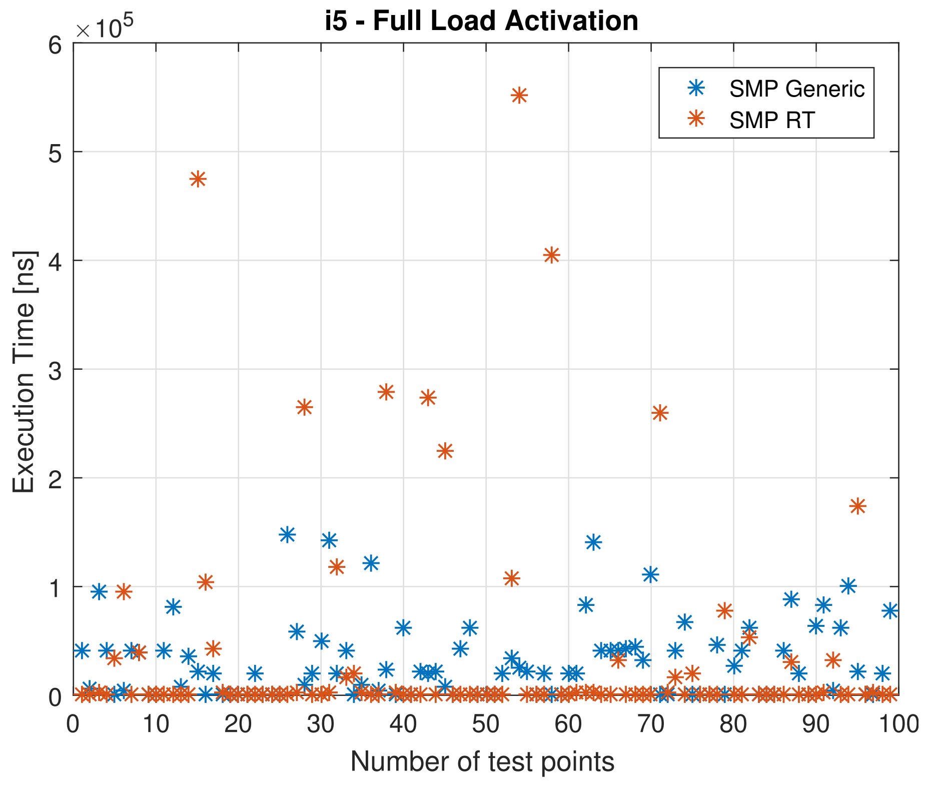

i7 - Average Load SetEvent
×10^6
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
0
10
20
30
40
50
60
70
80
90
100

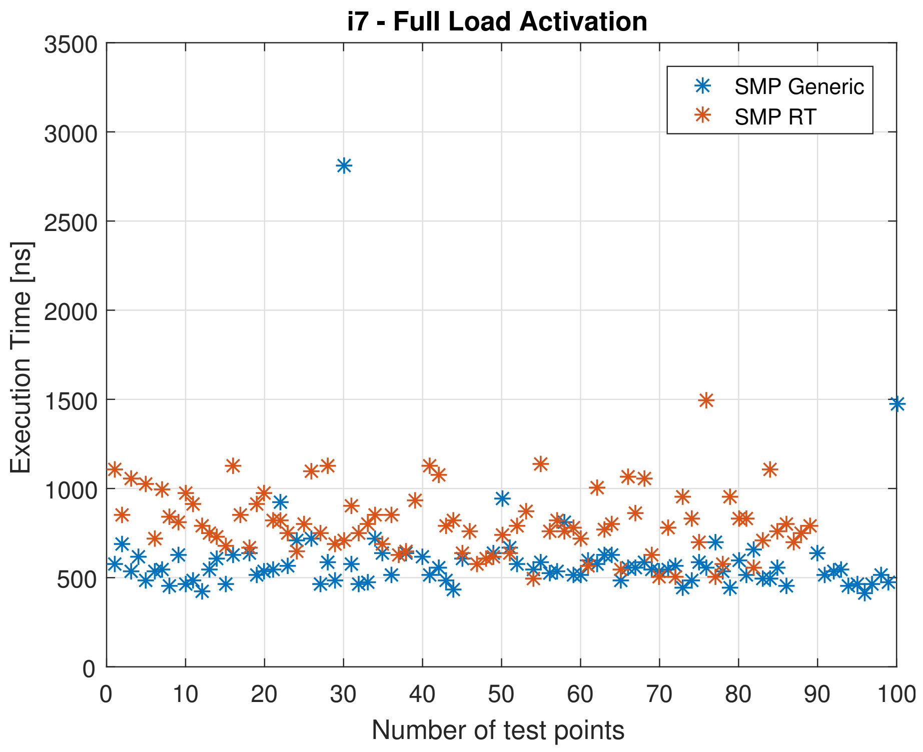

i5 - Average Load ClearEvent
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
140
160
180
200
220
240
260
280
300
320
0
10
20
30
40
50
60
70
80
90
100

