PUBLIC

TULIPP

H2020-ICT-O4-2015
Grant Agreement n° 688403

D3.2: Low power multicore RTOS release report

Lead Author: Antonio Paolillo, HIPPEROS S.A.

with contributions from:

Organisation no.	Organisation name	Participant Name
4	HIPPEROS	Olivier Desenfans
4	HIPPEROS	Vladimir Svoboda
4	HIPPEROS	Paul Rodriguez
2	TUD	Julian Haase
2	TUD	Muhammad Ali
2	TUD	Diana Göhringer

Reviewers

Organisation no.	Organisation name	Participant Name
2	TUD	Julian Haase
3	SUN	Emilie Wheatley

PUBLIC

1 Document Description

Deliverable number	3.2
Deliverable title	Low power multicore RTOS release report
Work Package	3
Deliverable nature	Report
Dissemination level	Public
Contractual delivery date	
Actual delivery	
Version	1.0

	Written by	**Approved by**
Name Signature	TULIPP consortium	

2 Version history

Version	Date	Editors	Description
0.1	2018-08-08	Antonio Paolillo	Initial draft
0.2	2018-08-21	Julian Haase	TUD's contributions and adding suggestions after review
1.0	2018-10-12	Antonio Paolillo	Document finalisation

3 Abstract

This deliverable represents the work carried out in the context of Task T3.2. This task covers the following content. HIPPEROS has designed the RTOS kernel with all the standard kernel features required for the use cases applications (interrupts, memory processes, IPC) and in particular the low power awareness and optimization in a multicore RTOS scheduler. The low power scheduling policies is configurable at design time, and then executed by the RTOS at run time. TUD has developed the low power technique called dynamic voltage frequency scaling which is controlled by the RTOS.

4 Table of Content

5 List of Figures

6 Introduction

Work package 3 focuses on implementing an operating system that matches the TULIPP guidelines and concepts from WP1, considers the choices done in WP2 and WP4 and delivers an integrated real-time operating system to WP5.
To ensure this, we have to work on the following actions:

- The design and development of a parallel real-time operating system that can handle the target platform specifics and constraints and is able to optimize low-power processing (tasks T3.1 and T3.2).
- The creation of the required standard APIs and runtime libraries for the RTOS (task T3.3).
- The extensions of the operating system with support for the hardware and software requirements of image processing (tasks T3.2 and T3.3).
- The instantiation of the reference platform.

Deliverable D3.2 represents the work carried out in task T3.2 spanning from M07 to M30 of the TULIPP project.

The objective of this task is to design and implement the RTOS kernel with all the standard kernel features required for the use cases applications and in particular the low-power awareness and optimization in a multi-core RTOS scheduler.

During this period, the main instance of the OS layer of the TULIPP platform went from a prototype in the early phases of the project to a complete, usable product that has been tested by the different partners and use case owners during integration phases of the project.

The deliverable presents the RTOS concepts and the specific developments like low power techniques that have been done to support the objectives of the TULIPP project.

7 RTOS Concepts

As a candidate for the OS instance, the TULIPP consortium choose HIPPEROS.

HIPPEROS is a family of real-time operating systems which targets high performance embedded platforms. One of its bigger strengths is its configurability which allows us to tailor the OS for specific applications and hardware.

HIPPEROS was designed with the idea that multicore processors will become the standard to run safety-critical embedded applications in the near future. While other RTOS were designed for single core processors, HIPPEROS is by design a multicore operating system. Its powerful master-slave architecture enables scalable system operation on any number of cores.

The HIPPEROS real-time microkernel is the first to provide new state of the art algorithms for scheduling, resource sharing and Inter-Process Communication (IPC). These make the operating system more reliable and more efficient in terms of CPU usage while keeping up with strict hard real-time constraints.

Reliability is at the heart of the system. All kernel structures are allocated at compile-time. Coupling this with the microkernel architecture, the memory footprint of the kernel stays low in all circumstances. This allows HIPPEROS to adapt to a variety of platforms, while the API remains common across the different configurations. This makes migrating applications on new hardware architectures seamless, enabling developers to fully focus on their application.

7.1 Embedded systems challenges

The development of embedded systems (or Cyber-Physical Systems, CPS) and software is typically affected by numerous technical challenges largely inherent to the domain. The embedded developer has to face issues such as stringent hard-real time constraints and safety regulations while developing applications that become more and more complex on platforms which work on limited power. As such, the range of issues addressed by HIPPEROS in its various versions is rather large and varied in nature.

7.1.1 Reliability

Embedded hardware platforms are not always equipped with reset switches, and for good reason. One of the key requirements of almost any embedded system is reliability: the application must be continually available to end users with little to no downtime or software crashes. In many cases such an event would have catastrophic consequences. Therefore, extra care must be taken to design systems in a way that prevents such failures.

7.1.2 Timeliness

Many embedded systems are responsible for the management of physical functionalities such as anti-lock braking systems or the fuel injection control in cars that are inherently subject to tight timing requirements. The challenge posed by these requirements is not that the system must execute software in a short time but rather that it must always be on time. This includes operating under circumstances possibly not foreseen during the development of the system.

This issue can be further divided into multiple categories of tightness. In a hard real-time application, a result which is known later than required is as good as a functionally wrong result. Some applications are said to be soft real-time, such as multimedia decoding. While such applications must deliver a quality of service for which timing is critical, inconsistently missing some deadlines does not make the application useless.

7.1.3 Efficiency

In many cases, the computing platforms used in embedded systems are limited in memory, cache and computing power. In modern, high performance systems, core temperature is also increasingly becoming a limiting factor. Finally, in certain application domains, the energy consumption of a system can be the single most important metric of comparison to competitors.

The over-provisioning of hardware components caused by inefficient use of resources can become extremely costly to manufacturers. This effectively means that any gain in terms of efficiency at the software level through a better use of resources tends to translate to a net decrease of costs or increase in functionality, especially when identified early during development.

In recent years, efforts to decrease over-provisioning have led to developments towards new hardware architectures involving multi-core platforms. The use of multi-core platforms allows manufacturers to pool a greater number of software functionalities in the same place thus reducing the amount of wiring compared to a solution with multiple single-core computing units, which can significantly reduce costs and weight.

7.1.4 Extensibility

Correcting defects and adding features to existing systems through new software versions increases the value of released products. In order to avoid re-writing code from scratch every time a change is required, software must be divided into independent building blocks that can be individually replaced, much like replaceable mechanical parts.

However, designing software to support this is not trivial and can be made very tricky in the presence of shared resources and inter-dependency between functionalities (services, drivers and so on).

7.1.5 Time to market and software development

The value of new electronics is increasingly defined by software quality and functionalities. However, developing software for embedded systems is notoriously difficult, especially for companies which historically created value from advances in hardware, not from software.

As shown in the rest of this chapter, embedded platforms and environments require software engineers to take into account far more complex issues than those they are usually familiar with on general purpose desktop and mobile platforms. Additionally, the available software development tools for embedded applications are both less powerful and more expensive.

The need for many applications to develop custom software at very low levels of abstraction translates to a significant increase in the time it takes to develop software for a complete product. On top of development time, the resulting software is also more likely to contain bugs or require deep re-designs late into development.

Any delay caused by software development can have serious consequences on the success of a given product in a competitive industry. As such, taking steps towards optimizing for short and predictable software development time is very valuable.

7.2 The HIPPEROS solution

Overcoming the challenges of embedded software development requires specific methods and tools. HIPPEROS provides such tools in order to facilitate the job of software engineers and let them unlock the potential of the platforms they use.

7.2.1 Common OS features

HIPPEROS offers a collection of familiar tools for application developers like any OS in its category. Those include a consistent process model, memory management, multi-threading, concurrency and locking primitives, inter-process communication channels, debug logs and others.

7.2.2 RTOS architecture

HIPPEROS follows a micro-kernel architecture. This means that only the most critical functions of the OS are executed in privileged mode on the CPU. In HIPPEROS, this essentially consists of memory management, scheduling and inter-process communication. The rest of the OS functionalities such as

drivers and network libraries are implemented as services which are user mode processes, just like any other task.

Traditionally, OS design is monolithic. That is, a vast array of OS functionalities is run in privileged mode.

Thanks to micro-kernel design, a fault in a given HIPPEROS task or driver cannot bring the whole system down. The small size of the kernel code makes it easier to test, understand and check for errors than more traditional kernel designs.

Of course, certain critical task may rely on other tasks and drivers to execute properly. The explicit isolation between tasks in this model not only makes the kernel simpler but also makes the interactions between individual tasks easier to engineer.

7.2.3 Scheduling

HIPPEROS implements state of the art scheduling algorithms to ensure the timeliness guarantees of tasks with minimal overhead, provided that the tasks have been appropriately analysed and prioritised at design time. HIPPEROS offers a flexible API for scheduling which makes it easy to implement a custom scheduling algorithm in case this is needed.

This facility allows designers to implement individual tasks in isolation and care about their scheduling at a later stage. Instead of being embedded in code, the scheduling of tasks in HIPPEROS is by default priority-based and supports event-driven tasks. This largely eliminates constraints on task periodicity and release model in general, thus expanding the space of applications that can be done by using HIPPEROS.

7.2.4 Resource sharing

The micro-kernel design of HIPPEROS makes sharing (hardware or software) resources between multiple tasks easy. On a bare metal system, the code of a task must explicitly manipulate the low-level interface of the resource it accesses. With an OS, the details of the (sometimes very error-prone) low-level code are hidden away in a dedicated task, a service, running independently of its clients. This service is accessed by its client tasks through the inter-process communication primitives of the kernel. Using an OS in such a way brings a twofold advantage.

Firstly, the details of the (sometimes very error-prone) low-level code are hidden away in a dedicated piece of code that can fail nicely and independently of other tasks, adding to global reliability. Secondly, the service is transparent to multiple users. Concurrent requests by multiple clients will be handled

sequentially (or with pre-emptions if it makes sense), which means that the tasks need not be aware of each other to be functionally correct, although there may be timing issues.

7.2.5 Power usage

HIPPEROS supports power management primitives for DVFS (Dynamic Voltage and Frequency Scaling). This allows the system to decrease the clock frequency and core voltage on the hardware platform in order to save energy. Decreasing clock frequency has an adverse effect on execution time and can therefore potentially harm timeliness guarantees.

With HIPPEROS, system designers willing to make use of DVFS have to specify a sustainable clock frequency for every task. That is, a frequency at which the task is guaranteed to complete within its deadline. With this information, HIPPEROS can automatically change the clock frequency and core voltage of the platform at runtime in order to minimize energy consumption without putting timeliness constraints at risk.

8 Operating System Architecture

The first thing to consider when designing an operating system is its overall architecture to meet the high-level requirements. One first needs to ask: what is the mission of a modern operating system?

The base requirements are hardware abstraction and multitasking. Simply put, the user wants to write code that is independent from the hardware target and run multiple applications at the same time without negative interference between them (e.g. a process accessing the resources of another process, corrupting its data structures).

This is where trade-offs come into place in the design. OS designed for performance will provide only limited restrictions on applications to reduce overheads at the cost of system stability. Meanwhile a hard real-time operating system will have to provide guarantees about the execution time of applications. It will provide isolation between applications to make sure that any unexpected behaviour (e.g. a bug in one of the applications running on the system) will not impact the stability of the whole system. As we will discuss later this has major impacts on the architecture of the OS itself.

