GENERIC HETEROGENEOUS OPERATING SYSTEM

Jérémie Lagravière - jerlag@utu.fi

MSc Thesis, Embedded Computing
Supervisor: Adj. Prof. Juha Plosila
Embedded Computer Systems Laboratory
University of Turku
Advisors: Prof. Johan Lilius & Dr. Sébastien Lafond
Embedded Systems Laboratory
Åbo Akademi University

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service
Tous les jours on retourne la scène
Juste fauve au milieu de l'arène
On ne renonce pas, on essaye,
De regarder droit dans le soleil

- Détroit
Nowadays computers and embedded devices are composed of more and more processing elements. Within one device, processing elements are used for different purposes and are based on diverse hardware specifications. A hardware platform composed of several processing elements is understood as heterogeneous when these processing elements do not share the same hardware specification. Heterogeneity has to be handled at a software level in order to provide maximum performances and efficiency. Creating application for a heterogeneous hardware platform means that heterogeneous memory, heterogeneous processing elements and heterogeneous communication channels have to be managed. Thus, the need for a software support providing an access and allowing the creation of application for heterogeneous platform is important. Many initiatives are focusing on creating heterogeneous operating systems and how to optimize the design of these operating systems in order to achieve better performance. However little work has been done regarding detailed exploration on how applications should be implemented for heterogeneous operating systems. In this thesis we focus on providing a model for a Generic Heterogeneous Operating System that clearly separates the responsibilities between the developer of applications and the Operating System’s roles. This model is the base for a future implementation of our solution: GHOS.

Keywords: heterogeneous operating system, many-core programming, multiprocessor architecture, scheduling, parallel programming.
I would like to express my deep gratitude to Adjunct Professor Juha Plosila for supervising the thesis and accepting me in the Master Degree Programme in Embedded Computing at the University of Turku.

I would like to express my very great appreciation to Professor Johan Lilius for giving me the opportunity of working in the Embedded System Laboratory (ESLAB) at Åbo Akademi and supervising the thesis.

I am particularly grateful for the assistance given by Dr. Sébastien Lafond, who helped to keep my progress on schedule.

I would also like to extend my thanks to the ESLAB staff members for their help and for welcoming me in their team.

I would like to offer my special thanks to Amandine Bugli who helped me a lot in the writing process.

Finally, I wish to thank my parents for their support and encouragement throughout my study.
# Contents

## 1 Introduction

## 2 State of the Art: Heterogeneous Operating Systems

2.1 Introduction ................................................. 4
2.2 HOS: Barrelfish ............................................. 10
2.3 HOS: Tessellation .......................................... 15
2.4 HOS: Helios ................................................ 21
2.5 Comparison of Solutions ................................. 25
2.6 Conclusion .................................................. 28

## 3 State of the Art: Scheduling for heterogeneous context

3.1 Introduction ................................................. 29
3.2 Signature Supported Het-Aware Algorithm .............. 30
3.3 Heterogeneous Earliest-Finish-Time (HEFT) Algorithm . 34
3.4 Task Scheduling using Unified Platform on Heterogeneous Multi-core Architectures ............................... 39
3.5 Conclusions .................................................. 41

## 4 The proposed Generic Heterogeneous Operating System: GHOS

4.1 Overview: Use Cases ....................................... 42
4.2 Low Level Application: GHOS ............................ 47
4.3 High Level Application: Creating application for GHOS ........................... 51
4.4 Build System ............................................... 53
4.5 Conclusion .................................................. 61

## 5 HOS & Specific Hardware Platform: Pandaboard / OMAP4460

5.1 Install Model for Helios on OMAP4460 .................. 64
5.2 Install Model for Tessellation on OMAP4460 ........... 67
5.3 Install Model for GHOS on OMAP4460 ................... 69
6 Future work and conclusions

6.1 Future Work & Further Improvements ........................................ 72
6.2 Conclusions ............................................................................ 73
6.3 GHOS Prototype: Overview ...................................................... 82
6.4 GHOS Prototype: Implementation ............................................ 84
6.5 GHOS Prototype: Results .......................................................... 87
6.6 Conclusion .............................................................................. 89
1 INTRODUCTION

Computer hardware systems are nowadays composed of more and more processors or processing elements. Within one device, processing elements are used for different purposes such as: basic computing operations, mathematical calculations, 3D optimization and sound processing. This means that the devices we use on a daily basis are increasing in complexity over time. In a given device, if the processing elements are specialized or based on different technological specifications, this device is understood to be heterogeneous. As example, the following devices are all based on heterogeneous designs:

- Playstation 3 & 4: using IBM Cell processor [1] [2] [3]
- Intel Ivy bridge and Sandy bridge processors [4] [5] [6]
- Texas Instrument Open Multimedia Applications Platform (OMAP) [7] [8]

These devices are used for different aims, and therefore their heterogeneity is used to perform different tasks such as: computing performance or energy efficiency. Creating programmes for these devices is challenging as a number of different skills, technologies and tools are needed.

Increasing hardware complexity implies more complex software. For the last few decades multiplying processing elements in devices has lead to discussion about parallel programming in the scientific community [9]. It is necessary to have methodologies and implementations in order to be able to interact with these devices in an efficient way. At present, in order to create programmes for heterogeneous devices many different technologies are needed, this leads to costs in terms of time and investment. These aspects represent a scientific and technical challenge. Scientific challenge: modeling an operating system for a heterogeneous device represents a long term effort and a selection among available paradigms and concepts is needed. Scheduling tasks on many processing elements working in parallel, in order to get better performance in terms of energy consumption or execution time, is a complex (NP-Complete) task that necessitates strong theoretical study [10, 11]. Technical challenge: designing a new operating system is a technical challenge particularly when the goal is to cover hardware heterogeneity while giving a simple way for a developer to create programs for these systems.
Providing a way to interact with a heterogeneous system is central to guaranteeing its usability. That is developers need a set of tools with which to create programs for a given heterogeneous platform. This set of tools provides an interface which gives access to optimization, efficiency and reliability. However this increases complexity and implies a stiff learning curve. Offering complex devices and a simple way to create programs for them is very important for vendors to easily attract software editors and developers. Facilitating programming for heterogeneous platforms is a critical success factor for a new design [12]. Providing tools to facilitate programming for a given platform is a good way to get developers involvement which is essential for the popularity and longevity of the platform.

In this work, we propose a model which aims to describe an operating system that is able to adapt to any hardware platform by giving a generic method of programming for the developer. Our supporting goal is to provide a simple and operational heterogeneous operating system that would be used as a proof of concept. To achieve these goals, we first provide a model following which we present and select concepts, paradigms and solutions in order to implement our heterogeneous operating system. This study contains two chapters reviewing prior research in heterogeneous operating systems (HOS) as well as in scheduling algorithms for heterogeneous hardware platforms. This literature review will be used as a theoretical and practical basis for our work and for the model of our generic heterogeneous operating system (GHOS).

Theoretical Foundation for Research

This study is based on prior research in three related and complimentary fields; Heterogeneous Operating Systems, Scheduling in a Heterogeneous context and research on the compilation and application building process for heterogeneous designs.

Research in existing heterogeneous operating systems is important as it allows us to identify features and functionalities that could be reused or perhaps should be avoided in the proposed solution. As a foundation this study will analyse in detail three existing heterogeneous operating systems. These systems under scrutiny are Barrelfish [13, 14], Helios [15] and Tessellation [16]. These three operating systems are based on different concepts and paradigms in order to manage heterogeneity. As the field of heterogeneous operating systems is not limited to the three under closer examination in this study we will extend our selection in order to give a brief, but global, view of this field of research and also reuse and/or exclude concepts, ideas and paradigms proposed in those additional systems.

The second important issue examined in this study is scheduling for a heterogeneous context. All operating systems have to deal with scheduling. Simply put scheduling concerns the organisation of tasks among the processing elements available. However, in a heterogeneous context, scheduling is a very active field of science: it is
clearly described as a NP-Complete problem [10, 11]. This is especially the case when the real time factor is added, it is then possible to create a complex and elegant scheduling algorithm. Nevertheless due to the amount of scheduling algorithms available, we will base our study on a set of selected algorithms and tools related to scheduling for heterogeneous platforms.

The *compilation and application building process for heterogeneous designs* [17, 18] involves planning the provision of an efficient way to create applications for the operating system in question. This study proposes a solution to describe and compile applications for a chosen hardware platform. Solutions are already developed to cover this aspect [15], however creating a dedicated solution is very important because the compilation and the application building system is the “visible” part of the system, i.e. the part of the operating system that will be used by developers.

As this thesis aims to provide a model for our solution GHOS (Generic Heterogeneous Operating System), it will study Heterogeneous Operating Systems and Scheduling in the a literature review (Chapters 2 & 3).

**AMEBA project**

This thesis is related to the AMEBA (Agent based Management of EmBedded dAta reservers) project, which aims to prepare for the arrival and demonstrate the interesting aspect of three-dimensional system on chip. Three dimensional chips integrate data storage, RAM and heterogeneous processing elements on one hardware unit. This thesis aims to propose a model for an operating system that is able to take advantage of the three-dimensional chip.

**Thesis structure**

The thesis reviews existing approaches which bring the reader closer to understanding the subject of heterogeneous operating systems for multi-core embedded systems. Chapter 2 of this study describes and analyses three existing heterogeneous operating systems (HOS). In chapter 3, addresses scheduling techniques and algorithms for the heterogeneous context. Chapters 2 and 3 also contain an analysis grid which evaluates the characteristics of each studied solution on the basis of applied criteria. The goal of these chapters is to give a detailed description of selected solutions. In chapter 4 we present a model of our generic heterogeneous operating system (GHOS). Chapter 5 presents the deployment of a heterogeneous operating system on a single board computer (Pandaboard) equipped with the system running on a Texas Instruments OMAP4460 chip. The selected solutions applied in this chapter are: Helios, Tessellation and our solution GHOS.
2 STATE OF THE ART: HETEROGENEOUS OPERATING SYSTEMS

The goal of this chapter is to present and describe operating systems that are designed for the heterogeneous context. The following introductory section provides a set of definitions for the concepts used in this chapter and an analysis grid (Table 2.1, page 9) which brings together a set of criteria that are used throughout the chapter in order to describe each operating system. The reasons for describing existing research and work in designing and implementing HOS is twofold: first, some of the concepts, notions or ideas presented in this section will be reused later in the thesis and therefore it is important to adequately discuss them beforehand. Second, by synthesizing the existing features of each HOS using an analysis grid, this study aims to point out what is missing, and then propose our own design in chapter 4.

2.1 Introduction

In the following sections of this chapter we present three HOS: Barrelfish [13, 14], Helios [15] and Tessellation [19, 16]. These HOS rely on paradigms and theoretical as well as practical choices that should be presented in order to give a detailed view of each of these HOS before these concepts are reused (Chapter: 4). The HOS that are described in this chapter are small operating systems, i.e. they mainly consist of a kernel and a set of deployment tools (compilers, build system & best practice). We study these systems in order to describe their main features and therefore to understand how these systems work.

2.1.1 Definition

It is important to provide adequate definitions for technical and theoretical concepts before moving forward in detail and discussion on HOS. As this study concerns oper-
ating systems, it is important to have a definition for them. In this study: “Operating systems are software environments that provide a buffer between the user and the low level interfaces to the hardware within a system. They provide a constant interface and a set of utilities to enable users to utilize the system quickly and efficiently. They allow software to be moved from one system to another and therefore can make application programs hardware independent. Program debugging tools are usually included which speed up the testing process. Many applications do not require any operating system support at all and run direct on the hardware” [20].

To define heterogeneity this study needs to use the concept of Instruction Set Architecture which is defined as follows: “An Instruction Set Architecture (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O.” [21]. This study examines heterogeneity, it follows that we consider that heterogeneity exists when different types of processing elements are used on the same platform in order to run in parallel to achieve the execution of a set of tasks. Two differences between processing elements are considered: different ISA and/or different frequencies. A generic definition of heterogeneous computing: “Heterogeneous computing systems refer to electronic systems that use a variety of different types of computational units. A computational unit could be a general-purpose processor (GPP), a special-purpose processor (i.e. digital signal processor (DSP) or graphics processing unit (GPU)), a co-processor, or custom acceleration logic (application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA)). In general, a heterogeneous computing platform consists of processors with different ISA.” [22]. To this definition we add the possibility of having Asymmetric single-ISA (ASISA) [23]. ASISA consist of exposing the same Instruction Set Architectures, but delivering different performance. This implies that operating systems and programs have to be designed specifically for such a context, so that the expected results are correctly produced and so that each processing element is used optimally. Optimum can be expressed in terms of performance, throughput or energy consumption.

