Adaptable OS Services for Distributed Reconfigurable Systems on Chip

Sufyan L. M. Samara

Design of distributed embedded systems

University of Paderborn

A thesis submitted to the
Faculty of Computer Science, Electrical Engineering, and Mathematics
of the
University of Paderborn
in partial fulfillment of the requirements for the degree of

Dr. rer. nat.

November 2010

Abstract

The ever guest for more computational capabilities leads to embedded systems which consist of multiple computational elements integrated on a single chip. An example is the integration of a reconfigurable fabric (FPGA) with a number of general purpose processors to form what so called a reconfigurable system on chip. These embedded systems are common to be distributed. This creates a flexible high performance distributed system. However, it is very complex when it comes to management.

Applications running on such systems are expected to be dynamic in regard of arriving and leaving the system. This increases the complexity as the resources and the demands would change unpredictably.

In this work, an OS service model, which efficiently adapts to the various changes in these systems, is presented. In addition, the algorithms and the methodologies, developed to allow this novel OS service model to interact with the application demands and the environment unpredictable dynamic variations, are discussed. Furthermore, the extensive evaluations of these algorithms are presented. Finally, a case study, which introduces the triple data encryption standard as prove of concept, is provided.

Zusammenfassung

Das ständige Streben nach immer größeren Rechenkapazitäten führt zu eingebetteten Systemen, die aus mehreren Verarbeitungselementen bestehen, die auf einem Chip integriert sind. Ein Beispiel dafür ist die Integration eines rekonfigurierbaren Gewebes (FPGA) mit mehreren Universalprozessoren, um ein rekonfigurierbares System auf einem Chip zu bilden. Typischerweise werden diese Systeme verteilt. Dieses schafft flexible, verteilte Hochleistungssysteme. Allerdings sind diese Systeme aus Verwaltungssicht hochgradig komplex.

Es wird erwartet, dass Anwendungen, die auf diesen Systemen laufen, dynamisch in das System hineinkommen und es verlassen. Dieses erhöht die Komplexität, da sich Ressourcen und Anforderungen unvorhersehbar verändern können.

In dieser Arbeit wird ein Betriebssystemdienstmodell präsentiert, welches effizient eingesetzt werden kann und sich den verschiedenartigen Veränderungen in diesen Systemen anpasst. Darüber hinaus werden Algorithmen und Methodologien diskutiert, die es diesem neuartigen Betriebssystemmodell erlauben mit den unvorhersehbaren Variationen der Anforderungen der Anwendungen sowie der Umgebung zu interagieren. Ferner werden extensive Evaluationen dieser Algorithmen präsentiert.

Das Dokument schließt mit einer Fallstudie des Triple Data Encryption Standards als Konzeptnachweis ab.

Acknowledgements

In the Name of ALLAH, Most Gracious, Most Merciful.

All praise and thanks are due to ALLAH (the one and only GOD, the creator of all and everything) and peace and blessings be upon his messenger Muhammad, he who said: "One who does not thank people does not give thanks to ALLAH, either." [1], I hereby thank my father, my mother, my aunt Fatimah, my family, the DAAD, Prof. Franz Rammig, my friends, and all who supported me in this work.

Muharram 1432 (December 2010)
Sufyan Lutfi Samara

[1] Tirmidhi, Birr 35, 1955; Abu Dawud, Adab 12, 4811

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

List of Algorithms

Glossary

$3DES$	Triple Data Encryption Standered, read (triple D. E. S.)
API	Applications Programming Interface, read (A. P. I.)
$ASIC$	Application Programmable Integrated Circuits, read (asic)
CLB	Configurable Logic Block, read (C. L. B.)
$CPLD$	Complex Programmable Logic Device, read (C. P. L. D.)
CPU	Central Processor Unit, read (C. P. U.)
$ComEl$	computational element
FDP	Fork Distribution Pattern set, read (F. D. P.)
$FPGA$	Field Programmable Gate Array, read (F. P. G. A.)
GFD	The General Fork Distribution Algorithm
GPP	General Purpose Processor, read (G. P. P.)
IM	A set of all the implementations of an OS service.
ISA	Instruction Set Architecture, read (I. S. A.)
LUT	Look Up Table, read (L. U. T.)
MC	Model Checking, read (M. C.)
MDE	Metadata Descriptor Entity used in execution graphs, read (M. D. E.)
$OSIF$	Operating System Interface, read (Osif)
OSR	OS Services Repository.
OS	Operating System, read (OS)
PAL	Programmable Array Logic, read (P. A. L.)
PLA	Programmable Logic Array , read (P. L. A.)
PLD	Programmable logic device, read (P. L. D.)
QoS	Quality of Service, read (Q. O. S.)
RC	Reconfigurable Computing, read (R. C.)
RH	Reconfigurable Hardware, read (R. H.)
$RSoC$	Reconfigurable System on Chip, read (R. soc)
SES	Small Execution Segment, read (ses)
$SPLD$	Simple Programmable Logic Device, read (S. P. L. D.)
$VHDL$	$VHSIC$ Hardware Description Language, read (V. H. D. L.)

$VHSIC$	Very High Speed Integrated Circuit, read (V. H. S. I. C.)
$WCET$	Worst Case Execution Time
WSN	Wireless Sensor Network, read (W. S. N.)
m	number of implementations in each OS service.
n	Number of SESs in each implementation of an OS service.

CHAPTER 1

Introduction

An embedded system can be defined as a system whose principal function is not computational, but which is controlled by a computer embedded within it [147]. Such systems are with an ever increasing uprising as many applications are moving into their realm. Examples include, but not limited to, home devices (e.g. burglar alarm, toys, and games), multimedia, industrial automated systems, and automotive applications (e.g. climate control, breaks, and engine control).

The demand of more computational capabilities and the evolution of electronic systems pushed traditional silicon designers into combining more than one computational element in one chip. This led to the design of complex hardware known as System on Chip (SoC).

Currently a SoC can contain one or many types of microprocessors, reconfigurable fabrics, application-specific hardware, and memories, all communicating via an on chip interconnection network.

Although the integration of more than one component (e.g. microprocessor) increases the complexity of software development, the integration of reconfigurable hardware on fabric, to form a Reconfigurable SoC (RSoC), would increase the complexity to a totally new level. This is because, unlike other components, these fabrics interconnection and functionality can be dynamically programmed and reprogrammed at runtime. Nevertheless, this also introduces the benefits of adaptation, flexibility, and high performance which made such systems very attractive [141].

Furthermore, the current trends are in the direction of networked SoCs. In most embedded applications, SoCs tend to exist in some sort of a distributed system. This introduces an additional constraint on the design of this kind of embedded systems: systems comprising a collection of embedded nodes communicating over a network and requiring, in most cases, a high level of dependability [158].

In order to utilize such systems efficiently, management software development, namely operating systems and middlewares, emerged as the most critical challenge. Such software would enable many useful facilities. For instance, two products could share the use of the same reconfigurable fabric for high performance execution, a product could utilize the reconfigurable fabric along with other computational elements simultaneously for maximum performance and adaptation, a product implemented on reconfigurable fabric can update itself, or a product already assembled with errors can be corrected dynamically at runtime and without changing the hardware platform.

An operating system or a middleware for embedded systems also has to monitor, coordinate, and optimize the usage of the different resources. This is very important because embedded systems are limited in resources. The general objective is then to utilize the available resources efficiently without violating any constraints. This objective, in addition to adaptation and reconfiguration, is the general direction of this work.

1.1 Motivation

In distributed RSoCs system, applications are expected to dynamically arrive and leave. With multiple processors and reconfigurable fabrics on one chip, application multitasking is another common property. This creates a dynamic and unpredictable change in demands, requirements, and environment resources.

Current management software, either middleware or operating systems, tend to run only on general purpose processors. Furthermore, their offered services usually are implemented with static resource requirements. This means that for every time an OS service is required to execute, for the same amount of data to be processed, the same amount of resources is needed. This fixed amount of resource requirements does not meet the dynamic application demands and the environment unpredictable change in resources.

In other words, the use of old management software for newly emerging embedded systems, namely RSoCs, does not efficiently utilize the computational capabilities offered by such systems.

An environment with dynamic resources availability and unpredicted application demands requires flexible and adaptable management. It requires an operating system service with the ability to provide a quality of service with dynamic and adaptable behavior in terms of execution time and resources requirements. An operating system service that could efficiently utilize the computational capabilities offered by a distributed multi-processor on chip system. An operating system service that could use many powerful techniques (e.g. hardware/software partitioning) offered by the integration of reconfigurable and general purpose computing. Such operating system service would enable many significant aspects such as resource sharing, fault tolerance, recovery, and dynamic adaptation to the variable applications demands and the changing in resources availability.

1.2 Thesis contribution

In this thesis, a novel OS service design/model is investigated. This design is intended for distributed RSoCs system. It enables an OS service to adapt its resources requirement to accommodate the variable and unpredicted applications demands. This includes the adaptation to provide for real-time applications, power optimization, performance, resource sharing, and load balancing. Furthermore, algorithms, strategies, and methodologies to support the adaptation in either single RSoC or distributed RSoCs were developed.

The work done in this thesis is well recognized by the research community as the majority of this work has been published in many highly ranked conferences. This also demonstrates the relevance of the investigations carried out in the scope of this thesis to the state-of-the-art research.

The OS service design was first presented in [7]. This includes the algorithms that investigate the adaptation of the OS service under limited or scarce RSoC resources. In [5] the model was generalized with formal description and algorithms to find pareto optimal solution to run an OS service. The challenges and the algorithms for distributing the OS services with load balancing in mind was presented in [6]. The communication overhead introduced from the new design and a workaround solution with a case study are presented in [2]

The developed service model allows for more than adaptation. It provides for error and fault tolerance and it has a built in support for recovery [8, 6, 3].

Other possibilities and methodologies involving biologically inspired algorithms for distribution, resource sharing, and cooperative execution of OS services are presented in [4].

1.3 Thesis structure

This thesis is structured as follows

Chapter 2 outlines some background knowledge necessary to the context of the subsequent chapters. It also summarizes relevant related work which includes software based (re)configurable embedded operating systems, reconfigurable hardware support in operating systems, and resource management support in embedded operating systems.

Chapter 3 presents the different distribution topologies, their weaknesses, their strengths, and the reasons for choosing the topology used in this work. Furthermore, it presents the structure of the middleware that exists on every RSoC to support the new design of the adaptable OS service.

Chapter 4 discusses the objectives required in an adaptable OS service running on a system of distributed RSoCs. It then introduces the novel design of the adaptable OS service which provides for the aimed objectives. An example is also shown to demonstrate the realtime adaptation ability of such OS service design.

Chapter 5 describes the OS service model formally and introduces the constraints, algorithms, and the methodology followed to obtain the required OS service configuration under different demands and criterions. It also discusses the partitioning problem and the use of pipelining technique as a way to minimize the inter partitions communication overhead.

Chapter 6 shows different developed algorithms for distributing the OS services over the RSoCs taking into mind the load balancing and routing. It also points out the use of encoding as a way to provide for error and fault tolerance.

Chapter 7 evaluates the methods and algorithms developed in this thesis. This includes the runtime algorithms used in finding an OS service configuration under

different criterion and the ones used for distribution and load balancing. The evaluation is carried out using the ShoX simulator.

Chapter 8 introduces a case study which uses the triple DES encryption standard as an OS service. The case study is used to compare the time required to execute a non-partitioned OS service and a partitioned OS service with and without pipelining. The case study shows the inter partitioning communication overhead and the different configurations obtained from the partitioned OS service. The ReconOS was used to manage the underlying platform.

Chapter 9 summarizes and concludes the work done in this thesis. It also presents the future directions which include developing tools for automated partitioning and for integrating an online model checker into the system.

CHAPTER 2

Background and related work

Early embedded systems consisted of microcontroller, memory, analog devices, and some I/O signals. The complexity of embedded systems has increased as more and more applications are utilizing them. Examples include digital cameras, mobile phones, handheld gaming consoles, and many electronic devices used in medical, multimedia, communication, and automotive industries.

The continuing quest for maximum performance and flexibility resulted in the emerge of heterogeneous architectures being combined on a single chip. An example of such architectures is the embedding of one or more General Purpose Processors (GPPs) into a reconfigurable fabric (e.g. Field Programmable Gate Array ($FPGA$)). This forms what so-called a Reconfigurable System on Chip ($RSoC$), see Section 2.1. In addition, the distribution of such heterogeneous embedded systems offers more computational power and capabilities [71, 72, 37]. However, this does not come without penalty. The development stages to design an embedded system is considerably complex [131, 127, 91]. It is an interdisciplinary activity involving many research areas. It goes from abstract level modeling and simulation, through software, hardware and platform design to hardware and software synthesis and testing [105].

As the complexity of embedded systems increases, the requirement for management applications or Operating Systems (OS) becomes a necessity. The role of an OS is to provide a transparency layer to applications in order to make the development cycle a little bit faster. An OS would provide for resource management, synchronization,

communication, multitasking, and any specific requirement such as reconfiguration for FPGA.

In this chapter, a background and related work is presented. In Section 2.1, we go through a brief overview of the reconfigurable systems and the flexibility offered by FPGA and the co-design. After that we outline some operating systems for embedded systems offering adaptation or reconfiguration support, see Section 2.2. Finally, a conclusion is given in Section 2.3.

2.1 Reconfigurable systems

Embedded systems can be classified into four categories. These are the Application Specific Integrated Circuit ($ASIC$), the Instruction Set Architectures (ISA) or the General Purpose Processor (GPP), the Reconfigurable Computing (RC), and the hybrid architecture which combines different architectures such as the ISA with the Reconfigurable Hardware (RH).

ASIC is characterized with its high performance. A hard-wired technology is used to customize it for the purpose of a specific application. As a result, an ASIC lacks flexibility and it suffers from a long and very expensive development cycle.

On the other hand, ISA or GPP offer much flexibility. Applications running on a GPP are stored as a sequence of instructions in memory. Any functionality has to be represented by a set of well-defined instructions, hence the name ISA. This however comes with the price of low performance but a relatively fast development cycle.

RC are custom hardware functions with reconfigurability characteristics. They combine flexibility, performance, and relatively low development cost and cycle. This brings them somewhere in between ASIC and ISA, see Figure 2.1.

RH or RC can be accompanied with GPP using different strategies, examples are the loose attachment using a standard bus such as PCI express as in Figure 2.2(a), connecting both of them to the local bus as in Figure 2.2(b), connecting both of them using a special dedicated bus as in Figure 2.2(c), or integrating both of them on one chip to form an RSoC as in Figure 2.2(d). This combination of heterogeneous computational elements in the same architecture allows drawing on strength from each other and increasing the overall system performance and efficiency.

A RH was used as a special configurable high performance accelerator to perform intensive computations [66, 63, 59]. Recently, many research activities and trends are

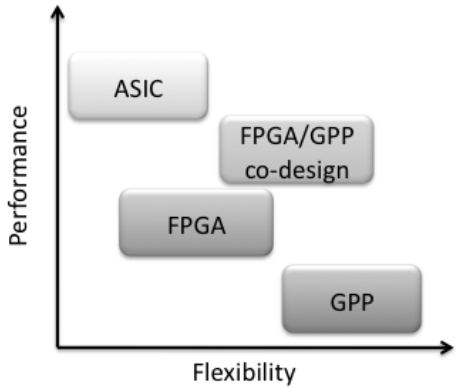
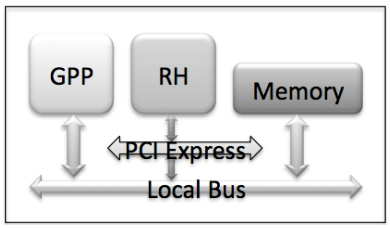
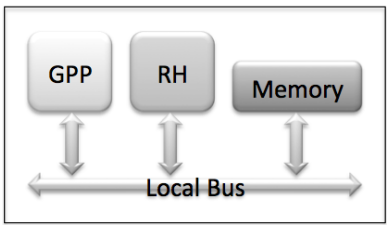


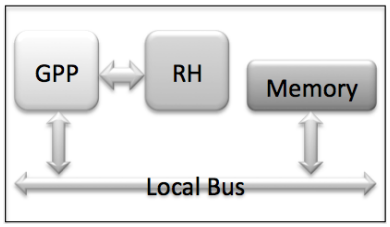
Figure 2.1: Flexibility and performance of embedded systems



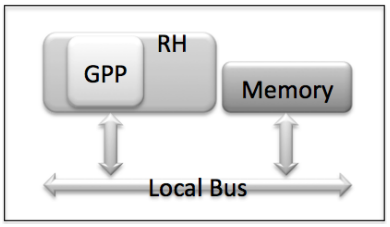
(a) RH is loosely connected to GPP using standard buses like PCI express.



(b) RH is attached to the local bus like the GPP



(c) RH is attached directly to GPP using special bus



(d) GPP is integrated into RH, (RSoC)

Figure 2.2: Different RH and GPP connection strategies

toward using RH as stand-alone processing unit [98, 9, 74, 67]. By utilizing the reconfigurability characteristics offered by the RH, hardware high performance resources can be shared and reused by many applications at runtime, see Figure 2.3. Re-

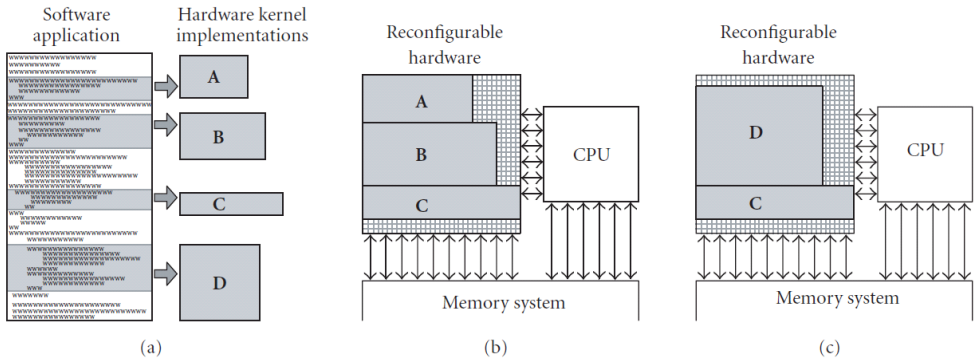


Figure 2.3: Reconfigurable computing implements compute-intensive application kernels (a) as hardware in Reconfigurable Hardware (RH) and the remaining code in software on a CPU (b). Run-time reconfiguration allows RH to implement circuits that would otherwise not fit simultaneously (c) [63]

configurable Hardware (RH) or Programmable Logic Devices (PLDs) are a class of integrated circuits that can be reprogrammed multiple times. They include many subclasses such as Simple PLD ($SPLD$), Complex PLD ($CPLD$), and FPGA. SPLD is normally composed of two logic planes, an AND- and an OR-plane, with one or both of them being programmable. Examples include the Programmable Logic Array (PLA)and the Programmable Array Logic (PAL). These are good for implementing simple sum of product functions. CPLD is a collection of SPLDs connected by a programmable interconnection network. FPGA is the related subclass of this work as it provides the required structure for runtime and partial reconfiguration techniques [73, 22].

2.1.1 FPGA and reconfiguration

FPGA is a programmable device consisting of a set of Configurable Logic Blocks (CLB), a programmable interconnection network, and a set of programmable input and output ports, see Figure 2.4. Other custom function units such as multipliers may also be included (e.g. Xilinx Virtex™ family[149]).