8.1 Hardware OS support

Before discussing the OS architectures, let us see what modern processors have to offer to the OS designer.

The first feature is privileged/unprivileged execution modes. For decades now processors have offered different execution modes for different pieces of code running on the target. Processors have at least one privileged execution mode to run system management code and one unprivileged execution mode to run user applications, the user mode. Some processor instructions simply cannot be executed when running in user mode. This means that code running in user mode cannot take control of the whole system.

The second feature is memory management. Applications are located in the main memory of the system. Processors provide features to protect the memory zones of applications: either a simple Memory Protection Unit (MPU) or a full-fledged Memory Management Unit (MMU). The MPU allows to restrict the access to certain ranges of memory addresses and forbid accesses from unauthorized applications. A MMU allows to virtualize the address space, i.e. let each application think that it has the whole memory space for itself. This enables what is called spatial isolation between applications. This is discussed further in the technical note on Paging.

8.2 Basic OS architecture

Aside from exotic systems, every modern OS is designed around what is called the kernel. The kernel is code that manages the applications and the resources of the system. Essentially, the kernel is all the code that runs in the fully privileged mode of the processor. It has full privileges over the system and can execute all the system control instructions. The kernel provides various services to user applications. These services are invoked using so-called system calls, gateways between the user and privileged worlds.

The architecture of the OS essentially depends on the functionalities that are provided by the kernel and the ones that are not. Two main philosophies exist: monolithic kernels and micro-kernels.

8.2.1 Monolithic kernel

The monolithic kernel philosophy is that the entire operating system runs in privileged mode. This means that things such as the file system, the TCP/IP stack, the USB drivers and so on also run in privileged mode. This has obvious advantages in terms of performance as every kernel functionality is only a function call away. This also means that a bug in any of these kernel modules can bring the whole system down. The Linux kernel is a known example of monolithic kernel.

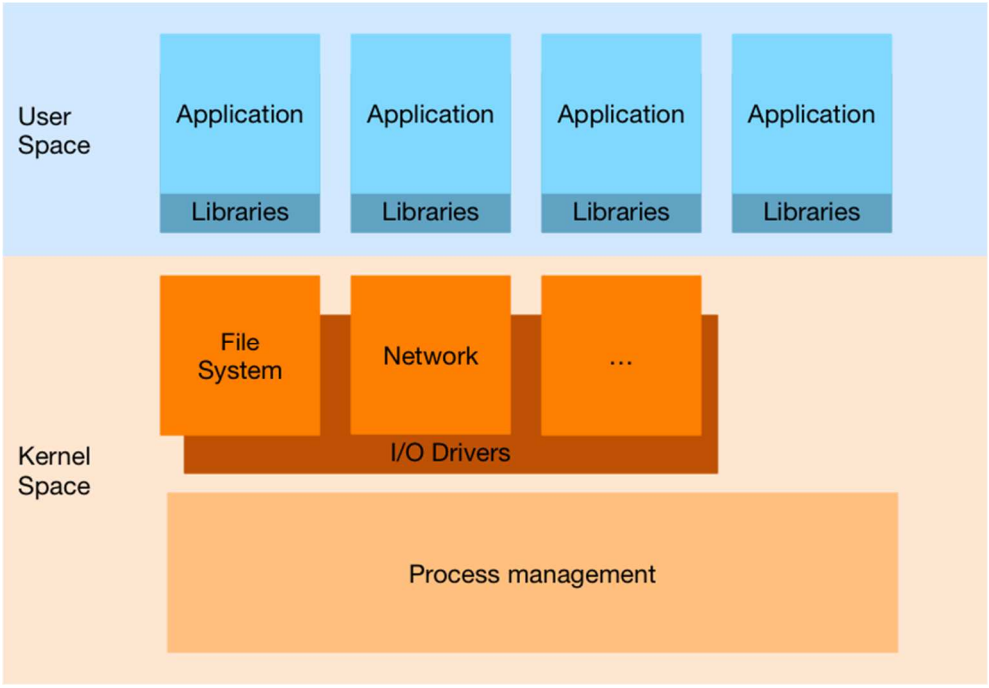

Figure 1 Monolithic kernel architecture. The OS services are part of the kernel. Applications use system calls to access these services.

8.2.2 Micro kernel

The micro-kernel philosophy takes the opposite approach: everything that can run in user mode should do so. This means that device drivers, communication stacks, and so on run in user mode. The actual features of the kernel vary between implementations, but usually only the application management features remain: scheduling (deciding which application can execute when) and system-managed inter-application communication. Operating systems that follow this philosophy are made of a set of system services running in user mode and the kernel at the centre.

This model drastically reduces the amount of code that runs in privileged mode at the cost of more expensive calls between operating system modules.

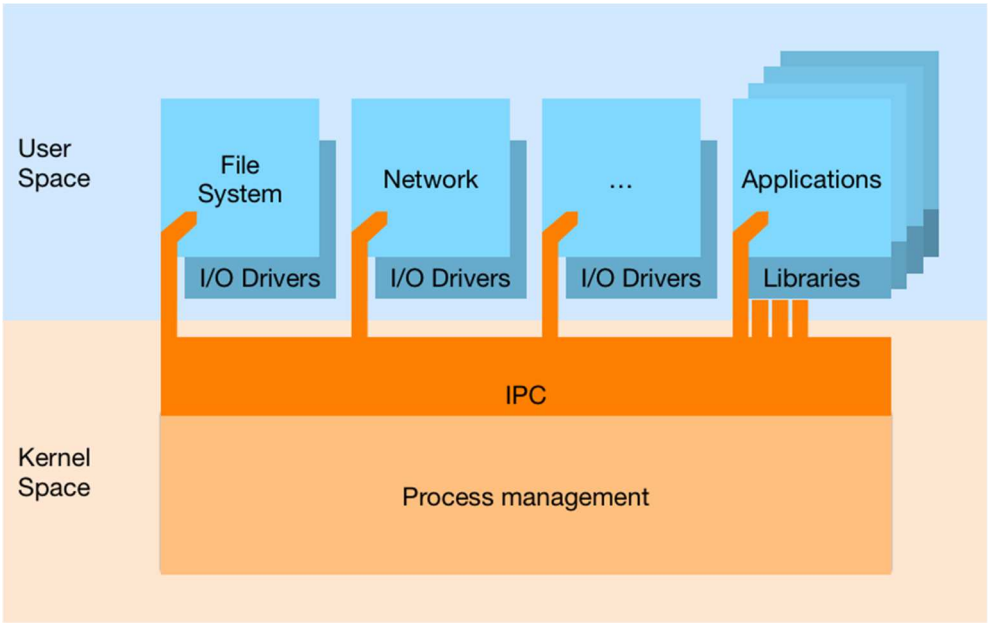

Figure 2 Micro-kernel architecture. The services run in user-mode. The applications access them using IPC.

8.3 Certification

In the context of certification, it is obvious that monolithic kernels do not pass the bar in terms of sheer cost. For example, a bug in the USB driver used for a non-critical logging application can bring the whole system down as the driver is running in privileged mode. The amount of code to certify at the highest level for the system is simply too large.

Micro-kernels on the other hand can go down to a few thousands of lines of code. Every OS module is separated as a single system task running in user mode; in the worst case a bug in one of them will

bring down the other applications that use it, but not the whole system. The micro-kernel approach is therefore suited for mission-critical certified applications.

This is the philosophy adopted for the design of the HIPPEROS RTOS.

8.4 Multi-core kernel design

Multi-core platforms have brought new challenges to kernel design. On a traditional single core processor, only a single application can run at any given moment. Multi-core processors can run several applications at the same time. This means that several hypotheses used for single core operating systems are no longer valid. Handling the concurrency between applications running in parallel requires entirely new kernel designs. For example, what happens when two applications running on different cores execute a system call? Are two instances of the kernel running at the same time? Can these instances manipulate the same data structures?

8.4.1 Symmetric kernel

In the symmetric kernel model, every core can run the full kernel code. This means that several instances of the kernel can run in parallel and try to access the same data structures. Therefore, the system-wide kernel data must be protected using locking primitives to avoid concurrent updates.

This approach has many pitfalls in terms of scalability: as more cores try to access the same data structures, latencies rise (Brandenburg, 2011).

8.4.2 Master-slave kernel

In opposition to the symmetric kernel model, the master-slave model follows a fully asymmetric approach. A single core in the system (the master core) runs the full kernel. Other cores (so-called slave cores) run a limited kernel. The principle is simple: the master core has ownership of all the kernel data structures. The slave cores can read these structures freely. If an event that occurs on a slave core requires to update these data structures, the limited kernel running on that core will send a request to the master core (using inter-core interrupts) to do the update. The master core is in charge of queuing the requests and executes them in the appropriate order.

This approach reduces the number of system-wide locks and improves the predictability of the system.

8.4.3 References

Brandenburg, B. (2011). Scheduling and Locking in Multiprocessor Real-Time Operating Systems. Chapel Hill.

9 Scheduling

The operating system's main objective is to support multi-tasking. In any multi-tasking system, the question of choosing among a set of executable jobs which one is going to be executed next (that is, scheduling) is central to the whole system.

In HIPPEROS, the unit of execution manipulated by the scheduler is a thread. Under the hood, a thread has its own stack and context but shares its address space with the other threads of a process. Threads are activated either when a process is activated or when another thread creates them. Only the abstract notion of currently executable jobs (interchangeably called ready jobs) actually matters to the scheduler, not the details of how these jobs come to become executable or how they cease being executable.

The role of the scheduler is merely to select which (there may be more than one) of such ready jobs are to be executed. To this end, the scheduler may consult a variety of data structures of the operating system. Such structures may include a table of task and thread characteristics, a time frame descriptor table or even a special structure built and updated by the scheduler module itself.

9.1 Task model

The threads created in a HIPPEROS system are dependent on a process, which is itself spawned at times defined by a task description. In HIPPEROS, tasks essentially dictate when the associated processes are activated, the values of their deadlines and for how long these processes can execute (before triggering watchdog mechanisms).

A task can either be unique (i.e. only spawning one process), periodic or sporadic. Periodic tasks activate processes at regular intervals whereas sporadic tasks are activated by exterior events (that is, another task makes a system call that activates such a task). The task model of HIPPEROS does not allow a task to have two processes overlap in time. That is, the previous process of a given task must be terminated before the next one is started. This allows for a few important simplifications in the data structures used by various schedulers.

HIPPEROS being a commercial operating system adapted to real use-cases, the tasks and therefore the threads running on a HIPPEROS system may be interdependent. That is, a thread may block its execution waiting for some event depending on the execution state of another thread or a system-wide interrupt. The scheduler must handle such self-suspending tasks efficiently. Additionally, a HIPPEROS system may have to run very frequent, short tasks together with long-running tasks. It is

necessary for the scheduler to handle preemptive scheduling i.e. the ability for a thread to be paused and saved at some point of its execution and continued at a later time. Otherwise, the frequent tasks may suffer intolerable delays caused by the long tasks. The dynamics of such a model may cause a job that is not currently running to change its status from executable to not executable (and of course, the other way around).

9.2 Monocore scheduling

The first embedded electronics used monocore platforms (or in any case, a sequential model of execution). Multi-tasking techniques were first developed for monocore platforms and are still widely used in embedded electronics today. Monocore scheduling consists in selecting one job for execution among the set of executable jobs.