Even if having multiple types of processing elements is the primary requirement to define heterogeneity, the criteria that characterize heterogeneity are not limited to these. In addition to the diversity of processing elements we can also take into consideration the non-uniform memory access (NUMA). Heterogeneity implies multiple
processing elements, which means that in this chapter and in the whole thesis is always about multi-processor context. Another criteria that determines the heterogeneity of a given system is the difference of Instruction Set Architecture among the processing elements. Figure 2.2 gives an overview of what is a heterogeneous context and how it may be handled by an operating system: there are several heterogeneous cores (x86, x64, ARM, GPU).

The multi-processor context implies that parallel programming is involved, parallel programming is considered at an operating-system-level in this thesis whereas "classic" parallel programming where developers have the responsibility of dividing the tasks, processes and functions so that they run efficiently in parallel. This means that the operating system is supposed to provide a way to make the development process as easy as possible for a multi-processor heterogeneous platform. Some of the HOS studied in this chapter make a real effort to provide solution to help developers to program easily for a complex heterogeneous platform.

Concerning the memory architecture, the selected HOS are designed specifically for NUMA architecture. “Non-Uniform Memory Access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to a processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors).”[24] NUMA is opposed in Symmetric Multi-Processing (SMP), in which all memory access are posted to the same shared memory bus. Figure 2.1 shows that the SMP structure is very different from the NUMA struc-
ture, the main difference and the one that we are interested in is that in NUMA each processing element has its own memory space and in SMP the processing elements share the same memory space.

2.1.2 Analysis Grid

In this chapter we have selected three HOS and for each of them some topics were systematically studied. From this topics we have identified a set of criteria used to compare HOS and present a synthesis of them. Before going further into the description of the chosen operating systems, it is necessary to give a definition and a justification of the concepts that are used in the analysis grid. These criteria are also considered, in this thesis, as questions that need to be answered.

Description of each criteria used in analysis grid:

- **OS structure** represents the way each studied HOS adapts its structure to the different processing elements. Basically, this criteria is about the size, the complexity and the amount of kernels needed to have the HOS to work properly.

- **Memory Management**: the studied HOS are designed for NUMA architecture, but as they must achieve a cooperative work between the different processing elements, the way memory is organized is an interesting criteria to understand how those HOS work.

- **Inter-Process Communication (IPC)**: because of the NUMA architecture, and the need of communication (parallel processing) between the processing elements, inter-process communication is an important functionality. Each HOS provides a specific features to cover this functionality.

- **Compile / Build Mechanism**: a set of tools is sometime provided to compile or prepare the application in order to be able to run it on the operating system. This kind of feature means that some responsibilities are given to the developer.

- **Centralization Level**: because of the multi-processor context, presented HOS need to propose a solution for de-centralization. This means that the OS adapts its structure in order to provide a multi-kernel hierarchy which is more or less centralized.
• **Scheduling**: in a heterogeneous context a specific algorithm can be needed in order to increase the performance or reduce energy-consumption. This criteria is about how the studied HOS provide a way to express and manage priorities between tasks.

• **Specific feature**: some HOS offer a specific feature that deserves to be described.

The analysis grid on Table 2.1 is a summary of what is presented in this chapter, more details are given for each HOS in the following sections of the chapter.
<table>
<thead>
<tr>
<th>OS Name</th>
<th>OS Structure</th>
<th>Inter-Process or Inter-Core Communication</th>
<th>Memory Management</th>
<th>Compile / Build Techniques</th>
<th>Centralization Level</th>
<th>Scheduling</th>
<th>Specific Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrelfish</td>
<td>Multiple kernels, one processing element &lt;= one kernel</td>
<td>Explicit inter-core and inter-process communication using message passing</td>
<td>“Share nothing” model, any data transfer involves a copy using message passing</td>
<td>Possibility of describing hardware in a knowledge database</td>
<td>The system is organized as a network of distributed systems communicating together</td>
<td>Each application can specify demands for bandwidth allocation or latency</td>
<td>Knowledge base to handle diversity of cores</td>
</tr>
<tr>
<td>Tessellation</td>
<td>A set of exo-kernel (thin layer) is deployed over hardware</td>
<td>Implemented using messaging for inter-partition communication</td>
<td>“Share nothing” model</td>
<td>-</td>
<td>Decentralized by structure, not a lot detail is given about this aspect of the OS</td>
<td>Scheduling is handled by the partition manager</td>
<td>Space-time partitioning</td>
</tr>
<tr>
<td>Helios</td>
<td>A set of micro-kernel is deployed over the processing elements</td>
<td>Transparent IPC using message passing. The system offers a transparent way to handle local message passing (same NUMA Domain) and remote message passing (different NUMA Domain)</td>
<td>“Share nothing” model</td>
<td>Two phase compilation is chosen, using an intermediate and generic language, which allows the application to be executed by using a maximum of optimizations for each processing element</td>
<td>A coordinator kernel is booted first in order to manage the namespace mechanism</td>
<td>Scheduling is done by using affinity mechanism, applications declare affinity between each other</td>
<td>Namespace mechanism to ensure IPC</td>
</tr>
</tbody>
</table>

Table 2.1: Analysis Grid
2.2 HOS: Barrelfish

In this section we introduce the HOS called Barrelfish [14] [13].

The logical structure of Barrelfish and its multikernel model [13] is presented in Figure 2.2. For each processing element a software stack is used, one part is hardware specific and the other one is generic. These two parts are called OS Node. Above an OS Node, an application can be implemented and run. An application can be limited to one OS Node (one processing element) or can be executed on multiple nodes (several processing elements). This means that the nodes need to communicate, and to achieve these communication Barrelfish uses only asynchronous message passing (AMP). As no state are shared (State Replication) and thanks to the asynchronous message passing mechanism, the OS is fully distributed, which improve the scalability and the adaptivity of the system.

Barrelfish gives an implementation of these design principles:

- Explicit communication between processing elements
- Hardware neutral implementation: only a small part of each OS Node has to be adapted to hardware. The behavior of the system is the same independently of the hardware.
• State are replicated instead of shared

These principles define the multikernel model according to Baumann et al. [13] and their implementation are visible in Figure 2.3. In the following sections, we describe each of the important concepts used to build the multikernel model in Barrelish.

2.2.1 OS Node implementation

In Figure 2.3 we can see that the OS Node presented in Figure 2.2 has been divided into two parts: a CPU Driver for the kernel space and a Monitor for the user space. In Barrelish the OS instances are factored on each core into a privileged-mode CPU Driver and a distinguished user-mode Monitor process. A CPU Driver is local to a core, all the inter-core communication are performed by monitors. The association of a monitor and a CPU Driver gives a typical monolithic set of functionality: scheduling, communication, low-level resource allocation. The CPU Driver is in charge of performing authorization, time-slices processes, mediating access to the core and its associated hardware (MMU, APIC, etc.). As the CPU Driver does not share any state with other processing elements, it can be fully event driven, single threaded and non-preemptable. Plus, as the CPU Driver is specific to the hardware it has to provide an implementation for a lightweight and simple asynchronous (split-phase) same core inter-process communication mechanism. This mechanism is then used by the monitor to provide more complex communication channels using shared memory (described in the sub-section Explicit Inter-core Communication).

Figure 2.3: Barrelish physical structure
Monitors are single-core, user space processes and are, as a consequence, schedulable. Monitors are in charge of running the agreement protocol process, that ensures the consistency of allocation tables and address space mappings. They are also responsible for the inter-core communication messages of the multikernel model. An application requesting an access to a global state has to ask the monitor to provide this access, the monitor will return a copy of a remote state.

The multikernel model implemented by Barrelfish is hardware-neutral by adapting the CPU Driver to the hardware and by using an intermediate layer called the Monitor. In particular the CPU Driver has to adapt its message transport mechanism and its interface to the hardware (Processing Elements and devices). Which means that a user of the operating system has to think the application in terms of message passing, but does not have to take care of the specifications of the different processing elements.

Managing diversity of cores

In Barrelfish, a System Knowledge Base (SKB) is implemented. It contains a representation of the machine’s hardware and current state. The SKB is built thanks to three different sources of information:

- “Online data”, those are data that are received when the system is working; it consists mainly of resource discovery, covering the hot-plugging of processor or memory.

- “Online measurement and profiling” those are data collected when the system is working. It consists of gathering information about the performance of the different processing elements and memory at runtime.

- “A priori knowledge” of the hardware platform is available through a description that is done offline.

Applications are able to make requests to this knowledge base in order to express needs for specific computing features. The requests are done in a generic way: independently of the hardware.

2.2.2 Explicit Inter-core Communication

Barrelfish uses explicit inter-core communication by implementing it through message passing, this technique has constraint and advantages that we describe in this section.
Explicit inter-core communication means that, communication data are exposed: what part of shared state are accessed when and by who. Contrary to implicit communication, explicit communication enhance the use of the system interconnect. As demonstrated in “Your computer is already a distributed system. why isn’t your OS?” by Baumann et al. [25], the machine increasingly resembles a network and as a consequence the OS has to behave as a distributed system. Because of the network resemblance it is then possible to model and implement well-known networking optimization. Additionally, using explicit message passing for all inter-core communication makes the analysis of the system amenable to human or automated tools. The structure of a system implied by using message passing is naturally modular, because components interacts with each other through a well-defined interface.

Barrelfish uses state replication and message passing to coordinate processes among processing elements. Message passing is used for every communication or coordination in Barrelfish not only because it provides modularity but also because it delivers good performance in terms of latency [13].

State Replication is a well-known technique to ensure OS scalability [26] [27] even if it is generally used as an optimization for shared-memory kernel design. Consistency is maintained by exchanging messages. Consistency maintenance can be long-run operations, therefore non-blocking communication are used. The use of the non-blocking behavior is double: a core can then execute useful work or sleep in order to save power.

From a technical point of view, Barrelfish uses a variant of User-level Remote Procedure Call (URPC) [28] between processing elements: a channel is implemented in a region of shared memory in order to transfer cache-line-sized messages point-to-point between a single writer and a set of reader-processing-elements. Thanks to this mechanism, all message interactions are abstracted using a common interface, so that classic operations such as "send" and "receive" are performed in a transport-independent way.

2.2.3 Memory Management

Even if, Barrelfish is a multikernel OS which is itself distributed it has to consistently manage physical memory. To do so, memory management in Barrelfish is performed by using capabilities, as defined in seL4 [29]. In this model, memory management is explicit: by using system calls that manipulate capabilities it is possible to make user-level reference to kernel objects or regions of physical memories. Thanks to this
model based on capabilities the CPU Driver is not responsible for dynamic memory
allocation, it is only in charge of checking the correctness of operations that manipu-
late capabilities. As a consequence, the responsibility of allocation and manipulation
of page tables (virtual memory management) is fully given to user-level code. The
choice of capabilities facilitates a clean decentralization of resource allocation which
improves the scalability. The use of capabilities implies a higher code complexity, but
it provides a uniform way to implement operations such as: copying and retyping.

2.2.4 Conclusion

We have studied Barrelfish as an example of multikernel OS. Pragmatic choices have
been done to develop this operating system in order to follow the multikernel model.
In the next section, we study an exokernel OS through the example of Tessellation,
which implements Space time partitioning.
2.3 HOS: Tessellation

In “Classification of Heterogeneous Operating System” by Sharma et al. [30], an exokernel is described as a tiny kernel whose most important task is to ensure protection and multiplexing resources. Exokernels are simpler than conventional microkernel or monolithic kernel. The main idea proposed by exokernel is to give a maximum of responsibility to the developer. The developer takes as many decisions as possible about hardware abstractions. "Exokernel can be seen as an application of the end-to-end principle to operating systems, in that they do not force an application program to layer its abstractions on top of other abstractions that were designed with different requirements in mind"[30]. Figure 2.4 presents a generic exokernel architecture: program interactions are more direct by using the exokernel than by using the "normal kernel". Figure 2.5 shows a possible exokernel system. The libOs (library operating systems) provides standard system abstractions such as virtual memory management, file system, network protocols. Application can also specialize or write their own libOS [31].

In this section, we use as an example the Tessellation operating system [19]. This OS is adapted for manycore platform and also for heterogeneous platform. Tessellation is based on an exokernel, and on the concept of Space-time partitioning (STP). STP is the mechanism that we study in detail in this section.
2.3.1 Space-time partitioning

"A spatial partition [...] is an isolated unit containing a subset of physical machine resources such as cores, cache, memory, guaranteed fractions of memory or network bandwidth, and energy budget" [19] Space-time partitioning (STP) performs multiplexing of all the partitions over the hardware. This is a virtualization of spacial partition. User-level scheduling is allowed by using sufficiently coarse granularity for the partitions. Scheduling and resource management in Tessellation is implemented at the partition granularity as shown in Figure 2.6. Within their partitions applications and OS services have exclusive control of the scheduling of their resources. Resource allocation and scheduling at the partition granularity are implemented by a thin layer which is the Tessellation Kernel as show in Figure 2.7.
Figure 2.7: Components of a Tessellation system. The Tessellation Kernel utilizes a combination of hardware and software mechanisms to enforce partition boundaries [19]

In the next sections we study how STP is implemented in Tessellation, from a hardware and software point of view.