Nearly all the elements of FPGA can be programmed and reprogrammed by the user. Depending on the complexity of a logical function, it can be implemented in FPGA

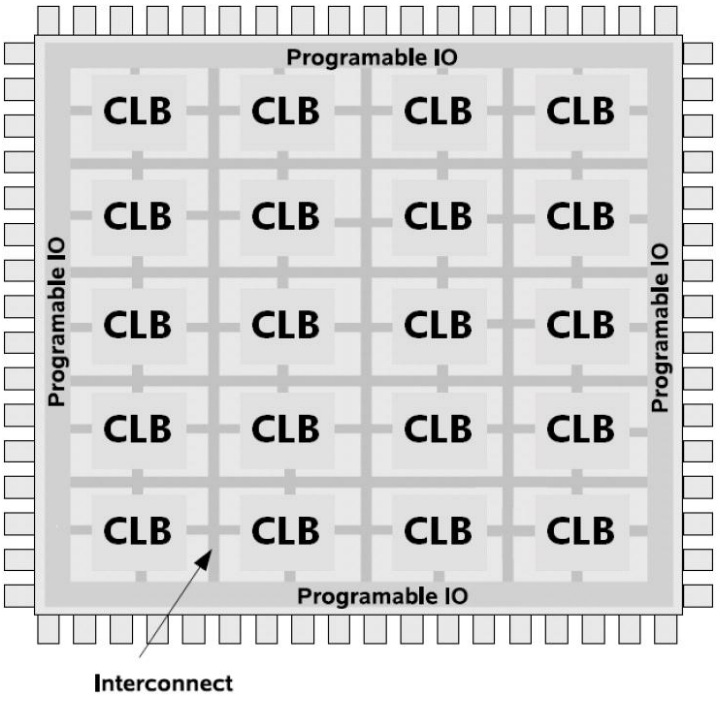
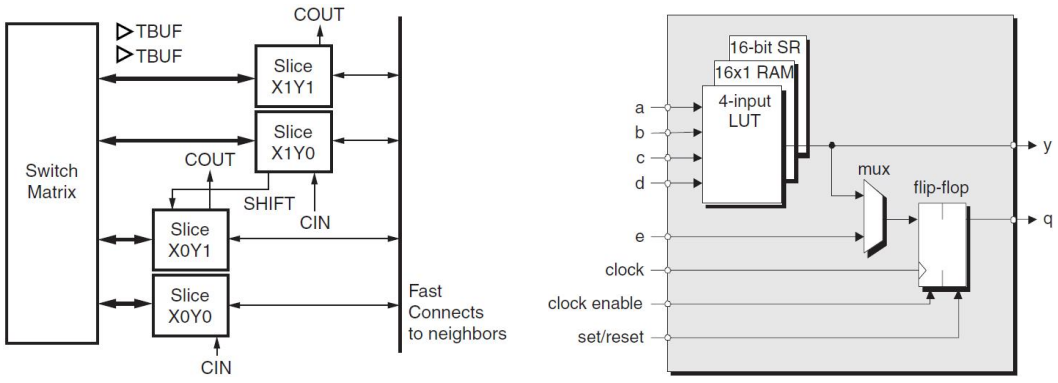


Figure 2.4: FPGA basic structure

using one or a combination of CLB's. The components in each CLB depend on the manufacturer, for instance a Xilinx Virtex II pro CLB consists of four similar slices split into two columns with two independent carry logic chains and one common shift chain, see Figure 2.5(a). A simplified slice contains a register, a multiplexer, and a Look Up Table (*LUT*) which can also be used as RAM or Shift Register (SR), see Figure 2.5(b). Many FPGA fabrics, such as Xilinx Virtex™ family [149] and Atmel FPSLIC™ family [15], are mounted with one ore more processor cores (e.g. PowerPC or AVR) on a single chip. Others[149, 12] use softcore processors (e.g. NIOS or MicroBlaze) to emulate an RSoC.

In this work, the Xilinx Virtex™-II Pro (*XP2V30*) is used in the case study presented in Chapter 8. The basic structure of this RSoC is shown in Figure 2.6. It incorporates two PowerPC405 processors on a single chip within an FPGA fabric. Each processor block is connected to the other components through the CoreConnect™bus architecture, composed of the Processor Local Bus (PLB), which provides low latency access to peripherals requiring high performance, and the On-chip Peripheral Bus (OPB) for Slow peripherals. Xilinx Virtex™-II Pro FPGA and its related software development tools provides for dynamic partial reconfiguration [151, 152, 122] which is a key element in resource sharing and runtime reusability.



(a) Virtex II pro CLB basic elements (b) Simplified contents of Virtex II pro slice

Figure 2.5: Virtex II pro CLB and slice basic elements[79, 107]

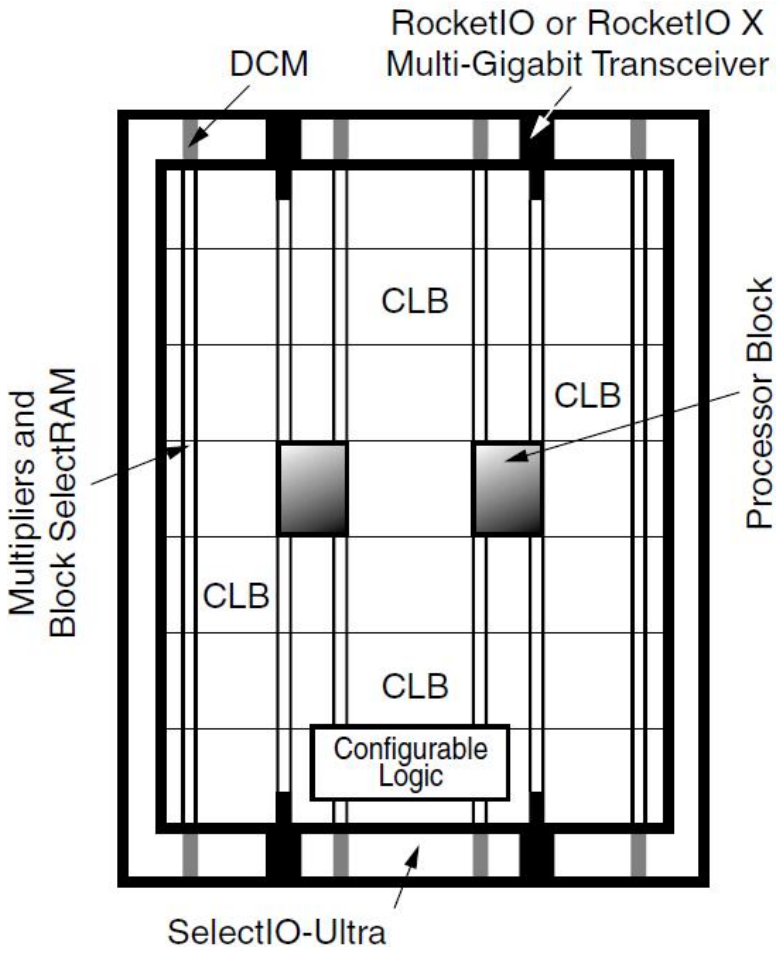


Figure 2.6: Xilinx Virtex-II pro structure [79]

2.1.2 Co-design and partitioning

The term hardware/software co-design appeared after the emergence of platforms combining Central Processor Unit (CPU) with several hardware accelerators (e.g. ASIC) [45]. Hardware/software partitioning determines what application functions should be run on hardware accelerators and which to be run on GPP or CPU. The emerging of FPGAs with high-performance GPPs have proven to be an important implementation medium for hardware/software co-design [17, 96, 101]. Since FPGAs are preexisting reprogrammable chips, they allow co-design to be used in both low-volume and higher-volume applications.

Many algorithms and systems have been developed to help determining which tasks to run on hardware (e.g. FPGA) and which to run on software (e.g. GPP).

The COSYMA[51] is an early hardware/software partitioning system. It was designed to implement applications with computationally intensive nested loops. Algorithms are used to determine how many of these nested loops should be implemented in hardware accelerator.

The global criticality/local phase (GCLP) algorithm[86] performs two sweeps to determine the partitioning. The global criticality sweep performed between software and hardware improves the performance of the application's critical path. The local sweep is then performed to reduce the hardware cost.

Another algorithm [148] starts with a general platform template. Using a separate processing element for each task, it tries to allocate each task on the fastest possible processing element. The algorithm then reduces the system cost by reallocating tasks and eliminating components.

Other algorithms involving hardware/software co-design include the LYCOS [102] algorithm, the SpecSyn [62] specify-explore-refine methodology, the COSYN [36] algorithm, the MOGAC [39] genetic algorithms, The Eles et al. [46] simulated annealing and tabu search comparison, and other related research like in [90, 38, 35].

All the presented approaches deal with static tasks placement at design time. They all try to find the fastest way to execute an application. Some of these algorithms take into account the minimization of one resource which is the hardware area or load. However, there is no optimal solution because the hardware/software partitioning problems are well-known to be NP-problems [14, 85, 61].

Code variation and resources

The hardware/software partitioning algorithms aim to improve the execution efficiency and performance of an application. However, these can also be improved by the coding style [32, 29]. Having a partition chosen to be executed on a specific computational element is not the end of story. One can choose different coding styles for different resources (e.g. power, hardware area) utilization. This is true even for code statements with similar functionality. For example, the synthesis of an IF-clause statement produces a different logic at the structure level from the CASE-statement [110]. The number of FPGA LUTs to represent a logic function as well as its estimated power consumption depends on the structural representation of that function [126].

The variations obtained by different partitioning, implementations, and heterogeneous designs allow for more flexibility. However, this introduces the problem of manageability and platform complexity which demands the existence of an OS. An OS acts as a transparent layer by providing a clear interface to an application while hiding the complexity of the low level platform details.

2.2 Operating system for embedded systems

Embedded systems are customized to meet different functional and non-functional requirements. Therefore, they are heterogeneous and limited in resources. In addition, many embedded systems are equipped with reconfigurable hardware and/or multiple general purpose processor units. Although this enables many powerful features such as flexibility and parallel processing, it also increases the complexity and requires special OS that provides lower cost and higher performance.

One solution is to develop an OS from scratch for each specific embedded system. However, this wastes a lot of development time, has poor flexibility, and makes it error-prone. Another approach is to develop an adaptable reconfigurable OS that can be customized to meet different needs and variations. This would lead to an underlying matured operating system and it would be an appropriate solution [106]. In this section we point out some of the existing OS systems and their support to adaptation and to hardware re-/configuration.

2.2.1 Configurability in embedded operating systems

Many embedded systems require an operating system with low cost, fault tolerable, quick time to market, and real-time and reconfiguration support. As a result, much research was conducted aiming to find such an operating system.

As there are many OS's designed for embedded systems, surveys became necessary. These surveys identify and classify the existed embedded OS's based on many criteria such as configurability (dynamic or static), architecture (monolithic or modular), and scheduling (realtime or non-realtime).

In [57], the author classified embedded OSs based on their ability for configuration. The survey classified *Exokernel* [49] and *SPIN* [18] with some form of configurability/extendability which is limited to a fixed predefined amount of functionality. In addition, the survey identifies a number of OSs with some form of reconfiguration capabilities. This includes *Choices* [30] with dynamic loading of its OS classes, *Coyote* [20] with dynamic change of event handlers, *2K* [92, 93] with dynamic loading of user components, *JavaOS* [128] with dynamic detection of drivers, *Jbed* [82] with user components downloading ability, *MMLite* [75] with OS components replacement capability, and *Pebble* [60] with dynamic OS services loading. The *VxWorks* [143, 136] and *QNX* [76] provide optional modules that can be statically or dynamically linked to the operating system. Other OSs such as *OS-Kit* [55], *PURE* [19], and *eCos* [44, 106] were classified to have no support for dynamic reconfiguration. These OSs provide components and functionalities which can be linked or executed to OS, hence configured, at design time. A similar survey based on OS configurability is also made in [137]

Another survey which concentrates on OSs for Wireless Sensor Networks (WSN) is done in [124]. In WSN, each wireless sensor node can be considered as a System on Chip (SoC) node, hence it has limited resources such as power, memory, and communication bandwidth. The author in [124] classified OSs based on architecture, execution model, reprogramming ability, realtime support, power management, portability and simulation support. The survey identifies the *SOS* [130] to support reconfigurability by runtime loading and removing of system modules. However, it has some problems like the lack of memory management and improper message handling between modules. *MantisOS* [21] support reconfiguration by providing a library built into the kernel which can be used by applications to write a new code. However, this requires a software reset and it is limited to remote login and changing of variables/parameters. *Contiki* [42] supports dynamic loading and unloading of services with some memory management problems. *CORMOS* [153] maintaining a static size table for allocating

and de-allocating of modules such as particular routing protocols. Modules related to user applications and core system extensions can only be loaded during system initialization. *DCOS* is a data centric OS with dynamic loadable modules. The loading of modules is not considered as a kernel task and is done separately.

Another operating system is the Organic Reconfigurable Operating System (*ORCOS*). It aims to use biologically inspired methods [77] to create a highly customizable but easily configurable operating system suitable for any kinds of embedded hardware. The *ORCOS* OS is based on the work done in [41].

Other surveys were conducted concentrating on realtime operating systems and general embedded operating systems in [16, 142].

Resource management

Because embedded systems are limited in resources, it is an important challenge to manage the resource wisely. As a consequence, many operating systems are equipped with some kind of resource management units. These units are either in the core of the operating system or co-existing as a middleware layer.

Examples of operating systems with some resource management like energy include *TinyOS* [123], *MantisOS* [21], *SenOS* [132], and *Nano-RK* [52].

In the realm of resource managers or middlewares we mention the Adaptive Resource Management (ARM) [43], the Dynamic QoS Manager (DQM) [23], the Quality-based Adaptive Resource Management Architecture (QARMA) [54], and the Flexible Resource Manager (FRM) [113].

The resource management in such systems can be summarized by assuring the realtime or high priority applications the resources they need to complete there tasks. However, none of these solutions can be applied straightforwardly for managing reconfigurable hardware resources. The main reason is that the reconfigurable hardware is changing its behavior per application, unlike the GPPs, which have a fixed hardware organization regardless the programs running on them.

In all of the above, OS or middleware is implemented to run on GPP. There is no support for any type of reconfigurable hardware. The reconfigurability/adaptability is merely related to loading or unloading of some system components or functionality either statically at design or dynamically at runtime.

One of the first to discuss the support of reconfigurable hardware in an operating system is Brebner [24, 25]. He proposed swappable logic units that can be switched in and

out of a partially reconfigurable device which is driven by an operating system. After that, many research parties were discussing the support of reconfigurable hardware in OS as outlined in the following section.

2.2.2 Operating systems and reconfigurable elements

Much research was carried out in the direction of supporting reconfigurable elements (e.g. FPGA) within operating systems. In [125, 95, 40], research is done for hiding reconfiguration latencies by prefetching, context switching and resource reusage among hardware tasks. Migration between software and hardware is discussed in [87, 10]

In [64, 140, 133, 139] an OS system model, architecture, and several algorithms to support and manage the sharing of resources in a reconfigurable computational element are proposed. OS includes services for a partial reconfiguration and scheduling of dynamically incoming threads.

In [144, 145, 146], they discuss an implementation of an operating system, *ReConfigME*, that has the ability to share its FPGA dynamically among multiple executing applications. Other computational elements were not considered.

In [104, 111], they present the design of an OS, *OS4RS*, that supports the reallocation and mapping of tasks to the heterogeneous computational elements of RSoC. The reconfigurable hardware is integrated into a multiprocessor system which is completely managed by the OS. *OS4RS* employs a dynamic two-level scheduling technique, top- and local-level scheduler. The top-level scheduler, implemented in software (GPP), stores the running tasks as a linked list. It is to map these tasks to the various heterogeneous computational elements. The local-level scheduler, can be implemented in either software or hardware (FPGA), manages the scheduling of its related mapped tasks on its local computational element. For task context-switching and migration between heterogeneous computational elements, they propose the saving of task states. However, the question remains of how they intend to translate between GPP register set and reconfigurable logic.

The *SHUM-uCOS* [156, 112] proposes a real-time OS for reconfigurable systems employing a uniform multi-task model. The OS manages the utilization of reconfigurable resources. Using static hardware task preconfiguration, it improves the parallelism of the tasks.

In [74], an OS, *BORPH*, is introduced. It is specifically designed for reconfigurable computers. The design process is improved by sharing the same UNIX interface among

hardware and software tasks. *BORPH* treats the reconfigurable computational elements as first-class computational resources instead of coprocessors. It contains three basic components: a conceptual hardware process and two sets of universal interfaces (input/output registers and hardware file input/output interface). The hardware tasks executed on reconfigurable computational elements do not share reconfigurable resources.

In [98, 100], the *ReconOS* is discussed. It offers a uniform hardware (FPGA) and software (GPP) thread model. This is done by modifying an $RTOS$, (e.g. eCos [44, 106]), to integrate hardware interface components called the OS Interface. Both the hardware and software threads are treated equally at scheduling time with no regard to constraints or resources management. *ReconOS* offers methods for threads communications and synchronization. Another related research with some similarities to *ReconOS* is presented in [116, 13]. The *ReconOS* is used in the case study presented in Section 8.1.

All the above research is concentrating on issues related to applications/tasks being implemented on reconfigurable computational elements. Yet, current state-of-the-art architectures do not provide efficient holistic solutions for accelerating multithreaded applications by reconfigurable hardware [154].

On the other hand, implementing OS services on FPGA would improve the performance and efficiency of OS [48]. One close related work tackling such area is the work done in [67]. OS services were implemented on both hardware and software with migration ability between the two implementations. Heuristic scheduling algorithms based on Binary Integer Programming (BIP) were developed for services assignment issues to either hardware or software. These algorithms have the complexity of $\theta(n^2)$. The reconfigurability of the system happen when services migrate from hardware to software or vice versa based on resources and balancing criterions. At the time of migration, both of the OS service implementations have to co-exist in the system. The presented OS has no support to distributed embedded systems, resource sharing, and provides a very limited adaptability to applications needs.

Our proposed system design provides a solution to such limitations by combining the multiple implementations with dynamic hardware/software partitioning to support an OS service with the objectives presented in Section 4.1.

2.3 Chapter conclusion

The combination of one or more GPP with an FPGA on a single chip provides an embedded system, namely RSoC, that provides both the flexibility and high performance. Much research has been carried out to find an OS suitable for managing embedded systems. The main aim was maximum performance and dynamic reconfigurability. These OSs were either dealing with the GPP part of the embedded system or providing support for applications to run on reconfigurable hardware. None except the work done in [67] considered runtime configuration and utilization of FPGA by the OS services themselves. Furthermore, many of the component-based reconfiguration methodologies utilize large components and do not address size, real-time performance, power, and cost issues. Another main problem with component-based systems is the configuration process itself. Issues such as selection, parameterizations of components, analysis, and choice of proper components are not fully addressed [57].

In a limited resource RSoC embedded system there are three main constraints to be considered. These are, execution time, power, and FPGA area or GPP load/utilization. The objectives for optimization and utilization of resources for an OS service is different from that of an application. An OS service is required to provide its service under variable constraints which are specific to each application and to the availability of resources at the time of request, see Section 4.1. Thus a new design is required to enable OS services to utilize the RSoCs platform in order to provide their services efficiently and dynamically at runtime.

CHAPTER 3

System design and architecture

3.1 System design

Distributing a number of RSoCs can be done based on many topologies. The centralized topology is one where an RSoC or a capable node in the system is used as a centralized coordinator. All communications, control, and management are performed through or with the acknowledgement of that RSoC. The benefit of such a topology is the relative easiness of management, data routing, and load balancing. On the other hand, if this coordinator RSoC fails, the whole system will fail as well. Such single point of failure is considered a non-tolerable risk for critical and real time applications.

Another topology is the fully distributed system, in which every RSoC has its own responsibility of coordination, data routing, load balancing, and resource sharing. In case of one RSoC failing, the system may not be affected and the system continues to operate normally. Such a system is more reliable in terms of fault tolerance and error recovery. The management of the fully distributed system is a work shared by all the RSoCs in the system. This requires effective algorithms for cooperation and coordination.

A better topology would be the one which combines the benefits of both; the centralized and the fully distributed topologies. A hybrid topology can work as centralized when it is beneficial, but without losing the fully distribution capabilities. This system could be constructed with a centralized RSoC which is important in the initialization

stage. The RSoCs do not need this centralized RSoC afterwards. However, the existence of such an RSoC is beneficial.

In this work, the hybrid topology is chosen. In the subsequent sections we discuss the reasons behind our choice. This is presented by pointing out the strengths and weaknesses of each one of the three topologies.

Having one topology in mind, we describe the architecture assumed in each RSoC to support the distributed cooperation to run an adaptable OS.

3.1.1 Centralized distribution

In this topology the distributed RSoCs are directly or indirectly connected to a centralized RSoC. It is also possible to logically divide the RSoCs into sets (colonies or clusters), as shown in Figure 3.1, which may or may not be connected. In each colony there is at least one node or RSoC operating as OS Services Repository(OSR) node. Every RSoC in a colony is connected to OSR. The OSR provides OS services needed for the RSoCs in the colony. It contains enough resources (memory, power etc.) to keep and manage the OS services and requesters.

In a colony, RSoCs can only exchange OS services via the OSR. Communication between colonies can be done through the OSRs. Every RSoC in a colony, consists of at least one FPGA and one GPP. RSoCs are to keep track of basic information about their resources such as the used area, and the current power consumption. This information are sent to OSR upon request.