9.2.1 Frame-based scheduling

Frame-based scheduling is a scheduling method where the job being executed (if any) is unequivocally determined by the current system time. Each job has its own reservation of the available execution time, known at design time. Of course, none of these reservations overlap in a correct schedule. The time intervals in which the jobs are slated to execute are periodic. All such times taken together form a repeating frame. This scheme of sharing a resource is often called time partitioning. Figure 1 shows an example of a frame-based schedule.

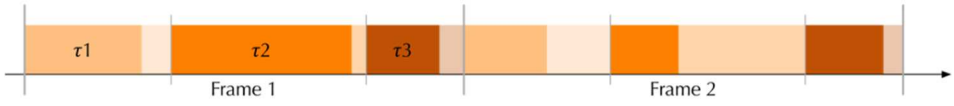

Figure 3 A frame-based schedule of three tasks.

Figure: Each task has a reservation (three units for the first task, four for the second task and two for the third task). Two consecutive frames are shown, where the actual execution times (the three saturated zones in each frame) of the tasks are different. This shows that the system idles between the end of a task and the beginning of the next task.

Frame-based scheduling is the approach used in the ARINC-653 standard. This approach guarantees that tasks will not interfere with each other unpredictably and always execute sequentially in a known and controllable order. Frame-based scheduling is preemptive, however it is difficult to handle task dependencies in this model. The latest time at which a task that can block waiting for another must be known in advance in order to reserve the next time slot for the task being waited on. Furthermore, frame-based scheduling is not efficient when handling tasks with sporadic release patterns. The time

frames allocated to sporadic tasks are reserved regardless of actual usage, which translates into considerable waste.

Frame-based scheduling as described in this document is not a work-conserving scheduling technique. Whenever a job is completed earlier than the end of its time frame, the rest of that frame is not used by another available job. This results in the behaviour of the system being very predictable. However, it also means that the unused part of each time frame is wasted, leading to over-provisioning and inflated costs.

9.2.2 Priority-driven scheduling

In systems that require more flexibility than what frame-based scheduling can offer (that is, very little), the job selected for execution may be chosen according to task priority. In such a scheme, the job from the task with the highest priority among the executable jobs is the one selected for execution. This type of scheduling is often referred to as Fixed Task Priority (FTP) scheduling.

This scheduling technique presents the advantage that tasks with high priority can be considered isolated from the potential adverse effects of tasks with lower priority. In comparison, frame-based scheduling ensures that all tasks are temporally isolated from each other. Priority-driven scheduling is work-conserving. With appropriate task release patterns, the fraction of platform capacity that can be used by the jobs can be arbitrarily close to 100%, although variations in job execution time will result in the actual utilization being lower.

A very common assignment scheme for FTP scheduling is Rate Monotonic (RM). RM scheduling attributes the highest task priority to the most frequent task (i.e. the one with the shortest period). Figure 2 shows an example of a RM schedule.

Name	WCET	Period
Task 1	1	3
Task 2	3	8
Task 3	2	12

Table 1: A task set of three tasks and their basic characteristics. The total utilisation of this system is just below one, which means that it is theoretically possible to schedule this system on one core.

PUBLIC

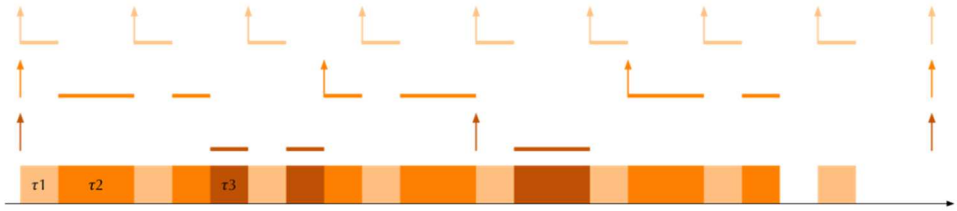

Figure 4 A Rate Monotonic schedule.

Figure: This is a schedule of three tasks that have the characteristics described in Table 1. The arrows pointing up show the release times of the jobs of each task and the thin lines show when each task is executed. The thick multi-colour line below shows the state of the processor (that is, either idle in white or executing one of the tasks). In particular, the execution of the first job of Task 3 is preempted by Task 1 and resumed later.

9.2.3 Earliest Deadline First scheduling

As implied by the name, Earliest Deadline First (EDF) scheduling will always schedule the executable job with the earliest deadline. Even though job deadlines may be seen as priorities, this scheduling method differs fundamentally from priority-driven scheduling introduced earlier. Indeed, in EDF scheduling, the priority of a task relative to other tasks may vary over time. Only the priority of a job is fixed.

Naturally, this added degree of liberty makes the design of the scheduler itself more complex. It is expected that EDF job switching overheads may be slightly longer than FTP job switching overheads.

EDF scheduling is shown to be dominating FTP scheduling on monocore platforms in theory. Buttazzo compared EDF and RM scheduling on a range of theoretical and practical characteristics, essentially concluding that EDF is the optimal choice for monocore scheduling bar very specific requirements (Buttazzo, Giorgio C. "Rate monotonic vs. EDF: judgment day." Real-Time Systems 29.1 (2005): 5-26).

9.3 Multicore scheduling

Modern embedded systems are increasingly complex and feature more computationally intensive tasks than a decade ago. There is also a drive to integrate more functionalities into single systems that were distributed on multiple platforms before, to save on weight and other costs. The industry is evolving towards multicore platforms to respond to these demands. The product vision of HIPPEROS includes providing an efficient framework using these novel embedded multicore platforms to application designers.

The transition from monocore to multicore platforms has profound implications on scheduling. The problem now consists in mapping a subset of the executable jobs to the available cores, instead of just one.

9.3.1 Partitioned multicore scheduling

Scheduling techniques that are said to be partitioned first consider the set of all tasks in the system and partition it into disjoint subsets, each attributed to one core. Then, a monocore scheduling technique is applied to each core in isolation. Jobs from tasks that are attributed to a given core cannot be executed on another core.

9.3.2 Global multicore scheduling

Partitioned scheduling is inherently limited in reactivity and maximum effective utilization by the constraint that tasks cannot move from one core to another. Indeed, a core may have nothing to execute while there are available jobs pegged on another core waiting for that core to finish executing its current job.

The ability of partitioned scheduling to use the platform effectively is limited by the quality of the task partitioning technique. In complex systems with many tasks, finding a good assignment of tasks to cores can be very complicated, the problem being essentially similar to multi-dimensional bin-packing in complexity. In pathological cases (i.e. specific task utilization values), partitioned scheduling may be wasting a significant portion (arbitrarily close to 50%) of the available computational power of a multicore platform.

Global scheduling solves these issues by allowing jobs to migrate from one core to another. A job can be preempted by another job on a given core and resume its execution later on another core.

Both EDF and priority-driven scheduling can be straightforwardly generalized to global scheduling. For both techniques, the global generalization consists in executing the set of ready jobs with the highest priorities (smallest deadlines for EDF). These global techniques are relatively simple to implement, but are known not to be optimal. Figure 3 shows an example of a Global EDF schedule.

Name	WCET	Period
Task 1	2	3
Task 2	6	8

PUBLIC

Task 3	4	12

Table 2: A task set of three tasks and their basic characteristics. The total utilisation of this system is just below two, which means that it is theoretically possible to schedule this system on two cores. This is the system of Table 1 with all execution requirements doubled.

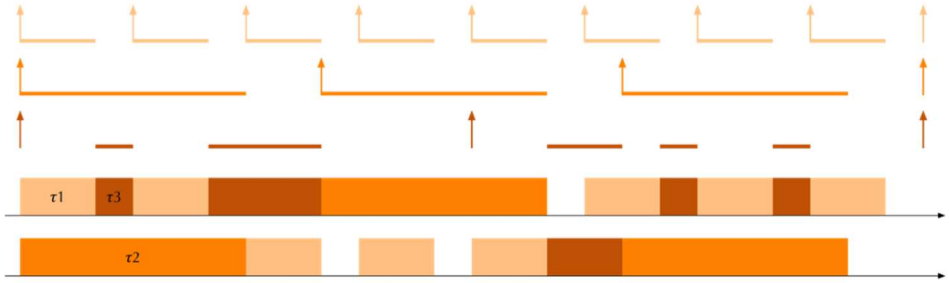

Figure 5 A global EDF schedule.

Figure: This is the Global EDF schedule of the system described in Table 2 on two cores. The job release times are the same as in Figure 2, however the schedule itself is obviously different. The two thick lines in the bottom of the figure show which task if any is executed on each core at every time instant. Note that in this schedule there is a job migration. The second job of Task 3 is preempted by Task 2 after executing two time units and is resumed one time unit later on the other core.

There are a number of more powerful global scheduling algorithms in the literature. However, such algorithms are all vastly more complex, generally require heavy pre-processing and are not optimal either, in the most general case (tasks with unpredictable release patterns).

9.4 Scheduling in HIPPEROS

HIPPEROS is a configurable operating system. Naturally, the scheduler implementation is part of the configurable subsystems of the OS. All HIPPEROS scheduler implementations provide a small standard API to the dispatcher module. This interface essentially lets the scheduler know which jobs are ready and lets the dispatcher ask the scheduler which of these ready jobs should be executed on each core.

There are currently four scheduler implementations available on HIPPEROS. The first two implement priority-driven scheduling in partitioned and global modes. Monocore scheduling and partitioned multicore scheduling use the same scheduler module. The second two schedulers implement EDF scheduling, also in partitioned and global variants.

Global scheduling in HIPPEROS is done through simple generalization of priority-driven and EDF scheduling logic, as described in a previous section.

9.4.1 Task-local scheduling

Each task can spawn a number of threads during execution. By default, all threads of a given task are scheduled according to the priority (or deadline) of the task that spawned it. If there are more than one thread ready for the same task, they are scheduled in FIFO order by default. This process-local scheduling can be changed by using standard API calls. Note that in this regard HIPPEROS differs from Linux: in Linux, changing the priority of a thread can change its priority relative to threads from a different process. This feature of Linux violates the timing isolation of processes and tends to make the design of a whole application slightly more complicated.

9.4.2 Thread scheduling scopes

Restricting all threads to strict task-based scheduling is limiting. In certain cases, an application designer may want to spawn a thread with absolute system-wide priority and a very small workload, to handle events where short latency is required, such as interruptions. Ideally, only the specific thread that needs this high priority should have it. Unlike most OS, HIPPEROS implements a scheduling scope feature which allows to do this without losing the advantages of task-based scheduling for all other threads of that task.

This feature is very simple to use: a thread can be defined to be part of the system scope and given a priority. All threads in the system scope are scheduled with one another as if part of a single, virtual task with absolute priority. It is however not possible to create a system scope thread with a priority lower than that of a process scope thread. This feature is very convenient to implement efficient interrupt handlers.

9.4.3 Scheduling and certifiability

The main advantage of using an RTOS in a certifiable system can be broadly summarized as isolation. One form of necessary isolation for critical systems is time isolation. That is, that one task behaving outside of defined bounds cannot affect the timing constraints of another task, except if they have to synchronize somehow.

The HIPPEROS scheduler ensures that all time-critical tasks meet their deadlines, if that is at all possible with the provided scheduling algorithms. Assuming that the system is not overloaded and task characteristics were correctly estimated, a simple mathematical analysis can tell the user if tasks will miss deadlines or not.