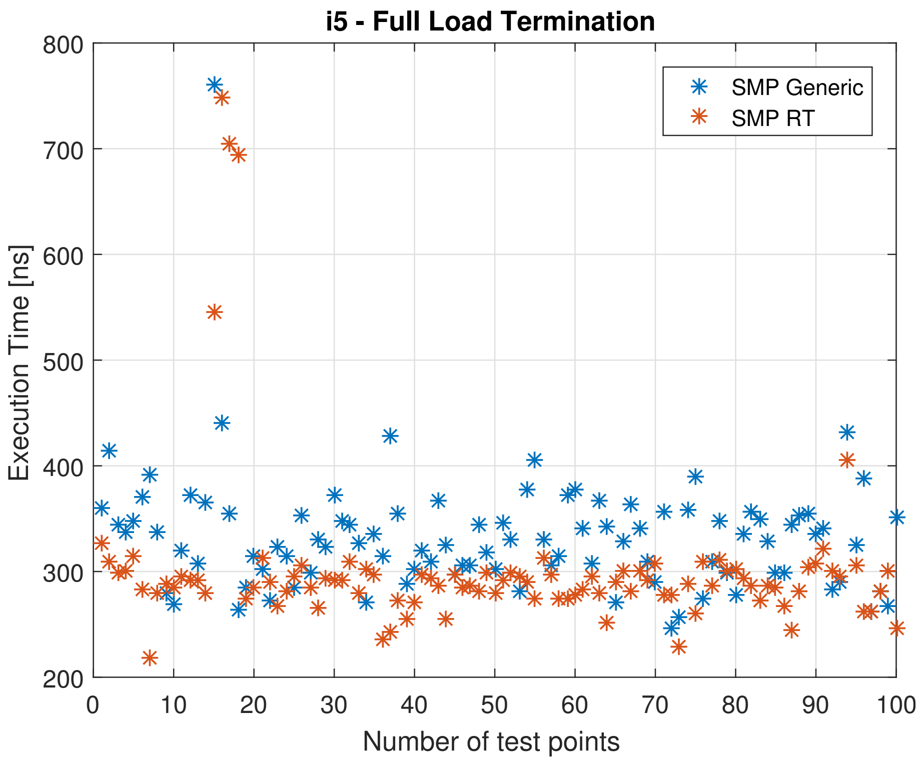

i7 - Average Load ClearEvent
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
900
800
700
600
500
400
300
200
100
0
10
20
30
40
50
60
70
80
90
100

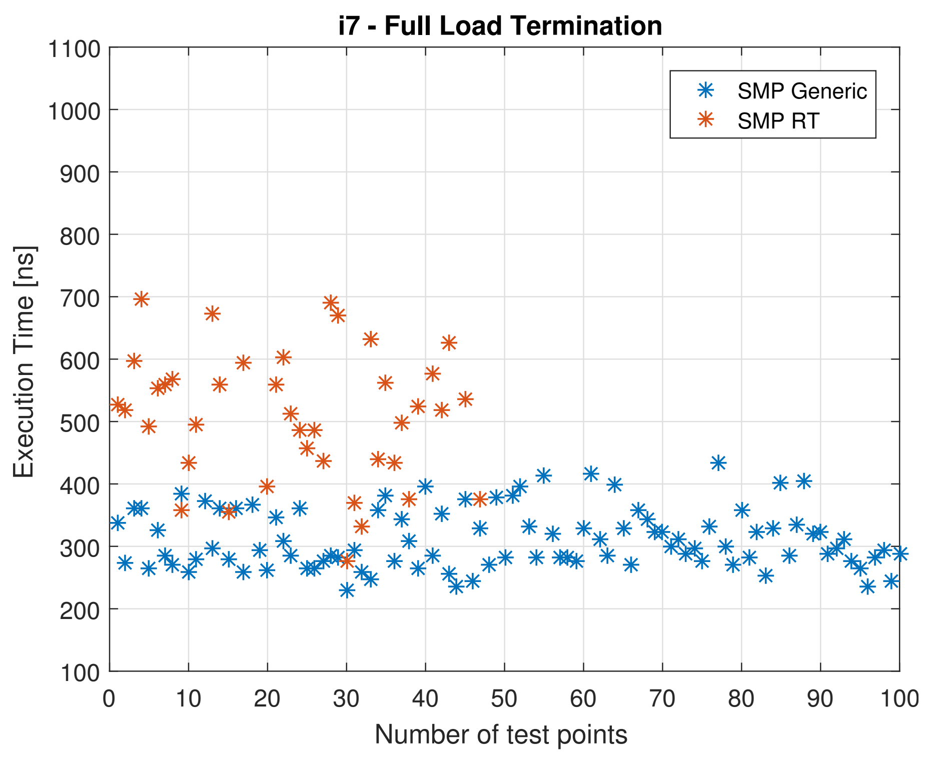

i5 - Average Load Startup
$\times 10^6$
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
10
20
30
40
50
60
70
80
90
100