2.3.2 Hardware Partitioning Mechanisms

In order to restrict the ability of application to run thread on the cores, the processing element are controlled by Tessellation’s kernel. The partitioning of the memory is performed by using classic mechanisms such as page-table and translation look-aside buffers (TLB).

Performance isolation is needed in order to get spatial-partitioning and to implement it hardware (processing elements, resources and memory) needs to be supported. This hardware support provides means to partition shared resources: caches, memory or network bandwidth, etc. Thanks to performance isolation the operating system is able to run arbitrary applications simultaneously (i.e. spatially distributed).

In order to measure performance, resource usage and energy consumption, Tessellation uses hardware performance counter (cf. Figure 2.7) to monitor application performance. These performance information are used to determine the runtime behavior of an application during its different phases.
Partition Mechanism Layer

In Tessellation choice has been done to have a "thin" layer which is hardware dependent, this layer is called *Partition Mechanism Layer* (cf. Figure 2.7). The Partition Mechanism Layer is responsible for "configuring available hardware mechanisms to enforce dynamic hardware partition" [19]. Through this mechanism an machine-independent API is provided. This API is used by the policy layer implemented in the Partition Manager (see next section).

2.3.3 Scheduling

The partition manager is in charge of determining which resources and how much of those resources should be allocated to each partition. The partition manager performs scheduling and resource allocation for applications and services partition, by taking in account load balancing and energy consumption.

In a more recent version of Tessellation (2010) the concept of Cell Model is introduced: a Cell is a container for parallel software components providing guaranteed access to resources [16]. A resource allocated to a cell is owned by this cell until it is revoked. There are two level of scheduling in Tessellation both of them relying on cells: distributing resources to Cells, scheduling with a Cell. It is important to note that thanks to the performance isolation principle described earlier the application within a Cell can have predictable and reproducible behavior.

To achieve this optimal scheduling in terms of performance -and- energy consumption, Tessellation implements a mechanism, on physical hardware, of partition-multiplexing by using virtualization. The direct consequence is that partition resources have to be *gang-scheduled*. This gang-scheduling mechanism is the only way to provide performance isolation to applications so they can have the inner scheduler to work properly (predictable and repeatable behavior).

2.3.4 Inter Partition Communication

As an application in Tessellation can be divided among several partitions, these partitions need to communicate with each other. In Tessellation communication are implemented by using message passing. But this operating system insists on the security aspect of message transmission. Messages are read-only and relinquished by the sender at the time of transmission. Messages are transmitted through inter-partitions channels.
There is a risk that the destination of the message is descheduled when it reaches the end-point of the channel. Therefore Tessellation defines that a destination has to be a resource (e.g., a queue in memory) so that the content of the message remains in memory in order to be available when the targeted partition will be scheduled again.

2.3.5 Space-time Partitioning for Efficiency and Heterogeneous Context

In Tessellation different kind of efficiency are targeted:

- Performance: execution time and memory allocation minimization
- Energy consumption
- Quality of Service (QoS)
- Hybrid behavior: this includes heterogeneous contexts.

Performance

Spatial partition giving exclusive access to resources provides a stable environment for applications that rely on different style of parallel programming and that have different performance objectives, for instance: guaranteed throughput vs interactive response. This gives the possibility to Tessellation to provide performance isolation and functional isolation.

Energy Consumption

Thanks to the partition mechanism and by relying on suitable hardware features, Tessellation can either put a whole partition in a low power state, or devote a minimal fraction of energy to a crucial function and restrict drastically the energy devoted to a non-crucial function. This latter functionality is particularly interesting for embedded device with a critical/crucial main functionality and a set of non-crucial functionality.

Quality of Service

By relying on performance isolation and strict control or inter-partition communication, Tessellation is able to ensure QoS. Resources can be dedicated to a partition that
requires QoS, such IP network handling external network traffic, performing intrusion
detection, etc. By dedicating resources to a partition, Tessellation ensures that this
partition will receive sufficient CPU and memory resources to guarantee its quality of
service requirements.

**Hybrid Behavior**

By using an efficient inter-partition mechanism, it is possible with Tessellation to split
application into multiple sub-application partitions. Usage of this functionality can
be diverse: for performance, a virtual reality game might isolate real-time compon-
ents from background tasks via partitioning; for security a web-browser could isolate
untrusted webpages or plugins in separated partition.

Concerning heterogeneous architecture, partition can cover several processing ele-
ments with different computational abilities.

**2.3.6 Conclusion**

We have studied in detail Tessellation OS which relies on an exokernel. The partition
mechanism implemented in this operating system is an interesting concept that can
cover some of the requirements for the generic heterogeneous platform that we want to
model in chapter 4. Before reusing the ideas and concepts from this operating system,
we will study Helios OS in the next section, and in chapter 5 we will study in detail an
implementation of Tessellation over the system-on-chip Texas Instrument OMAP4460.
2.4 HOS: Helios

Helios [15] has been built with the idea of using satellite kernel. Helios is designed to facilitate the task of writing, deploying and tuning application for heterogeneous platform.

Helios is based on satellite kernels, this type of kernel exports only one single set of abstractions independently of the processing elements diversity and their performance characteristics.

Helios manages in a specific way the non-uniform memory architecture (NUMA), in fact they are considered as shared-nothing multiprocessor which means that each NUMA domain (NUMA domain: set of processing elements and their local memory) runs its own satellite kernel and manages its resources independently. In a large multiprocessor environment, Helios can achieve better performance by scaling up, thanks to explicit boundaries between NUMA domains.

Additionally, Helios simplifies application deployment by providing an affinity metric, which basically offers the possibility to declare the preferences of an application to be executed at a certain location on the hardware platform. This affinity metric is centered around the notion of message passing channel: a positive affinity means that the OS has to know that two components will benefit from being executed in the same domain, implying the use of fast-message passing (zero copy message passing, this mechanism is described in detail later in this section).

The Helios research team has chosen a two-phase compilation strategy in order to cover the diversity of processing elements that could be available on a machine. The applications implemented for Helios have to be compiled to an intermediate language. Then the generated intermediate language, is compiled to the specific instruction set of each of the available processing elements. A benefit of the intermediate language is that it encapsulates multiple implementations of a specific feature which can be optimized for different architectures.

All these features are described in the following parts of this section. The objective of this section is to give an overview of Helios, by studying each mechanism modeled and implemented in this HOS.
2.4.1 Satellite kernel: definition & implementation

Satellite kernels are microkernels, each of them contains: a scheduler, a memory manager, a namespace manager (defined later), and functionality to coordinate communication between other kernels.

The main goal of Helios satellite kernel architecture is to provide a single set of abstractions even though the operating systems covers many platform, which implies many programmable devices.

To achieve that goal, four guidelines are defined by the Helios research team [15]:

- Avoid unnecessary remote communication: all requests should be, as much as possible, served locally (inside the same NUMA domain)

- Require minimal hardware primitives: by respecting the preceding guideline, Helios should always require a minimal set of hardware primitive.

- Require minimal hardware resources: Helios should be able to run on platform offering slower CPU and less RAM (low power embedded platform) than general purpose computers.

- Avoid unnecessary local IPC: as local message-passing is slower than a system call, private resources to a process should be managed by the satellite kernel and accessed a system call.

As Helios is based on Singularity RDK [32], it implements most of its functionality: satellite kernels, message passing and affinity.

2.4.2 Transparent IPC

To implement transparent IPC the Helios research team solved two problems. First, an application has to be able to name another application or OS service, wherever this application or service is executed. Secondly an application should run indifferently whether it executes on one or multiple satellite kernel(s). To meet the first requirement, Helios implements a single and unified namespace. Basically, application, OS Services or even drivers can expose their functionality, platform type etc. in the namespace. The second requirement is met by implementing local message passing (LMP) (inside the same NUMA domain) and remote message passing (RMP) by using
the same abstraction (interface). Thanks to this mechanism, an application can communicate with another application such as an operating system service, as if the service was on the same satellite kernel.

**Namespace**

A service can be registered in the namespace so that other application can find it. The communication between a process and a service is done by using message passing.

The coordinator kernel, which is the first to boot in Helios, manages the namespace mechanism. This coordinator kernel has three roles concerning namespace: it allows application to register in the namespace, this means that an application can declare that it is listening for subscribers (waiting for connection); the coordinator kernel is also in charge of the binding mechanism, basically it ensures the retrieval of the requested service or application and then establishes a direct channel of communication between the service and the subscriber; finally, the coordinator kernel performs the removal of the entries in the namespace. This namespace mechanism is centralized, which is an exception in the Helios operating system as this is the only centralized component.

**Remote Message Passing**

In terms of implementation, Helios makes a difference between local message passing and remote message passing, but offers the same abstraction to use them. When two processes have to communicate over a local channel, it implies that they share the same address space, thread scheduler, exchange heap. This allows the sender to pass a pointer to the receiver instead of copying the pointed-to data. Remote message passing is about communication between processes that run on different kernels, which means that the processes do not share the same address space, exchange heap or thread scheduler. In that case, Helios performs a copy of data instead of passing pointers. This is a crucial point in Helios, tuning an application in order to encourage local message passing or remote message passing is a matter of affinity between applications, this mechanism is described in the next section.

### 2.4.3 Scheduling: Affinity mechanism

The affinity policies allow any application or service running on Helios operating system to express a preference to run on the same kernel of another application or service
(positive affinity), or to run on a different kernel (negative affinity). As a matter of convenience, processing elements can be addressed by application affinities in order to express a need of an application to run on a specific processing element. In fine thanks to the affinity policies the message passing structure is defined by relying on the preferences of each application or services.

To express affinity, Helios provides a XML syntax which is used to write manifests. These manifests are used at compile time, and transformed into a common intermediate language (CIL).

A positive affinity means that a process will benefit from a classic zero-copy message passing channel. A negative affinity means that a process will benefit of its isolation from the other processes it has to communicate with. The default value for affinity (if not specified), is 0, which means that there is no placement preference for the application. If a developer does not have any knowledge about the hardware topology of the platform, it is still possible to define the dependencies between applications in order to maximize the performance.

In detail, a positive affinity can have two meaning: it expresses a tight-coupling relationship between two processes or it expresses the preference of a process for a certain kind of processing elements. Concerning negative affinity, it is often used in order to avoid resource contention, because, by nature, a negative affinity expresses a preference to avoid a specific satellite kernel. Negative affinity can be used also as a self-reference affinity: a self-negative affinity means that every instance of the same process should run on a different satellite kernel.

## 2.4.4 Conclusion

By relying on satellite kernels, two-phases compilation, transparent IPC and affinity mechanism, Helios provides to the developer a set of tools that facilitates the handling of a heterogeneous platform. The affinity policies are a simple and efficient way to manage the performance and the scalability of application by giving the possibility to express preferences such as "this application should run on a different kernel than that application".

In order to provide a clear view on this chapter, in the next section, we propose another analysis grid which aims at describing what are the common use-cases for heterogeneity management and how well the solutions presented in this chapter cover these use-cases.
2.5 Comparison of Solutions

This section aims at giving criteria to define different kind of heterogeneity and how the solutions presented in chapter 2 cover these criteria. The different kind of heterogeneity can be considered as use-cases and the analysis grid we present in this section are a simple and efficient way to provide a basis to compare solutions.

In chapter 2, we studied respectively Heterogeneous Operating Systems and scheduling algorithms, which have a set of constraints and objectives in common, which we are focusing on in this section to be able to compare them.

This section aims at answering two questions:

- What use-cases are covered by the solutions described in chapter 2;
- What are the criteria privileged by each solution.

2.5.1 Methodology

In this section we present how we have built a set of criteria to describe and compare solutions from chapter 2.

Criteria

We have chosen a set of four criteria to build our analysis grids and be able to compare them:

- ASISA: Asymmetric Single Instruction Set Architecture [23] (Defined page: 30). Simple definition: having a platform with multiple processing elements having all the same instruction set but using different clock frequencies;
- Heterogeneous ISA: This is the "classic" heterogeneity: multiple processing elements with different instruction set;
- NUMA: Non-Uniform Memory Architecture (Defined page: 6). Processing elements have access to local memory domain but can also require data that are located in remote memory domain;
- Shared Memory: All processing elements have access to the same memory space, and thus have to share its usage.
We focus our analysis on criteria dealing with processing elements and memory which can be considered as the main elements of heterogeneity. Heterogeneous processing elements imply the need for heterogeneous computing, and heterogeneous memory architecture have consequences on how heterogeneous operating systems work.

**Representation**

To show how each solution from chapter 2 covers each criteria we created a table which gathers criteria and solutions. The analysis grid provides two levels of information: the first one is by associating a color to each criteria to show how well the solution covers the criteria, the second one is a synthetic explanation of the solution’s coverage.