Compared to an embedded platform without any OS, the RTOS scheduler enables users to design each task of their system without paying attention to interferences from unrelated tasks. In a bare-metal system, the tasks would typically be fused into one large loop, forcing frame-based scheduling. HIPPEROS like other RTOS encapsulates each task into its own process making it far easier to develop and maintain. Additionally, this model allows changes in scheduling without affecting the actual code of the tasks. Conversely, the behaviour of a task can be altered without necessarily having to reconsider everything.

10 Virtual Memory Management

10.1 Space isolation

Current embedded systems perform a number of different operations. When using a real-time operating system, these different operations are performed by different user processes. This allows to split the logic between multiple user processes. It also makes the development of these processes easier as it requires less synchronization between the software engineers. In order to make this responsibility split effective, the real-time operating system must guarantee time and space isolation between the processes. Whereas the time isolation is provided by the scheduler (see the corresponding technical note), paging is the way space isolation is provided to processes.

More specifically, by creating a private address space for each process, each of these processes can safely act as if it was the only process in the system. As the complexity of the certification process grows exponentially with the size of an application, the ability to split the application in smaller parts that can mostly be certified independently is of great value. Moreover, paging is also used to protect the critical data of the system (e.g. the kernel memory).

From a safety, reliability and certification point of view, space isolation and therefore paging are must-have features.

10.2 Address spaces

In any processing platform, most of the peripherals, the main memory and the CPU are interconnected through a communication bus. The components connected to that bus exchange information through the bus; they must be able to send their request to a peripheral or another: the destination. When a person wants to send a message to someone, they provide a phone number, an email address or a postal address. Similarly, in hardware each peripheral is assigned an exclusive set of physical addresses. Generally, peripherals have a contiguous range of physical addresses and they react to any operation (read or write) occurring in this range.

The address range given to a peripheral may change from platform to platform. This mapping of physical addresses to peripherals is specified by the memory map of the target platform.

PUBLIC

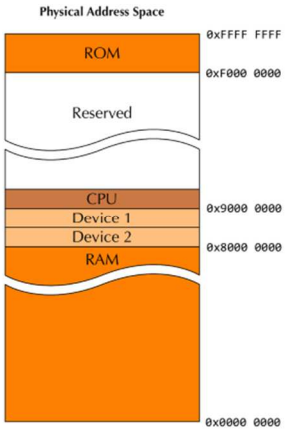

Figure 6 Example of the memory map of a platform.

Figure: We can see that there are 2GB of RAM at the beginning of the physical address space (from 0x00000000 to 0x7FFFFFFF), 2 memory-mapped devices, the CPU configuration register, a reserved address range and 256MB of Read-Only Memory.

One should note that without an additional protection mechanism, every bus observer (which could be a thread running on a processing core) could access the entire address space: it could reconfigure peripherals, modify/read the memory of another thread or even make the system crash by performing illegal operations. In order to prevent the failure of one process impacting the entire system, the concept of virtual memory has been introduced.

The idea of the virtual address space is to place a filter between an observer (code executing on a processing core) and the interconnection bus. This filter can be used to:

- Forbid access to a range of physical addresses.
- Only allow certain types of accesses to some part of the memory (read-only, not executable...).
- Give different access rights for privileged/unprivileged code.
- Give the illusion to a process that it owns the entire address space.
- Give access to the physical address X through the virtual address Y.

PUBLIC

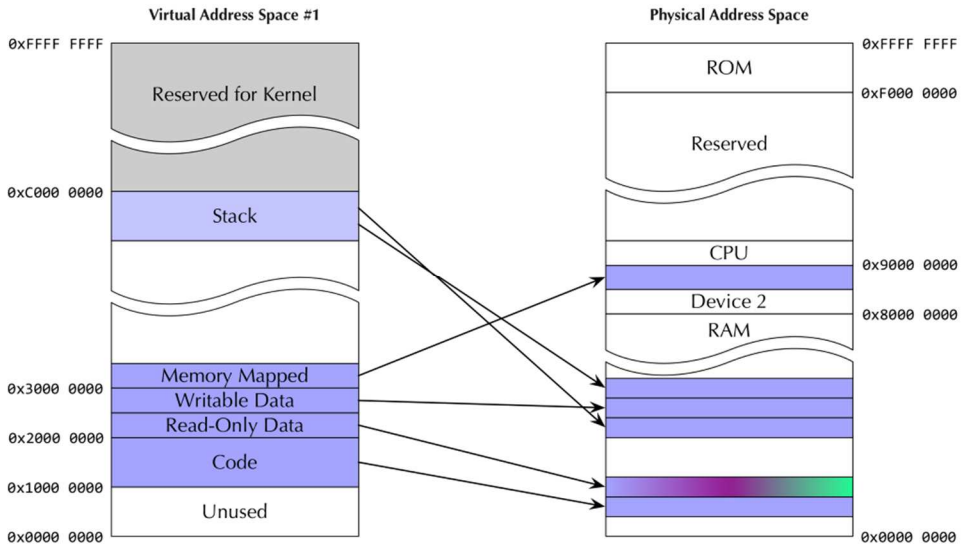

Figure 7 A possible mapping from a virtual address space of a process to a physical address space.

This filter is interchangeable at run-time and we can therefore provide a different filter for each process. Each process has then its own virtual address space while there is still one and only one physical address space.

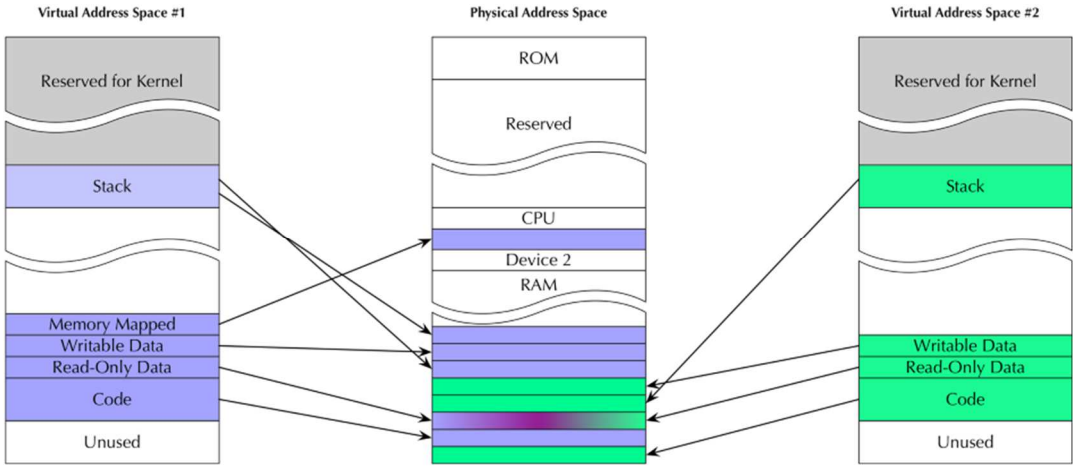

Figure 8 Two virtual address spaces.

Figure: with 2 different virtual address spaces, we can have 2 different views of the physical address space. Nothing prevents to share the same physical memory from 2 different virtual address spaces (to implement inter-process communication for example).

10.3 Paging in HIPPEROS

Modern processors provide space isolation through the usage of a memory management unit (MMU), each processing core having one. Therefore, HIPPEROS relies on the existence of a MMU to configure virtual address spaces.

In such systems, the peripherals always have exclusive address ranges that are some whole number of pages wide. The size of a page is generally 4KB. This constraint allows to divide the physical memory in page frames where the size of each frame is equal to the page size. The virtual address space is itself divided in pages where the size of the smallest pages is also equal to the page size (thus the name).

The size of a virtual address space is huge (232 bits or 4GB on 32-bits architectures) compared to the page size. Consequently, it would be expensive to configure the access to each page frame independently for each virtual address space. This is why different methods are supported by the hardware and implemented in HIPPEROS to reduce the cost of the paging. The main method relies on the possibility to configure pages of different sizes: With larger pages, we need fewer of them to cover an address space.

Most of the memory management units rely on hierarchic page tables to cover the entire virtual address space at smaller costs. Each entry (called a page table entry) at a certain level in hierarchy can either directly map a physical address or reference a page table of the next level. The number of maximum paging indirection levels depends on the address space (2 or 3 levels for 32-bits architectures, 4 to 5 levels for 64-bits architectures). The HIPPEROS mapping algorithm uses the least possible page table entries to satisfy every mapping request.

PUBLIC

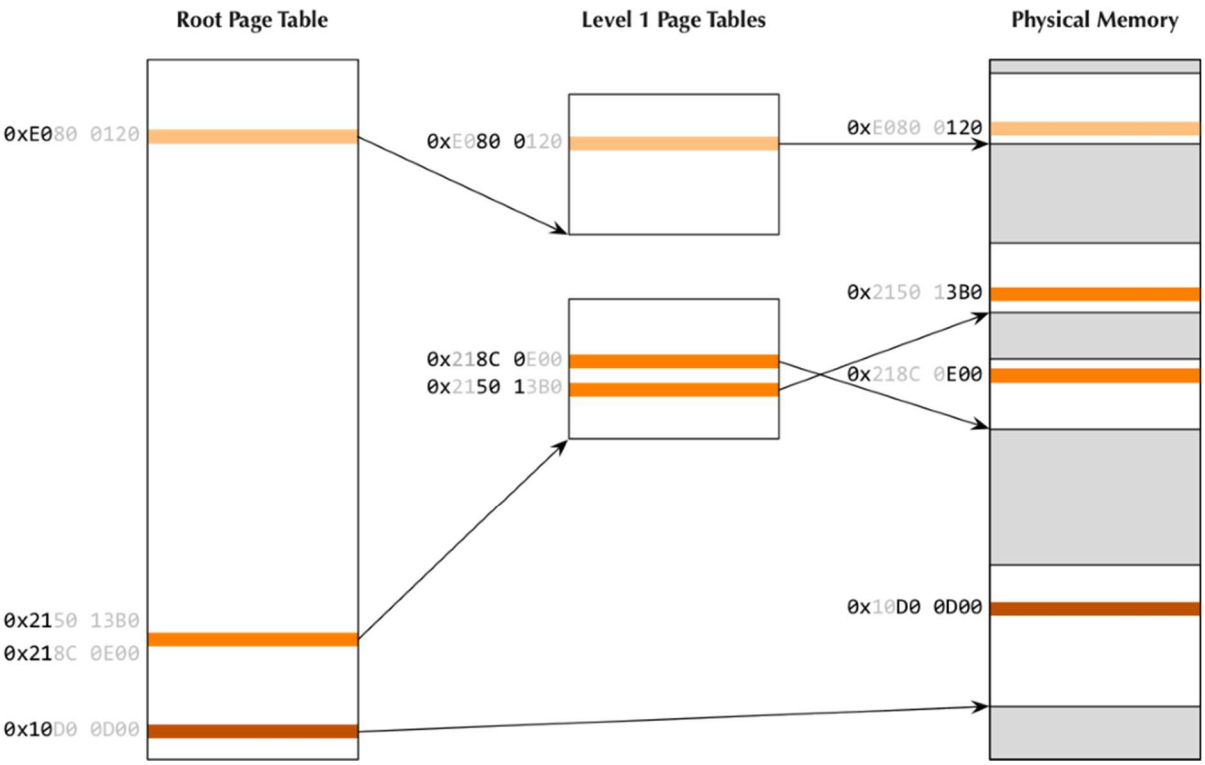

Figure 9 Multi-level paging as implemented in HIPPEROS.