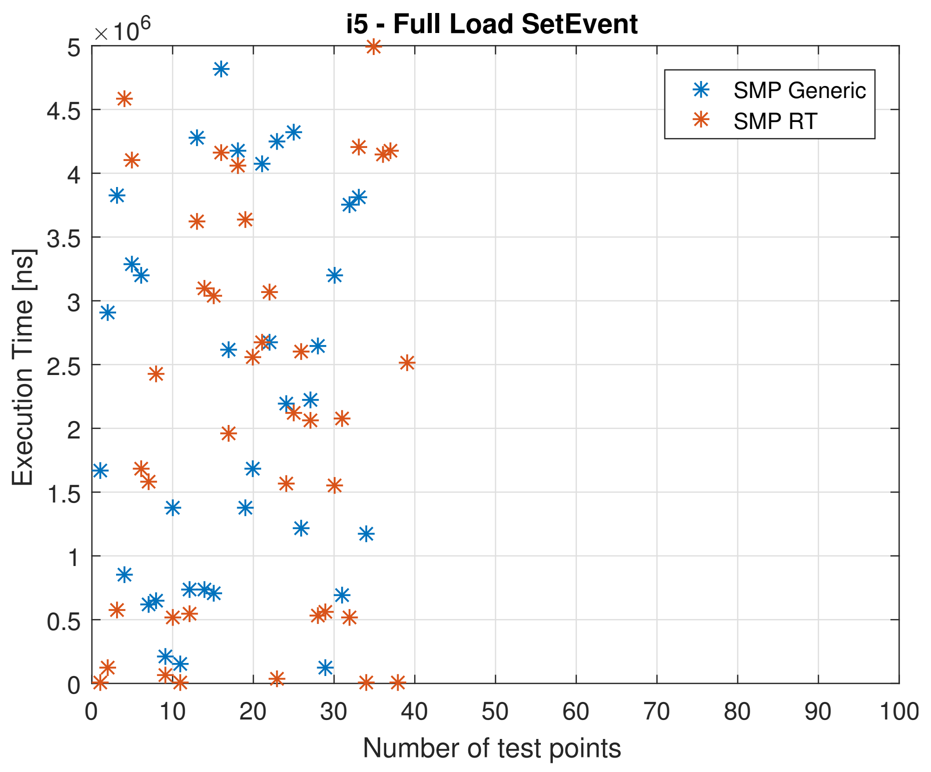

i7 - Full Load Startup
×10^6
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
0
10
20
30
40
50
60
70
80
90
100

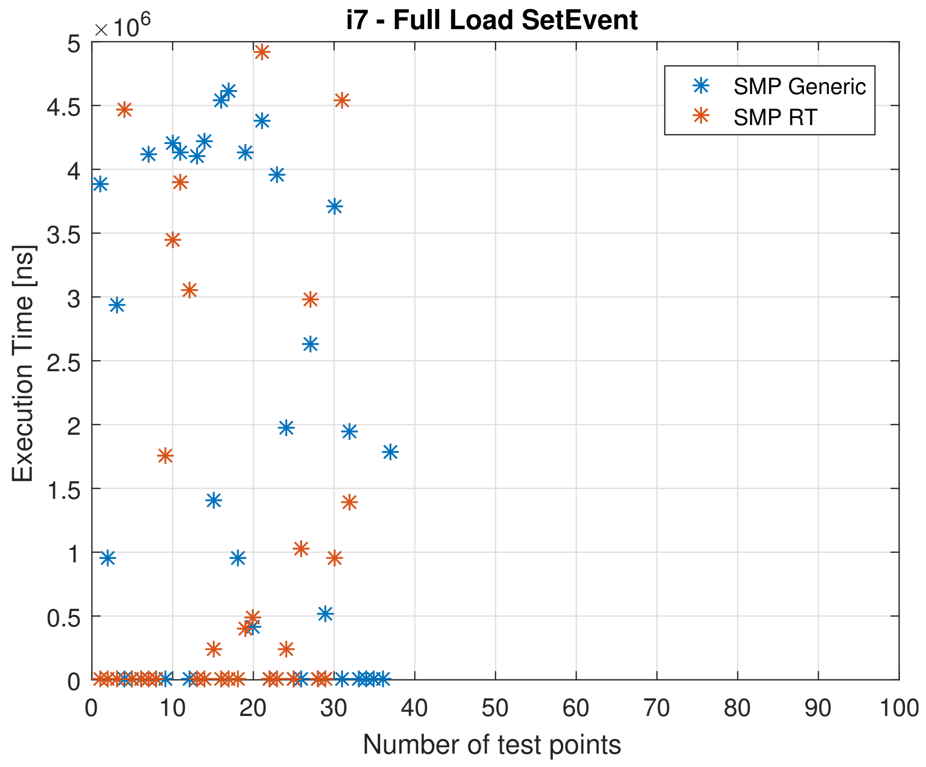

i5 - Full Load Activation
×10^5
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
0
1
2
3
4
5
6
0
10
20
30
40
50
60
70
80
90
100