When an RSoC joins a colony, it sends all of its resource information to the OSR belonging to that colony. The OSR keeps a record of every RSoC in the colony. The RSoC record is updated when a change happens in the RSoC resources or a service is being requested and it is executed on that RSoC.

The change in an RSoC resource can be due to a newly arriving task or due to a change in the RSoC's current tasks resource requirements. This depends on the applications dynamics expected in a distributed system, the scheduler, and whether a manager and a profiler are used. An example of the influence imposed by a manager/profiler combination is presented in [113]. The author, in [113], uses the so-called Flexible Resource Manager (FRM) to control the share of resources each application/task is using.

RSoCs other than the OSR hold no OS services. This is to save as much resources as possible for applications/tasks. Another reason is that not all of the OS services are needed every time. In case an application/task needs a service, a request is sent to the

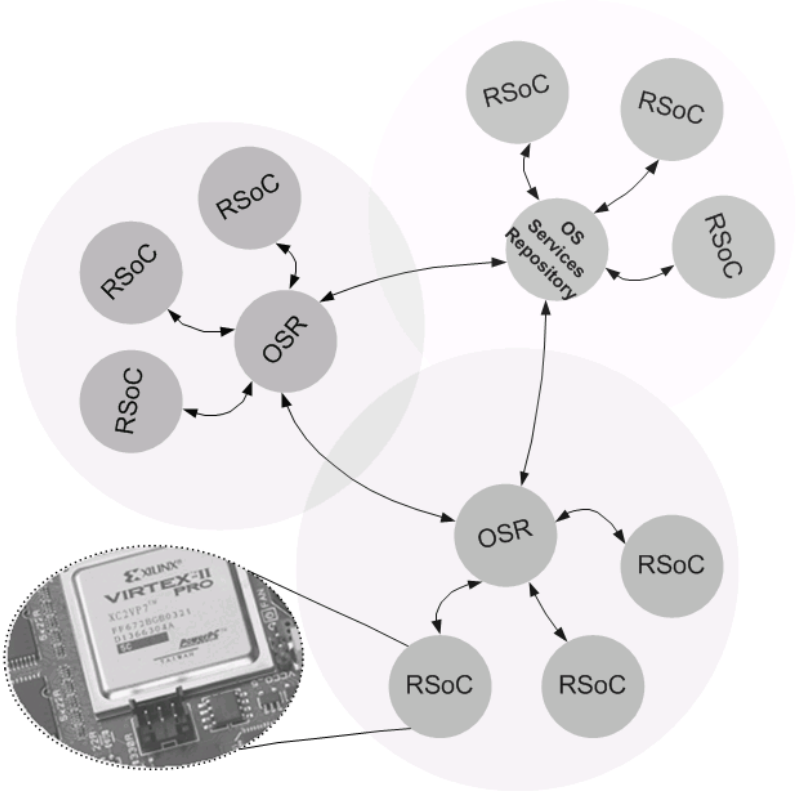


Figure 3.1: RSoCs centralized distribution topology

OSR with information about the current RSoC resource status. The OSR finds a suitable service configuration, see Chapter 5, to execute on the requested RSoC and sends it back. Upon receiving the service, the RSoC uses it to process the application/task request. When the service is no longer needed, the RSoC can discard it to free more resources for applications/tasks.

In the centralized topology, all the OS services, services scheduling, recovery algorithms, and adaptation control are done in the OSR. The RSoCs are merely providing for services execution and resource status reporting. So OSR plays an important role in this topology which makes it, in case it fails, a non-tolerable hazard.

The failing of OSR in a colony will lead to a whole colony crashing. This single point of failure can be worked-around by providing some replica of the OSR, however, this still is not acceptable, especially if the RSoCs are working in uncontrolled critical and risky environments [114]. Moreover, the RSoC which to hold a complete OS and to act as an OSR is required to have enough resources to manage both the OS

and the requesters. Replicating such a capable RSoC may not be a feasible solution. Furthermore, the OSR may become a bottleneck of the whole system in case of many requesters.

3.1.2 Fully distributed topology

Another solution is to eliminate the OSR. This requires the distribution of all the OS services over the RSoCs in the distributed system. Doing so we have no longer any single point of failure. One service can be replicated or encoded to exist on more than one RSoC. A failing RSoC may have a little influence on the running system. Some encoding techniques can be adopted to recover lost data. This remains with some limitations as it depends on the number of the RSoCs in the system and on the type of the used encoding. More discussion on encoding is presented in Section 6.1.2. However, in case of a fully distribution topology, many problems arise like load balancing, service localization, routing, and resource sharing.

RSoCs are limited and heterogeneous in resource amount and availability. Some may have a lot of tasks/applications running while others have a few. The sharing of resources which each RSoC offers to keep and maintain an OS service, has to be relatively fair with relevance to its application/task load and the amount of its resources.

On the other hand, distributing OS services over RSoCs require an effective way of finding these services when needed. Locating services can be performed using many techniques. One technique can be the requester RSoC to send a search message to allocate a service when needed. An announcement message is another technique. A broadcast message can be sent from each RSoC about its services. A record of where each OS service is located, is maintained in each RSoC. Or, some RSoCs can be chosen to maintain records about each service and where it can be located. These RSoCs are supposed to be well-known to all the others.

Concerning the above techniques, each one has its benefits and drawbacks. For instance, the first technique is not deterministic. Though it saves resources by not storing records about services locations, it has no guarantee of finding the location of an OS service in deterministic time. The last technique suffers the single point of failure as in the centralized distribution. It is faster in finding where a service is located, but in case the RSoC containing the records fails, the whole system may fail as well. The second technique however is more tolerable to failure. It is faster in OS service allocation but it requires storing a record of the services locations in each RSoC. Moreover, if not carefully managed it may cause communication flooding.

The routing of data between the requester RSoC and the RSoC maintaining and running the requested service is another important issue. A routing path is required to be maintained to assure the requested QoS. However, allowing all RSoCs to use the same path simultaneously may cause a non-tolerable delay. Moreover, requesting a service from the same RSoC every time, even when there are alternatives, may drain the resources (e.g. power) of that RSoC. So the routing and the allocation has to be managed carefully to avoid such non-preferable situations.

Having found a service location and having established a routing path are not the end of story. The way the services are distributed and if the location of the service is chosen wisely are key elements in minimizing communication time. The number of copies of each service in the system also play an important role. It remains to decide whether it is better to migrate data to the RSoC holding the service and process data off site or to migrate the service and process data locally. All of these routing and migration decisions are relatively easy to manage in the centralized topology. In the fully distribution topology however, this can occasionally lead to drastically complex and sometimes unresolvable matters.

3.1.3 Hybrid distribution topology

Distributing the services over RSoCs increases the management complexity. On the other hand, it remains more tolerable to RSoCs failure. However, if the benefits of both the centralized and the fully distributed topologies were to be combined in a new topology, this might be a better choice. This can be done by adding a capable RSoC with enough resources to act as a centralized node. In the meanwhile, allow the other RSoCs to act and work as in fully distributed topology, see Figure 3.2.

The existence of centralized node in the initialization stage enables fast OS service discovery and localization at execution time. The centralized node is used to distribute the OS services at the initialization stage. Hence, it is able to create the necessary discovery and routing records to find and execute an OS service. Moreover, the distribution of OS services can be balanced as the information about the RSoCs resources could be collected in advance. However, the balancing is not trivial. The distribution, although it might be balanced, it is not optimal. The distributed OS services are heterogeneous in both of their requirements and execution resources. Moreover, the balancing is performed taking into account three main conflicted RSoC resources. In its substructure, this balancing distribution is very similar to the knapsack problem which is a well-known NP complex problem.

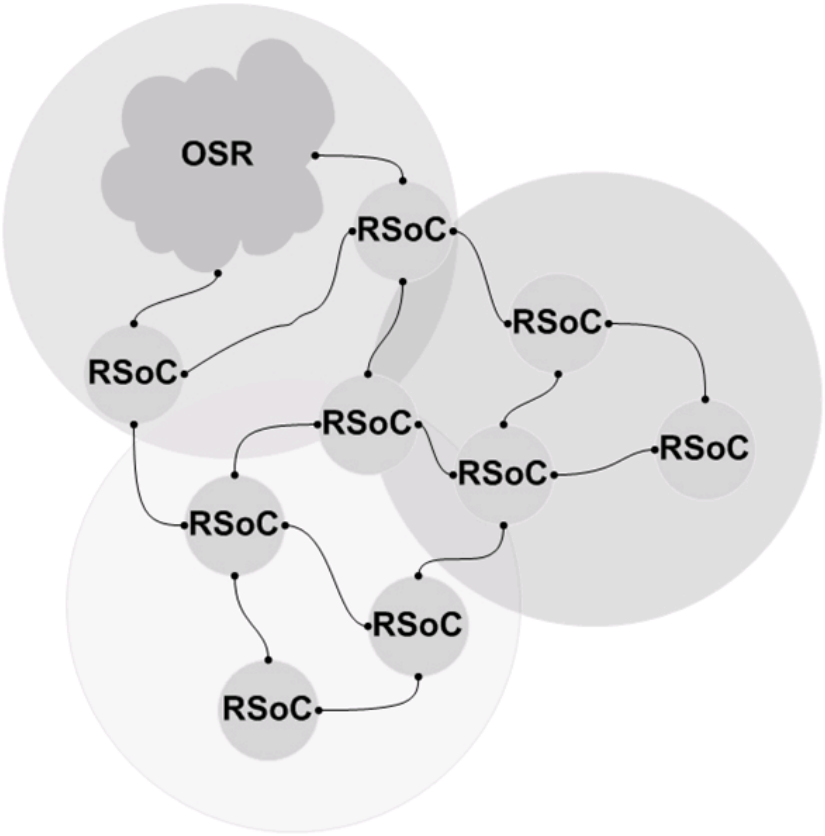


Figure 3.2: RSoCs hybrid distribution topology

In terms of routing, a service may be located away from where it is frequently used, or two dependable services may be distributed apart from each other. These issues are very difficult to resolve at the initializing stage due to the dynamic behavior of the system. However, at runtime, and after collecting some profiling information about the execution of services and the application/task requirements, the system can migrate services and reallocate them to enhance the communication and minimize delay time.

After the initialization stage, the system continues to operate as in fully distributed topology. The existence of a centralized node, although beneficial, is not required. The centralized node would provide a replica of the OS. It also would contain complete routing and distribution information about all the services and the RSoCs in the system. Therefore, such node could provide faster and efficient assistance. Moreover, in case of an RSoC failure, it would provide a faster recovery. Nevertheless, the system provides for recovery and for an OS replica even when the centralized node ceases to exist. This is done by using OS services encoding during the distribution.

The records created during the initialization stage are very important for service execution, adaptation, and recovery. They contain information which helps finding an execution profile suitable to the requester application/task requirement. When these requirements are changing due to the dynamic behavior of the system and of the applications/tasks, these records are used to change the execution profile of the service to adapt to the current variations. In case of an execution failure during service execution, the records are also used to recover the service and if possible to continue execution using other non-faulty profiles without loosing much of the processed data.

Every service record is encoded and saved in more than two distinct RSoCs. Depending on the encoding used, the service record can be rebuilt even if one or two related RSoCs failed. The service record can still be rebuilt even so all the RSoCs holding the records fail. This however may take more time than rebuilding from RSoCs holding some encoded parts of a record. Having done so, it exports a location pointers record to all the RSoCs in the system. After this step, the RSoCs can operate with no regard to a centralized node.

To support such operations, each RSoC is assumed to have the following architecture.

3.2 RSoC architecture

In the RSoC distributed system, every RSoC is consisting of at least one FPGA and one GPP. The RSoCs are assumed to be connected to each other either directly or indirectly through other RSoCs. Although the means of communication and the routing of data packets is irrelevant to the topic of this thesis, an RSoC can be assumed to be using wireless communication. Having its own limited power supply, limited FPGA area, and other limited resources, an RSoC is required to carry out its obligations involving communications, resource management, and supporting various types of applications.

To provide for these obligations and for the adaptable OS services discussed in Section 4.2, every RSoC has an architecture composed of a middleware and an optional virtual machine.

3.2.1 Middleware

Traditionally, an application runs on top of an operating system, using its services to fulfill its goal. Besides providing these services, the operating system acts as an indirection to the hardware by managing the available resources to meet the application

requirements. In our work, since the OS services have been partitioned into small blocks, see Section 4.2, another layer, namely a middleware, is required to provide necessary services to these blocks.

The middleware separates the OS services from the applications/tasks. Hence, it is the one that translates the requests/responses between RSoC hardware, OS services, and applications/tasks. To carry out its duties, the middleware intended to run on RSoC is composed of five main logical units. These are: The reconfiguration unit, the resource monitoring unit, the control unit, the link unit, and the schedular unit. See Figure 3.3.

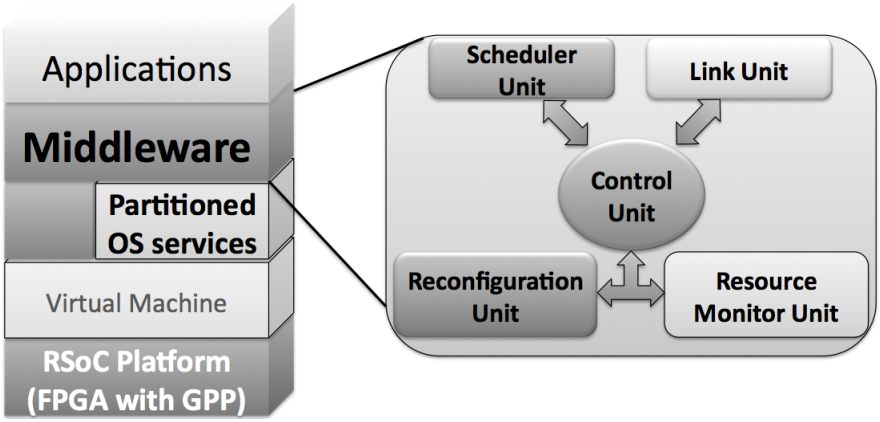


Figure 3.3: RSoC and middleware architecture

The reconfiguration unit

The reconfiguration unit can be further divided into two main units. The FPGA/GPP reconfiguration unit and the communication unit. The FPGA/GPP reconfiguration unit contains the proper routines and drivers to run an entity on either FPGA or GPP depending on its implementation. This includes the runtime partial reconfiguration of the FPGA.

The communication unit has the job of obtaining new OS services. This may require communication with other RSoCs or the OSR. The communication unit contains the necessary routing procedures and drivers to manage the communication and its underlying related hardware. Additionally, it can find the information about each OS service. These include where to find the service and whether or not alternatives exist.

The process of finding an OS service depends on the control unit. This is because the decision of which OS service/configuration to request and when to request it, is done by the scheduler and the control unit respectively. For that, this unit is directly connected to the control unit.

Another direct connection exists between the reconfiguration unit and the resource monitor unit. This is required to keep track of the resources consumed during the reconfiguration unit activities. Examples include communication power, memory usage, and FPGA area.

Although the reconfiguration unit is very important to maintain a reliable system, many parts of this unit (e.g.: network routing) is out of this thesis scope. However, when necessary, a brief discussion will be brought up which is related to this unit operations.

The resource monitoring unit

To keep the system up-to-date with all the occurred changes, an active resource monitor is integrated into the middleware of every RSoC. The role of the monitor is to keep track of and inform the control unit about the status of the RSoC resources. Any request to resource allocation has to be through this monitor, or at least, to notify it if such an allocation occurs. The monitor maintains a set of descriptors used to identify the available amount of resources. In RSoCs the most important descriptors are time, power, and area. These descriptors involve complex analyzes and in many cases may affect each other.

The area descriptor representation is architecture dependant. In RSoC, the area descriptor is divided into two parts. The hardware (FPGA) area descriptor and the software (GPP) area descriptor. The hardware area descriptor represents the number of look-up-tables (LUT) or the number of configurable-logic-blocks which can be used for implementation. An implementation of a service or an application intended to run on FPGA is normally represented using hardware disruption languages such as $VHDL$. These representations are compiled to use a number of LUTs and other hardware specific units (e.g.: multipliers). Obviously, an FPGA has a limited number of LUTs and other specific units. Hence, the descriptor would hold information about the number of used and available LUTs and the other specific units. On the other hand, implementations intended to run on a GPP are compiled into a number of instructions sequences which are stored in memory. So a software area descriptor may represent the available and used memory. However, this is not accurate, especially if we are dealing with real time applications/tasks. A better representation is the GPP utilization or GPP bandwidth.

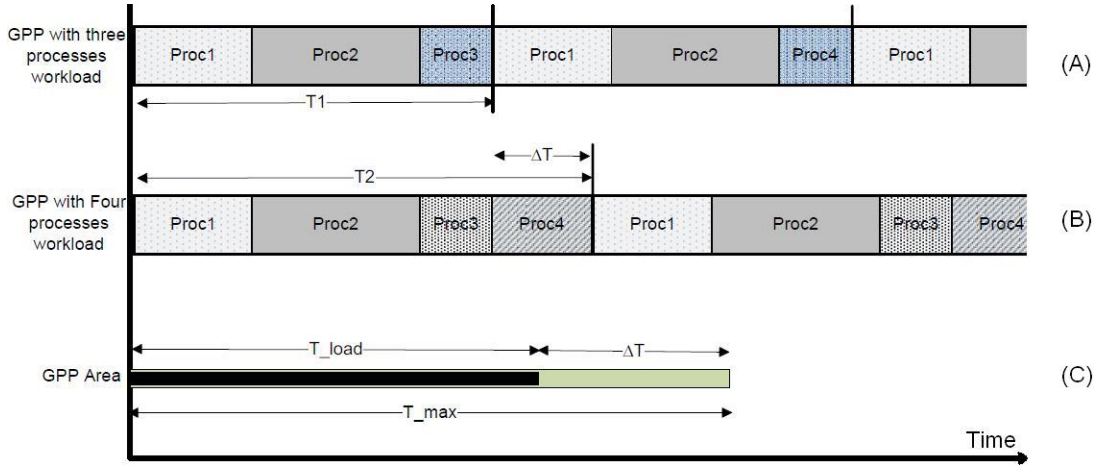


Figure 3.4: GPP area representation

To explain such a representation see Figure 3.4. Figure 3.4-(A) shows a GPP workload with three processes. Each process can be a normal application/task or a service. For simplicity, let us assume that a scheduler has already scheduled the processes shown and this schedule satisfies the time requirements of these three processes. In order to complete the operation of each process, the GPP will cycle every $T1$ period with the same schedule. Now, let a new process enter the system, see Figure 3.4-(B). To schedule the new process, the GPP has to cycle every $T2$ interval. If the new schedule also satisfies the timing requirements of all the processes, we could say that in Figure 3.4-(A) the cycle interval is $T2$ with a free slot equals to $\triangle T$. A more clear description is shown in Figure 3.4-(C). Where T_{max} represents the maximum time interval the GPP can cycle without violating the timing requirements of the scheduled processes. T_{load} represents the time consumed by the scheduled processes and $\triangle T$ the available slot.

The bandwidth can be described as $1/T$. A GPP would have a maximum bandwidth of $1/T_{max}$ and each process would use some of this bandwidth. The GPP utilization of a process is the workload percentage of that process. This is the percentage of the required time for a process in one cycle to the total cycle period. GPP bandwidth or utilization can be referred to as GPP area. This kind of software area representation makes the amount of the total GPP area depending on the current workload and the scheduler algorithm.

The power descriptor holds information about the amount of power an RSoC has. The representation of available power can be measured by program instructions/units or as power units. Each instruction of a program (application/task or service) can be estimated by a number representing the amount of power consumed by running

that instruction. The total power can then be measured as the maximum amount of instructions an RSoC can run. If a program is implemented to run on FPGA, the power analysis would depend on the structural design of the program. In this analysis, each structural unit (e.g.: an AND gate) is related to some amount of power consumption per clock cycle. On RSoCs, the amount of available power can be obtained from a special circuit designed to measure power. This can then be represented as power units, where a power unit is the smallest amount of power consumption that can be measured in the system. Other power consumption measurements can also be estimated using models, simulation, or explicit analysis [78, 150, 31, 121, 89].

In case of separate power for FPGA and GPP, the descriptor can be divided into two descriptors, the hardware power descriptor and the software power descriptor. The amount of power available can affect the time descriptor. This is true, because the amount of power will limit the number of clock cycles or the number of instructions which can be executed.