On architectures relying on page tables, the MMU is configured by giving the physical address of the root page table. Switching to another virtual address space is done by giving the address of another page table. Only privileged code (the kernel) can configure the MMU.

10.3.1 Access rights

HIPPEROS heavily uses the access rights possibilities given by the paging mechanism to protect the kernel from user processes and to protect user processes from other user processes.

In particular:

- Only pages mapping code are marked as executable. This means that it is impossible for a malicious/bogus process to modify the code.
- The constant data (that must not be modified) is mapped to read-only pages.
- The kernel resides in high memory and it is mapped as only accessible from privileged code. Consequently, no user process can access (read or write) kernel memory.
- Each process has its own address space. Consequently, processes are isolated.

10.3.2 Performance

Page tables are big structures stored in the main memory. As the code always uses virtual addressing, this means that the MMU has to translate virtual addresses to physical addresses on every CPU instruction. This translation takes some time. To alleviate this issue, the MMU stores some virtual-to-physical translations in order to reduce the performance penalty. This information is stored in a small cache called the TLB (Translation Lookaside Buffer). In some situations, some maintenance operations must be performed on the TLB. The kernel also ensures the coherency of the TLB in multi-core platforms.

The penalty then consists of:

- A loss of usable physical memory for the application, as the page tables take some space.
- CPU cycles spent to load the page table entries.
- CPU cycles spent for TLB maintenance.
- CPU cycles spent to change the active page table.

10.4 Conclusions

Paging provides an efficient way to provide space isolation. The penalty of using paging is negligible compared to the benefits it brings. In the context of certifiable systems, this feature is a must-have and has no alternative on modern platforms. In particular, it allows to increase the reliability of the system and to reduce the certification cost.

PUBLIC

11 Inter-Process Communication

Using a real-time operating system to design embedded systems provides several serious advantages against designing the system in a bare-metal environment. One of such advantages is the ability to easily split the different functionalities of the system in several tasks (processes). This allows to develop them and certify them (mostly) independently. However sometimes the tasks still have to communicate between them: this is called inter-process communication.

HIPPEROS is a micro-kernel, where the least possible amount of code is run in a privileged mode. In an operating system with a micro-kernel, most of the code handling the devices (called drivers) is isolated in independent system processes. When a user process wants to perform a request on a device, it needs to communicate with the driver responsible for that device using the aforementioned inter-process communication feature of the OS.

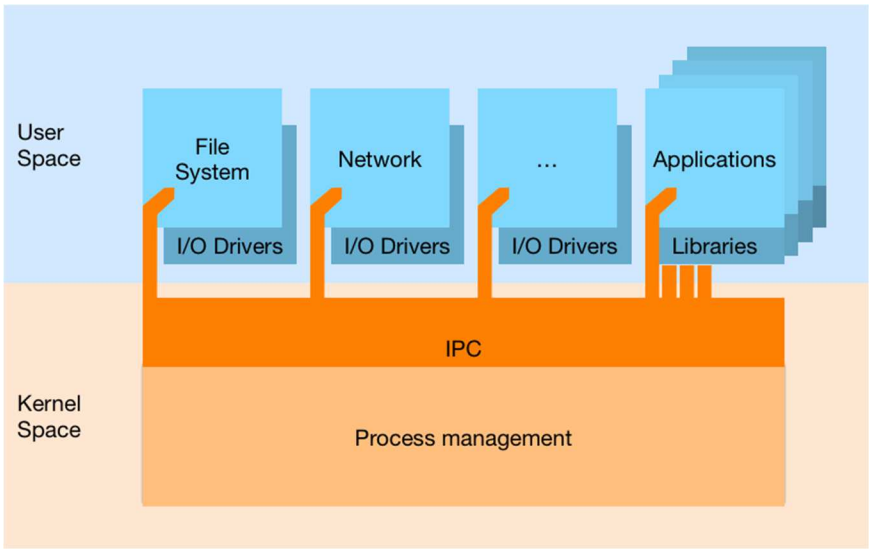

Figure 10 Illustration of how IPC allows different use space processes to interact together.

There are 2 different types of inter-process communication supported in HIPPEROS:

- Synchronous IPC which uses a shared-memory mechanism for one-to-one communication.
- Copy-Buffer IPC which uses in-kernel buffers for many-to-one communication.

Both these mechanisms can be used transparently, in a mono-core or multi-core environment. This simplifies the design of user applications.

11.1 Synchronous IPC

Synchronous IPC channels are used when two processes need to exchange data. This mechanism uses memory that is made temporarily available to a process using the paging mechanism. This allows to ensure that:

- Other processes do not have access to the exchanged data.
- The sender process that initially wrote the data does not modify it after having sent the data.
- The receiver process that reads the data only starts to read it after the sender finished writing.

The feature is made of 4 actions, 2 for the sender and 2 for the receiver:

- Get the buffer of a channel in write mode (sender).
- Notify the receiver that there is data to be read on the channel (sender).
- Get the buffer of the channel in read mode (receiver).
- Notify the sender that the data of the channel has been read (receiver).

Two processes that communicate through synchronous IPC can successively be sender and then receiver (or the opposite). For that they could either use two different channels, or only use one channel and exchange their role (that should be part of their communication protocol).

Synchronous IPC transfers information between processes through shared memory (that is, modifying page permissions) rather than copying. This means that this is an efficient way to transfer big chunks of data quickly. The synchronisation primitives add overhead but ensure the data coherency of the accesses, which is necessary for almost any use.

11.2 Copy-Buffer IPC

Synchronous IPC is an efficient mechanism when it comes to communication between two processes. However, sometimes many different processes need to communicate with one specific process (for example a driver). In that situation, a more efficient way to handle this communication is to use the copy-buffer IPC feature of HIPPEROS. Copy-buffer IPC allows many-to-one communication through communication channels. A channel can have at most one receiver, but many processes can send messages to that channel. When a message is sent over a channel, the receiver is notified, and it is able to read the message and to retrieve information about the sender. This last information can be used by the receiver to answer the sender, using a different channel. This is used by servers in client-server architectures. Typically, drivers act as servers in copy-buffer IPC.

The integrity of the message is guaranteed by the kernel, but this has a cost: the data is copied first from user space to the kernel, and then again from the kernel to user space. The size of these in-kernel buffers is limited and that may no suit all types of communication, such as long blobs of data. However, nothing prevents processes to use copy-buffer IPC to initiate the communication between 2 processes and then to use synchronous IPC to exchange large chunks of data if need be.

11.3 Interface

The API for copy-buffer IPC is inspired by the POSIX API for UDP communications. Processes open channel descriptor, they can then bind it to a channel, and then they can receive or send messages using the channel descriptor.

In order to use the copy-buffer IPC mechanism, a thread must first obtain a channel descriptor and then bind it to a channel. This will allow the thread to receive the messages sent to that channel but also to send message to other channels. When a thread receives a message from a channel, it also gets information about the sender so that it can answer to it. Once a thread has finished to use a channel, it must close its descriptor.

When a thread wants to receive a message from a message but that no messages are available, the thread is blocked until a message is available. This way, a server can provide functionalities to its clients without consuming the resources, that can then be reused by other processes.

The same interface is available for both mono-core and multi-core builds of HIPPEROS. There are however some restrictions when using the copy-buffer IPC in multi-core in order to increase the reliability of the system. The main restriction is that some of these operations (open of channel descriptors and their binding) must always be performed by the master thread (which is the first thread executed when a process is launched).

Inter-process communication in itself is one of the main building blocks of micro-kernels.

12 Mixed-Criticality

Mixed-Criticality applications are primarily concerned with the isolation of tasks. A multitude of hardware and software components in a real-time system are shared between tasks and can potentially lead to less critical tasks interfering with more critical tasks. In a RTOS, ensuring mixed criticality operation means that the OS ensures as much temporal isolation between tasks as possible across criticality levels. In other words, while temporal isolation between any two tasks is already an important feature of RTOS design, additional steps must be taken to ensure that less critical tasks cannot cause more critical tasks to fail. Of course, there may be interferences caused by hardware constraints or application behaviour that the OS cannot eliminate or mitigate and that cannot be accurately accounted for.

As such, the challenges resulting from the use of modern general-purpose platforms in the context of mixed criticality real-time systems must be faced using an array of techniques ranging from application design to hardware design, including RTOS design. This document describes a set of OS level solutions aimed at temporal isolation and efficient use of resources in mixed criticality applications.

Together, it is expected that these techniques allow HIPPEROS to functionally support a wide range of mixed criticality applications with efficient use of processor resources. However, the challenges posed by temporal isolation are much more difficult to tackle at the software level and it is also usually difficult to evaluate the impact of isolation techniques on a real platform.

This document begins with a short state of the art on mixed criticality in RTOS. Then a description of the relevant core features of HIPPEROS is given. The third chapter discusses how this design can be used in applications.

12.1 State of the art

12.1.1 Scheduling

Industrial applications involving mixed criticality tasks usually isolate criticality levels by splitting them between multiple separate platforms. Alternatively, a single multi-core platform can be partitioned by allocating a number of cores to each criticality level. However, this approach causes concerns in terms of isolation as cores can potentially interfere with each other through shared memory, caches and buses.

12.1.2 Time and space isolation

Perhaps the most critical component in terms of timing unpredictability, hardware takes a major place in mixed criticality literature.

In particular, steps have to be taken to mitigate cache interferences between tasks. However due to the often nonstandard and poorly documented nature of cache architectures on modern platforms, the OS only has limited means to achieve this mitigation. Typical simple techniques to deal with this problem are to disable caches altogether and, stated more generically, to partition cache space between tasks statically. Obviously, both are undesirable from a performance standpoint and lead to severe under-utilisation of the platform's computing power in certain cases. Nevertheless, for the most critical functions such methods may be necessary.

The system's memory allocation mechanism must also be aware of memory banks and buses in order to avoid unnecessary contention.

12.1.3 Software resources

In addition to the hardware resources, the OS and services also manage software resources such as those necessary for paging, queues of system calls and IPC messages. In general, any structure that requires synchronisation between tasks is a cause of concern for the potential implications in terms of temporal dependencies.

Again, partitioning is the preferred approach to these issues in the industry. Memory mapping is made static, and multiple kernels are run independently on the same platform each with its own structures and set of tasks. Finally, the use of pure software services is also either partitioned or time-sliced between tasks or criticality levels. Hypervisors are sometimes employed to enforce strict isolation when the OS doesn't offer such functionality. In that case, multiple instances of the OS are run alongside each other on one platform and the hypervisor provides an additional layer of resource (memory and CPU) allocation.

12.1.4 Multi-WCET execution

Academic works have been produced on techniques aimed at mitigating the impact of highly critical tasks upon platform utilisation. The theory derives from the observation that the strict static techniques used to evaluate the WCET of critical tasks give results that are usually considerably higher than the average execution time of such tasks or even the highest observed. In particular, there is a difference between the WCET evaluation methods used for critical tasks and "non-critical" (but still real-time) tasks.