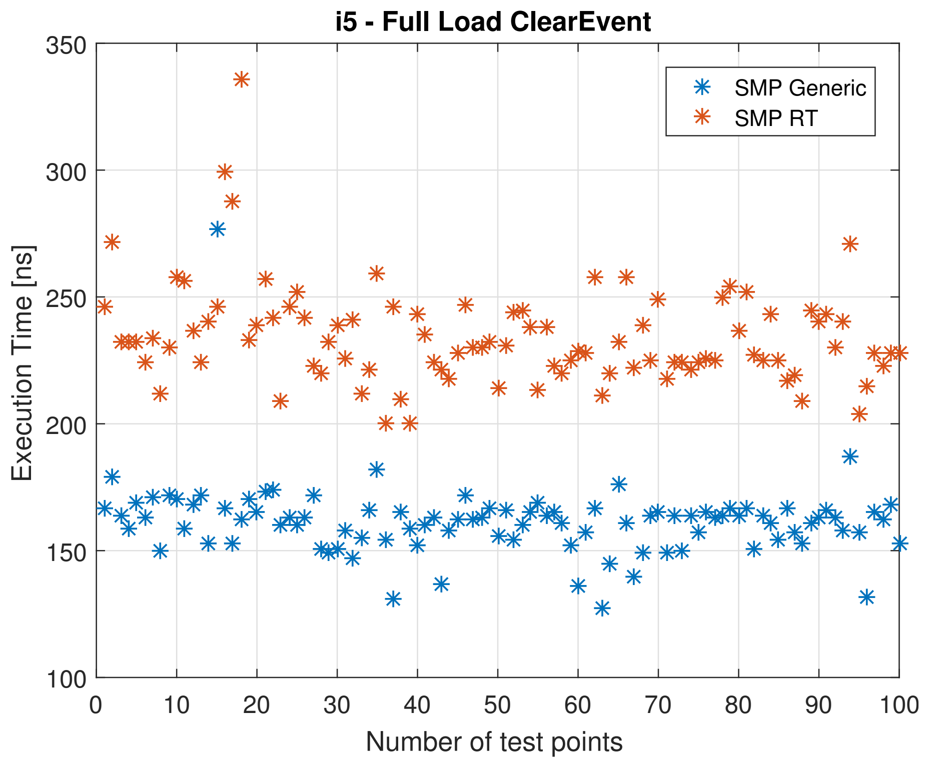

i7 - Full Load Activation
SMP Generic
SMP RT
Execution Time [ns]
Number of test points
0
500
1000
1500
2000
2500
3000
3500
0
10
20
30
40
50
60
70
80
90
100

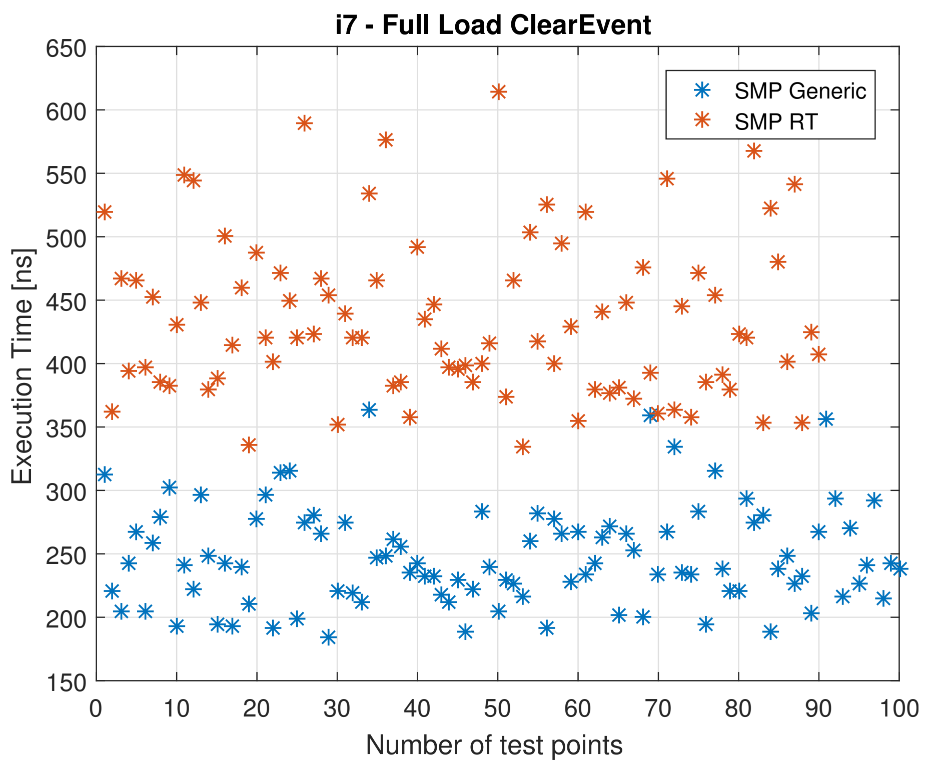

i5 - Full Load Termination
SMP Generic
SMP RT
Execution Time [ns]
Number of test points
200
300
400
500
600
700
800
0
10
20
30
40
50
60
70
80
90
100