The time descriptor is important for scheduling. It is more related to the process executing on an RSoC and it holds information which estimates the process Worst Case Execution Time $WCET$. This descriptor should not be confused with the GPP utilization which represents the percentage of the time reserved for a process in each GPP cycle. The WCET can be obtained by measurements [134] or by static analysis [135, 33, 70, 50]

The applications link unit

The adaptation of OS services presented in Chapter 5, depends on the information provided on both the RSoC resources and the applications/tasks. The applications link unit provides the necessary means of communication to transfer the applicaitons/tasks demands to the middleware. These demands are then represented as constraints or demanded Quality of Service (QoS) an application/task requires in order to complete its operation correctly. For instance, a realtime task may require a service to response in a define period of time.

As there may be more than one application/task executing OS services in one RSoC, it is the responsibility of this unit to provide the necessary channel support to link each application/task with its corresponding OS service. The link provides the necessary support to perform the communication transparently. This is not a trivial task as services may exist on the same RSoC which the application resides on, on another RSoC, or even partitioned on more than one RSoC.

In order to perform such duties, this unit works closely with the control unit. The cooperation is realized by establishing the appropriate communication of information to accommodate the applications/tasks requirements.

The scheduler unit

A distributed RSoCs system with adaptable partitioned services requires special algorithms and support. This unit holds the algorithms needed to schedule new services, reconfigure a service, or recover a service from expected fault. To realize this, the scheduler unit has to account for many factors. These include efficient resource usage, meeting multi criterion constraints, accounting for communication and FPGA reconfiguration time, and whether the execution is made on the same RSoC, on a different RSoC, or distributed on more than one RSoC.

To carry out its obligations, the scheduler unit needs to communicate with the other middleware units to obtain the required information. These communications are done through the control unit which analyzes and provides the necessary data needed by this unit.

The control unit

One important aim of this unit is to make the partitioned OS services appear monolithic to themselves and to the applications using them. It provides basic services such as memory allocation and resource management needed to run the OS services. The control unit treats the partitions of an OS service as components with defined interfaces. This is achieved by encapsulating the arriving partitions (e.g: from OSR) in component containers, which provide the required interface to manage the life-cycle of a partition on an RSoC.

The other aim of this unit is to glue the other units of the middleware so they work consistently with each other. All important decisions are taken by the control unit based on the analysis of the data provided by the resource monitor unit and the link unit.

In relevance to the work

Every unit of the middleware is very important to the whole system. However, the majority of the work done in this thesis is related to some parts of the control unit and

the scheduling unit. The full investigation of the other units, though consequential, is out of this thesis scope. Nevertheless, some related matters will be briefly discussed as the context requires it.

3.2.2 Virtual Machine

The virtual machine is an optional layer which may be needed if too many computational elements are within an RSoC. A virtual machine could be used as a method to reduce the number of the considered computational elements, but may also slow down execution. A virtual machine would provide the possibility to map one service/process to many computational elements transparently.

Nevertheless, even without a virtual machine, the OS services work and adapt quickly, see Chapter 5. However, it will always be a matter of storage and area versus speed and performance. This is because the OS service design requires a number of service implementations equivalent to the number of computational elements. Although these will adapt and execute faster, more resources are needed to confine such implementations. On the other hand, the virtual machine would also require some resources to operate. It will also slow down the execution of an OS service in comparison to a native platform execution. In the end, it remains to decide which used method consumes less resources and whether more performance is needed or not.

3.3 Chapter conclusion

Each distribution topology has its strengths and weaknesses. A preferable option is a hybrid topology which combines the benefits of the different topologies. The chosen hybrid distribution topology has a centralized node which is required at initialization stages for load balance distribution and the building of execution routing tables.

To enable the distributed RSoCs to support the different intended characteristics and objectives, each RSoC is assumed to have a middleware that provides for the novel OS service model, see Chapter 4. The middleware contains the proper units to manage and coordinate the RSoC resources to communicate, configure, and run an OS service.

3. SYSTEM DESIGN AND ARCHITECTURE

CHAPTER 4

OS Service structure

An operating system is normally consisting of many services. These services are selectively executed upon an application demand. A normal OS service, even in embedded OS, is developed as a unit which is implemented to run on a GPP. Such an OS service has the properties of fixed resources requirement and process time. In other words, if provided the same input stream, such an OS service will always require the same resources and execution time to provide the processed output. This however does not utilize any flexibility implied by the multiple heterogeneous computational elements which may exist in embedded systems such as RSoCs. Moreover, if resources are dynamically changing, which is highly expected in distributed embedded systems, some computational elements may be heavily loaded while others may be idle. Such a scenario may prevent a fixed resources OS service from executing. However, if the OS service has flexible resources requirements, it could adapt itself to execute on other available resources.

Currently, the adaptation of embedded OS systems is merely related to whether to include, exclude, or extend an OS functionality at design time. The whole OS always has to reside on every embedded system. This means that every RSoC in a distributed embedded system has to have its own OS. In addition, if applications on one RSoC are using the resources required to execute an OS service, the only option for that OS service is to wait until the resources are available again. These limitations, among many others, urge the need for a new OS service design which utilizes the current

multi computational heterogeneous platforms and provides for the increasing application demands.

4.1 OS service design objectives

As aforesaid, each RSoC in the distributed system is consisting of at least two different computational elements (e.g.: FPGA and GPP). To utilize these embedded systems, each OS service exists in a number m of implementations which equals the number of the different Computational Elements (ComEl). In other words, for each OS service, there exists an implementation $Im \in IM$ which can run on a computational element $comel \in ComEl$, where IM is the set of all the implementations for an OS service. This enables an OS service to run on a computational element that has relatively less computational load cl than the others.

Now lets consider that each computational element $comel_i$ has a maximum computational load of mcl_i, a current computational load of ccl_i, and an available computational load of acl_i, where $mcl_i = ccl_i + acl_i$. If an OS service implementation Im_i has an estimated computational load scl_i, then this implementation can run on the computational element $comel_i$ if and only if $scl_i \leq acl_i$. In general, an OS service can run on an RSoC if one of its implementations has a computational load less than or equal to its corresponding computational element's available load. However, if every computational element $comel_i$ of one RSoC have $acl_i < scl_i$, then the executing of the service is not possible. But what if the summation of all the computational load in all the computational elements is bigger than any service computational load, that is, what if $\sum acl_i > scl_j,\ where\ i,\ j \geq 0$. Such an option encourages the objective of designing an OS service that has the means to distribute its computational load to run on wherever computational load is available.

The adaptation to applications variable demands is another important objective. For instance, two applications may require different responses time from an OS service. Enabling OS service to reconfigure itself, allows the support for such variable demands. However, assuming that an OS service has the reconfiguration ability, there would be a configuration with a minimum response time or resource usage. So, why not just use that configuration every time?

The OS service reconfiguration refers to the ability of an OS service to run using a different resource combination. This yields to different outcomes in terms of response time and resources usage amount. Confining OS service to its minimum response time configuration would again raise the issue of fixed resource usage. Normally, a

minimum response time means faster execution which, in most cases, means more resource usage. So again, what if the required amount of resources to run an OS service is not available at the time the OS service is invoked? To avoid this problem, it is important that an OS service has the ability to reconfigure itself to provide for different resource usage, hence, different application demands.

Another essential objective is the fault tolerance and recovery. A reliable system imposes the possibility of some fault percentage. Although this percentage varies from one system to another, a distributed system is most likely to be exposed to a non-negligible fault percentage. The handling of these faults varies as their corresponding problems. These include the missing of data through communication, the partial or total failing of one or more RSoCs, and the computational errors. The recovery from these faults has to be fast and efficient. For instance, in computational and communication errors, the recovery process has to support resuming from the last known correct point and not just start from the beginning.

Obviously an OS service intended for a distributed RSoCs system has to provide for distribution and for resource sharing. As RSoCs are limited in resources, there will be a case where one RSoC can not run an OS service and the only remaining option is to use resources from other RSoCs. This sharing of resources has to be done fairly and without harming the RSoC that its resources being used. Hence, load balancing techniques and algorithms are required to support the introduced novel OS service design and to allow a fair resource sharing and distribution.

The support for OS services localization and routing is an additional requirement for distributed OS services. It should be noted that the routing here does not refer to network messages and packets routing, but to the process of finding the convenient RSoCs to execute a distributed OS service. This has to account for fair resource sharing among the RSoCs without violating any constraints imposed by the applications/tasks. Moreover, services may have more than one copy or version among the system. Hence locating a service copy or version and the operation to transfer the data through the related RSoCs is what we refer to by OS services localization and routing, see Section 6.3.

To enable these objectives and requirements, a new design model for an OS service has been developed. This novel design enables OS services to adapt themselves to the variable resource availability, application/task demands variations, and the distribution requirements. Moreover, the model provides for fault tolerance, recovery, resource sharing, and load balancing.

4.2 OS service design

Every RSoC in the distributed system is consisting of at least two different computational elements (e.g: GPP and FPGA). All the RSoCs are assumed to be connected to each other either directly or indirectly, see Section 3.1. Every RSoC keeps track of the information about its resources (e.g. used area, current power consumption) which can be retrieved at any time.

To provide for every computational element on an RSoC, each OS service exists with a number m of implementations that equals the number of different computational elements. For instance, if the distributed system has RSoCs, each consisting of one GPP and one FPGA, every OS service would have two implementations; one that can run on GPP and another which can run on FPGA.

To fulfill our objectives, see Section 4.1, each implementation of an OS service is partitioned into the same number n of execution blocks, where each block is called a Small Execution Segment (SES).

In each implementation, the $SESs$ are indexed from 1 to n, where 1 means the first one in the execution sequence and n means the last one to be executed. In addition, each SES is marked with a notation that indicates the implementation to which it belongs. For instance, in an OS service with a hardware implementation (intended to run on an FPGA) and a software implementation (intended to run on a GPP), the $SES_{h,i}$ has an execution sequence index i and belongs to the hardware implementation.

In one OS service, all the $SESs$ with the same index in all the implementations have the same functionality. This means, if they are provided with the same input stream or sequence, they will all produce the same output. Because of this, and due to the fact that n $SESs$ are needed to have a complete OS service, at each execution sequence index i any SES with the index i can be chosen regardless of its implementation. This allows the execution of one OS service in a variety of m^n configurations, where m denotes the total number of implementations for an OS service. This implies the same number n of $SESs$ in every implementation belonging to the same OS service. However, in case some of these implementations have fewer $SESs$, dummy $SESs$ can be added to reach the n number of SES.

This novel design can be modeled as shown in Figure 4.1. The model consists of entities and directed data paths. An entity has three types: $START$, $FINISH$, and SES. The $START$ entity is used to hold meta-data information about the OS service, its $SESs$, and its implementations. These meta-data are very important to find a suitable OS service configuration under some specific constraints. It is also required

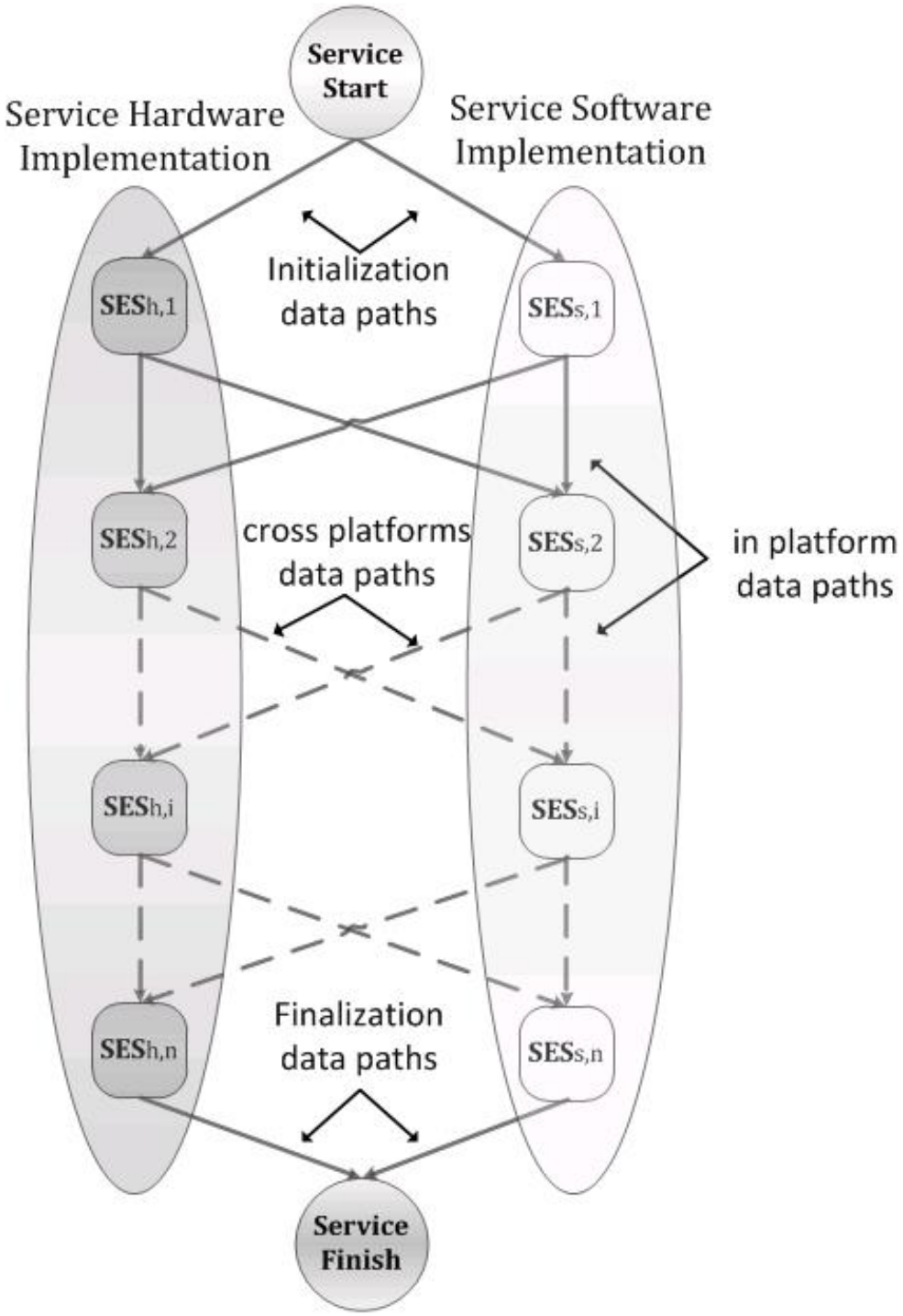


Figure 4.1: OS service with two partitioned implementations

when $SESs$ are distributed over the RSoCs in the distributed system. The $FINISH$ entity holds information to check, refine, and finalize the processed data obtained from a SES with the execution sequence index n.

The directed data paths represent the direction of data flow between two entities and the cost of such data flow. The cost denotes communication time and any non-negligible resource, if any. In contrast to the direction of data flow, the directed data paths can be classified into four types: Initialization data paths, finalization data paths, in platform data paths, and cross platforms data paths.

The initialization data paths connect the $START$ entity with the $SESs$ that have an execution sequence index 1. An initialization data path represents setup resource (e.g. time) required to start the OS service execution from the related $SES_{x,1}$, where x denotes an implementation type. The finalization data path connects a SES with execution sequence index n with the $FINISH$ entity. It accounts for any required resource (e.g. time) to transfer processed data to its final stage. The in platform data path connects any two successive $SESs$ in the same implementation. On the other hand, the cross platforms data path connects any two successive $SESs$, where the first SES is in one implementation x and the other SES is in a different implementation y, and the two implementations belong to the same OS service.

The SES entity is encapsulated as shown in Figure 4.2. The identification field

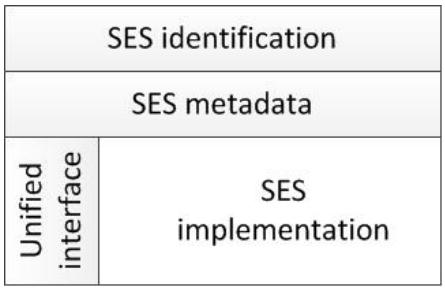


Figure 4.2: SES general structure

defines the SES implementation, its execution sequence index, and to which OS service it belongs. The meta-data field holds information about the SES execution requirements. These include the SES worst case execution time ($WCET$), the SES estimated power consumption, and the SES area allocation, see the resource monitoring unit in Section 3.2.1. In addition, the meta-data field contains two records: The alternatives record, and the next successive record. These two records are required in the distributed execution for routing, fast fault recovery, and migration. The structure and the usage of these two records are discussed in Section 6.3.

To allow a generalized SES initiation, a unified interface is required. This however has to be managed without causing the SES to loose its individuality. The middleware uses this interface to communicate signals for SES execution management (e.g. initiation and interruption). On the other hand, a SES uses the interface to notify the middleware about the status of the execution. The SES specific parameters can be passed to the SES as a pointer to a record in memory.

4.2.1 $SESs$ execution and applications

The operations of these novel OS services have to be transparent to applications. This is done with the help of the middleware. The link unit of the middleware, see Section 3.2.1, provides the applications with the proper standard application programming interface (API) to access OS services.

Once an application with the potential to use an OS service is introduced into the system, the middleware begins the process of the OS service configuration. The middleware first checks if there exists a previous OS service configuration in the system which suits the current demands and requirements. In such a case nothing has to be done but to wait until the OS service is required to begin execution.

If there was no suitable OS service configuration on the RSoC, the middleware has to obtain one either from OSR or from the other RSoCs in the distributed system. With regard to the RSoC resource status, an OS service configuration is to be found such that it does not violate any constraint. In case the RSoC does not have enough resources for any OS service configuration, other solutions may be explored. Such solutions may include running the OS service on another RSoC or using more than one RSoC to run the OS service.

Having an OS configuration ready, the application can request the OS service by calling the API offered by the middleware. Once a request is received, the middleware prepares the parameters and any other needed information inside a record in the memory. Using the unified interface, the middleware sends a signal to the first SES with a pointer to the memory record in order to start execution. When the first SES finishes execution, it sends a notification signal to the middleware with a pointer to the record of the processed data in memory. The middleware transfers these data to the next SES as an initiation signal and a memory pointer. This process continues until the OS service finishes processing the data after the last SES being executed. The processed data is then finalized and transferred by the middleware to the requester application.

If, during the OS service $SESs$ execution, the RSoC resources changed due to a new coming application/task, the middleware can request another configuration which suits the new resource status. In case a fault or an error occurred, the OS service can be reconfigured to avoid the faulty SES. The recovery process may resume from the last SES which provided a correct processed data.

To carry out these operations, efficient algorithms that evaluate and find OS service configurations are required. Additionally, because these OS services are designed to operate on distributed RSoCs, it is required to distribute them taking into mind the issues of load balancing, localization, and execution routing. These along with other related matters are discussed in the subsequent chapters.