In practice, a real-time system can implement multiple modes of execution and switch between them according to the status of a critical task after some execution time threshold has been met. Under normal circumstances, the system is utilised efficiently allowing low criticality tasks a lot of computing power. In the event that a critical task has not completed before a given execution time threshold, the system enters another mode where low criticality tasks receive less computing power. Naturally, such techniques only make sense if critical and non-critical tasks are mixed on the same core. Analysis techniques exist to keep the loss of computing power of non-critical tasks to a minimum.

12.2 Mixed-Criticality in HIPPEROS

This section covers the design of the mixed criticality solution implemented in the HIPPEROS kernel. The use of mixed-criticality affects the behaviour of the OS in specific, well-defined components. Namely, the scheduler, the task description system, the system call layer and the event handler.

In relation to the state of the art, this chapter goes into the details of how the multi-WCET and related techniques for mixed criticality scheduling are put into practice in the HIPPEROS kernel.

12.2.1 Criticality levels

The system supports three types of tasks: highly critical tasks, real-time tasks and best effort tasks. Best effort tasks are free to be released at any rate and to consume an arbitrary amount of computation time. However, they will be considered with the lowest priorities by the scheduler and may therefore not get the resources needed to perform their purpose. Such a system makes the design of long-running best effort tasks relatively easy.

Real-time tasks have design-time defined periods (or inter-arrival times), deadlines and worst-case execution times. The scheduler will pre-empt best-effort tasks to execute real-time tasks to completion. Real-time tasks must be divided into two sub-categories: compressible and incompressible. Compressible real-time tasks allow the OS to reduce their arrival rate (increase the periods) within pre-defined limits, therefore allowing some form of QoS adjustment. If the compressible task is a sensor reading, the readings will simply be less frequent while the task is in a compressed state. Incompressible real-time tasks simply do not allow such changes.

Finally, highly critical tasks have the characteristics of incompressible real-time tasks but additionally provide an intermediate execution time limit between zero and their worst-case execution time. The use of this intermediate execution time limit is described in the section on mode switching.

Criticality and priority are different concepts. To guarantee the deadlines of regular real-time tasks, some highly critical tasks may have lower priority than some regular real-time tasks. In this sense,

priority is merely a tool to ensure that the proper scheduling decisions are taken to meet all deadlines. Criticality is a degree of assurance that temporal (and other) constraints will be met for a given task.

12.2.2 Mode switch

Two modes of operation are defined. Under normal circumstances, the system uses all resources as efficiently as possible. Best effort tasks are allowed to run with limited constraints and compressible real-time tasks are released at the nominal rate. If a highly critical task reaches its intermediate execution time limit, then the system enters a critical mode of execution.

In critical mode, best effort tasks may be terminated and compressible real-time tasks have longer periods. A longer period may mean that a process from a compressible real-time task currently ready to be executed could be terminated, depending on the maximum guaranteed delay between two activations. This mechanism is known in the literature as elastic mixed-criticality scheduling.

As a result, there is more available computation time and fewer interferences on highly critical tasks from the other tasks. This design achieves the double goal of providing highly critical tasks with the assurance that they will meet their deadlines even in the event of extremely long execution times and allowing other tasks to use the otherwise wasted resources when execution times are closer to the average. In particularly extreme cases of pessimism in the WCET evaluation of highly critical tasks, it might be that a running system in its production environment never switches to critical mode.

When the system enters high criticality mode, it is expected that this should only be a temporary state. The safest time to go back to normal mode is when the entire system is idle (that is, there are no active jobs). This is the condition used in HIPPEROS, and the user has to ensure that such an idle time will actually happen, otherwise the system will remain in high criticality mode indefinitely.

12.2.3 Mixed-Criticality overheads

The implementation of the mixed-criticality feature in the HIPPEROS kernel requires the use of which check for execution time overruns (to determine when a criticality mode switch must occur). HIPPEROS timeguards are lightweight objects that affect timing in an essentially negligible capacity which can be mitigated the same as task-switching overhead, because it is easily predictable.

The overhead of the mode switch itself is more important. This overhead is linear with the number of threads in the system. Because mode switches are rare events and the time they take is very predictable, this overhead is easily mitigated by provisioning this time on the intermediate and total execution time limits of high criticality tasks. Note that the switch back from high criticality mode to low criticality does not actually impact highly critical tasks. Indeed, it can only happen when the system becomes idle in high criticality mode.

12.3 Mixed-Criticality and certifiability

Without mixed-criticality it is effectively not allowed to mix user programs of multiple assurance levels into a single platform without elevating the certification level of every piece of that system to the highest certification requirement of the group. The mixed-criticality implementation in HIPPEROS aims to relax that requirement by offering techniques to enforce maximum temporal isolation between tasks of different levels if need be. This is done where necessary, without entirely giving up on efficient use of the platform.

In addition, the HIPPEROS multi-WCET mixed-criticality solution is particularly useful when the certification level of a given task imposes strict static WCET measurement methods. Such methods might over-estimate the actual WCET of the application by one or even several orders of magnitude. Figure 1 illustrates the difference between WCET evaluation methods. In a system without the mixed-criticality feature, the reservation has to be made regardless. In a mixed-criticality system like HIPPEROS, an intermediate "WCET" value can be chosen to increase the processor time available to other tasks. The system ensures that if that intermediate execution time limit is indeed reached, the rest of the tasks will be affected in a way that guarantees that the critical task will complete its execution regardless.

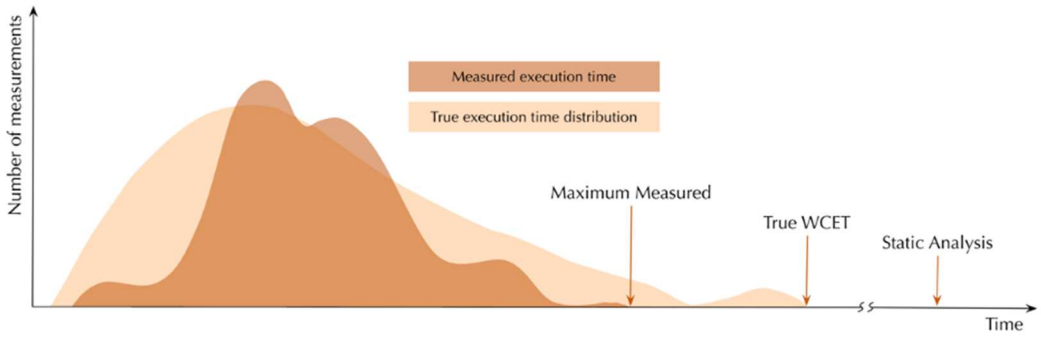

Figure 11 Effective measured execution times in relation to actual WCET.

Figure: The more stringent the WCET evaluation method, the more overprovisioning is necessary. Static WCET Analysis is used in very critical applications and results in extreme pessimism.

The core principle that governed the design of the mixed-criticality feature of HIPPEROS is flexibility. It is up to the user to choose which way to handle critical mode is appropriate for each task. That may involve immediately killing all tasks that are not maximally critical whenever the system enters critical mode, simply letting them go and increasing periods or any variation in between.

13 Platform Support

The HIPPEROS kernel is fully compatible with the ARMv7-A ISA and partially compatible with the ARMv8-A ISA. The following sections describe the current state of the port.

13.1 ARMv7-A

13.1.1 Port status

The HIPPEROS kernel is fully compatible with the ARMv7-A architecture and has been tested fully on multi-core Cortex-A9 processors.

13.1.2 Interrupts

HIPPEROS supports the ARM Global Interrupt Controller (GIC) version 1. Several ARM timer models are supported (Private Timer, Global Timer, several SoC-specific timers). The global timer is useful for dealing with multi-core systems.

13.1.3 Multi-core

HIPPEROS supports multi-core Cortex-A9 systems, up to 4 cores (the maximum number of cores using this architecture).

13.1.4 Memory management

The HIPPEROS kernel supports full memory virtualisation using 2-level page tables. The supported memory models are:

- No MMU: bare-metal operating system without memory protection.
- Single Page Table: common address space for all the user applications. There is memory protection between the applications and the kernel but each application can access the address range of others.
- Full virtualisation: each application is compiled and linked as a single ELF file. Each application has its own address space and has no means of accessing the memory used by another application. The kernel is fully protected from user access.

L1 caches are supported.

13.1.5 Supported targets

HIPPEROS runs on the following Cortex-A9 SoCs:

- Freescale i.MX6Q
- Xilinx Zynq-7000
- Texas Instruments OMAP4

Deliverable 3.1 describes in detail the implementation of the support for the Xilinx Zynq-7000 board.

13.1.6 Current/future developments

The support for ARMv7-A is mature. Only developments required for new projects will be considered (new SoCs, new processors, etc).

13.1.7 Dynamic partial reconfiguration on PL

The Zynq-7000 family provides an FPGA block inside the SoC called Programming Logic (PL). HIPPEROS provides support for the dynamic (partial) reconfiguration of the FPGA at run-time which offers the flexibility to reconfigure a design with different Reconfigurable Modules (RMs) at runtime reusing the same hardware resources on the FPGA floorplan. This is particularly useful when running multiple computation-intensive applications in parallel or when the size of the FPGA is limited. Therefore, the design can fit on a smaller FPGA, which results in lower power consumption and costs. DPR is still experimental at this point but the full reconfiguration through the PCAP port during run-time is fully functional. For more details see Deliverable D3.3.

13.2 ARMv8-A

13.2.1 Port status

The HIPPEROS kernel runs on a Cortex-A53 target. HIPPEROS supports the Xilinx UltraScale+ (on the TE0820 board provided by SUNDANCE).

13.2.2 Interrupts

Interrupts are functional, using the ARM GICv1 & GICv2.

13.2.3 Multi-core

The kernel port only supports single core processors for now.

13.2.4 Memory management

Only the No MMU memory manager is supported at the moment.

13.3 Future developments

We intend to support the other architectures present on the UltraScale+ platform, including the real-time cores.

13.4 Power management

There are mainly two ways of implementing power management at operating system and kernel level:

- Dynamic Voltage and Frequency Scaling (DVFS): this feature consists in changing the operating voltage and processor frequency at run-time, allowing to trade power against performance as required by the current workload. Raising the frequency allows to execute more CPU cycles per second but requires raising the voltage level powering the CPU. Voltage and frequency values have a significant contribution in the power dissipation (and then the energy consumption) of the target processor.

- Dynamic Power Management (DPM): this feature consists mainly in deactivating the part of the circuit that are not used for computation. For example, if only one CPU core is used by the running application, the other cores can be disabled therefore cancelling their contributions to the overall power consumption of the chip.

To support these two power management policies, low-level kernel drivers must be implemented to change the configuration of Phase-Locked Loop (PLL) circuits and voltage regulators. These drivers are highly platform-specific and require specific implementation in privileged mode (kernel).

In HIPPEROS, DVFS is supported on the imx6q platform. The Zynq-7000 allows to select a frequency at boot time, but the chip does not allow to provide both DVFS **and** real-time guarantees. The operating point is then selected offline and remains the same during run-time. Due to this case, TUD's team developed these low power techniques based on FreeRTOS to keep the soft real-time constraints of the application. For more details, go to chapter 15 at page 47.

DPM is implemented in HIPPEROS on ARMv7a through hardware-automate multi-core sleep states, triggered by specific instructions (I.e. infinite loop, idle state, Wait For Interrupts (WFI) instructions).