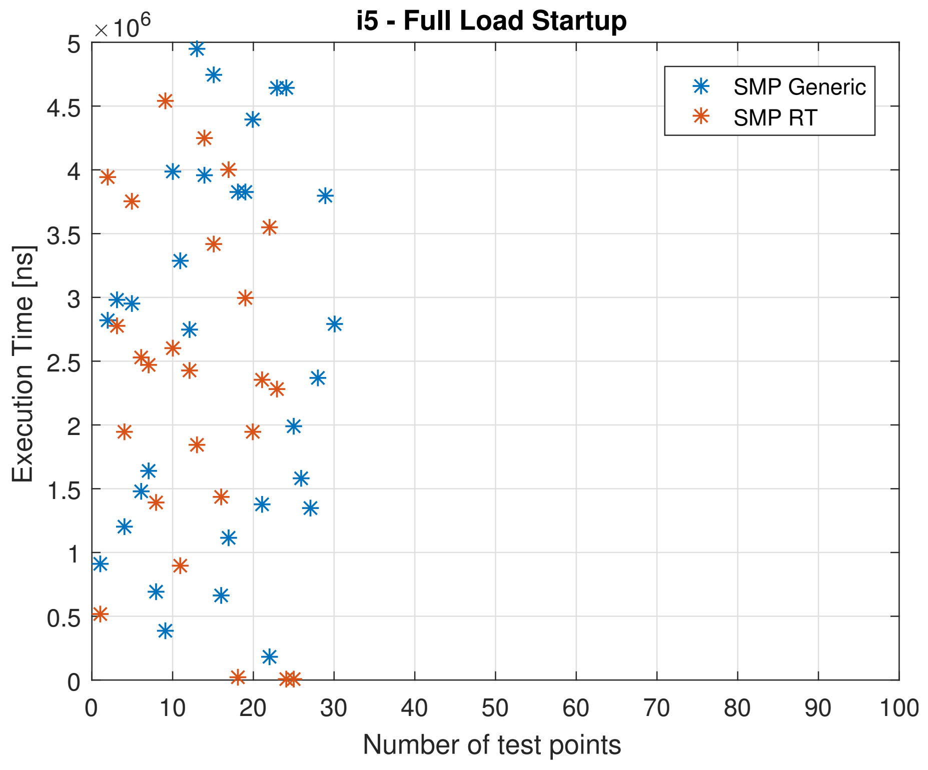

i7 - Full Load Termination
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
1100
1000
900
800
700
600
500
400
300
200
100
0
10
20
30
40
50
60
70
80
90
100

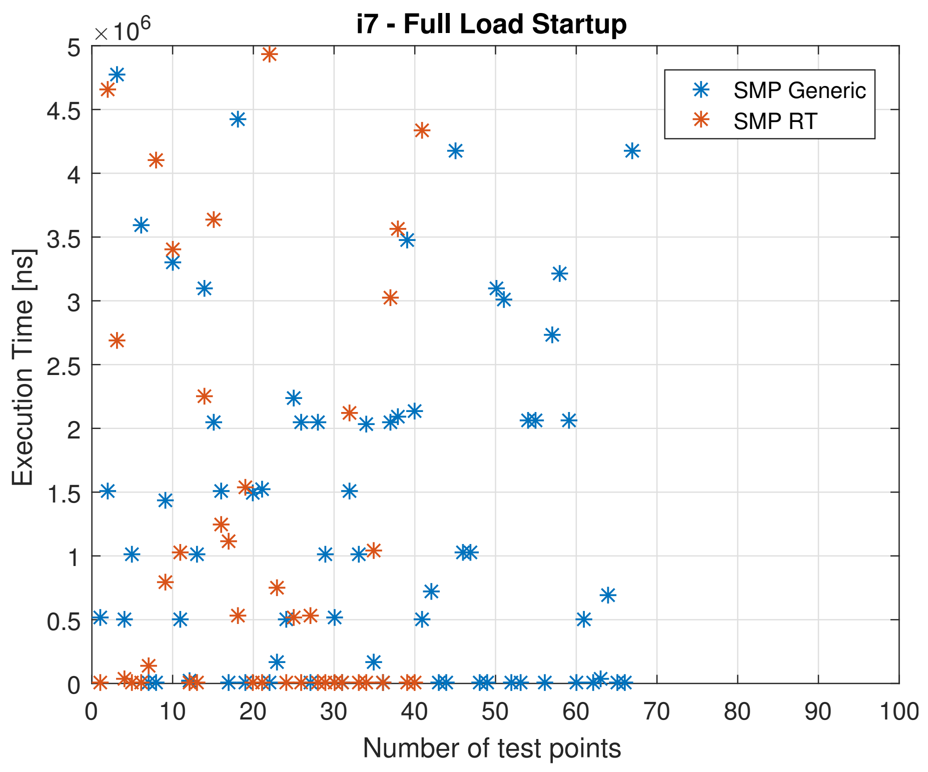

i5 - Full Load SetEvent
$\times 10^6$
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
10
20
30
40
50
60
70
80
90
100