**Color code**

We have chosen four background-colors to present different levels of coverage:

- **White**: impossible to determine if the solution is actually able to cover or not the criteria;

- **Red** (Dark Grey in black & white): the solution is able to cover the criteria but it is not optimized to cover it and/or the solution does not specify anything about the criteria but by design it is, at least, able to cover partially the given criteria;

- **Green** (Light Grey in black & white): the solution covers the criteria and offers optimization or tuning possibilities for this criteria.
Barrelfish can distinguish two processing elements having the same ISA but different frequencies, but it is only possible at runtime with the online data and online measurement and profiling, no direct optimization can be done by the user.

Barrelfish handles ISA heterogeneity mainly thanks to its system knowledge base and the CPU driver. Barrelfish thanks to its share-nothing model for memory management is clearly optimized for NUMA.

Barrelfish is optimized for NUMA at the expense of a shared-memory management, particularly because of its communication model based on transparent message passing which implies.

Helios provides the affinity mechanism to handle heterogeneity but it is not demonstrated that this mechanism can or cannot handle ASISA.

The affinity mechanism provides an abstraction for managing heterogeneous ISA. Helios is optimized for NUMA mainly thanks to its management of local message passing and remote message passing in a transparent way.

Helios is not clearly optimized for shared-memory as it focuses mainly on providing primitives for NUMA.

Although Tessellation is designed to handle heterogeneous platforms, it is not clearly stated that the partition mechanism, which is central for this operating system, gives the possibility of managing ASISA.

The partition mechanism allows the manipulation of partitions among processing elements having heterogeneous ISA. Tessellation and its partition mechanism is clearly optimized for NUMA, i.e. structurally separated memory.

The partition mechanism implies a strongly segmented vision of memory management and even if the partition mechanism is claimed to be flexible and light-weight it is not based on concepts optimized for shared memory.

<table>
<thead>
<tr>
<th></th>
<th>ASISA</th>
<th>Heterogeneous ISA</th>
<th>NUMA</th>
<th>Shared Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrelfish</td>
<td>Barrelfish can distinguish two processing elements having the same ISA but different frequencies, but it is only possible at runtime with the online data and online measurement and profiling, no direct optimization can be done by the user.</td>
<td>Barrelfish handles ISA heterogeneity mainly thanks to its system knowledge base and the CPU driver</td>
<td>Barrelfish thanks to its share-nothing model for memory management is clearly optimized for NUMA.</td>
<td>Barrelfish is optimized for NUMA at the expense of a shared-memory management, particularly because of its communication model based on transparent message passing which implies.</td>
</tr>
<tr>
<td>Helios</td>
<td>Helios provides the affinity mechanism to handle heterogeneity but it is not demonstrated that this mechanism can or cannot handle ASISA.</td>
<td>The affinity mechanism provides an abstraction for managing heterogeneous ISA.</td>
<td>Helios is optimized for NUMA mainly thanks to its management of local message passing and remote message passing in a transparent way.</td>
<td>Helios is not clearly optimized for shared-memory as it focuses mainly on providing primitives for NUMA.</td>
</tr>
<tr>
<td>Tessellation</td>
<td>Although Tessellation is designed to handle heterogeneous platforms, it is not clearly stated that the partition mechanism, which is central for this operating system, gives the possibility of managing ASISA.</td>
<td>The partition mechanism allows the manipulation of partitions among processing elements having heterogeneous ISA.</td>
<td>Tessellation and its partition mechanism is clearly optimized for NUMA, i.e. structurally separated memory.</td>
<td>The partition mechanism implies a strongly segmented vision of memory management and even if the partition mechanism is claimed to be flexible and light-weight it is not based on concepts optimized for shared memory.</td>
</tr>
</tbody>
</table>

Table 2.2: Analysis Grid

The solutions presented in table 2.2 even if they rely on very different concepts and paradigms have the same coverage of the chosen criteria. However, even if these solutions have limited coverage concerning ASISA and shared-memory, their implementation is still flexible enough to actually run on platform providing this kind of architecture.
2.6 Conclusion

The three HOS presented in this chapter share some technical and conceptual aspects such as: transparent/explicit IPC using message passing, avoiding shared memory, giving abstract programming independently of the hardware heterogeneity. The main differences appear with the kernel or the programing style: Tessellation, for instance, has a kernel which implies a thin abstraction layer [33], whereas Helios which provides a strong abstraction of hardware.

Other differences are visible by looking at the scheduling algorithms: Helios relies on affinity and Tessellation provides space-time partitioning and performance isolation to ensure quality of services. Barelfish has a complex structure for scheduling and relies on orders given at user level. Some of these features, techniques and concepts will be reused in Chapter 4.

One aspect is missing concerning the studied HOS: the scheduling algorithm is never given or described in detail. The scheduler is a central functionality of any operating systems, and in heterogeneous context the complexity and the responsibility are increased compared to homogeneous context. The scheduler is in charge of the execution of tasks and on which processing element each task should be executed. In the following chapter we study in details a set of scheduling solutions for heterogeneous context.
3 STATE OF THE ART: SCHEDULING FOR HETEROGENEOUS CONTEXT

This chapter is fully dedicated to the topic of scheduling and is part of the State of the Art of the thesis. The goal of this chapter is to present a set of scheduling algorithm in order to compare them and to reuse some of their concepts.

3.1 Introduction

The scheduler is a central functionality of an operating system. In heterogeneous operating systems the scheduler design can be a complex question. In a heterogeneous context the responsibilities of the scheduler are larger than in a homogeneous context. The scheduler must decide which task should be allocated to what processing element at time T, by taking in account the different kind of processing elements.

In chapter 2, HOS have been described in detail, however in most research papers used in this thesis, the scheduling algorithm for each HOS was not clearly described. Scheduling for heterogeneous systems is an active field of research and we need theoretical material in order to propose our model of heterogeneous platform in chapter 4.

A classic scheduler is in charge of giving access to resources for process, threads or data flows. For HOS the scheduler can be also in charge of giving access to the suitable processing element, this can happen in real time or be pre-determined at compile time. By nature, in a heterogeneous context, processing elements are not interchangeable, which means that the scheduler choice to allocate a task on a particular processing element is not neutral and has to be done by taking in account efficiency criteria. The efficiency criteria can be the optimization of execution time, memory usage or energy consumption.

The goal of this chapter is to provide a description of a set of scheduling algorithms
specifically developed for heterogeneous contexts. The concepts and implementations that are studied in this chapter will be reused in chapter 4. In this chapter, we will focus on three different solutions for scheduling tasks in heterogeneous contexts.

We propose the following analysis grid in order to visualize and sum up the aspects that we will focus on three scheduling solutions that are described in detail in this chapter.

<table>
<thead>
<tr>
<th>-</th>
<th>Full Name</th>
<th>Main feature(s)</th>
<th>Main limitation(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HASS</td>
<td>Heterogeneous Aware Signature Supported</td>
<td>Signature based scheduling, Classes and Partitions to identify, categorize and schedule dynamically threads over processing elements</td>
<td>Many sources of imprecision: average number of instructions per cycle involved in the signature calculation, offline scheduling implies a cost in terms of accuracy</td>
</tr>
<tr>
<td>HEFT</td>
<td>Heterogeneous Earliest-Finish-Time</td>
<td>Accurate Recursive calculation for prioritizing tasks, low complexity, Two-phases scheduling: prioritizing tasks and processor selection, full online (dynamic) scheduling solutions</td>
<td>The calculations even with low complexity represents a high cost in terms of computing time, but also in terms of implementation time, heterogeneity characteristics (specific instructions, cache structure and size) is not taken in account</td>
</tr>
<tr>
<td>StarPU</td>
<td>-</td>
<td>Framework offering functionality in order to implement heterogeneous applications, multiple scheduling policies, data management enhanced for heterogeneity: moving data to memory before it is going to be needed</td>
<td>Application using StarPU are not generic: they are specifically developed to comply with StarPU, heavy rules to respect for the developer, stiff learning curve</td>
</tr>
</tbody>
</table>

To cover heterogeneity the three chosen scheduling algorithms implement different techniques. Each of these techniques provides advantages in terms of efficiency, accuracy and also implies costs in terms of precision or computing time.

### 3.2 Signature Supported Het-Aware Algorithm

In this section we will present the algorithm called Heterogeneity-Aware Signature Supported [34]. In HASS heterogeneity can be defined as a difference of computing power (frequency and cache size) among the processing elements, which implies that the difference of architecture (ISA) is not central in this study. The concept of ASISA (Asymmetric Single Instruction Set Architecture) [23] is used to define what kind of heterogeneity is taken in account in HASS.

HASS is a static scheduling algorithm using off-line monitoring to prepare and perform task scheduling. This static nature implies limitations, which are described in [34]:

30
• Only one signature per application: for each application a signature file is embedded in the executable, this signature file is data-dependent, which means that it does not cover different input sets. In other words, the accuracy provided by the signature mechanism is decreased because it is defined for a fixed amount of data.

• Signature implies cooperation from the application development side: in order to have signature for each application developers have to be involved. This represents an increased amount of work and a bias concerning the quality of the obtained signatures. However, the possibility of including signature’s creation process in the compiling phase represents a way of reducing the amount of work for this task.

• HASS does not take in account phase changes: this means that HASS cannot outperform the best static assignment. The main reason for this is that the signature persists for the lifetime of a program.

HASS also has advantages which are: better scalability, simpler implementation (thanks to the absence of dynamic scheduling), support for short-lived threads which would, in the case of dynamic scheduling, spend the major part of their lifetime in the performance monitoring stage. The explanations for these advantages are given in the following sections.

In the following sections we study in detail the mechanisms developed and offered by HASS. First we will describe how signatures are created and then we study how HASS performs task scheduling using signatures.

### 3.2.1 Signature calculation

The signature of an application is a summary of the architectural characteristics of one application. The goal of the signature is to prepare the prediction of relative performance of one application on different processing elements. The signature must be architecture independent, and it must be available at run-time: which is why the signature is included into the binary file of each application.

HASS is focused on ASISA, which means that processing elements having the same ISA but different clock frequency and/or different cache size represent the heterogeneous aspect of the scheduling algorithm. In order to predict performance variations of one application the degree of memory-boundedness is taken in account. A
high rate of memory access implies, an increase probability for the application to stall the core often, which means that clock-frequency will not be a significant parameter for performance improvement. This is why the signature takes into account the memory behavior of each application.

Memory-boundedness is captured by measuring application’s reuse-distance profile. A reuse-distance is the number of memory accesses between two memory access to the same memory location. The reuse-distance profile is the distribution of reuse distances. The goal of using reuse-distance profile is to accurately estimate the last-level cache miss rate for any cache configuration [35] [36]. These estimated miss rates are the contents of the signature [34].

To obtain the reuse-distance profile, the application developer has to use a specific profiling tool in order to include the signature information in the binary file. This means that the responsibility of providing adapted and relevant input data set during the profiling step is given to the developer. The developer must run the profile tool by ensuring that the application runs in a nominal scenario.

### 3.2.2 Using Signatures for Scheduling

At runtime, the signatures for each application are available, and they are used to estimate thread performance on each type of processing element on the hardware platform. By using signature, HASS calculates at runtime an abstract and hypothetical completion time. The calculation of the abstract completion time is based on two different parts:

- Execution time: by assuming a constant number of cycles per instruction, the execution time is calculated and represents the amount of time needed to execute. The clock frequency is also taken in account in the calculation.

- Stall time: at runtime the operating system provides the memory access latency and by using the signature a predicted miss rate is obtained. By using these two indicators coarse approximation is performed to calculate stall time.

By relying on thread’s caching behavior, cores’ cache size and clock frequency HASS proposes a prediction of performance of different threads running on different cores.
3.2.3 **HASS Scheduler**

HASS scheduler emphasizes scalability, and its structure and mechanisms are designed to improve it. HASS scheduler is based on a set of key abstractions that we define in this section:

- **Processor class**: is a set of properties such as clock frequency and cache structure. If two processing elements belong to the same class, it means that they are identical. With HASS, heterogeneity exists when at least two classes exist. HASS do not manage dynamic class creation, which means that HASS cannot take into account hot-plugging of processing elements.

- **CPU-partition**: owing to the amount of information in processor classes can be important due to the many-core nature of processing elements, the cores are also grouped in **CPU-partition**. Each partition belongs to the same processor class, although each class can contain one or more partitions and each core must belong to exactly one partition. A partition keeps a counter of **runnable threads** (currently running or ready to run), this counter is updated in real time time.

When entering the system, the threads iterates through all processors classes in order to compute an estimation of their performance based on the signatures and according to the properties of the class.

The scheduling algorithm is composed of these steps:

1. New **threads** enter the system

2. Iterates through all **processor classes** and perform an estimation of their performance based on the signatures and according to the properties of the class.