Power management is currently not supported on the Zynq UltraScale+ platform.

When these low-level power management primitives are available, the kernel is able to *use* them through “governors”, which are module responsible to the decisions regarding power management. These modules are heavily coupled to the scheduler, which as presented earlier in this deliverable is responsible to decide which task runs on which core and at what moment. These governors enabled to set the running speed of a task and must be considered when analysing the timing behaviour of the system.

13.5 Heterogeneous hardware and dynamic reconfiguration

Both Zynq architectures provide an interesting feature: the on-board CPU is coupled with some FPGA area, that is reconfigurable from the CPU during run-time. This allows to implement scheme such as Dynamic Partial Reconfiguration (DPR).

To be able to load a bitstream into the FPGA from the CPU, the system must provide device support for the Processor Configuration Access Port (PCAP). The HIPPEROS-TULIPP distribution comes with the PCAP driver built-in directly usable: a user can put its collection of static bitstreams into the SD card of the target board and load these bitstreams at run-time using the driver. For *partial* reconfiguration, HIPPEROS only provides experimental support.

Using hardware accelerated modules (running in the FPGA) is another way to design low-power systems. Indeed, by massively exploiting the inherent parallel architecture of FPGAs, these designs have a very good *performance per watt* ratio, allowing to design high performance yet low-power applications.

14 Release Process and Deliveries

The HIPPEROS development team, in charge of the kernel and RTOS code base, applies an agile software lifecycle methodology. The development is articulated around target releases, that are composed of a target release date and a set of features to prototype, design, develop, test, document and integrate.

HIPPEROS created RTOS packages specific for the TULIPP platform. We encoded the technical requirements related to it and implemented the TULIPP RTOS distribution. This package has the particularity to be configured with full memory isolation and ELF tasks (the kernel is independent from the files). There are two HIPPEROS distributions for the TULIPP OS instance: one for each supported platform:

- Zynq-7000, based on the ARMv7a architecture;
- Zynq UltraScale+, based on the ARMv8 architecture.

HIPPEROS has a mature shipping process to create and document usable package.

The following releases were distributed to the TULIPP consortium and to the members of the Advisory Board:

- 17.02
- 17.04
- 17.06
- 17.09 and its associated hotfix, 17.09.1
- 17.12
- 18.03
- 18.05
- 18.06

Each release date correspond to the version number (year.month) and the releases used during both integration meetings were 17.06 (M17, June 2017) and 18.06 (M29, June 2018). The TULIPP-HIPPEROS RTOS was shipped to the following partners: FRA, NTNU, SUN, SYN, THL, TUD/RUB and the following ABers:

- Adiuvo Engineering
- Evidence
- Ikerlan

PUBLIC

- Inesctec
- Think Silicon
- Univaq
- Utia CAS

15 Dynamic Voltage and Frequency Scaling

The Dynamic Voltage and Frequency Scaling (DVFS) is a low power technique to reduce the power and as a consequence the energy consumption of the entire system. This method scales the core frequency and the supply voltage of the hardware. DVFS allows devices to perform needed tasks with the minimum amount of required power.

There is a proportionality between the clock frequency and the power of the core: higher frequencies effect higher proportional power consumption of the processor. Actually, it exists a square correlation between the supply voltage and the power consumption. Reducing the operating voltage thus has a significantly lower power consumption. If both parameters, the clock frequency and the operating voltage are simultaneously decreased, the reduction of the power consumption is even in the third power. These findings are implemented in the DVFS technique by defining the maximum possible clock frequency and operating voltage for the processors and controlling them during operation.

In this chapter, we describe a software solution to implement DVFS and power measurement using FreeRTOS and Xilinx SDSoC on ZC702, as this platform provides the needed voltage regulators. The approach can be easily ported to similar platforms (as mentioned in section 13.4).

15.1 Dynamic Voltage Scaling

The Dynamic Voltage Scaling (DVS) is a power management technique where the operating voltage used by the hardware is modified during runtime depending upon circumstances and on the needed performance of the application. DVS to increase voltage is known as overvolting and decreasing voltage is known as undervolting. Undervolting is done in order to reduce power consumption where energy comes from a battery and thus is limited, or in rare cases, to increase reliability. Lowering the voltage of the circuit results in a decrease of the dynamic and leakage current. Although, it also increases the gate delay. The idea is to scale voltage as low as possible and maintain a reliable performance of the design [1]. As a benefit, undervolting leads to a reduction of temperature and cooling requirements, and possibly allowing hardware like a fan to be omitted. As opposed to this overvolting is done in order to increase computer performance.

Many different implementations have already been done in this field to improve the power consumption of FPGAs. A software and hardware method for DVS is described in [2]. A complete PL-side setup, using MicroBlaze IP, Dual-Port RAM (DPRAM) and I²C was defined in order to scale the voltage and perform power monitoring. In [3], the importance of Logic Delay Measurement Circuit (LDMC) is described. First, voltage scaling is done and then a suitable frequency is selected on which

the system should operate. Consequently, gate delay can be adjusted. [4] present a self-adaptive voltage control in order to solve the problems of implementing DVFS for low power FPGAs.

15.1.1 Implementation

During the development stage of an application, it is required to optimize the power dissipation of the hardware and improve the performance of the HW/SW-Codesign. This dissipation can be changed through appropriate DVS-technique at runtime. Xilinx Zynq-7000 SoC ZC702 (EK-Z7-ZC702-G) provides suitable hardware components to measure and control the power dynamically. The SoC communicates with the power controller through the Power Management Bus (PMBus) using the I²C interface. The core and auxiliary voltages are supplied to the board using power regulators. An application ensures from the PS-side that the desired power is achieved. Furthermore, it monitors dynamically the power dissipation of the reconfigurable hardware [9].

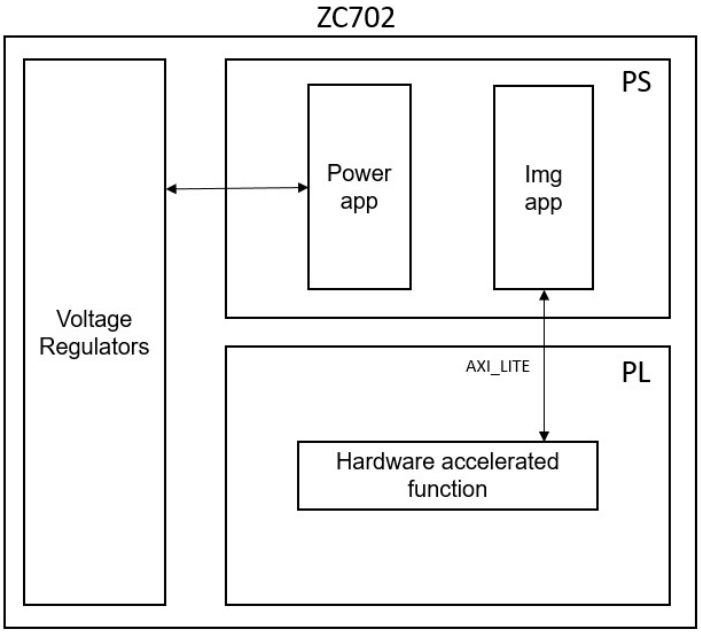

Figure 12 System Overview. Power app: application for DVS and power measurement, Img app: Image processing application with hardware accelerated function on PL

The application is built on two main tasks. The first one monitors the power dissipation and controls dynamically the voltage scaling of the PL-side. The second one contains an image processing application. Xilinx SDSoC permits only to use one of the dual-core processors so that an operating system was required to schedule both functions. FreeRTOS is also deployed to keep the soft real-time requirement of the application. A pre-emptive priority-based scheduler, guarantees that either the power monitoring task or the image processing application runs at each clock cycle. The power

monitoring (described as Task 1) has a higher priority than the image processing application (described as Task 2). At the beginning, Task 1 starts to measure power of PL. Then it is blocked by a delay of approximately 95ms. The scheduler enables the execution of the Task 2 in that delay. Both tasks are running periodically and are executed periodically after every approximately 360ms. Task 2 is pre-empted multiple times by Task 1 to provide power measurement from the PL-side. Table 1 shows the overview of the task scheduling. A median filter is running on the PL-side in Task 2. Xilinx SDSoC allows to accelerate functions of the software application into the PL-side of SoC. An overview of the implemented hardware architecture is shown in Figure 12. For this project, the most interesting part is acceleration of the image processing application in hardware. In this case, voltage scaling and power measurement of the PL was done. This improved the total measurement time of power consumption by 14 times as compared to measuring all voltage rails, due to less computation. The voltage rails of the board allow to measure voltage and current of each rail supplied to the board. Voltage scaling of PL was adapted by setting the voltage level in software and can be set dynamically.

Table 1 Overview of scheduled tasks in FreeRTOS on one ARM-core

Application Function	Execution Time [ms]
Power Monitoring	262.05
Image Processing Application	95.32

15.1.2 Evaluation

Table 2 Resource utilization of the median filter

Resource	Used	Total	% Utilization
DSP	0	220	0
BRAM	18	140	12.86
LUT	9842	53200	18.5
FF	11984	105400	11.26

DVS was tested by scaling the voltage to different levels dynamically and monitoring the power consumption of the PL. The image processing function used for evaluation of DVS is a 3x3 Median Filter for 1920x1080 16-bit images. It is processing 1-pixel per clock cycle (plus overhead). The resource

utilization of hardware generated by SDSoC for filter median moved to the PL-side (operating at 100 MHz) is shown in Table 2. The voltage of normal execution of the image processing application without any voltage scaling was approximately 1.0V and was taken as reference. For evaluation, different voltage levels were set at 0.85V, 0.90V and 0.95V. Average power consumption after voltage scaling was measured and compared with power consumption of the normal operation.

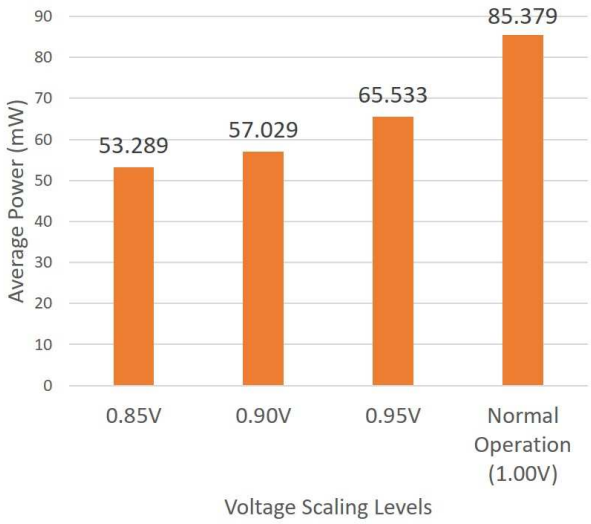

Figure 13 Average power consumption

Figure 13 shows the results of the test. It can be noted from the chart that power consumption with scaling is lower as compared with normal execution of image processing application. For 0.95V scaling, power consumption is reduced by 23.2%, for 0.90 V scaling, it is reduced by 33.2% and for 0.85V scaling, it is reduced by 37.6% as compared to normal execution without any scaling. It was also interesting to note the execution time of the median filter. With hardware acceleration, the median filter executed at an average time of 20.880 ms and execution on software took an average of 653.785 ms. This means that with hardware acceleration the median filter executed 31 times faster as compared to software execution. DVS results show that power consumption can be reduced up to 37% using voltage scaling.