i7 - Full Load SetEvent
×10^6
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
Execution Time [ns]
0
10
20
30
40
50
60
70
80
90
100
Number of test points
SMP Generic
SMP RT


i5 - Full Load ClearEvent
350
300
250
200
150
100
Execution Time [ns]
0
10
20
30
40
50
60
70
80
90
100
Number of test points
SMP Generic
SMP RT


i7 - Full Load ClearEvent
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
650
600
550
500
450
400
350
300
250
200
150
0
10
20
30
40
50
60
70
80
90
100

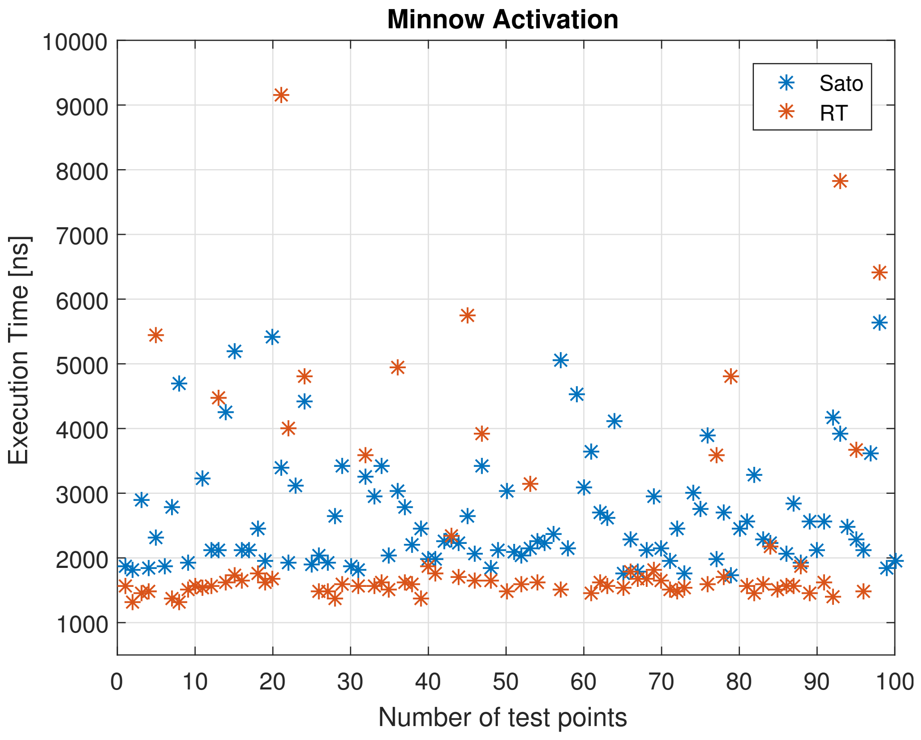

i5 - Full Load Startup
×10^6
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
10
20
30
40
50
60
70
80
90
100