3. Each thread assigns itself to a partition, to do so, it parses the **list of all partition** and then a performance estimation is performed using the ratings from the previous step and the current number of **runnable threads** per core in the partition.

4. The partition with the highest estimated performance is selected.

This set of steps is called **regular assignment** which has a linear complexity. Each time a thread accumulates a certain amount of CPU time in its partition, the regular assignment process is repeated, this is called a **refresh**.
3.2.4 Conclusion

Clearly the accuracy is not the first goal of HASS algorithm as there are many sources for inaccuracy:

- As mentioned earlier in this section, the signature calculation is data dependent, and as it is computed off-line and as the responsibility of this calculation is given to the developer, the bias in this calculation is a potential source of massive unadapted and inaccurate measurements which will impact directly the scheduling algorithm at run-time;

- There is an assumed number of cycles per instruction, which implies an important level of imprecision in the calculation of application’s completion time. This assumption has an important consequence on the way the scheduling algorithm will allocate tasks to processing elements.

In [34], it is said that HASS has been designed by performing a trade-off between simplicity and accuracy, at the expense of accuracy. By providing an off-line signature calculation process, HASS algorithm offers a new way to perform scheduling for heterogeneous platform based on ASISA multi-core processors. Signature based scheduling gives the possibility to create a complex and adaptive scheduling policy for our solution presented in chapter 4. This signature mechanism represents the main interest for our study and this is why we presented HASS in this chapter.

3.3 Heterogeneous Earliest-Finish-Time (HEFT) Algorithm

In this section we study the algorithm called Heterogeneous Earliest-Finish-Time [37]. This algorithm is designed to schedule application on a bounded number of heterogeneous processors.

3.3.1 Upward and Downward Rank Calculation

In HEFT tasks are ordered by their scheduling priorities which are based on the calculation of their upward (from exit point of the task graph to the entry point) and downward ranking (entry point to exit point of the task graph). This algorithm offers
low complexity and is designed to operate a static scheduling of a set of tasks on heterogeneous processing elements. The **upward rank** of a task is defined recursively by this formula:

\[
rank_u(n_i) = \overline{w}_i + \max_{n_j \in \text{succ}(n_i)} (\overline{c}_{ij} + rank_u(n_j))
\]

- \(\text{succ}(n_i)\) is the set of immediate successors of task \(n_i\)
- \(\overline{c}_{ij}\) is the communication cost of edge \((i, j)\)
- \(\overline{w}_i\) is the average communication cost of the task \(n_i\)

The upward rank is calculated recursively, starting from the exit task \(n_{exit}\) to the entry task \(n_{entry}\). For \(n_{exit}\) the upward rank is equal to:

\[
rank_u(n_{exit}) = \overline{w}_{exit}
\]

\(rank_u(n_i)\) represents the length of the critical path from task \(n_i\) to the exit task \(n_{exit}\) including the computation cost of task \(n_i\).

The **downward rank** of a task \(n_i\) is recursively defined by this formula:

\[
rank_d(n_i) = \max_{n_j \in \text{pred}(n_i)} \{rank_d(n_j) + \overline{w}_j + \overline{c}_{i,j}\}
\]

\[
rank_d(n_i) = \max_{n_j \in \text{pred}(n_i)} \{rank_d(n_j) + \overline{w}_j + \overline{c}_{j,i}\},
\]

With \(\text{pred}(n_i)\) the set of immediate predecessors of task \(n_i\).

The downward ranks are calculated by recursion, starting from the entry task of the graph. For the entry task \(n_{entry}\) the downward rank is equal to zero. The downward rank of a task \(n_i\) \((rank_d(n_i))\) represents the longest distance between the entry task and task \(n_i\) excluding the computation cost of the task \(n_i\).
The HEFT algorithm is described by the pseudo code below:

1. Set the computation costs of tasks and communication costs of edges with mean values
2. Compute upward ranking for all tasks by traversing graph upward, starting from the exit task
3. Sort the tasks in a scheduling list by non-increasing order of upward ranking values
4. while (there are unscheduled tasks in the list)
5.   |
6.   Select the first task "N", from the list for scheduling
7.   for (each processor "P" in the processor set "P_set")
8.   |
9.     compute earliest finish time (EFT) for N on P using the insertion based scheduling policy
10. |
11. Assign task N to the processor P_j that minimizes EFT of task N
12. |

HEFT consists of applying a scheduling algorithm for a limited number of heterogeneous processing elements, through two major phases: a task prioritizing phase for calculating the priorities of all tasks and processor selection phase for selecting the tasks depending on their priorities and scheduling each task on the “best” processor, which minimizes the task’s finish time.

Figure 3.1: A sample set of 10 tasks, for HEFT [37]
Tasks Prioritizing Phase

In that phase the upward ranking \((rank_u)\) of each task is calculated and then the tasks are stored in a list sorted by decreasing order of \(rank_u\). The decreasing order of \(rank_u\) preserve the precedence constraints [37].

Processor Selection Phase

The HEFT algorithm processor selection phase consists of an insertion-based policy which considers the possible insertion of a task in an earliest idle time slot between two already scheduled tasks on a processing element. The search for an appropriate idle time slot of a task \(n_i\) on a processing element \(p_j\) starts when all input data of \(n_i\) that were sent by its immediate predecessors tasks have arrived at processing element \(p_j\). The search will continue until a suitable idle time slot is found. A suitable idle time slot is a time slot that is capable of holding the computation costs of task \(n_i\). The complexity of the HEFT algorithm is:

\[
O(e \times q)
\]

where \(e\) is the number of edges and \(q\) is the amount of processing elements.

Figure 3.2 represents the schedules computed by the HEFT algorithm for the sample Directed Acyclic Graph of Figure 3.1. Schedule length comparison to other static scheduling algorithm:

- For HEFT the total schedule length is 80 unit of time
- For Dynamic Level Scheduling [38] the total schedule length is 91 unit of time
- For Mapping Heuristics [39] the total schedule length is 91 unit of time
3.3.2 Conclusion

We presented HEFT in order to introduce the notion of upward and downward ranking calculation as well as the two phases scheduling algorithm. Having a task-prioritizing and a processor selection phases is a straightforward and clear way of organizing the scheduling of application on heterogeneous processing elements. Plus, HEFT emphasizes on having a low complexity to deliver efficient scheduling, and this is an interesting advantage for our goal which is embedded platforms.
3.4 Task Scheduling using Unified Platform on Heterogeneous Multi-core Architectures

In this section we will analyze the unified platform for task scheduling called StarPU and one of its scheduling policy for heterogeneous context. StarPU [40] is a framework that allows the programmer to create programs that are not dependent of any hardware platform. In other words, StarPU gives a homogeneous way of programming, on heterogeneous platform. To achieve this, StarPU implements different techniques concerning:

- Task versioning and Task parallelism
- Data management
- Performance optimization

3.4.1 Global structure

With StarPU the notion of “codelet” is used: a codelet models a multi-version kernel of computation. In other words, a codelet is a structure that describes on which architectures a kernel of computation can be run. A task in StarPU, consists of associating a codelet with a data set.

To execute a task StarPU has a generic procedure:

1. StarPU gets the task sent asynchronously by the application;

2. Thanks to the chosen scheduling policy (described later) the task is allocated to a specific processing element;

3. The data needs to be available for the chosen processing element and the data transfer has to be hidden to the programmer. StarPU will replicate or migrate the data to the right memory. This is possible because a “distributed shared memory” (DSM) is used to centralize the information about the data set;

4. Once the data are moved, the computation is moved to the chosen processing element;

5. When the execution is done the application gets a notification (optional), the tasks that were waiting can be scheduled.
In this procedure, the scheduling policy has a very important role: it determines on which processing element the task will be executed.

### 3.4.2 Data Management

In order to ensure the availability and the consistency of the data for the different processing elements, StarPU provides a library that implements a software version of the DSM. This means that a high level interface is usable by the programmer, in order to represent data as an abstract object which can be moved and checked in a generic way. StarPU uses a write-back approach for the data management, this policy of data transfer is qualified as “lazy”, which means that the transfer are performed “just in time”. For instance, if a matrix has to be read from a processing element $p$, StarPU checks if this matrix is available for $p$, if not then the matrix is replicated on $p$.

### 3.4.3 Scheduling

As the data transfers are time consuming, StarPU takes in account the locality for scheduling tasks. StarPU provides a high level interface which give the possibility to describe scheduling policies. This high level interface is a generic way that is given to the programmer in order to write portable scheduling policies. Writing a portable scheduling policy consists in choosing a structure for the task queues, the way to organize these queues and in writing the methods which will be called when a task is submitted or selected in one of these queues.

StarPU provides the following set of policy [40]:

- greedy : Greedy policy with support for priorities
- no-prio : Greedy policy without support for priorities
- ws : Greedy policy based on Work Stealing
- w-rand : Random weighted by processor speeds
- heft : Heterogeneous Earliest-Finish-Time
3.4.4 Conclusion

StarPU provides a framework to manage heterogeneity by giving a set of rules and tools to the developer. StarPU is not a scheduling algorithm, instead it gives to the developer the choice among a set of scheduling algorithm. This is a very flexible way to manage scheduling for heterogeneous platform: once the application is implemented, the scheduling policy can be changed so that the developer can perform benchmark and evaluate which policy offers the best performance.

3.5 Conclusions

In this section we have presented the HEFT algorithm particularly, the HASS algorithm and the StarPU framework which gives the possibility to create application for heterogeneous platform. Scheduling for heterogeneous platform is a wide-topic and our goal in this section was to give an overview of different available solutions. Our objective is to build a simple, efficient and generic platform for heterogeneous context, therefore the concepts presented in this section are a important base for our solution presented in chapter 4.
In this chapter, we propose a model for our *Generic Heterogeneous Operating System* (GHOS). The generic aspect has two significations:

- Generic adaptation to hardware platform: porting the operating system for a hardware platform must be simple and must not involve heavy modification of the operating system structure to have it working properly;

- Generic programming of application: when developing an application the developer should not care about the architecture of the hardware platform.

In order to present an ideal and simple representation of our Generic Heterogeneous Operating System: we propose a model that describes each component of our solution. In this chapter we describe both GHOS (section 4.2) and the set of tools, GHOS-T (*Generic Heterogeneous Operating System - Tools*), that we provide in order to create application and deploy GHOS. The goal of this generic aspect is to emphasize simplicity for GHOS users and also to anticipate the fact that GHOS will probably change a lot during implementation phase.

### 4.1 Overview: Use Cases

In this section we present a use-case diagram, relying on UML 2 [41], in order to show how the responsibilities are divided in GHOS-T. This use case diagram is visible in Figure 4.1. This Use Case Diagram aims at showing which are the actors involved in using GHOS and what are their responsibilities. In addition, describing this use-case diagram is a way to explain and define the vocabulary we use in GHOS-Tools (GHOS-T). In Figure 4.1 there are three packages:
• High Level Application: contains all actions and actors related to the creation and preparation of application.

• Build System: describes what happens at compile time

• Low Level Application: describes what kind of action are performed by GHOS

From a vocabulary point of view, high level applications are applications running on GHOS, low level application in our case is GHOS and the Build System is the set of tools and templates that we provide in order to be able to build and compile both high and low level applications.

In this section we succinctly describe each package in order to give an overview of GHOS and GHOS-T.
Figure 4.1: Use Case Diagram for GHOS
4.1.1 **Package: Low Level Application**

Low Level Application is the operating system, in our case GHOS, in this package there are two actors:

- **Node**: GHOS typical functionality:
  - Execute Tasks: by using High Level Application built by the Build System tasks are executed by the Node (In this document High Level Application are defined as a set of tasks).
  - Local Scheduling: as there will probably be multiple tasks, we need a simple scheduling policy at a local level. *Local* in this context relate to the fact that a Node is an instance of GHOS running on a particular core and memory space.
  - Send and Receive Data: a node will always receive orders from a Coordinator Module and will have to send back replies.

- **Coordinator Module**: is part of the node, it can send orders to nodes (including itself) in order to divide the work depending on the heterogeneous directives from the executed High Level Application. Heterogeneous directives are message passing functions that triggers a transmission of data and the beginning of a new task, they are detailed in section 4.3.
  - Schedule Tasks On Node: the Coordinator Module has the responsibility of taking in account heterogeneous directive and allocate tasks to the designated *type of core* (see subsection 4.4.1)
  - Send and Receive Data: the Coordinator Module sends and receives data to and from Nodes after they have executed allocated tasks

Our Low Level Application, GHOS, is fully described in section 4.2

4.1.2 **Package: High Level Application**

High Level Applications are applications destined to run on GHOS. In this document we define an application as a set of tasks and a task as a set of operations. Although this package is not directly part of GHOS it is important to describe it, as creating application represents the main interaction with our system.