4.2.2 Example: Realtime adaptation

To understand the benefits gained from different OS service implementations and partitioning, let's assume that we have an OS service X with two implementations. Each implementation has just two $SESs$ as follows:

$$Im_s = \{SES_{s,1}, SES_{s,2}\},$$

$$Im_h = \{SES_{h,1}, SES_{h,2}\}.$$

Where s denotes software implementation which executes on GPP and h denotes hardware implementation which executes on FPGA. For the sake of discussion we define each SES as a tuple of $\{release\ time,\ Worst\ Case\ Execution\ Time\ (WCET)\}$. Let's assume that the former $SESs$ are defined as follows:

$$SES_{s,1} = \{r_1, 3\},$$

$$SES_{s,2} = \{r_2, 5\},$$

$$SES_{h,1} = \{r_3, 1\},$$

$$SES_{h,2} = \{r_4, 2\}.$$

Now we want to schedule the real-time task defined below.

$$t_3 = \{8, 2, 3, 24\}.$$

Here a task is defined by the tuple $\{release\ time,\ WCET\ before\ OS\ service\ call,\ WCET\ after\ OS\ service\ call,\ and\ absolute\ deadline\}$. This task is to execute on GPP and call the service X. Let the target RSoC have one GPP and one FPGA with 6 blocks of available area. For the sake of simplicity, let's assume that each time slot will

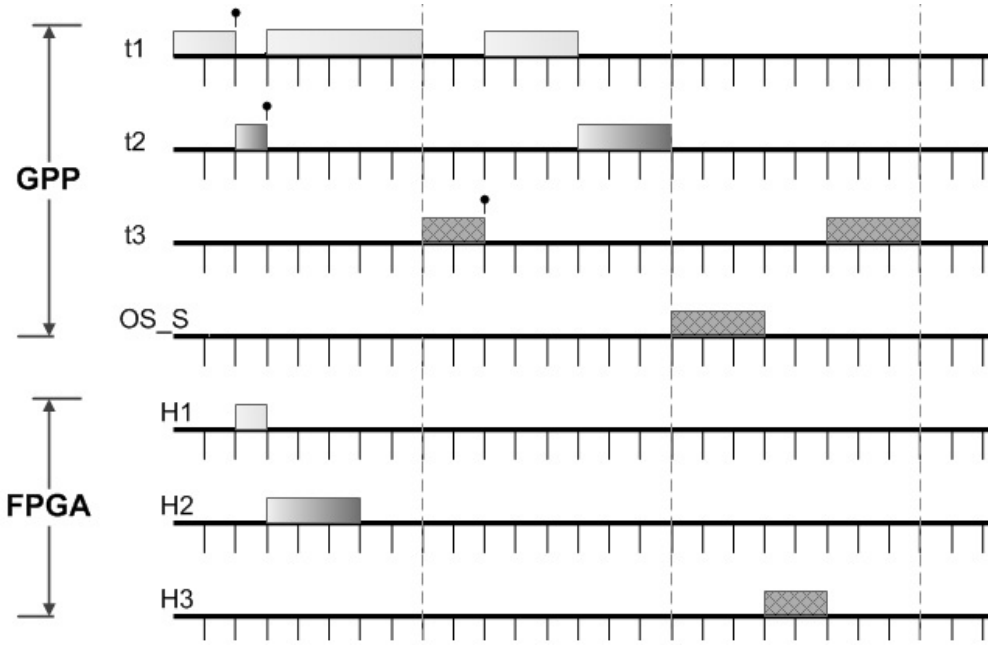


Figure 4.3: Scheduling configuration 1. OS_S: OS service time slot on GPP, H1: FPGA area-time slot scheduled for t1, H2: FPGA area-time slot scheduled for t2, and H3: FPGA area-time slot scheduled for t3.

need to allocate one block of the FPGA area if executed on the FPGA. Considering the scheduling configuration seen in Figure 4.3. The RSoC is already having the tasks, t_1 and t_2 which are using both the FPGA and the GPP as following:

$$t_1 = \{0, 2, 8, 13\},$$

$$t_2 = \{2, 1, 3, 16\}.$$

Here each task is defined as the tuple $\{release\ time,\ WCET\ before\ FPGA\ usage,\ WCET\ after\ FPGA\ usage,\ and\ absolute\ deadline\}$. In this schedule we see that task t_1 is scheduled with FPGA usage of one slot and task t_2 with three slots. The remaining available FPGA area is just two slots. Hence, task t_3 can only be scheduled with the OS service configuration $SES_{s,1}$ and $SES_{h,2}$ with $r_1 = 16$ and $r_4 = 20$.
On the other hand, scheduling configuration 2, see Figure 4.4, has task t_1 and t_2 defined using the above task tuple definition as:

$$t_1 = \{0, 2, 3, 8\},$$

$$t_2 = \{2, 1, 3, 13\}.$$

Here task t_1 is scheduled with three FPGA slots and task t_2 with also three slots of FPGA usage. At this point FPGA has no remaining available area. Hence task t_3 can

only be scheduled with the OS service configuration $SES_{s,1}$ and $SES_{s,2}$ with $r_1 = 12$ and $r_2 = 16$.

As seen we have been able to schedule t_3 without missing the deadline in both sched-

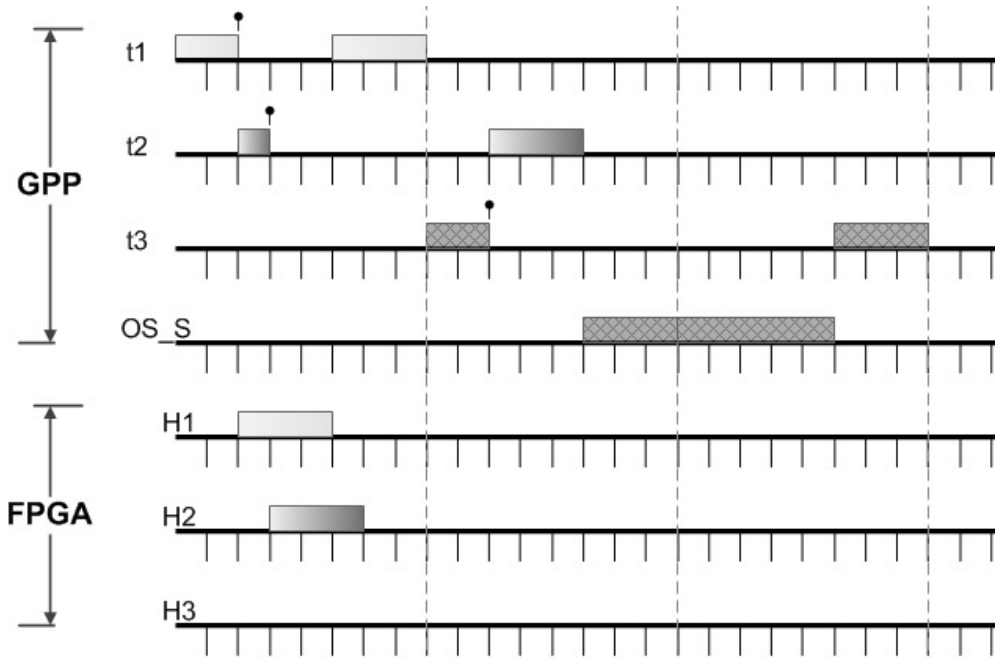


Figure 4.4: Scheduling configuration 2. OS_S: OS service time slot on GPP, H1: FPGA area-time slot scheduled for t1, H2: FPGA area-time slot scheduled for t2, and H3: FPGA area-time slot scheduled for t3.

ules using different service configurations. The above two examples assume nothing about the RSoC power, $SESs$ communication time, and OS service release and finish time. However, if the power is limited, it has to be considered as well, as different configurations have different power requirements. In scheduling configuration 1 and 2, more flexibility could be achieved if dynamic reconfiguration of the FPGA is allowed. However, the time to reconfigure, that is the time to replace an already used area of the FPGA with new data, should not affect the deadline of one task or another.

Another benefit gained from such partitioning is fault tolerance and adaptation. If at one point in the execution, a failure in one SES is discovered, another configuration from the point of failure which does not have the faulty SES could be set, provided that constraints, like timing, do allow this.

4.3 Chapter Conclusions

An OS service designed for a distributed RSoC system has many objectives. These include multiple architectures support, reconfiguration, hardware utilization, adaptation to applications variable demands, adaptation to resources dynamic change, fault tolerance, and recovery.

These objectives can be achieved if an OS service exists in multiple implementations, with each implementation partitioned in the same number of functionally equivalent blocks, namely $SESs$, as the other implementations. This enables the execution of one OS service using many different configurations, with each configuration having a different response time and resource requirements.

The variations in configurations resource requirements and response times, allow an OS service to adapt to various application demands and environments. Such an adaptation can take place with regarding to one constraint or to multiple constraints such as area and realtime deadlines.

CHAPTER 5

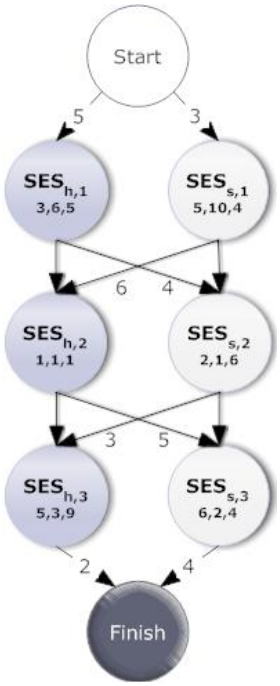
OS service configurations and adaptation

The partitioning of OS services into $SESs$ enables an OS service to be executed in many different ways or configurations. This is because for any OS service, the $SESs$ with the same execution indices have the same functionality behavior, see Section 4.2. Hence, for each execution index, any SES with the same index can be chosen from any implementation belonging to that OS service.

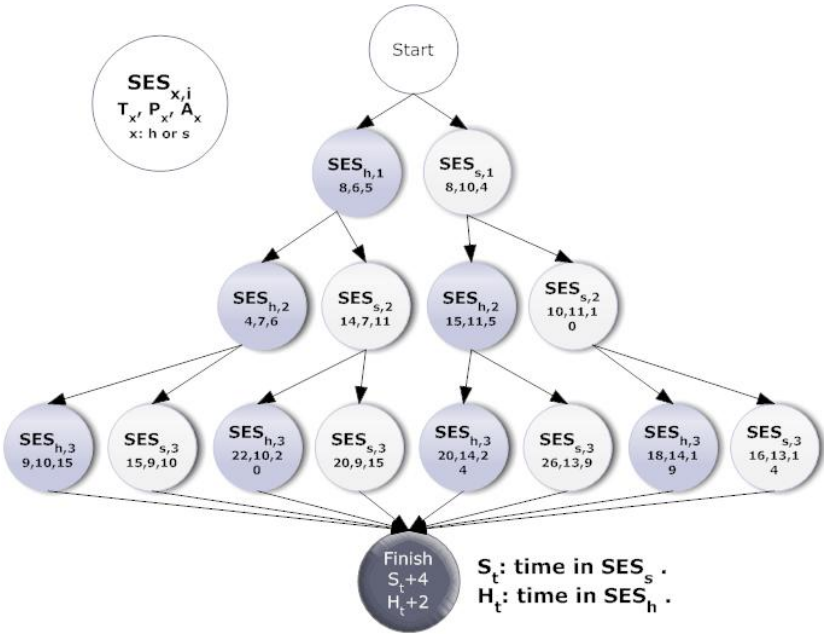
The aim is then to find, at runtime, a configuration that enables an OS service to run under specific resources combination without violating any constraint enforced by the requester application.

The search for a configuration can be done exhaustively by exploring all the possible $SESs$ combinations to run an OS service, see Figure 5.1. Although this will always lead to an optimal solution, it is not practical. For n number of $SESs$ in each implementation $Im \in IM$, the time spent on exhaustive search would be in the order of $O(m^n)$, where $m = |IM|$. If the calculations are to be stored, an amount of $m * (m^n - 1)$ storage elements are needed.

The exhaustive exploring method may be preferable for OS services with small number of $SESs$ and implementations, however, it is out of the question for a large number of $SESs$.



(a) A simple model of an OS service with each implementation partitioned into three SESs.



(b) The exhaustive exploration of all the possible configurations to run the OS service in 5.1(a)

Figure 5.1: An example of an exhaustive exploration of the configurations of a simple OS service model

To search the configurations at runtime, efficient methods are required. This can be achieved using general algorithm designing techniques such as dynamic programming [34].

5.1 OS service model formal definition

In a distributed RSoCs system, an OS exists as a collection of services. Each service has at least two implementations, a hardware and a software implementation. This is due to the fact that each RSoC is assumed to have an FPGA and a GPP.

To generalize the problem lets consider an OS with X number of services. Each service S_i in the OS has m implementations, see Definition (5.1).

$$\begin{aligned} &\exists X \in \mathbb{N}; X \geq 1; \exists m \in \mathbb{N}; m \geq 2 \mid \\ &\qquad \forall i; 1 \leq i \leq X; \exists S_i \in OS \quad and\ S_i = [Im]^m \end{aligned} \tag{5.1}$$

All the implementations of an OS service are partitioned into the same n number of $SESs$, see Figure 5.2(a). Hence, an OS service implementation Im_k can be defined in (5.2) as a sequence of n $SESs$.

$$\exists n, k \in \mathbb{N}; n \geq 1; k \geq 1 \mid Im_k = \bigcup_{l=1}^{n} SES_k, l \tag{5.2}$$

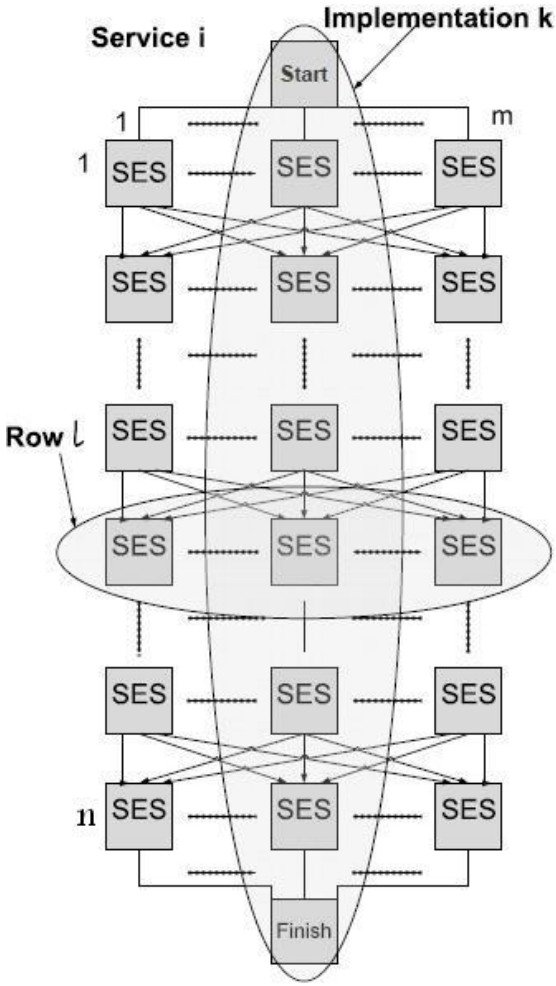
Every SES in an implementation Im_k is connected to all its successive $SESs$ using directed communication edges CE, defined in Definitions (5.3) and (5.4), see also Figure 5.2(b).

$$E \subseteq [SES]^2 \tag{5.3}$$

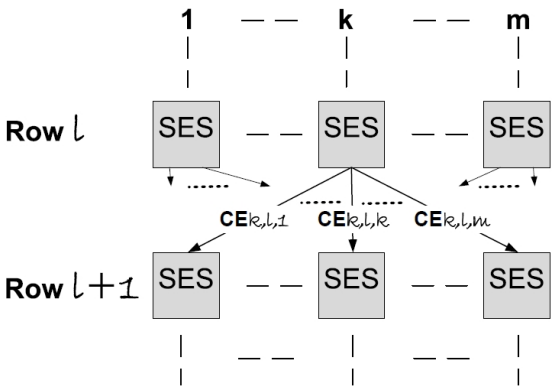
$$\begin{aligned} &\exists k' \in \mathbb{N}; 1 \leq k' \leq m; \forall k'; \exists e \in E; \\ &\exists SES_{k,l} \in Im_k; \exists SES_{k',l+1} \in Im_{k'}; Im_k, Im_{k'} \in S_i \mid \\ &\qquad e = (SES_{k,l}, SES_{k',l+1})\ and \\ &\qquad CE_{k,l,k'} = (e, SES_{k,l}, SES_{k',l+1}) \end{aligned} \tag{5.4}$$

A SES can be abstracted by its resource descriptors, Definition (5.5), where each SES contains three main descriptors which are: its $WCET$ Time (T), its estimated Power consumption (P) and Area (A), see Section 4.2.

$$SES_{k,l} \equiv \{A_{k,l}, T_{k,l}, P_{k,l}\} \tag{5.5}$$



(a) A general representation of an OS service with m implementations each partitioned into n SESs.



(b) A general representation of the edges of the OS service model in 5.2(a)

Figure 5.2: General OS service model with a general edges representation

The directed communication edges can also be defined by the descriptors representing the resources used in communication. These descriptors also include communication time (d_{ct}), communication power (d_{cp}), and communication area (d_{ca}). However, in many cases, the most important descriptor for these edges is the communication time descriptor (d_{ct}).

$$CE_{k,l,k'} \equiv \{dca_{k,l,k'}, dct_{k,l,k'}, dcp_{k,l,k'}\},\ where \tag{5.6}$$

$$k,\ k'\ denote\ implementations\ indices,\ and$$

$$l\ denotes\ execution\ sequence\ index,\ see\ Figure\ 5.2$$

To every OS service, a $START$ entity is added. The $START$ entity has all of its descriptors set to zero. This entity is connected to every SES with execution sequence index 1 in each implementation Im_k of the OS service by an initialization edge ($I_k; 1 \leq k \leq m$). These edges represent any resources (area ($I_k.a$), time ($I_k.t$), and power ($I_k.p$)) required in order to start the corresponding $SES_{k,1}$ of the Im_k implementation. The $FINISH$ entity is added to the end of every OS service. The $FINISH$ entity also has no valuable descriptors. A finalization edge ($F_k; 1 \leq k \leq m$) connects every $SES_{k,n}$ in every OS service implementation $k; 1 \leq k \leq m$ to the $FINISH$ entity. The descriptors of the finalization edge represent the resources (area ($F_k.a$), time ($F_k.t$), and power ($F_k.p$)) required to get the processed data to its final destination. Again, the most probably important descriptor in both the initialization and the finalization edges is time. Nevertheless, in case of network communication, the power descriptor may also be of a non-negligible value.

5.2 Optimal configurations and constraints

The configuration of an OS service depends on the constraints enforced by the requester application and the available resources at the time of request. However, in order to minimize the evaluation time, that is, the time required to take a decision and to find a configuration, boundaries of the maximum/minimum required resources are needed to be defined. In other words, it is required to define which OS service configuration is considered to be optimal in using a resource and which is considered to have the maximum resource usage.

Finding the minimum resource requirement allows an early decision of whether an OS service configuration can be found for a specific resource combination. In case of no configuration can be found, other means, such as finding another RSoC or using more than one RSoC to run an OS service configuration, can be carried out.

On the other hand, defining the maximum resources boundaries allows limiting the exploration of the solution space. This is done by rejecting any OS configuration that may have resources requirements which would exceed the defined maximum boundaries. Additionally, in case of much resources availability, these limits help the quick finding of a Pareto optimum configuration.

To find the minimum boundary for each resource usage, we evaluate to find the OS service configuration which is optimized towards using as low amount of that resource as possible. In RSoC, three main resources are considered. These are: Time, power, and area. So as a first step, an OS service is evaluated to find the optimized configuration to run in minimum time, then to run in minimum power, and finally in minimum area. These evaluations can be performed using the dynamic programming designing technique [34].

5.2.1 Single criterion optimization algorithm

Let's consider the minimum possible amount of resource required for a chunk of data to be processed by passing it from the $START$ entity all the way through $SES_{k,l}$. If $l = 1$, for any implementation Im_k, there is only one way that the data could have gone, and so it is easy to determine how much resources have been consumed to get through $SES_{k,l}$. For $l = 2, 3, ..., n$, however, there are m choices: the data could have been processed by any $SES_{k',l-1},\ where\ 1 \leq k' \leq m$ and then through $SES_{k,l}$. The resources consumed by passing the data to $SES_{k,l}$ is then defined by $CE_{k',l-1,k}$.

Now, we will consider a criterion $\mathcal{C} \in \{time,\ power,\ area\}$ to identify the one resource descriptor we want to minimize. Throughout this discussion, we will use $\mathcal{C},\ SES_{k,l},\ and\ CE_{k,l,k'}$ as a general representation to denote a single criterion (descriptor) representation instead of using subscriptions to represent every descriptor. For instance, instead of using $\mathcal{C}.t,\ SES_{k,l}.T,\ and\ CE_{k,l,k'}.dct$ to denote time. The same also applies for $I_k\ and\ F_k$.

Let us assume that the minimum usage of resource $\mathcal{C}$ to get to $SES_{k,l}$ is through $SES_{k',l-1}$. The key observation is that the data must have been processed with the $SESs$ combination that yields a minimum $\mathcal{C}$ from the $START$ entity through $SES_{k',l-1}$. Why? If there is another $SESs$ combination to get through $SES_{k',l-1}$, we could substitute this $SESs$ combination to yield a minimum $\mathcal{C}$ through $SES_{k,l}$, which contradicts our assumption.

In general, a minimum resource usage with n $SESs$ combination contains within it a smaller number of $SESs$ combinations with minimum resource usage (optimal sub-

structure). This means that we can construct an n $SESs$ combination with minimum resource usage by constructing the minimum resource usage combination of $n-1$ $SESs$.