15.2 Dynamic Frequency Scaling

Dynamic frequency scaling (DFS) is a technique whereby the frequency of the hardware can be adjusted during runtime depending on the actual needs and circumstances, to conserve power of the

system and reduce the amount of heat generate by the SoC [6,7]. DFS helps preserving energy storages, decreases cooling cost and is useful as security measure for overheated systems. In [8] a hardware solution is presented. In this part, DFS was implemented on ZC702 Xilinx board and controlled through software using FreeRTOS and a custom platform on Xilinx SDSoC. Figure 14

15.2.1 Implementation

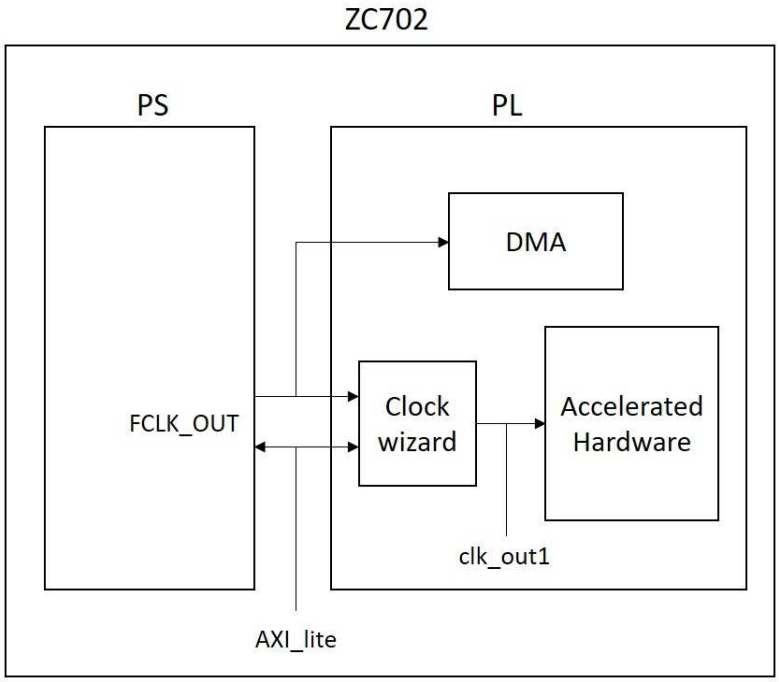

Figure 14 Block diagram of dynamic frequency scaling

A custom platform in SDSoC was implemented in order to implement DFS. This custom platform included a re-configurable clock wizard in the design and I²C bus was enabled in the processing system block (ARM). I²C bus was enabled so DVS and power management application can be executed on this platform. DVS and DFS applications are both independent from each other and both are controlled via software. Re-configurable clock wizard was used to change frequency of a hardware accelerator in PL for image processing application, which was used as a test case. Re-configurable clock wizard allows to change frequency of the clocks, defined by the user at runtime. This can be achieved by setting clock reconfiguration registers [5]. The frequency for DMA is set at 133.33344 MHz and is set from processing system block (ARM) and only the frequency of the hardware accelerator is dynamically changed through clock wizard.

DFS was defined as a separate task along with voltage scaling and power monitoring application and image processing application. When frequency scaling is required, the priority of DFS task is created and its priority is set as the highest, so it start to execute. First, the image processing task is deleted and then clock reconfiguration registers are set for the required frequency. Finally, a new image processing task is created which will execute at the defined frequency and the priority of DFS task is set lower then image processing application and power monitoring application.

15.2.2 Evaluation

For Evaluation of DFS, an image processing application was used with one function (median filter) implemented as hardware accelerator on PL-side. This hardware accelerator was executed on different frequencies, which were changed dynamically and the power of PL was measured using the power management application. Figure 15 shows the average power of the PL under different frequencies set for hardware accelerator.

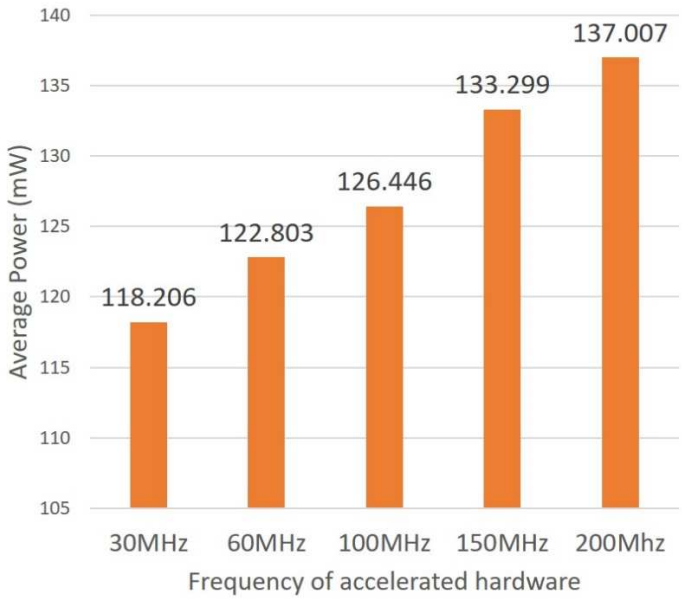

Figure 15 Average power of PL with different frequencies for the hardware accelerator using DFS (PL average voltage is 1.0V)

Normal execution frequency for the system is 60MHz. From Figure 15, we can see that with frequency scaling of 30MHz, the power consumption is reduced by 3.74% with respect to 60MHz. The image processing function used for evaluation of DVS is a 3x3 Median Filter for 1920x1080 64-bit images. The resource utilization of the filter median moved to the PL-side (operating at 60 MHz) is shown in Table 3.

Table 3 Resource utilization of image processing hardware in PL.

Resource	Used	Total	% Utilization
DSP	0	220	0
BRAM	18	140	12.86
LUT	12656	53200	23.79
FF	13199	105400	12.41

15.3 Evaluation of DVFS

Evaluation of both DVS and DFS was also done based on different scenarios. Voltage scaling with 0.90, 0.95 and 1.0 volts was done along with frequency scaling at 30MHz, 60MHz, 100MHz and 150MHz. Figure 15 shows average power of PL using voltage scaling at 1.0 volts with different frequency levels.

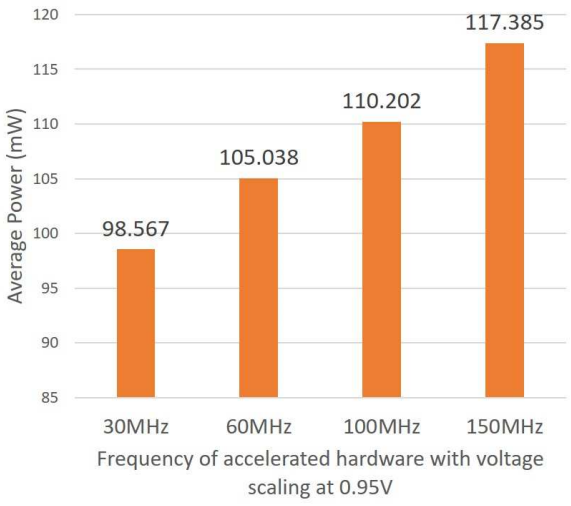

Figure 16 Average power of PL with different frequencies for hardware accelerator and with voltage scaling at 0.95V.

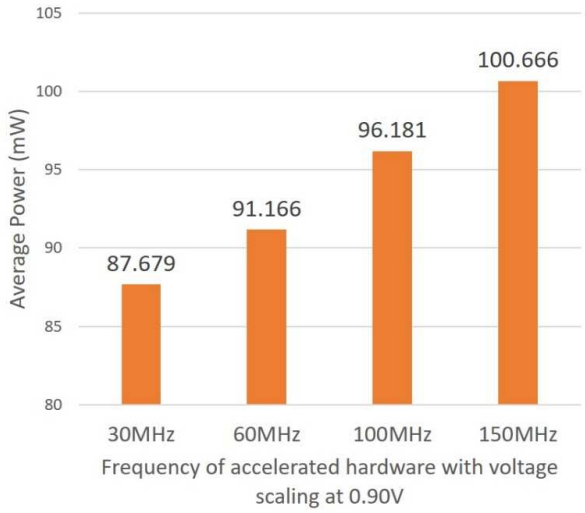

Figure 17 Average power of PL with different frequencies for hardware accelerator and with voltage scaling at 0.90V

Both Figures above show average power of PL using voltage scaling at 0.95 and 0.90 volts respectively, with different frequency levels. Comparing the most scaled result i.e. voltage scaling of 0.9 volts with

frequency of 30MHz we have an average power of 87.679mW. Which is 28.60% less as compared to normal execution of the system (i.e. average power of 122.803mW at 1.0V with 60MHz.)

DVS and DFS in a system can improve both power consumption and performance of a system. For future work, a power management algorithm should be investigated with respect to the image processing application and implemented in order to control both DVS and DFS. A hardware implementation of DVS and DFS on the FPGA would be very beneficial, since it will allow to run applications on PS independently of DVS and DFS.

16 References

[1] C. T. Chow, L. S. M. Tsui, P. H. W. Leong, W. Luk, and S. J. E. Wilton, "Dynamic voltage scaling for commercial fpgas," In Proceedings of the International Conference on Field-Programmable Technology (FPT), pp. 173–180, Dec 2005.

[2] A. F. Beldachi and J. L. Nunez-Yanez, "Accurate power control and monitoring in zynq boards," In Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–4, Sept 2014.

[3] A. Nabina and J. L. Nunez-Yanez, "Adaptive voltage scaling in a dynamically reconfigurable fpga-based platform," ACM Transactions on Reconfigurable Technology and Systems (TRETS), vol. 5, no. 4, p. 20, 2012.

[4] S. Ishihara, Z. Xia, M. Hariyama, and M. Kameyama, "Architecture of a low-power fpga based on self-adaptive voltage control," In Proc. of the International SoC Design Conference (ISOCC), pp. 274–277, Nov 2009.

[5] Xilinx Inc., "Clocking Wizard v5.3", in 2016 LogiCORE IP Product Guide, Oct 2016.

[6] P. R. Chafi, M. Moradi, N. Rahmanikia and H. Noori, A platform for dynamic thermal management of FPGA-based soft-core processors via Dynamic Frequency Scaling., Tehran: 2015 23rd Iranian Conference on Electrical Engineering, 2015.

[7] M. Agarwal, S. Singh, N. Agrawal, A. Kumar and B. Pandey, Frequency scaling based thermally tolerable Wi-Fi Enable 32-bit ALU design on 90nm FPGA, Mathura: 2015 Communication, Control and Intelligent Systems (CCIS), 2016.

[8] J. L. Nunez-Yanez, M. Hosseinabady and A. Beldachi, Energy Optimization in Commercial FPGAs with Voltage, Frequency and Logic Scaling, IEEE Transactions on Computers (Volume: 65, Issue: 5, May 1 2016), 2015.

[9] A. Podlubne, J. Haase, L. Kalms, G. Akgün, Muhammad Ali, H. ul Hasan Khan, A. Kamal and D. Göhringer, "Low Power Image Processing Applications on FPGAs using Dynamic Voltage Scaling and Partial Reconfiguration," in Proc. of the Conference on Design and Architectures for Signal and Image Processing (DASIP), Aug 2018. (Submitted)