In this package, we have identified two actors:
• Developer: the developer has three functions:
  – Create High Level Application: basically programming, respecting templates provided by the Build System
  – Provide High Level Application Description: using a template provided by the Build System in order to prepare its deployment
  – Provide Compiler Information: using a template provided by the Build System in order to prepare its compilation

• Integrator: has two functions
  – Provide Compiler Information: can be responsible of describing compilation information
  – Describe Target: using a template provided by the Build System in order to describe the hardware platform, see Code 4.2, Code 4.3 and Figure 4.6

High Level Applications are fully described in section 4.3

4.1.3 Package: Build System

The build System is in charge of providing templates to the Developer/Integrator in order to achieve the building process of both High and Low Level Applications, this actor has four actions:

• Provide Template: a set of XML Schema document are provided in order to describe how to write XML files that will be use as templates for different purpose: describing High or Low Level Applications, defining compilation process and describing Target (hardware platform)

• Check Template Validity: the Build System is in charge of validating each template (XML files) based on its model (XML Schema), if one template is not valid the build process is stopped

• Build Low Level Application: relying on template, the build system, build and compile Low Level Application for the chosen Target

• Build High Level Application: relying on template, the build system, build and compile High Level Application for the chosen Target
• Create Binary File: this operation is there in order to anticipate the technical deployment constraint on hardware platforms that will probably require a binaries of High and Low Level Applications to be packed together in order to be sent on the Target.

The Build System is described in section 4.4.

4.2 Low Level Application: GHOS

GHOS is the core solution that we provide in this document, we have based this operating system on a set of rules and guidelines related to the objectives and constraints described below.

4.2.1 Objectives and Constraints

The main objectives and constraints for building GHOS are:

• Performance: the goal of using GHOS is to improve performance criteria such as execution time and possibly energy consumption

• Generic: the hardware must be handled so that developer creating applications (High Level Application) for GHOS should not have to modify the application’s code depending on hardware architecture

• Scalable: GHOS must be able to handle an undefined number of processing elements and communication channels so that executed application are able to distribute tasks efficiently over the processing elements

• Modularity: GHOS and GHOS-T are divided into modules and components that interact through interfaces, these modules and components are based on the Separation of Concerns Principle which states that "A computer program should be divided into distinct sections, such that each section addresses a separate concern."

4.2.2 GHOS Kernel

In GHOS each processing element runs an instance of the kernel of the Operating System. A GHOS Kernel provides two main functionality: scheduling and commu-
communication. The scheduling and execution of tasks are performed by an entity called the Node. The communication are handled by the Coordination Module. The architecture of the GHOS Kernel is visible in Figure 4.2. The Node and the Coordination Module are in constant interaction: the Coordination Module updates the queue of tasks to be executed and the Node, when needed, updates the queue of tasks that are sent by the Coordination Module to other Node.

**Description of GHOS Kernel**

This description is based on Figure 4.2. The Node and the Coordination Module are running in parallel and each of them have different roles

- A Node has two roles
  - If the *Receive Queue* is not empty, the Node reads the tasks and execute all the operations it contains
  - If, in a task, one of the operation is a Send operation then the Node updates the *Send Queue*. A Send operation is a function call to GHOS-API (See Chapter 6.2), it is used to transfer data and triggers the execution of a new task on any available Node.

- A Coordination Module has two roles
  - It listens for incoming tasks and updates the *Receive Queue*
  - It listens for updates on the *Send Queue* and perform a Send operation

**4.2.3 Scheduling**

In GHOS Kernel, in Figure 4.2, the scheduling is performed at two different level:

- Scheduling at a Node Level: greedy scheduling: jobs are received and executed as soon as possible in order of arrival;

- Scheduling between Nodes: the scheduling map is given by the High Level Application as in Figure 4.4, the action of sending work to other GHOS-Kernel is performed by the Coordination Module.
In Figure 4.4, each box is a job performed by a Node and each arrow is a send/receive operation performed by Coordination Module. For the case of the three boxes marked as "3D", if the hardware platform has only one processing element tagged as "3D", the three "3D" jobs will be allocated to the same Node.

By relying on the hardware description document in Code 4.3 and on the description of a High Level Application in Figure 4.4, we propose a graphical representation of the schedule performed by GHOS in Figure 4.3. We provide execution time values for each jobs in Figure 4.4:

<table>
<thead>
<tr>
<th>Task Name</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.1: Execution time for each task in the proposed example of High Level Application
The total execution time of all the jobs is 10 and the length of the schedule applied to GHOS is 7 (cf 4.3).

<table>
<thead>
<tr>
<th>Type of Core</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3D – nVidia</td>
<td>t3</td>
<td>t4</td>
<td>t5</td>
</tr>
<tr>
<td>GeneralPurpose – Core2Duo</td>
<td></td>
<td></td>
<td>t6</td>
</tr>
<tr>
<td>VideoEncoding – VideoEncoding – A9</td>
<td>t1</td>
<td>t2</td>
<td></td>
</tr>
<tr>
<td>Time &gt;</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 4.3: Schedule for an Instance of High Level Application

### 4.2.4 Inter-Node Communication and Memory Management

In GHOS, inter-process communication are transparent IPC, which from a technical point of view are message passing operations. These message passing operations provide two main functionality: data transfer and triggering the execution of a new task on a processing element.

In GHOS-API (See Chapter 6.2) we provide two functions: Send and Receive. The Send function specifies which data to transfer and which type of processing element is supposed to receive and process these data and which task from which application has to be executed. The Send function does not require a specific processing element as a parameter, but simply a type of processing element (as describe in Code 4.2), which means that any processing elements of the specified type can execute the request sent by the Send function. The role of the receive function is to listen for incoming task and data.

Thus, these functions (Send & Receive) provide a solution to handle communication between processing elements, data transfer and part of heterogeneity management.
4.3 High Level Application: Creating application for GHOS

In GHOS, High Level Application are the applications running over the GHOS-Kernel. A High Level Application is created by a developer, who is given a set of guidelines to respect in order to create a working program for GHOS. To achieve the creation of an application for GHOS the developer has to:

- Program the application by using the provided primitives, which are used to allocate jobs to the right type of core;
- Describe the application in an XML Document, using the provided template (XML Schema, see Code 4.4 and Code 4.5) (In Figure 4.1 the actor having this role is the Integrator);

![Diagram of High Level Application](image)

**Figure 4.4: Dependency graph for an Instance of High Level Application**

In Figure 4.4 we propose an example of visual representation for an Instance of High Level Application in which we describe tasks dependencies. From this diagram we propose an example of pseudo-code in order to show how the code produced by the Developer should be implemented. This example of code is visible in Code 4.1. In Code 4.1, by relying on the Type of Processing Element (provided in the target file, see Code 4.2 and Figure 4.6) the Developer creates a program that allocates jobs to different Processing Elements. By using the Send() function the data are sent and distributed by using a *tag* which is used to identify the *Send Request*. Each Receive() function specifies the *tag* it is listening for. The interest of this methodology is that no
specific processing element is targeted by the developer, GHOS is in charge of finding a GHOS-Kernel of the right type which is available to perform a job. A visual representation of the schedule of this specific High Level Application running on GHOS is visible in Figure 4.3.

```c
main()
{
  if(processingElementType == GeneralPurpose)
  {
    #GeneralPurposeCode
    Send(videoData, videoType, tagVideo)
    Send(3DData, 3DType, tag3D)
    #GeneralPurposeCode
  }
  if(processingElementType == 3D)
  {
    Receive(3DData, tag3D)
    #3DCode
    Send(3DData, 3DType, tag3D1)
    Send(3DData, 3DType, tag3D2)
    Receive(3DData, tag3D1)
    if(received)
    {
      #3DCode
      Send(GeneralPurposeData, GeneralPurposeType, tagGeneralPurpose3)
    }
    Receive(3DData, tag3D2)
    if(received)
    {
      #3DCode
      Send(GeneralPurposeData, GeneralPurposeType, tagGeneralPurpose4)
    }
  }
  if(processingElementType == Video)
  {
    Receive(VideoData, tagVideo)
    #VideoCode
    Send(GeneralPurposeData, GeneralPurposeType, tagGeneralPurpose1)
  }
  if(processingElementType == GeneralPurpose)
  {
    Receive(GeneralPurposeData, tagGeneralPurpose1)
    #GeneralPurposeCode
    Send(GeneralPurposeData, GeneralPurposeType, tagGeneralPurpose2)
  }
  if(processingElementType == GeneralPurpose)
  {
    Receive(GeneralPurposeData, tagGeneralPurpose2)
    Receive(GeneralPurposeData, tagGeneralPurpose3)
    Receive(GeneralPurposeData, tagGeneralPurpose4)
    #GeneralPurposeCode
    exit();
  }
}
```

Code 4.1: Example of code for High Level Application based on Figure 4.4 (The message passing operations are used to transfer data and tasks)
4.4 Build System

In our solution we include the building system which provides to the developer a set of tools in order to achieve the application deployment on the chosen hardware platform. In this section, we first present a general model for the building system and then we present each key-element of this general model.

In Figure 4.5, we present the general model for building application for GHOS. This model contains a set of components interacting with the central component which is the BuildSystem. Each component has a different role, which we describe succinctly below:

- The build system: Compile high level application and/or low level application by relying on the description file specific to compiler and target (hardware platform);

- The target: is the hardware platform, in our model it is parsed by the build system by relying on a description file written in XML which is presented in this section (see Code 4.2);

- The compiler: is the program that creates the application file binary, in our model it is called by the build system by relying on a compilation file;

- High Level Application: program that are supposed to run on the chosen hardware platform

![Diagram of Build System Model](image-url)
• Low Level Application: the operating system we want to install on a given target. This means that the build system is able to deploy the operating system itself for a given target.

The building system is open for extension\textsuperscript{1}, which means that by design it allows future evolution without implying heavy changes on the whole model structure: there can be many targets, many high level applications, many low level application and the build system should be able to handle them. However for our first implementation we will focus on implementing a build system that handles a set of high level application for only one low level application running on one target.

\subsection{Target Template: Handling the Hardware}

In order to compile application for GHOS, a description of the hardware platform is required. This description is done in a file (described in this section), which is parsed during the compile process. In the General Model (Figure 4.5), this description of the hardware platform is called the target template.

In order to build application, for a specific target, we provide a model (cf Code 4.2 and Figure 4.6) implemented in XML Schema. Providing a XML Schema means that a XML document has to be implemented by following the rules of this XML Schema. The XML Schema can be used to validate the structure of the XML document (cf Figure 4.7).

Figure 4.6 provides a visual representation of the rules required to describe a hardware platform. Code 4.3 provides an example of a hardware platform described in XML syntax complying with the rules from the schema. We propose a description of each element and attributes available in the hardware description model:

• \texttt{<core>} is used to describe one processing element, the element itself does not contain data, it is defined by a set of attributes, a \texttt{<core>} and its set of attributes can be replicated multiple time in the target template.

  – type: describe the type of processing element. This value is very important as it declares and defines which kind of task will be executed on the core.

\textsuperscript{1}In object-oriented programming, the open/closed principle states “software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification”\cite{1} that is, such an entity can allow its behaviour to be modified without altering its source code Source: Wikipedia
indicate which piece of code is supposed to execute on which type of core; the allowed types of core are described in the XML Schema 4.7, they can be added, modified or deleted;

- name: give a name to the <core>, if the same value is used multiple times it means that we designate the same core: in the last two cores of 4.3 "A9" is used twice, this means that the core called "A9" will be used for both Video Encoding and General Purpose.

- compiler: contain a link to a compilation file specific for this <core>

- frequency: this attribute is optional, the expected value type is an integer. The frequency attribute will be used for scheduling purpose: between two (or more) processing elements having the same type, the frequency can be used to differentiate these processing elements and allocate process with a higher priority to the processing elements having the higher frequency. There is no specific unit to use (Hertz, Mega-Hertz, Giga-Hertz), the only restriction is that the same unit has to be used in the document: if "Hertz" is chosen for one core, it has to be used for all the cores.

- <platformName> contains the name of the hardware platform. It is used to identify a specific target. In the case of multiple targets it is important that only

By using this template we provide a simple way to describe a platform and a simple way to parse this template.

Figure 4.6: Model for Target (hardware platform) description
Code 4.2: Model for hardware platform description (XML Schema)
Code 4.3: Instance of hardware platform description (XML complying with XML Schema presented in Code 4.2)

Figure 4.7: XML Document complies with XML Schema rules; the schema can be used to validate the XML document structure

4.4.2 High and Low Level Application Template

To prepare the building process of High and Low Level Applications the Integrator (4.1) have to provide templates. The purpose of these templates is to gather information that are not directly related to compilation and not related directly to hardware handling. At this step of the design of GHOS and GHOS-Tools we know that these template will exist and will be required by the Build Process, but as they totally depend on technical aspects we provide an implementation for the XML Schema that define them.