The construction of n $SESs$ minimum resource usage combination can be done recursively in terms of finding sub-combinations with minimum resource usage for $l = 1, 2, ..., n-1$. Let $\mathcal{C}_k[l]$ denote the minimum possible resource usage to process data from the $START$ entity through $SES_{k,l}$. Our goal is to determine the minimum resource usage to process data all the way through an OS service, which we denote by $\mathcal{C}^*$. The data has to be processed all the way through SES_n on any implementation k and then to the $FINISH$ entity. Since the minimum resource usage of these combinations (configurations) is the minimum resource usage of an OS service, we have

$$\mathcal{C}^* = min_{1 \leq k \leq m}(\mathcal{C}_k[n] + F_k)$$

It is also easy to find $\mathcal{C}_k[1]$. To get data processed through $SES_{k,1}$ on any implementation Im_k, a data just goes directly to that SES through the initialization edge I_k. Thus,

$$\mathcal{C}_k[1] = I_k + SES_{k,1}$$

Now let us consider how to compute $\mathcal{C}_k[l]$ for $l = 2, 3, ..., n$ (and $1 \leq k \leq m$). Recalling that the minimum resource usage through $SES_{k,l}$ is the minimum resource usage through $SES_{k',l-1},\ where\ 1 \leq k' \leq m$, and then directly to $SES_{k,l}$ through the correspondent edge $CE_{k',l-1,k}$. Hence we have:

$$\mathcal{C}_k[l] = min_{1 \leq k' \leq m}(\mathcal{C}_{k'}[l-1] + CE_{k',l-1,k} + SES_{k,l})$$

The $\mathcal{C}_k[l]$ values give the minimum resource usage of sub-combinations of $SESs$. To keep track of how to construct a minimum resource usage $SESs$ combination, we define $\mathcal{M}_k[l]$ to be the implementation number whose $SES_{k,l-1}$ is used in an optimal minimum resource $SESs$ combination through $SES_{k,l}$. Here, $1 \leq k \leq m$ and $l = 2, 3, ..., n$. $\mathcal{M}_k[1]$ is not defined because no SES precedes $SES_{k,1}$. We also define $\mathcal{M}^*$ to be the implementation k whose $SES_{k,n}$ is used in the optimal minimum resource usage configuration through the entire OS service execution. The $\mathcal{M}_k[l]$ values are used to trace an optimal resource usage configuration. Based on this analysis, the evaluation for minimum resource usage can be simply found recursively. However, such recursive algorithm would have a run time which is exponential in n. Nevertheless, by reversing the top down method of the recursive algorithm, a better method could be used in computing the $\mathcal{C}_k[l]$ values. This can be done because for $l \geq 2$, each value of $\mathcal{C}_k[l]$ depends only on the values of $\mathcal{C}'_k[l-1],\ where\ 1 \leq k' \leq m$. By computing the $\mathcal{C}_k[l]$ values in the order of the increasing SES execution index number l, we can compute the minimum resource usage combination for an optimal OS service execution. With this bottom up algorithm the evaluation can be done in $\Theta(n)$ time.

Algorithm 5.1 Single criterion optimal SESs configuration

Require: $x, OS\ service : \{SESs, CE, I, F\}$ //x is the criterion we want to optimize
1: $\mathcal{C}_s[1].t \leftarrow I_s.t + SES_{s,1}.T$
2: $\mathcal{C}_s[1].a \leftarrow I_s.a + SES_{s,1}.A$
3: $\mathcal{C}_s[1].p \leftarrow I_s.p + SES_{s,1}.P$
4: //... {The same (lines 1-3) done for $\mathcal{C}_h.y[1]$ where y is substituted with t, a, and p.}
5: **for** $l \leftarrow 2$ to n **do**
6: //... {DC: denote decision criterion. s: denote software. h: denote hardware.}
7: $DC_s \leftarrow getDCs(SESs, l, x)$
8: $DC_h \leftarrow getDCh(SESs, l, x)$
9: //... {First, evaluate assuming software SES is taken at execution index l.}
10: $DC1 \leftarrow DC_s + CE_{s,l-1,s}.x$
11: $DC2 \leftarrow DC_s + CE_{h,l-1,s}.x$
12: **if** $\mathcal{C}_s[l-1].x + DC1 \leq \mathcal{C}_h[l-1].x + DC2$ **then**
13: $\mathcal{C}_s[l].x \leftarrow \mathcal{C}_s[l-1].x + DC1$
14: $\mathcal{M}_s[l] \leftarrow s$
15: **else**
16: $\mathcal{C}_s[l].x \leftarrow \mathcal{C}_h[l-1].x + DC2$
17: $\mathcal{M}_s[l] \leftarrow h$
18: **end if**
19: //... {Next, evaluate assuming hardware SES is taken at execution index l.}
20: $DC1 \leftarrow DC_h + CE_{h,l-1,h}.x$
21: $DC2 \leftarrow DC_h + CE_{s,l-1,h}.x$
22: **if** $\mathcal{C}_h[l-1].x + DC1 \leq \mathcal{C}_s[l-1].x + DC2$ **then**
23: $\mathcal{C}_h[l].x \leftarrow \mathcal{C}_h[l-1].x + DC1$
24: $\mathcal{M}_h[l] \leftarrow h$
25: **else**
26: $\mathcal{C}_h[l].x \leftarrow \mathcal{C}_s[l-1].x + DC2$
27: $\mathcal{M}_h[l] \leftarrow s$
28: **end if**
29: **if** $x = t$ **then**
30: $\mathcal{C}_s[l].p \leftarrow \mathcal{C}_s[l-1].p + CE_{\mathcal{M}_s[l],l-1,s}.p + SES_{s,l}.P$
31: ... {updating the rest but not the time criterion}
32: **else if** $x = a$ **then**
33: $\mathcal{C}_s[l].t \leftarrow \mathcal{C}_s[l-1].t + CE_{\mathcal{M}_s[l],l-1,s}.t + SES_{s,l}.T$
34: //... {updating the rest but not the area criterion}
35: **else**
36: $\mathcal{C}_s[l].t \leftarrow \mathcal{C}_s[l-1].t + CE_{\mathcal{M}_s[l],l-1,s}.t + SES_{s,l}.T$
37: //... {updating the rest but not the power criterion}
38: **end if**
39: **end for**
40: **return** $\mathcal{C}, \mathcal{M}$

5.2.2 Evaluating for a suitable OS service configuration

For an RSoC with just one FPGA and one GPP we have OS services with each of them having a software and hardware implementations. Algorithm 5.1 shows how to find a single criterion optimal $SESs$ combination (or configuration) for such OS services. In the algorithm we substitute k by either s to denote software or h to denote hardware. The algorithm requires the Worst Case Execution Time (WCET) (T), the required implementation area (A), and the estimated power consumption (P), for every SES in both of its software and hardware implementations. It also requires the setup time (I), to start the execution from either of the first $SESs$, the finish time (F), to get results after executing one of the last $SESs$, and the Communication Edges (CE) connecting the successor SESs which are either in different implementations or in the same implementation.

The algorithm computes the best $SESs$ configuration that is optimized with regard to the resource criterion given in x. Where x can have the value t for $WCET$, a for area, or p for power. It also calculates and stores the values of all the other non-criterion required resources. The area criterion can be further optimized for hardware area hx or software area sx. However, in this algorithm a weighing function is assumed which combines both the hardware and the software areas into a single criterion a. After running the algorithm, it returns two lists, the first one is the $\mathcal{C}$ list, which holds the amount of resources needed to optimally execute from the $START$ entity all the way through every $SES_{k,l}$. The amount of resources and the bath chosen are evaluated taking into account the optimization of the resource criterion provided by x.

The second list is $\mathcal{M}$. It contains the information about the $SESs$ implementations which were chosen to reach every $SES_{k,l}$ with optimal x resource usage. The $getDCs$ and $getDCh$ are select statement functions which take as arguments the $SESs$, the execution sequence index l, and the decision criterion x. As result, they return the value of the decision criterion x in $SES_{s,l}$ and $SES_{h,l}$ respectively. The Algorithm 5.1 runs in $\Theta(n)$ time. This is because it contains only one for-loop which runs for about n times. The other elements in the algorithm can be estimated with constant values.

In a limited RSoC resources environment, where applications are demanding an OS service, the configurations have to be chosen to use the resources which are less demanded by applications. Normally, one resource is more demanded by applications than the others. In such a case, the targeted OS service configuration is not the optimal one. Nevertheless, we target a suitable OS service configuration which runs on the limited RSoC resources without violating any constraint and with minimum usage of

the relatively heavily demanded application resource. In order to obtain such an OS configuration, we use a heuristic algorithm, see Algorithm 5.2.

Initially, we use Algorithm 5.1 to find the timing optimal $SESs$ configuration ($TOSC$), the area optimal $SESs$ configuration ($AOSC$), and the power consumption optimal $SESs$ configuration ($POSC$). From these configurations, a suitable configuration is chosen. A suitable configuration is optimized to use less amount from the relatively heavily demanded application resource without violating any constraint.

Algorithm 5.2 GetSuitableSESsConfigurations

Require: OS service, RSoCConstraints, optimizationRequirements
1: TOSC ← SingleCriterionOptimalSESsConfiguration(OS service, t)
2: AOSC← SingleCriterionOptimalSESsConfiguration(OS service, a)
3: POSC ← SingleCriterionOptimalSESsConfiguration(OS service, p)
4: **if** One constraint is less than required optimal resource in any optimal $SESs$ configuration **then**
5: No possible configuration can be found, notify for alternative solutions, exit algorithm
6: **else**
7: begin with the optimal requested as a main optimization requirement such as TOSC
8: **end if**
9: **for** all $SESs$ in the chosen configuration that are with different implantations in the second or the third optimal configuration and from the last SES to the first **do**
10: try exchange functionally equivalent SES between implementations if that would satisfy the constraints and minimize the other required resources.
11: **if** exchange success **then**
12: mark as possible configuration
13: **end if**
14: **if** no configuration was marked **then**
15: Notify for alternative solutions.
16: **end if**
17: **end for**
18: **return** $SuitableConfigurations$

For instance, suppose an application requests an OS service X with an expected worst case response time given by T_{req}. For the sake of discussion, lets assume that the RSoC has a limited hardware area given by A_{ava} and a plenty of the other resources. The main algorithm, Algorithm 5.2, starts by looking at $TOSC$. If the total execution time of the $TOSC$ configuration is the same as the constraint T_{req}, and the hardware area fits in the available area A_{ava}, the algorithm sends the configuration as a critical one with no alternatives. On the other hand, if T_{req} is greater than $TOSC$, the algorithm looks at the $SESs$ that are implemented using hardware in the $TOSC$, and tries, if possible (no constraints violation), to exchange each one of them with the $SESs$ from the other implementations. The algorithm keeps doing that as long as there is a hardware SES which has not been tried. Other switching between $SESs$ implementations may also be carried out if it leads to a resource usage minimization. The algorithm marks every

suitable configuration found as another possible alternative configuration. When done, it returns the configuration with the lowest hardware area. The marking of alternative configurations makes it faster to find another configuration in case of some faults or errors. In case of no configuration found, other solutions or exploration techniques may be examined. These may include finding another RSoC with enough resources to run the OS service, or using more than one RSoC to share resources in order to run the OS service. These solutions however depend totally on the constraints and especially on the execution and the communication timing constraints.

Configurations limits

The ability to find a configuration depends on the availability of resources and on the enforced constraints from the requesting applications. However, there are situations where no configuration can be found.

Section 7.1 investigates the limits after which the finding of an OS configuration is not possible. The evaluation compares an RSoC with scarce resources to the resource required by an optimal OS service configuration. For instance, there is a very low probability to find a configuration for an OS service with optimal area A_{opt} on an RSoC with available hardware area$= 0.5 * A_{opt}$ and available software area$= 0.5 * A_{opt}$.

To know such facts is very important. This would save a lot of time which would be wasted by trying to find a configuration. Instead, such time could be used in finding other solutions. These include using or sharing other RSoC(s) resources in the distributed system. However, efficient and fast distribution processing implies an efficient OS services distribution and discovery procedures. This is discussed in Chapter 6.

5.2.3 Evaluating for Pareto optimal OS service configuration

A Pareto optimum can be a targeted OS service configuration. This could be in a case where an RSoC has plenty of resources to run any OS service configuration. A Pareto optimum configuration helps avoiding preemption which is caused by the dynamic change in resources.

To find a Pareto optimum configuration, we first define the boundaries or the threshold values to confine our search. Two main threshold values are used to confine the solution space for the targeted configuration. These are the minimum and the maximum amount of resources required to run an OS service configuration. For every resource, that is for

area, time, and power, we will define the threshold of the minimum and the maximum value for a valid OS service configuration.

The minimum resource threshold value can be found by obtaining the optimal OS service configuration for each one of the three descriptors. This is done by using the single criterion optimal $SESs$ configuration algorithm, see Algorithm 5.1. For instance, to find the minimum time threshold value for an OS service, we evaluate to find the time optimal service configuration $TOSC$. As this configuration is optimized to run as fast as possible, its running time is considered the minimum threshold value for time.

In order to find the minimum resource value which can be used by a configuration, we need to evaluate for an optimal resource value from the $START$ entity all the way through $SES_{k',n}$ and finally the $FINISH$ entity. To achieve that, we find the optimal resource value through each $SES_{k',l}$, where $1 \leq l \leq n$, and $1 \leq (k \; or \; k') \leq m$.

Along the evaluation for optimal resource value through each $SES_{k',l}$, we can also evaluate to find the usage amount of the other non-optimized resources. These will be used later to find the maximum threshold values.

Equation (5.7) defines how to evaluate to find the optimal value for the area descriptor regarding $SES_{k',l}$. Along with the minimum area, we also calculate the values of the other non-optimized resources, see Equations (5.8) and (5.9) for time and power respectively. From these equations, only the area value is optimal or has the minimum resource usage.

$$\exists k' \in [1,m] \mid A_{k',l} = min_{1 \leq k \leq m}(A_{k,l-1} + SES_{k',l}.A + CE_{k,l-1,k'}.dca) \tag{5.7}$$

$$T_{k',l} = T_{k',l-1} + SES_{k',l}.T + CE_{k,l-1,k'}.dct \tag{5.8}$$

$$P_{k',l} = P_{k',l-1} + SES_{k',l}.P + CE_{k,l-1,k'}.dcp \tag{5.9}$$

The optimal value for the time descriptor through $SES_{k',l}$ can be obtained using Equation (5.10). The other non-optimized resource usage values for area and power can be found using Equation (5.11) and Equation (5.12) respectively.

$$\exists k' \in [1,m] \mid T_{k',l} = min_{1 \leq k \leq m}(T_{k,l-1} + SES_{k',l}.T + CE_{k,l-1,k'}.dct) \tag{5.10}$$

$$A_{k',l} = A_{k',l-1} + SES_{k',l}.A + CE_{k,l-1,k'}.dca \tag{5.11}$$

$$P_{k',l} = P_{k',l-1} + SES_{k',l}.P + CE_{k,l-1,k'}.dcp \tag{5.12}$$

Finally, Equation (5.13) is used to find the optimal value for the power descriptor. The non-optimized usage resource values regarding area and time can be found using Equation (5.14) and Equation (5.15) respectively.

$$\exists k' \in [1, m] \mid P_{k',l} = min_{1 \leq k \leq m}(A_{k,l-1} + SES_{k',l}.P + CE_{k,l-1,k'}.dcp) \tag{5.13}$$

$$A_{k',l} = A_{k',l-1} + SES_{k',l}.A + CE_{k,l-1,k'}.dca \tag{5.14}$$

$$T_{k',l} = T_{k',l-1} + SES_{k',l}.T + CE_{k,l-1,k'}.dct \tag{5.15}$$

The initial optimal value through $SES_{k,1}$ which is used in evaluating the optimal area, time, and power configurations can be found using the Equations (5.16), (5.17), and (5.18) respectively.

$$A_{k,1} = SES_{k,1}.A + I_k.a \tag{5.16}$$

$$T_{k,1} = SES_{k,1}.T + I_k.t \tag{5.17}$$

$$P_{k,1} = SES_{k,1}.P + I_k.p \tag{5.18}$$

The optimal value for a configuration is known after finding the optimal $SESs$ combination all the way till $SES_{k,n}$. Using the Equations (5.7), (5.10), and (5.13) we can evaluate the required optimal value for an optimal OS service area configuration, optimal OS service time configuration, and optimal OS service power configuration by adding any resources needed for finalization at index n to the calculated optimal resource value. See Equations (5.19), (5.20), and (5.21).

$$A_{opt} = min_{1 \leq k \leq m}(A_{k,n} + F_k.a) \tag{5.19}$$

$$T_{opt} = min_{1 \leq k \leq m}(T_{k,n} + F_k.t) \tag{5.20}$$

$$P_{opt} = min_{1 \leq k \leq m}(P_{k,n} + F_k.p) \tag{5.21}$$

The optimal resource value for an OS service is the amount of that resource used by the OS service configuration which is optimized to use the minimum amount of that resource. In our discussion we will refer to these optimal values as the minimum threshold limits, A_{min} for area, T_{min} for time, and P_{min} for power, to obtain a Pareto optimal OS service configuration.

While evaluating for the area, time, and power optimal configurations, we also calculated the amount of resources required by the other non-optimized descriptors. For example, while evaluating to find the configuration with the optimal area, Equation (5.7), we have also calculated the amount of estimated power, Equation (5.9) and time (5.8), consumed by the configuration. The found configuration would have an optimal area usage, however, the other resources are most probably not optimal. These non-optimal values are used for defining the maximum threshold values.

A maximum resource threshold value is defined as the maximum value of that resource found in any of the three single-criterion optimal configurations. For instance, the maximum power threshold is the maximum value of the Equations (5.9) and (5.12). These equations are used for evaluating the required non-optimized power requirements, while evaluating for the optimal configurations for area and time respectively. The value of Equation (5.13) is not considered because it always has the minimum amount of power that a configuration could consume. The three maximum thresholds are represented as A_{max} for area, T_{max} for time, and P_{max} for power. Another possibility would be the maximum amount of that resource available on the RSoC. However, this may lead to unnecessary expansion of the search space.

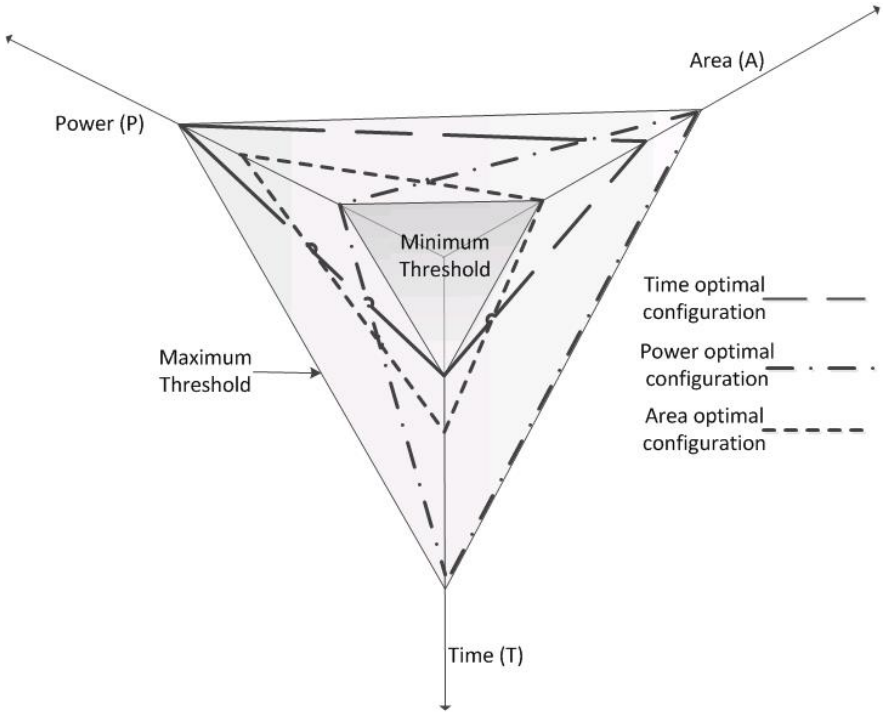


Figure 5.3: Maximum and minimum thresholds to find Pareto optimal OS service configuration

The three single criterion optimal configurations for time, area, and power with the minimum and the maximum threshold values can be represented as coaxial-triangles in a three axis plane as shown in Figure 5.3. The minimum threshold is represented as the triangle which has its three corners defined by the minimum value on each axis. The maximum threshold triangle has its three corners defined by the maximum value on each axis. The Pareto optimal configurations can be represented as the triangles which

are defined to have corners in between the corners of the maximum and the minimum threshold triangles, with as close as possible to the minimum threshold triangle.