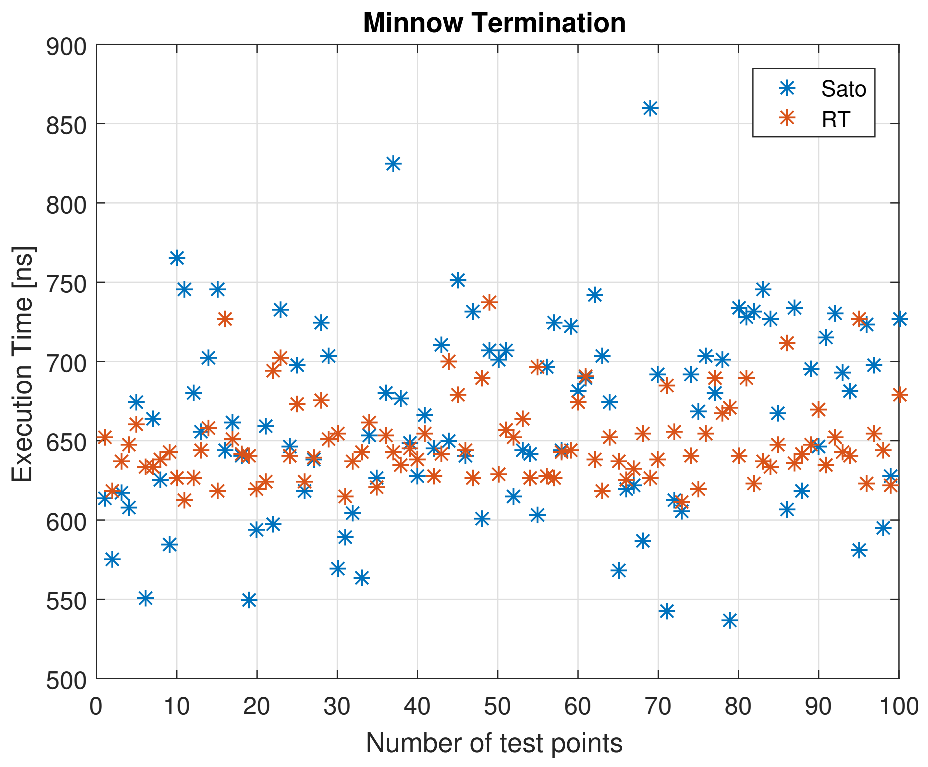

i7 - Full Load Startup
$\times 10^6$
Execution Time [ns]
Number of test points
SMP Generic
SMP RT
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
0
10
20
30
40
50
60
70
80
90
100