Code 4.4 presents the schema to be respected in order to create a High Level Application description file. An instance of High Level Application description file is presented in Code 4.5. This description file must contain a set of four elements:

- applicationName: is a simple string in order to give a name to the application
• startingCore: indicate on which core the application is supposed to start, the Low Level Application running on this core will activate by default its Coordinator Module

• preferedTypeOfCore: will be used for scheduling purpose: if the required core is not available then this application will run on the core indicated in this XML Element

• targetFile: point to the target file

This files will evolve when we reach the implementation phase, at a modelling level it is important to anticipate the need for this file to exist and define the basic interaction with the other components of GHOS-T.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="highLevelApplication">
    <xs:complexType>
      <xs:sequence maxOccurs="1" minOccurs="1">
        <xs:element maxOccurs="1" minOccurs="1" name="applicationName" type="xs:string"/>
        <xs:element name="startingCore" type="xs:string" maxOccurs="1" minOccurs="1"/>
        <xs:element name="preferedTypeOfCore" type="xs:string" maxOccurs="1" minOccurs="1"/>
        <xs:element maxOccurs="1" minOccurs="1" name="targetFile" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Code 4.4: Model for High Level Application description file (XML Schema)

```
<?xml version="1.0" encoding="UTF-8"?>
<highLevelApplication xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="file:highLevelApplicationSchema.xsd">
  <applicationName>myHighLevelApplication</applicationName>
  <startingCore>GeneralPurpose</startingCore>
  <preferedTypeOfCore>GeneralPurpose</preferedTypeOfCore>
  <targetFile>targetFile.xml</targetFile>
</highLevelApplication>
```

Code 4.5: Instance of High Level Application description (XML complying with XML Schema presented in Code 4.4)

4.4.3 Build Process

Building High and Low Level Applications with GHOS-T is done in a set of steps: the easiest way to describe it is to use an Activity Diagram [41], visible in Figure 4.8.
In Figure 4.8, the action called "For each Target Compile High Level Application" is complex and deserves a detailed description visible in pseudo code sample below. The `send()` instruction mentioned in the pseudo-code is described in section 4.3.

```
For each Messaging Instruction in High Level Application Code
{
    Update list of target used by app
    Compare list of target with target description file
    if compare() does not match:
        {  
            Send Error
            Terminate Process
        }
    Update memory map relying on data type in Messaging Instruction
    Update schedule relying on designated type of core in Messaging Instruction and target file
}
For each target in list of target compile High Level Application
```

The first line of the pseudo-code above uses "Messaging Instruction", this concept is defined in 4.3
Figure 4.8: Activity Diagram for GHOS-T Build Process
4.5 Conclusion

In this chapter we have presented our Model for both GHOS and GHOS-T, to achieve this we relied on a set of concepts that were covered in state-of-the-art chapters of this thesis. We have provided a solution to build GHOS in respect of the specifications described in the introduction of this chapter. In the next chapter we present a Model to install Heterogeneous Operating Systems on a specific hardware platform: Panda-board equipped with Texas Instrument OMAP4460 chip. In this chapter, we propose an example of installation for two HOS presented in the state-of-the-art and one example for GHOS.

In chapter 6.2, we present a technical description for an implementation of a GHOS Prototype, running on a simulated Texas Instrument OMAP4460.
5 HOS & SPECIFIC HARDWARE PLATFORM: PANDABOARD / OMAP4460

From September 2012 to February 2013 we have performed experiments on the Pandaboard ES equipped with an Texas Instrument system on chip (SoC): OMAP 4460. This SoC is composed of many processing elements and other components which are describe in Figure 5.1. From this block diagram we consider mainly three main components:

- Dual Cortex-A9 Multi-Processor Unit
- Dual Cortex-M3 Multi-Processor Unit
- DSP Subsystem based on a derivative of the TMS320DMC64x+

Additionnaly, the Pandaboard is based on a shared memory design: all processing elements access to the same memory space through the "L3 interconnect".

The goal of this chapter is to present how could be installed on the Pandaboard, some of the operating system described in chapter 2. Then we present how our solution, proposed in chapter 4, can be installed on this hardware platform. To achieve this goal we propose a simplified version of the OMAP4460 block diagram which is available in Figure 5.2. In this diagram we keep three processing elements, the inter-connection channel and the RAM shared among the PEs.
Figure 5.1: Pandaboard Block Diagram [43]
5.1 Install Model for Helios on OMAP4460

In this section we use Helios Operating System as an example, in order to present a model for its installation on the Pandaboard ES-OMAP4460. Helios is based on "Satellite kernel" which are microkernels, in Figure 5.3 we present a simplified version of one Helios satellite kernel. This figure contains the main elements for a Helios satellite kernel: scheduler, memory manager, namespace manager, communication manager. Details for all these functionnality are given in chapter 2. In Figure 5.4, each red rectangle labelled "Helios Kernel" is an instance of the Kernel presented in Figure 5.3. In this installation each cores needs a kernel instance which will be in charge of scheduling, memory management, communication with other kernels and namespace management. One of the kernel install on the dual Cortex-M3 has been declared as "Coordinator" (green rectangle in Figure 5.4), this means that this kernel will boot first and will be the central point of the Namespace management.

![Figure 5.2: Simplified OMAP4460 Block Diagram, based on [43]](image)

Even if the memory is shared, the way the memory management is implemented in Helios imply the use of only one kind of interface which give access to message passing functionality; the mechanism for message passing in Helios is described in chapter 2.

Helios’ global structure makes it scalable: when a processing elements is added, a new kernel instance is needed. This makes Helios installation model extensible, but with some limitations: as each kernel needs to communicate with the coordinator kernel there is a risk of bottleneck when the amount of namespace-related request increases.

Helios’ mechanisms are not only visible in its kernel organization and implementation, but it is also based on mechanisms that are important before or during compile time. The main mechanism of Helios is the affinity mechanism, in Figure 5.5 this is
why we have represented a set of applications using affinity installed on Helios Simplified Kernel on OMAP4460. In this Figure we have defined three application:

- Application 1: with a positive affinity and instantiated once
- Application 2: with a positive affinity and instantiated once
- Application 3: with a negative affinity and instantiated twice

Figure 5.3: Simplified Helios kernel based on [15]

In Figure 5.5, the application deployment is done respecting affinities choices for each application, this implies that the two instances of Application 3 because of their negative affinities are allocated to two different cores of the same processing element. Therefore the negative affinity expresses also a self-negative affinity, which means that Application 3, as long as it is possible and feasible, needs to run "alone". The two other applications have positive affinities and then they use the remaining processing elements not used by Application 3.

Figure 5.4: Helios Simplified Satellite Kernel on OMAP4460 based on [15] [43]
Helios has a decentralized structure which improves scalability, however the coordinator kernel represents a risk of bottleneck. Each process needs to communicate with the coordinator kernel for Namespace-related purpose (register, query). With the affinity mechanism Helios covers in a simple way the allocation of tasks onto processing elements. The affinity mechanism avoids for the developer an increased amount of work to perform task mapping, task allocation and hardware platform description. This is done at the expense of accuracy and flexibility: the affinity mechanism implies that there is no evolution of the affinity at run-time, which means that an application is always considered in the same state and always requires the same access to processing elements. Additionally, to provide more accuracy in task allocation, affinity mechanism should rely on a description of the hardware platform which is not available with Helios.

Figure 5.5: Set of Application Installed on Helios Simplified Kernel on OMAP4460
5.2 Install Model for Tessellation on OMAP4460

In this section we present a model for installing Tessellation on Texas Instrument System on Chip OMAP4460. Tessellation OS is presented in detail in chapter 2. Tessellation is based on the concept of "microkernel" which means that a lot of responsibilities are given to the developer so that when the applications are created they include close-to-hardware-instruction. As it is visible in Figure 2.4 (Page: 15), the application running on the Exokernel can interact in a more direct way with the hardware compared to what is allowed with "classic" kernels. Basically Tessellation provides a scheduling and a resources sharing mechanism. The interest of Tessellation relies its space-time partitioning mechanism which is described in chapter 2.

![Simplified Tessellation Kernel](image)

**Figure 5.6: Simplified Tessellation Kernel, based on Figure 2.7**

A simplified Exokernel architecture of Tessellation is visible in Figure 5.6. The entire block is the kernel, which is divided in two main sections:

- **Partition Management**: which contains the scheduler and the partition manager. This part of the kernel can send and receive requests and replies to and from applications;

- **Partition Implementation**: interacts with the partition management layer and the hardware, no application has access to this layer. Hardware in this case is taken in a wide view, it includes: processing elements, memory and also hard-drive and/or memory card.
Relying on Figure 5.2, we propose an install model of Tessellation on Texas Instrument OMAP4460 visible in Figure 5.7. Tessellation Kernel covers the hardware and deals with its space-time partitioning. In Figure 5.8, we instantiated one application that runs on Tessellation. The application is allowed to request Tessellation kernel for: querying resources, scheduling constraints (such as priority) and receiving partition re-sizing. Each application running on Tessellation must include the Library OS functionality in order to be able to work in cooperation with Tessellation’s Kernel. This means that using Tessellation is not neutral for the application’s code. The increased complexity on the developer side has at least one main advantage which is: performance. Tessellation clearly emphasizes its features towards performance (in this case execution-time): by giving more freedom to the developer and a complex system for managing hardware partitioning including dynamic partition re-sizing, it opts for an optimized management of heterogeneity. In our solution presented in chapter 4 we have chosen different paradigms that lead to an operating system that does not interfere heavily with application’s code, thus our solution is very different from Tessellation. In the next section we present our model to install our solution on Texas Instrument OMAP4460.
5.3 Install Model for GHOS on OMAP4460

In this section, by relying on the data-sheet [43] and on GHOS and GHOS-T described in chapter 4, we provide an install Model for GHOS on OMAP4460. This section also describes the work-flow for the Developer, and the first step in this work-flow is to describe the target which is the SoC OMAP4460. In Code 5.1 we provide a description file for OMAP4460 as a Target for GHOS installation. In Figure 4.2 we provide a synthetic visual representation of the GHOS Kernel and in Figure 5.10 we provide a graphical representation of GHOS installed over Texas Instrument OMAP4460.

<table>
<thead>
<tr>
<th>Task Name</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.1: Execution time for each task in the proposed example of High Level Application
A GHOS kernel is installed on each processing element of the OMAP4460, and in Figure 5.10 we have added a label on each core in order to show which core is tagged for "General Purpose", "Video Encoding" or "3D" tasks. In order to provide an image of how the tasks are mapped with GHOS we will use the High Level Application dependency graph from Figure 4.4. For each task in Figure 4.4 we have described the execution time in Table 5.1.

In Figure 5.9 we provide a table containing the calculated schedule for our installation of GHOS over OMAP4460. The total execution time is 6. We provide in Figure 5.11 a graphical representation of the tasks described in Figure 4.4 running over GHOS installed on OMAP4460. In Appendix A (Page: 82), we propose a prototype for GHOS, GHOS-T, a simulated OMAP4460 and a High Level Application.

Figure 5.9: Schedule for High Level Application running on GHOS over OMAP4460

<table>
<thead>
<tr>
<th>Type Of Core</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeneralPurpose – Cortex M3 – 1</td>
<td>t1</td>
</tr>
<tr>
<td>GeneralPurpose – Cortex M3 – 2</td>
<td>t2</td>
</tr>
<tr>
<td>VideoEncoding – Cortex A9 – 1</td>
<td>t3</td>
</tr>
<tr>
<td>GeneralPurpose – Cortex A9 – 2</td>
<td>t1</td>
</tr>
<tr>
<td>3D – Cortex M3 – 1</td>
<td>t4</td>
</tr>
<tr>
<td>3D – DSP – C64+</td>
<td>t5</td>
</tr>
<tr>
<td>Total</td>
<td>6</td>
</tr>
</tbody>
</table>

Code 5.1: Target File describing OMAP4460 for GHOS
Figure 5.10: GHOS over OMAP4460 based on section 4.2 and [43]

Figure 5.11: High Level Application Schedule running on GHOS over OMAP4460
6 FUTURE WORK AND CONCLUSIONS

6.1 Future Work & Further Improvements

This thesis has set the foundation for the future work of implementing an operating system for heterogeneous platform. The model that we provide for GHOS is minimal, however it gives guidelines for a first implementation which is the logic future work. GHOS’ model will require improvements concerning more technical aspects such as compilation or inter-kernel communication management.

A hardware platform such as the Pandaboard equipped with OMAP4460 requires drivers to handle the hardware and provide API to use this hardware. As the time of the writing of this thesis the driver to communicate with the DSP of the OMAP4460 is still a work in progress on Texas Instrument’s side. In any case, a first implementation of GHOS can be done on a simpler hardware platform. Another room for improvement is actually the need to specify and implement the way GHOS is adapted to a hardware platform.