Deducing our discussion, we can now evaluate to find a Pareto optimal configuration with the resources requirement A_{p-opt}, T_{p-opt}, and P_{p-opt} as defined in the Equations (5.22), (5.23), and (5.24) respectively.

$$A_{min} \leq A_{p-opt} \leq A_{max} \tag{5.22}$$

$$T_{min} \leq T_{p-opt} \leq T_{max} \tag{5.23}$$

$$P_{min} \leq P_{p-opt} \leq P_{max} \tag{5.24}$$

Algorithm 5.3 Pareto optimization for OS services

Require: OS service $SESs$ and CE
$Acf_{opt} = evaluate\ for\ the\ configuration\ optimized\ for\ area$
$Tcf_{opt} = evaluate\ for\ the\ configuration\ optimized\ for\ time$
$Pcf_{opt} = evaluate\ for\ the\ configuration\ optimized\ for\ power$
$Evaluate\ A_{max},\ T_{max},\ P_{max},\ A_{min},\ T_{min},\ and\ P_{min}$
$First - conf = Acf_{opt}$
$Second - conf = Tcf_{opt}$
$Third - conf = Pcf_{opt}$
$Opt - conf = Acf_{opt}$
repeat
  for all SES_l in First-conf from last to the beginning **do**
    if SES_l is not the same as the SES_l in the Second-conf or the Third-conf **then**
      Try changing SES_l to the one in the Second-conf and then to the one in the Third-conf
      if changing would give a configuration that is better than Opt-conf and there is no resource greater than its maximum allowed value **then**
        Opt-conf=the new configuration after SES_l changed
      end if
    end if
  end for
  if First repeat **then**
    $First - conf = Tcf_{opt}$
    $Second - conf = Acf_{opt}$
    $Third - conf = Pcf_{opt}$
  else
    $First - conf = Pcf_{opt}$
    $Second - conf = Acf_{opt}$
    $Second - conf = Tcf_{opt}$
  end if
until Three times repeated
return Opt-conf

Algorithm 5.3 uses the above analysis to search for a Pareto optimal configuration for an OS service. It requires the OS service $SESs$ and the communication edges CE.

Using Algorithm 5.1, Algorithm 5.3 finds the three optimal configurations for time, power, and area. From these configurations, the algorithm extracts the minimum and the maximum threshold values. The search for a Pareto optimal OS service configuration begins by first choosing a configuration from the three optimal configurations as the Pareto optimal configuration. It then starts comparing the $SESs$ in the different configurations and tries to exchange them. Every exchange is asserted to be within the limits of the maximum and the minimum threshold values. A resource weighting function (e.g.: euclidian function) is then used to compare the configuration obtained after the exchange with the current Pareto optimal configuration. If the resources required by the new configuration are closer to the minimum threshold resources, it will be chosen as the Pareto optimal configuration. The process continues until we have tried the exchanging of all $SESs$ in the three single-criterion optimal configurations. This algorithm has a polynomial running time of $\Theta(n)$.

5.2.4 Wrap up example

To demonstrate some of what we have presented, let us consider the OS service shown in Figure 5.4. This OS service has two implementations: a hardware and a software implementation. Each implementation is partitioned into four $SESs$. For the sake of this example, the resources requirements of each SES was chosen randomly. The communication edges between these $SESs$ are merely representing the communication time which has been also randomly chosen.

Using Algorithm 5.1, we find the $\mathcal{C}_k[l]$ for time, power, and area, see Tables 5.1, 5.3, and 5.5 respectively. These tables show the resources used taking one resource as an optimal criterion from the $START$ entity up until $SES_k[l]$. In addition, they show the amount required by the other non-optimized resources. For instance, the minimum time required for running the OS service from the $START$ entity up until $SES_s[2]$ can be obtained from Table 5.1, which is $\mathcal{T}_s[2] = 23$. In addition, we require 11 units of power and 14 units of area. The value denoted by $\mathcal{C}^*$ represents the total amount of the optimized resources needed to run a complete OS service optimized configuration from the $START$ to the $FINISH$ entity, where $\mathcal{C}^* \in \{\mathcal{T}^*, \mathcal{P}^*, \mathcal{A}^*\}$.

Although the optimized required resources are shown in Tables 5.1, 5.3, and 5.5, we still need to know how to obtain such optimal configurations. This however can be acquired from the Tables 5.2, 5.4, and 5.6 respectively.

The $\mathcal{M}$ holds information about which implementation k of $SES_k[l-1]$ was chosen to reach $SES'_k[l]$ with optimal $\mathcal{C}$ resource usage. For instance, looking at Table 5.2 we

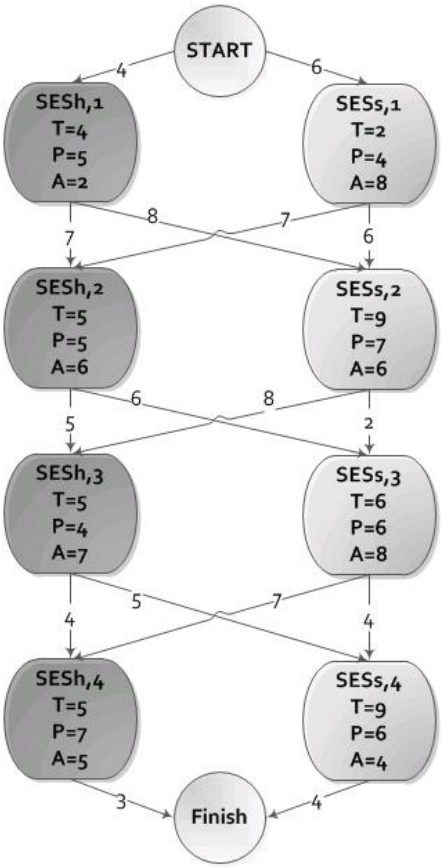


Figure 5.4: An OS service example with random generated meta-data. The communication edges represent only the communication time.

l	1	2	3	4
$\mathcal{T}_h[l]$	$\mathcal{T} = 8$	20	30	39
	$P = 5$	9	13	20
	$A = 2$	14	21	26
$\mathcal{T}_s[l]$	$\mathcal{T} = 8$	23	31	44
	$P = 4$	11	17	23
	$A = 8$	14	22	26

Table 5.1: Time optimized $\mathcal{C}_k[l]$ with $\mathcal{T}^*$=42

l	2	3	4
$\mathcal{M}_h[l]$	s	h	h
$\mathcal{M}_s[l]$	s	s	s

Table 5.2: Time optimized $\mathcal{M}_k[l]$ with $\mathcal{M}^*$=h

know that $SES_h[2]$ can be reached with minimum execution time from the previous SES whose implementation is $\mathcal{M}_h[2] = s$, that is from $SES_s[1]$.

l	1	2	3	4
$\mathcal{P}_h[l]$	$T = 8$	20	30	39
	$\mathcal{P} = 5$	9	13	20
	$A = 2$	14	21	26
$\mathcal{P}_s[l]$	$T = 8$	23	32	44
	$\mathcal{P} = 4$	11	15	19
	$A = 8$	14	22	25

Table 5.3: Power optimized $\mathcal{C}_k[l]$ with $\mathcal{P}^*$=19

l	2	3	4
$\mathcal{M}_h[l]$	s	h	h
$\mathcal{M}_s[l]$	s	h	h

Table 5.4: Power optimized $\mathcal{M}_k[l]$ with $\mathcal{M}^*$=s

l	1	2	3	4
$\mathcal{A}_h[l]$	$T = 8$	20	38	47
	$P = 5$	10	16	23
	$\mathcal{A} = 2$	8	15	20
$\mathcal{A}_s[l]$	$T = 8$	25	33	52
	$P = 4$	12	18	22
	$\mathcal{A} = 8$	8	16	19

Table 5.5: Area optimized $\mathcal{C}_k[l]$ with $\mathcal{A}^*$=19

l	2	3	4
$\mathcal{M}_h[l]$	h	s	h
$\mathcal{M}_s[l]$	h	s	h

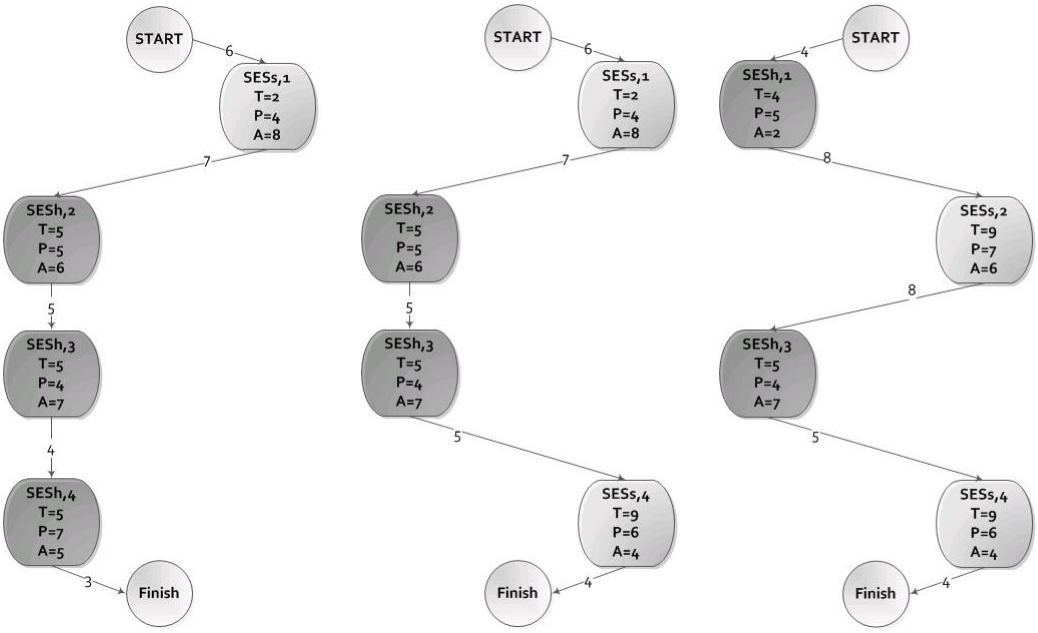
Table 5.6: Area optimized $\mathcal{M}_k[l]$ with $\mathcal{M}^*$=s

The $\mathcal{M}^*$ holds the type of the implementation we chose to obtain the optimal resource usage $\mathcal{C}*$. To get the optimal configuration, we trace back the $SESs$ chosen in order to obtain the optimal resource usage configuration. This tracing is done in backward manner from the $FINISH$ entity to the $START$ entity. The tracing is possible using $\mathcal{M}_k[l]$. We begin by setting $Im \leftarrow \mathcal{M}^*$ and choosing $SES_{Im}[n]$ to be the last SES in the optimal configuration. Then from $l = n, ..., 3, 2$, we chose $SES_{Im}[l]$ as another SES in the optimal configuration by setting $Im \leftarrow \mathcal{M}_{Im}[l]$.

The optimal OS service configurations for time, power, and area can be obtained from Tables 5.2, 5.4, and 5.6 respectively. These configurations are shown in Figure 5.5.

The Figures 5.5(a), 5.5(b), and 5.5(c) show the single-criterion optimal configuration for time, power, and area respectively. The configuration which uses as much minimum h area as possible but executes within a 46 time can be found at run time using Algorithm 5.2. This configuration is shown in Figure 5.5(d).

From the three single-criterion optimal configurations we can define the corners of the minimum threshold triangle, see Figure 5.3. The triangle has its corners defined as 42 on the time axis, 19 on the power axis, and 19 on the area axis. The corners of the maximum threshold triangle would have the values of 52 for time, 22 for power, and



(a) OS service time optimal configuration.

(b) OS service power optimal configuration.

(c) OS service area optimal configuration.

(d) OS service configuration optimized to run in 46 time and have minimum h usage.

(e) An OS service Pareto optimal configuration.

Figure 5.5: The OS service different criterion optimized configurations

26 for area. The chosen Pareto optimal configuration for the OS service has to have its resource requirements between the maximum and the minimum threshold values, with as close as possible to the minimum threshold values, see Section 5.2.3. Using Algorithm 5.3, we get the Pareto optimum OS service configuration, shown in Figure 5.5(e), with the values of 42 for time, 21 for power, and 20 for area.

5.3 SESs granularity and pipelining

So far we have assumed the OS services to be in different implementations, with each implementation partitioned into the same number of $SESs$. We know that the number of implementations is related to the number of the different computational elements in the system. We also know that the level of the flexibility and adaptability introduced in an OS service is proportional to the number of its implementations and to the granularity of which these implementations are partitioned.

To increase the adaptation of an OS service we have to increase the number of $SESs$ by minimizing the partitioning granularity. However, this minimization is limited or governed by the $SESs$ inter-communication overhead. The problem now is to define a lower bound for $SESs$ granularity in terms of execution time and the $SESs$ inter-communication overhead.

On the other hand, the structure of this novel OS service design allows the use of pipelined execution. This powerful technique can be utilized to reduce or provide some tolerance to the $SESs$ inter-communication overhead.

5.3.1 Partitioning granularity

Each OS service consists of a number of implementations that is equal to the number of different computational elements in the distributed system. These implementations are partitioned to the same number of $SESs$. This process enables many useful characteristics such as adaptation and self-healing. It also introduces an undesired overhead of the inter SESs communication overhead. The key element is to find a trade-off between the desired and the undesired characteristics by finding a proper partitioning granularity.

In general, the granularity of which to partition the implementations depends on two factors: The architectures of the computational elements in the system and the inter-communication overhead.

The developed OS service design imposes the partitioning of all the implementations of an OS service into the same number of SESs. All the SESs with the same execution index in all implementations belonging to one OS service have the same functionality behavior. This introduces the first partitioning limitation. The partitioning of one implementation may not be as flexible as the other OS service implementations. For instance, one can argue that the partitioning of an implementation for a Reduced Instruction Set Computing (RISC) architecture can be as small as one instruction, however, in an implementation for a Complex Instruction Set Computing (CISC) architecture, one instruction may map to many RISC instructions. Hence, the minimum partitioning in this case is limited by the CISC implementation.

Nevertheless, architectures are not the major limiting factor of OS service partitioning. In order to run a partitioned OS service, a successive sequence of n $SESs$, regardless of implementations, is required. Obviously, the more fine-granular the partitioning, the higher the inter-communication overhead, but also the better the adaptability.

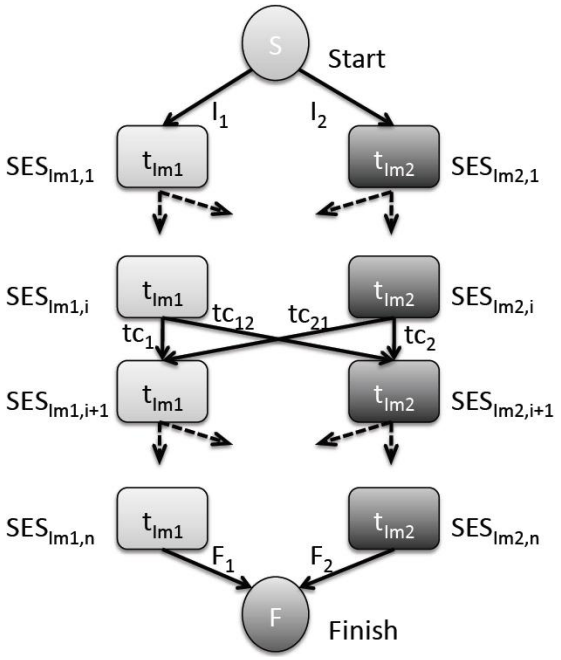
Given a non-partitioned OS service with $WCET$ T, we want to find the maximum number n of $SESs$ with granularity as fine as possible, bearing in mind the communication overhead. Let an OS service X have two non-partitioned implementations, Im_1 and Im_2, with $WCET$ T_{Im1} and T_{Im2} respectively. Now let each implementation be partitioned into $SESs$ with the $SESs \in Im_1$ having a $WCET$ t_{Im1} and the $SESs \in Im_2$ having a $WCET$ t_{Im2}. Here, $t_{Im1},\ t_{Im2}$ are the $WCET$ for the smallest partition granularity possible in $Im_1,\ Im_2$ respectively (see Figure 5.6(a)).

Based on this, we can define $T_{Im1} = n{*}t_{Im1}{+}c_1$, and $T_{Im2} = n{*}t_{Im2}{+}c_2$, where $c_1,\ c_2$ are constants ≥ 0. Note that we still did not consider any inter $SESs$ communication overhead.

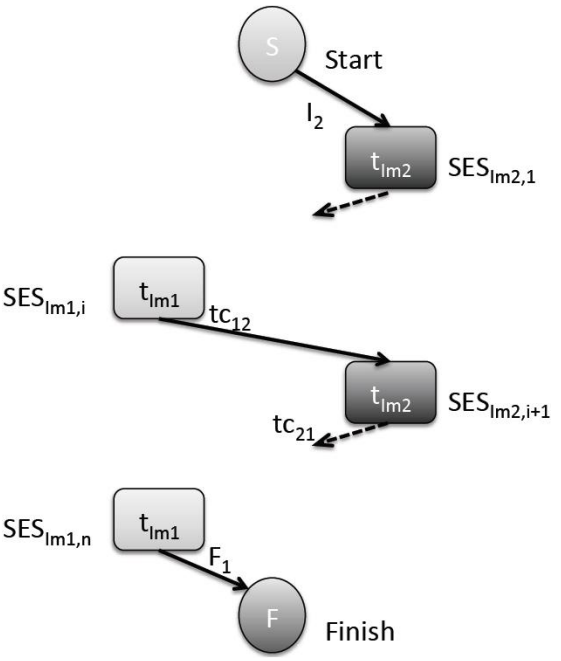
For a service to complete its execution, n successive $SESs$ are required to execute and communicate regardless from which implementation each SES is. The communication between $SESs$ occurs either between two successive $SESs$ in the same implementation, denoted tc_1 and tc_2 in Figure 5.6(a), or between two successive $SESs$ in different implementations, denoted tc_{12} and tc_{21} in Figure 5.6(a). The time needed for two successive $SESs$ in different implementations to communicate is usually greater than the time needed if they were in the same implementation.

Recalling our assumed OS service X, we can define the Worst Case Service Execution ($WCSE$), in terms of inter $SESs$ communication overhead, as follows:

$$WCSE = \bigcup_{i=1}^{n-1} \{(SES_{Im_{x,i}}, SES_{Im_{y,i+1}})\},\ where$$



(a) OS service with two implementations partitioned to the minimum possible fine granularity allowed by both architectures.



(b) The configuration with maximum inter $SESs$ communication overhead (the service $WCSE$).

Figure 5.6: An OS service example with the configuration of maximum inter $SESs$ communication overhead

$$x,\ y \in \{1,2\},\ and\ x \neq y.$$

This means that the $WCSE$ is the combination of $SESs$ in which each SES in a set of two successive $SESs$ $\{SES_i, SES_{i+1}\}$ belongs to a different implementation than the other, see Figure 5.6(b). The time to execute the $WCSE$ of Service X can be computed using Equation (5.25).

$$T_{WCSE} = K + \frac{n}{2}t_{Im1} + \frac{n}{2}t_{Im2} + C. \tag{5.25}$$

Where,

$$K = max(I_x) + max(F_x),\ x \in \{1,2\}.$$

$$C = \frac{n-1}{2}tc12 + \frac{n-1}{2}tc21.$$

Assuming that each implementation is partitioned into $SESs$ of the minimum size possible, then t_{Im1} will be the minimum $WCET$ for SESs $\in Im_1$, and t_{Im2} will be the minimum $WCET$ for $SESs \in Im_2$. From this we can define the minimum $WCET$ in $WCSE$ as $t_{min} = min(t_{Im1}, t_{Im2})$. Using the last definition we can define the $WCET$ for service X without considering the communication overhead as $T_{WCET} = n * t_{min}$, see Figure 5.7(a).