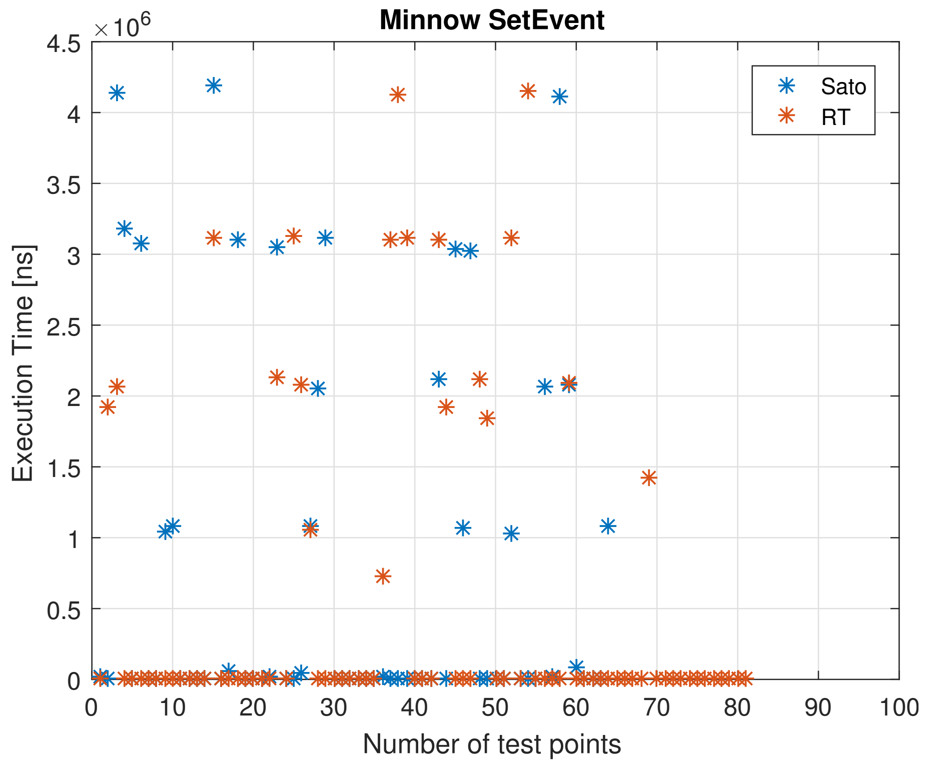
# D

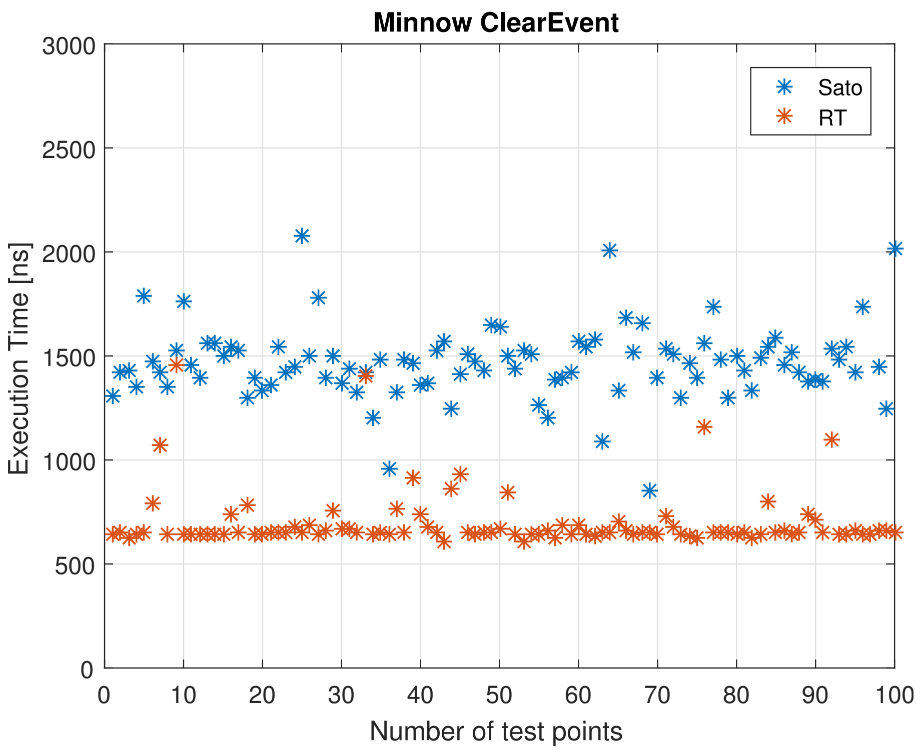
# MinnowBoard MAX Performance Results

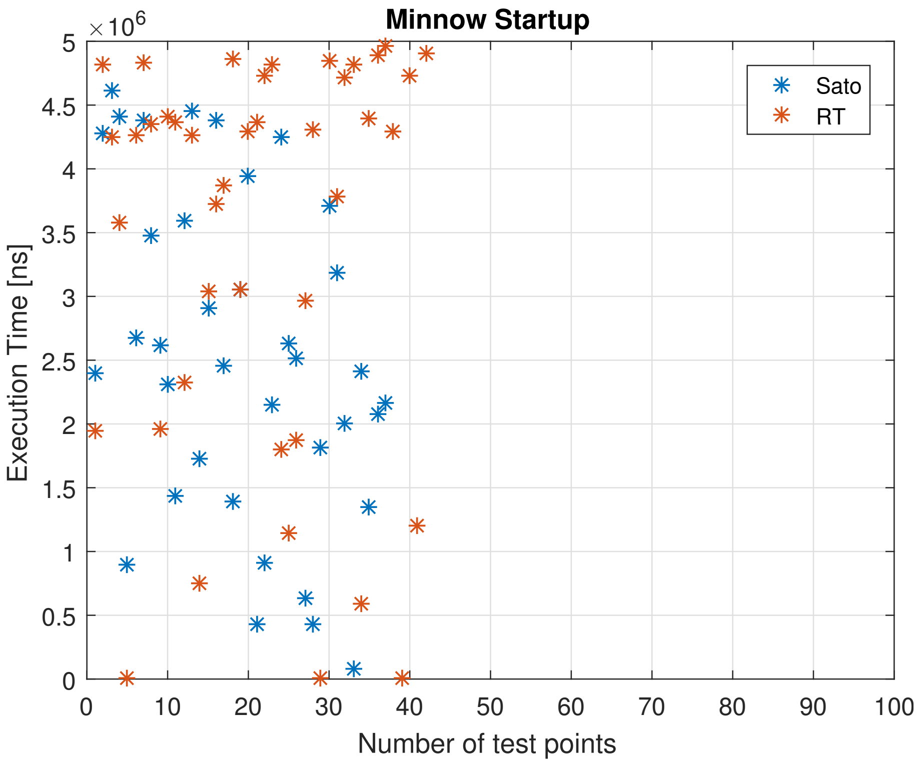
The test results from the embedded alternative, *MinnowBoard MAX*, for the two kernels that we used to test our application on: the mobile *Sato* and an RT patched Linux minimal image.




Minnow Termination
Execution Time [ns]
Number of test points
Sato
RT
900
850
800
750
700
650
600
550
500
0
10
20
30
40
50
60
70
80
90
100


Minnow SetEvent
$\times 10^6$
Execution Time [ns]
Number of test points
Sato
RT
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
10
20
30
40
50
60
70
80
90
100


Minnow ClearEvent
Execution Time [ns]
Number of test points
Sato
RT


Minnow Startup
×10^6
Execution Time [ns]
Number of test points
Sato
RT

# E

# Pthread Test Source Code

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdint.h>

void *newborn(void *ptr);

typedef unsigned int TimerTick;
struct timespec t;
TimerTick CurrentTicks;
unsigned int start, middle, stop;
long MeanActiv = 0;
long MeanTermin = 0;

TimerTick Timer_GetTicks(void) {
        clock_gettime(CLOCK_REALTIME, &t);
        CurrentTicks = t.tv_nsec;
        return CurrentTicks;
}

void *newborn(void *ptr) {

        int *state = 0;

        middle = Timer_GetTicks();
        //pthread_cancel(state);
        pthread_exit(state);
}

int main(int argc, char *argv[]) {
        int i, err;
        pthread_t thread1;

        for (i = 0; i < 200; i++) {
                start = Timer_GetTicks();
                err = pthread_create(&thread1, NULL, newborn, NULL);
                pthread_join(thread1, NULL);
                stop = Timer_GetTicks();

                MeanActiv += (middle - start);
                MeanTermin += (stop - middle);
        }

        printf("Mean Activation Time:  %ld [ns]\n", MeanActiv/200);
        printf("Mean Termination Time: %ld [ns]\n", MeanTermin/200);
        return 0;
}
```