Concerning further improvements, the provided Model is extensible and should support evolution related to the implementation of scheduling policies management such as the one presented in Chapter 3. Providing a way to offer more accurate scheduling solutions is important to actually give the possibility of exploring different kind of optimizations: energy consumption, execution time. The Model will also evolve during the implementation process this is why we have designed GHOS’ model in a modular and extensible way.

As a basis for further implementation we propose in Appendix A a prototype for GHOS, GHOS-T with a simulated hardware platform and a High Level Application running on this platform.
6.2 Conclusions

Since electronic and embedded devices are becoming more and more complex, and as the heterogeneity requires to be handled, the need for adapted solution on the operating-system-side is critical. Providing OS-Level solution for heterogeneous platform is a scientific and technical challenge. In fine, our goal is to prepare the arrival of three-dimensional chip (AMEBA project) that gather Data storage, RAM and heterogeneous processing elements.

We have studied a set of existing solutions of heterogeneous operating system and scheduling algorithms for heterogeneous context. This matter has been discussed in chapter 2 and chapter 3. Then we have provided a model for our solution GHOS in chapter 4. And we finally presented a model for installing two solutions presented in chapter 2 and GHOS on the Pandaboard equipped with the System On Chip Texas Instrument OMAP4460.

In this Thesis, we have provided a model to cover heterogeneity and build an operating system that we have called GHOS. Our goal is to pursue the study on this topic and implement a simple and efficient operating system for heterogeneous platform. In Appendix A we provide a first implementation of GHOS and GHOS-T in a prototype called GHOS-Prototype.
LIST OF FIGURES

2.1 SMP vs NUMA .............................................. 6
2.2 Barrelfish multikernel model .................................. 10
2.3 Barrelfish physical structure ................................... 11
2.4 Exokernel architecture [30] ................................. 15
2.5 Possible exokernel system [31] ............................... 16
2.6 Space-Time Partitioning in Tessellation: a snapshot in time with four spatial partitions [19] ................................. 16
2.7 Components of a Tessellation system. The Tessellation Kernel utilizes a combination of hardware and software mechanisms to enforce partition boundaries [19] .............................................. 17
3.1 A sample set of 10 tasks, for HEFT [37] ...................... 36
3.2 Scheduling of task graph in Figure 3.1 [37] ................. 38
4.1 Use Case Diagram for GHOS ................................. 44
4.2 GHOS Kernel ................................................ 49
4.3 Schedule for an Instance of High Level Application .......... 50
4.4 Dependency graph for an Instance of High Level Application .... 51
4.5 General model for the building process, based on [42] ........ 53
4.6 Model for Target (hardware platform) description .......... 55
4.7 XML Document complies with XML Schema rules; the schema can be used to validate the XML document structure .............. 57
4.8 Activity Diagram for GHOS-T Build Process ............... 60
5.1 Pandaboard Block Diagram [43] ............................. 63
5.2 Simplified OMAP4460 Block Diagram, based on [43] ......... 64
5.3 Simplified Helios kernel based on [15] ....................... 65
5.4 Helios Simplified Satellite Kernel on OMAP4460 based on [15] [43] .............................................. 65
5.5 Set of Application Installed on Helios Simplified Kernel on OMAP4460 ................................. 66
5.6 Simplified Tessellation Kernel, based on Figure 2.7 .......... 67
5.7 Tessellation Kernel on OMAP4460, based on [43, 19, 16] .......... 68
5.8 Application running on Tessellation Kernel on OMAP4460 .... 69
5.9 Schedule for High Level Application running on GHOS over OMAP4460 ................................. 70
5.10 GHOS over OMAP4460 based on section 4.2 and [43] .......... 71
5.11 High Level Application Schedule running on GHOS over OMAP4460 .... 71
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>GHOS Prototype: Class Diagram</td>
<td>83</td>
</tr>
<tr>
<td>6.2</td>
<td>Receive Registered Application: Activity Diagram</td>
<td>85</td>
</tr>
<tr>
<td>6.3</td>
<td>Receive Task: Activity Diagram</td>
<td>86</td>
</tr>
</tbody>
</table>
LISTINGS

4.1 Example of code for High Level Application based on Figure 4.4 (The message passing operations are used to transfer data and tasks) . . . . . . 52
4.2 Model for hardware platform description (XML Schema) . . . . . . 56
4.3 Instance of hardware platform description (XML complying with XML Schema presented in Code 4.2) . . . . . . . . . . . . . . . . . . . . . 57
4.4 Model for High Level Application description file (XML Schema) . 58
4.5 Instance of High Level Application description (XML complying with XML Schema presented in Code 4.4) . . . . . . . . . . . . . . . . . . . 58
5.1 Target File describing OMAP4460 for GHOS . . . . . . . . . . . . . 70
6.1 GHOS Prototype running three instances of High Level Application over simulated Texas Instrument OMAP4460 . . . . . . . . . . . . 88
6.2 GHOS Prototype running one instance of High Level Application over simulated Texas Instrument OMAP4460 . . . . . . . . . . . . . 89
BIBLIOGRAPHY


78


79


80

We have implemented a prototype for GHOS and GHOS-T, this chapter contains the description of our prototype for GHOS (GHOS-P).

6.3 GHOS Prototype: Overview

In our prototype for GHOS we have created an instance of the model given in chapter 4. In this section we propose an overview of the implementation of GHOS Prototype (GHOS-P). This prototype has three main goals: simulating a heterogeneous hardware platforms composed of many processing elements, running GHOS, running application(s) on GHOS and providing timing measurement.

6.3.1 Functionality

In this section we propose a list of all GHOS functionalities of GHOS that are covered by our prototype:

- Simulating hardware platform (target)
- Simulating processing elements (cores)
- Prototype implementation of GHOS Kernel, GHOS API, High Level Application and GHOS-T
- Memory management
- Message Passing
- Time measurement
6.3.2 Technology

To implement our prototype we have chosen Python, in order to benefit of this language’s advantages, such as: thread, simplicity, flexibility, simple object-programming. We use SQL to simulate memory behaviour: read, write and dynamic memory allocation. To describe the hardware platform (target) we use a XML file based on the structure described in the XML Schema (cf. Code 4.2). The interest of using SQL is that it provides a basic framework to simulate a typical memory behavior which avoid the need to implement an entire simulated memory environment. SQL offers functions to store, retrieve, update and delete data that were particularly convenient for developing GHOS-P.
6.4 GHOS Prototype: Implementation

In this section we describe in detail the implementation of GHOS-P. GHOS-P is programmed in Python using object-oriented programming, which means that the implementation relies on a set of class composed of methods and attributes (cf. Figure 6.1).

6.4.1 Global Structure

GHOS-P is composed of six classes and one script which we describe in this section. Most of the classes contains attributes, and for each attributes we provide a getter, setter, a constructor and, if necessary, an *add* method. Below we provide a description of each class, attributes and methods

The Memory class provides one method which gives access to a connection object to the memory space. The memory space is implemented in SQL, and we have chosen sqlite as a SQL library.

The Core class is composed of two attributes. Name is a string object that contains the name of the core. Type is an array that contains all the different type of core allocated to the core. This means that GHOS-P is able to handle multiple type of core for one core. For example, a core can defined as able to process Video and 3D instructions.

The Target class is composed of four attributes and three methods. This class provides the needed information and behaviour to simulate a heterogeneous hardware platform. Cores is an array of core composing the hardware platform. Name is the name of the hardware platform. TargetFilePath is a string object pointing the address of the XML File (In GHOS-P we use the same XML file as in Code 5.1) used to define the Target. Attribute ghosKernel is an array containing a set of GHOS Kernel instances, the number of instances is equal to the number of cores. Method installGhosKernelOnEachCore: creates an instance of a GHOS Kernel for each core in the array cores. Method startAllGhosKernel calls the method start on each GHOS Kernel, as each GHOS Kernel inherits from Thread, the start method launches a thread running the GHOS Kernel. A Core
The GhosKernel class provides the attributes and methods that define the behaviour of a GHOS Kernel as defined in chapter 4. The GhosKernel class inherits from thread, which means that it has to implement a run method, which contains the code that will be launched as a thread when calling start() on a GhosKernel instance. In our implementation the run method contains an infinite loop that executes sequentially node() and coordinationModule(). Attribute id is an identifier generated automatically when a GHOS Kernel is created. Attribute task is the current task to be executed by the GHOS Kernel, this attribute is a HighLevelApplication object. Method node execute the task and then removes it, so that it is not executed multiple times. Method coordinationModule runs sequentially two methods receiveRegisteredApplication and receiveTask. Method receiveRegisteredApplication performs a verification in memory in order to fetch a task which is starting. This methods is described in Figure 6.2. Method receiveTask reads the memory in order to retrieved a task to be executed that corresponds to one of the core type that is attached to the GHOS Kernel. This method is described in Figure 6.3.

Figure 6.2: Receive Registered Application: Activity Diagram
The GhosAPI class provides a set of methods that are meant to be used at the user-space level. Which means that GHOS Kernel interacts directly and in an independent way with any kind of hardware related task. However we need to provide a way for High Level Application to send and receive messages, perform join operation (one task receiving data from multiple sources), initialize memory space. The Methods Send and Receive are classical message passing functions. Method SyncJoin is a function that determines if a task requiring many data to be received, is ready to start. Method ReceiveForJoin is a special receive operation for tasks that require to receive messages from multiple sources. Method isStarting returns true if the name of the task given in parameter is marked as "starting" in memory. Method taskDone registers a tag (A tag is a specific state of a High Level Application) given in parameter as "done" in memory. Method isDone return true if a task is marked as "done" or returns false otherwise. Method registerApplication writes in memory a High Level Application marked as "registration". Method initializeMemorySpace instantiate the memory in order to create a memory space, in our case it creates a SQL file. Methods defineEndingTime and getEndingTime are used for measuring time in High Level Application.

The HighLevelApplication class provides attributes and methods in order to write application running over GHOS-Kernels by using GHOS-API. Attribute startingCore is a string object containing the name of one of the core available in the target, the application will start running on this core. Attribute name is a string object used to give a name to the application. Attribute id is generated automatically when a High Level
Application object is created, it is used to identify a specific High Level Application, all the methods in GhosAPI that requires HighLevelApplication information use the \textit{id} in order to guarantee that GHOS Kernels process the right instruction for the right application in the case of there are multiple applications running on the target. Attribute \texttt{tag} is used to point at a specific state of the application. Attribute \texttt{endTime} contains the time when the application has finished running. Method \texttt{execute} contains the code that is executed by the GHOS Kernel. Method \texttt{receiveTag} is used to check the value of the attribute \texttt{tag} in order to detect if a specific task of the application\footnote{Reminder: an application is seen as a set of tasks} has to be executed. Method \texttt{register} writes in memory that the application needs to start on its \texttt{startingCore}.

\textbf{The Launcher script} executes a set of operations in order to start GHOS Prototype.

\subsection{High Level Application}

For GHOS-P we have chosen to write one High Level Application by relying on the task graph visible in Figure 4.4. We have changed all the execution time to two units of time (UT).

\section{GHOS Prototype: Results}

Running GHOS Prototype with the High Level Application described in the previous section on the target simulating a Texas Instrument OMAP4460, gives the output visible in Code 6.2. The total execution time is 9.94 units of time, which means that the High Level Application was able to scale thanks to GHOS Kernel mechanism, as there are seven tasks that take two units of time each to execute the global amount of execution time is 14 units of time.

By instantiating and registering the High Level Application we can run multiple applications on GHOS-P which give the output visible in Code 6.1, which is the output for three instances of High Level Application. The total execution time is 23.19 units of time and the global amount of execution of time is $3 \times 7 \times 2 = 42$ units of time. Speedup factor for one application running on GHOS-P:

$$\frac{14}{9.94} = 1.4$$
Speedup factor for three applications running on GHOS-P:

\[ \frac{42}{23.19} = 1.81 \]

Code 6.1: GHOS Prototype running three instances of High Level Application over simulated Texas Instrument OMAP4460
6.6 Conclusion

We have implemented and described a prototype for GHOS and GHOS-T. In this chapter we have described the structure of GHOS-P and we have run High Level Application with GHOS-P. By running one High Level Application GHOS-P has achieved execution with a speedup factor of 1.40 and with three High Level Application GHOS-P has achieved execution with a speedup factor of 1.81. The implementation of GHOS-P has been done by simplifying GHOS and GHOS-T models, a further improvement can be an implementation of GHOS-P using systemc in order to add: clock, accurate timing, priority management and memory management without using SQL. Implementing GHOS-P over a hardware simulator is also an interesting continuation before going to actual implementation on a heterogeneous hardware platform.

Python code for GHOS-Prototype is available at this URL: https://sourceforge.net/projects/ghos/.

Code 6.2: GHOS Prototype running one instance of High Level Application over simulated Texas Instrument OMAP4460