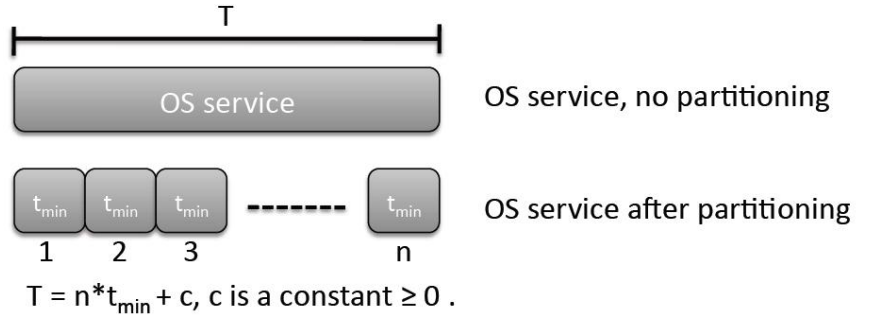
Let β be defined in Equation (5.26) as the time ratio of an OS service $WCET$ with the inter $SESs$ communication overhead to the OS service $WCET$ without the inter $SESs$ communication overhead, see Figure 5.7(b).

$$\beta = \frac{T_{WCSE}}{n * t_{min}}. \tag{5.26}$$

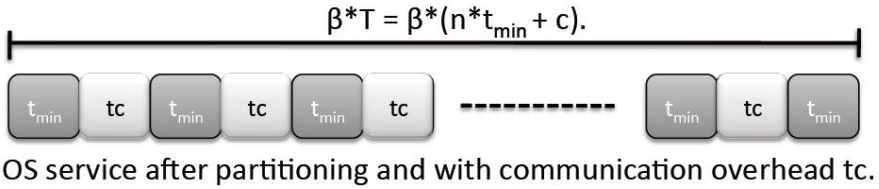
Let T_{WCSE} be limited to an upper bound of $\beta * T = \beta * n * t_{min}$ where β is a heuristic value and $2 \geq \beta > 1$. When $beta$ is 1, it means that there is no inter $SESs$ communication overhead, when it is 2 it means that each inter $SESs$ communication overhead requires the same time as the $WCET$ of a SES. The value of $\beta = 2$ is chosen as an upper bound in reference to Section 5.3.2.

Substituting Equation (5.26) in (5.25) we get

$$t_{min} \geq \frac{K + \frac{n-1}{2}(tc21 + tc12)}{n * (\beta - 1)},\ 2 \geq \beta > 1 \tag{5.27}$$



(a) Partitioned OS service with each partition has as minimum $WCET$ t_{min} as possible. No communication overhead is considered



(b) The partitioned OS service with t_c inter communication overhead.

Figure 5.7: A fine granularity partitioned OS service example with and without inter $SESs$ communication overhead

The communication time $tc21$ and $tc12$ are equal to the maximum time between any two successive $SESs$ communicating with each other. Equation (5.27) gives an estimation of the minimum allowable SES granularity in terms of the underling inter-communication time. Figure 5.8 shows some values for a minimum granularity limit of a SES when $tc21 = 75\mu s$ and $tc12 = 65\mu s$.

The communication between $SESs$ can either be control messages or data. Examples of control messages between an RSoC middleware and the $SESs$ include: a $START$ signal to initiate the processing procedure, a $DONE$ signal to indicate the end of data processing, an $ADDRESS$ signal as a pointer to the data to be processed, and other messages intended for fault recovery, distribution, and obtaining metadata about the SES itself. To gain access to the data to be processed, the $SESs$ use shared memory. This requires no extra complex synchronization as there is no competing between any two $SESs$ to access such data at the same time. The middleware controls when a SES

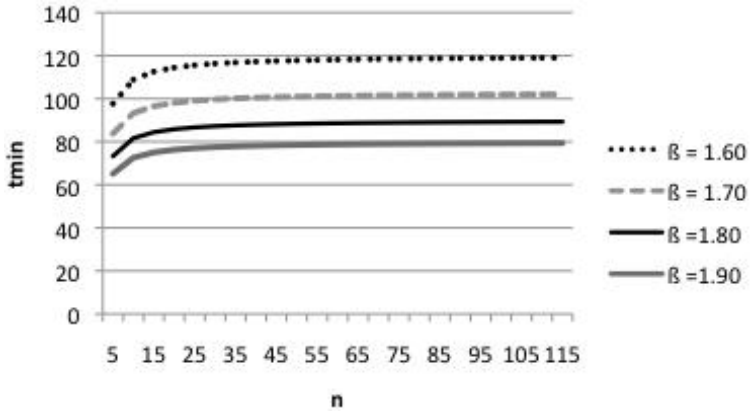


Figure 5.8: A comparison of the minimum allowable SES granularity with regards to the underlying inter-communication time. All values in μs

can access the data. This accessing is commenced sequentially depending on specific conditions being satisfied.

The data to be processed is better suited if it is divided into chunks. This will enable the use of pipelining as discussed in Section 5.3.2. Moreover, it will allow controlling the $WCET$ of each SES as this is normally proportional to the size of the processed data chunk.

5.3.2 Pipelining execution and communication time

Although the partitioning of an OS service into $SESs$ improves execution flexibility, adaptation resource variations, optimization, and fault tolerance, it has its drawbacks as well. The added inter $SESs$ communication overhead may increase the $WCET$ of a service significantly. This is especially true when the chosen configuration of an OS service involves much inter $SESs$ communication between $SESs$ in different implementations.

Although this non-negligible communication overhead is unavoidable, the $SESs$ of an OS service provide a workaround solution. The partitioning into $SESs$ enables the execution of a service using pipelining. This, if used, can reduce the $WCET$ to be even less than that of the non-partitioned OS service.

To explain this, let us assume a non-partitioned OS service X which can process a data chunk $d_i \in D$ with a $WCET$ T, where $D = \bigcup_{i=1}^{\omega} d_i$, and ω is the total number of data chunks. Let the OS service X be partitioned into n $SESs$ of minimum size and equal execution time t_{min}, where, t_{min} is the $WCET$ time required by a SES to process a

data chunk d_i. Normally it is preferable to have a communication time which is less than processing time. However, to see the benefits gained by using pipelining we will consider an OS service execution configuration in which all inter $SESs$ communication times being the worst case communication time $tc = t_{min}$. Hence, the total communication time in the chosen complete OS service configuration will be $(n-1) * tc$. This sums the total $WCET$ of the OS service configuration to $(n-1) * tc + n * tc$ = $2{*}n * t_{min} - t_{min} \leq 2 * T$.

Following what was assumed, a non-partitioned OS service requires a total time of $2 * T$ to process two data chunks. For ω data chunks we need a time of $\omega * T$. The partitioned OS services, if no pipelining used, requires a total time of $2 * \omega * T$ to process ω data chunks (see Figure 5.9).

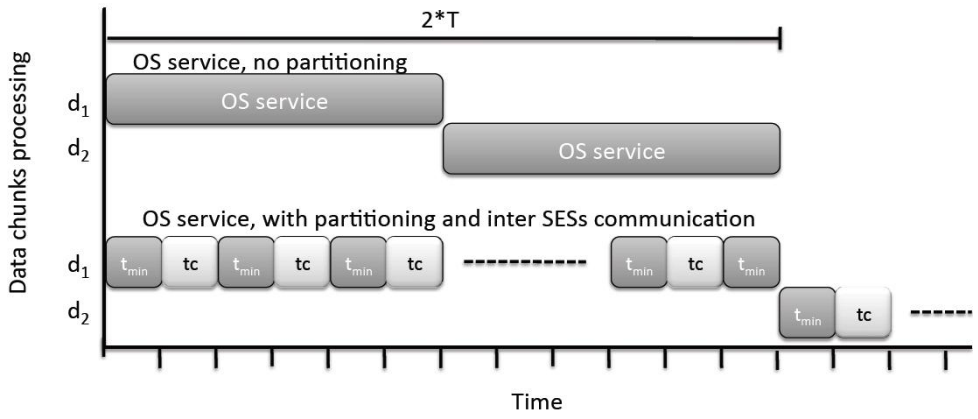


Figure 5.9: The time needed to process one data chunk d_i for non-partitioned OS service X and the partitioned OS service X with worst case inter $SESs$ communication time tc

Using pipelining (see Figure 5.10), we could have the first SES in a service execution begin processing the next data chunk as soon as it finishes processing the first data chunk. In the meanwhile the processed data will be transferred to the successor SES in the service execution. Following this pattern, every SES will begin executing a new data chunk at the same time the processed data is being transferred to a successor SES. Doing so, the total $WCET$ would be minimized to have the value defined in Equation (5.28).

$$T_{pipeline} = 2 * T + m * \frac{T}{\mu} = 2 * T + m * t_{min} \tag{5.28}$$

For a service processing one or two data chunks, the non-partitioned version of a service will be faster. However, for processing a large number of data chunks, the

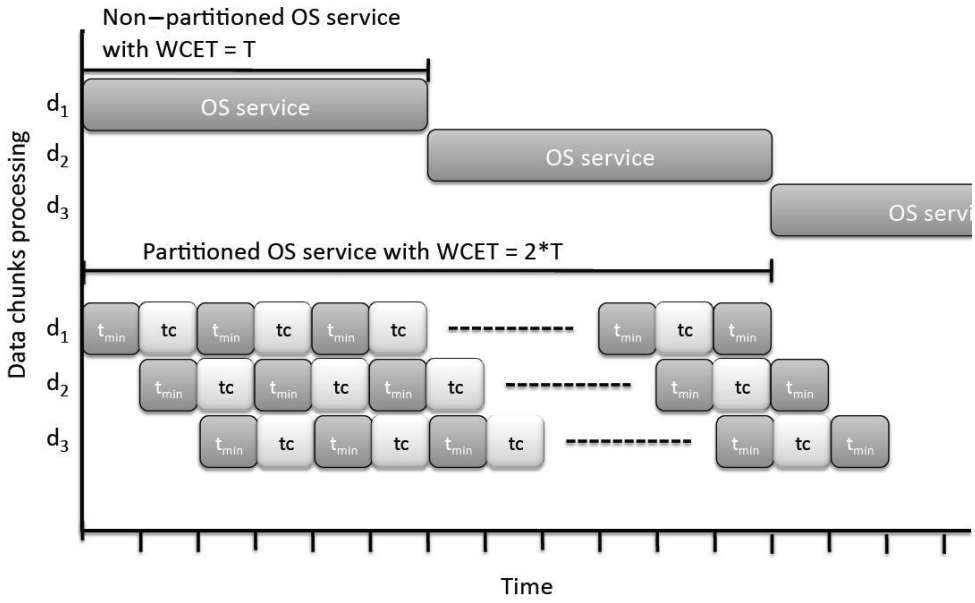


Figure 5.10: The time needed to process three data chunks $d_1 - d_3$ for non-partitioned OS service X and the pipelined partitioned OS service X with worst case inter $SESs$ communication time tc

pipelined partitioned service will have the upper hand in a lower execution time (see Figure 5.11).

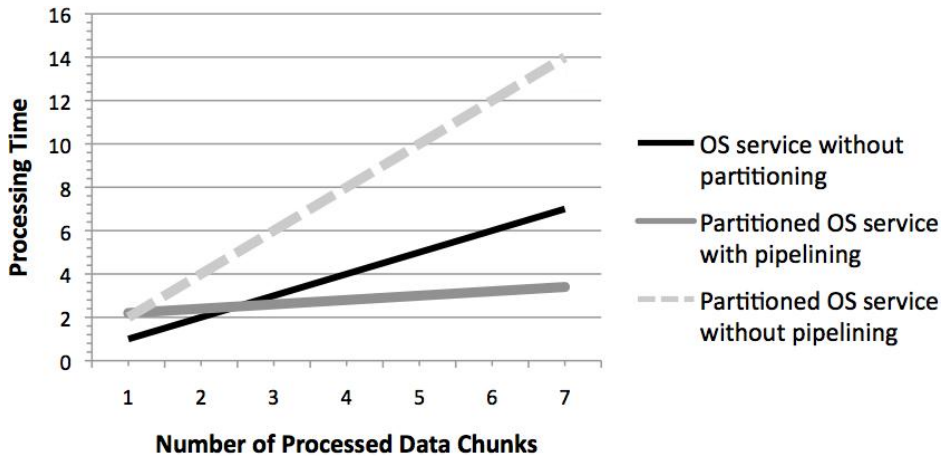


Figure 5.11: TA comparative representation of the time needed to process a number of data chunks using non-pipelined partitioned service, non-partitioned service, and a pipelined partitioned service

Although this discussion shows a possible solution to the communication overhead problem, it remains adequate to mention that the main objective of partitioning is not to have a faster execution, but to provide for adaptation to resource variations and applications demands expected in a distributed RSoCs system.

5.4 Chapter Conclusions

The multiple implementations and partitioning allow for many possibilities to execute an OS service. Each possibility has different resource requirements, hence the adaptation. However, finding the proper configuration for a specific resource environment and a set of constraints requires efficient and fast algorithms.

This chapter presented the formal description of the OS service model and the considered constraints. The developed exploration algorithms explore an OS service model to find a proper configuration under some specific criterions. These may be finding such a configuration which allows the OS service to run under scarce resources or a configuration which has a Pareto optimal usage of resources. An example was presented which uses these algorithms to find different configurations of an arbitrary OS service under some chosen constraints.

As this OS service model deals with partitioning, some discussion on the granularity of these partitions was presented. Furthermore, the use of pipelining technique is introduced as a way to overcome the inter partitions communication overhead.

CHAPTER 6

OS services distribution

Distributed embedded (RSoCs) systems are in general limited in resources. However, they are desired to deal with an unpredictable spectrum of applications. An OS designed for distributed embedded system is faced with two contradicting requirements. An OS mostly is preferred to provide a wide range of services to support a broad range of applications. On the other hand, it is expected to use as little resources as possible to allow enough resources to be utilized by applications. All of this has to be done without violating any constraint imposed by the requested QoS.

There is no doubt that the introduction of another computational element enables the execution in parallel. However, if these are not of the same kind (e.g.: one is an FPGA and the other one is GPP), we would be given the advantage of heterogeneous execution. This is because a task executed on GPP may have a response time and resource usage which is significantly different from its response time and resource usage if implemented on FPGA.

The OS service design, introduced in Section 4.2, utilizes this heterogeneity to allow for flexibility and adaptability in scheduling real-time applications/tasks. Moreover, it allows for an OS service to be distributed by distributing its $SESs$ across the RSoCs in the system. Doing that allows resource sharing and fault tolerance. In addition, RSoCs can share to hold a wide variety of OS services. Nevertheless, basic middleware is still required to exist on each RSoC in the system, see Section 3.2.1. This middleware

provides the necessary means and units to manage and realize the intended objectives, see Section 4.1.

6.1 $SESs$ distribution

In a distributed system, it is highly expected for applications/tasks to arrive (execute) and leave dynamically. In other words, a static set of applications/tasks for one RSoC may not be known in advance. As a result some RSoCs may need to execute OS services or have some functionalities that are previously not present or anticipated. One solution to overcome such a problem is to have a complete OS copy on each RSoC. However, this means the reservation of resources for some OS services on each RSoC which may never be used. Additionally, it would prevent applications from having the luxury of using more resources, if needed, to speed up their computations which may be necessary sometimes. Moreover, this contradicts our assumption of RSoCs having limited resources. Nevertheless, we still want an RSoC to have access to any service that can be offered by the OS to support a wide range of applications. We also want the OS service to have the attributes of adaptability, fault tolerance, and recovery.

6.1.1 Heterogeneity and Distributed Environments

The RSoCs in the distributed system are assumed to be heterogeneous. Although heterogeneity often denotes the diversity of hardware, we refer here to heterogeneity which occurs in resources with equal functionality. For instance the size of FPGAs, the power consumption or storage capacity. Both heterogeneity areas meet different needs in distributed environments. Mainly, there are two tasks to be solved. First, the identification of resource entities offering the required functionality. Second, allocating the resources according to an objective function, e.g. even percental usage. Here, we concentrate on the second one. Thus, we denote with heterogeneity different sizes of the same resource type.

Heterogeneity can be resolved in different ways. If dynamical changes in a distributed environment like failing, inserting or removing resources are allowed, simple static solutions are inefficient [157, 103]. A classical dynamical approach is Consistent Hashing aka. Distributed Hash Tables(DHT) [88]. It can be used to balance uniform resources and handles dynamical changes efficiently. In case of heterogeneity it is combinable with capacity normalization, which typically increases the model complexity. This is because the number of entity representors depends on the normalization

factor of each entity. This factor is determined for each entity and either described by the ratio of its capacity to the smallest entity or to the smallest capacity difference. This unpredictable behavior makes it hard to use such an approach in restricted environments like embedded systems. So, models resolving heterogeneity in a dynamical environment directly with less complexity and being independent of the heterogeneity gap are needed. To mention some models we refer to SHARE, SIEVE, DHHT, SPREAD [27, 129, 108, 26]. Those models are nearly independent of this gap, but intensively use random values and inherit side effects from balls into bins problems, namely deviation and logarithmic complexity overhead. The deviation problem can be handled by multiple choice [109], but the complexity overhead remains as well as the necessity to embed fault tolerant codecs, see Section 6.1.2. This embedding is possible if distinct entity selection is provided.

6.1.2 Fault Tolerance or Availability

Fault tolerance concerning the availability of data, can either be solved by redundant data segments or by using a specific encoding scheme. However, reliability requires the segment placement on distinct devices. The method decision depends on the storage or performance available to apply the decoding and needed to fulfill additional constraints. In Storage Area Networks a quiet common scheme is used by RAID IV or V [115]. Here, the advantage of XOR parity encoding is given by using less storage compared to duplication. Furthermore, the encoding is easy to calculate on controllers. However, this compensates only one failing device. A later published scheme RAID VI includes a two dimensional parity scheme, tolerating any of two devices to fail. The code is based upon spacial Reed-Solomon-Codes [119], which actually is a k-dimensional MDS Code [53]. However, even by using $k = 2$ the code is still computation intensive, whereas the Liberation Code [120] offers a two dimensional parity information still determinable by XORs. So, the application domain decides which code is applicable and eventually the evolution of the domain and its future perspective. Here, we will simply rely on the idea that we can use an $\eta = m + \lambda$ codec providing us $\binom{m+\lambda}{\lambda}$ possibilities to reconstruct the original information. For this we have to determine access patterns of $m + \lambda$ distinct entities capable to allocate all available resources in the system.

On the other hand, at one point, there may be no suitable OS service configuration for one RSoC, see Section 5.2.2. Or the OS services repository, which may be a key role, fail to exist. So although the partitioning of services implementations enables the OS services to adapt to many tasks/applications requirements, the limitation introduced from the resources such as FPGA area and power consumption may hinder it. More-

over, in a distributed system this may happen to be on one RSoC but not on the other. This means resources may exist on other RSoCs which allows faster configuration to be chosen.

So in order to benefit from the heterogeneity introduced in RSoC resources and to avoid single point of failure introduced in centralizing the OS, see Section 3.1.1, we distribute OS services $SESs$ across all RSoCs, see Section 3.1.3. Moreover, we want to be able to share the resources to execute any configuration using the collaboration of many RSoCs.

6.1.3 Distribution stages

Due to the heterogeneity of resources, some RSoCs may be able to execute a complete service or even a mini OS using its own resources while others may not have the chance to execute even a part of an OS service. To overcome such a variation, we want to view transparently all the RSoCs as if they are one when it comes to resource usage, and as many when it is related to fault tolerance and error recovery. To realize this, we distribute the OS services $SESs$ across all the RSoCs in a way that insures load balancing, fast $SESs$ discovery, error tolerance, and fault recovery. These objectives can be archived in two stages, the initialization stage and the discovery and routing stage.

6.2 The Initialization stage

As mentioned before, we consider a heterogeneous distribution topology which combines the attributes of the fully decentralized and the centralized systems, see Section 3.1.3. This is because we have an OSR which has all the OS services and can act as a centralized node. However, this OSR is only necessary in initialization and can act as a backup in the other stages.

In the initialization stage we want to distribute all the $SESs$ of the OS services across the RSoCs in the system. As both the $SESs$ and the RSoCs are heterogeneous, a fair distribution is required to balance the percentage utilization of the resources in the distributed RSoCs. Hence, the ability of an RSoC to provide for any requests to execute these $SESs$ without violating any constraints.

One solution is to use random distribution. A repository is assumed in the initial stage to hold the OS services $SESs$. These $SESs$ are distributed randomly on RSoCs.

However, if an RSoC receives a SES which exceeds its $SESs$ resource reservoir, the SES is rejected and it is propagated to another RSoC. The RSoC $SESs$ reservoir is a resource percentage reserved for the sake of storing and running a number of $SESs$.

To allow a kind of balancing, a simple algorithm which involves an updated polarity number and a simple counter is used.

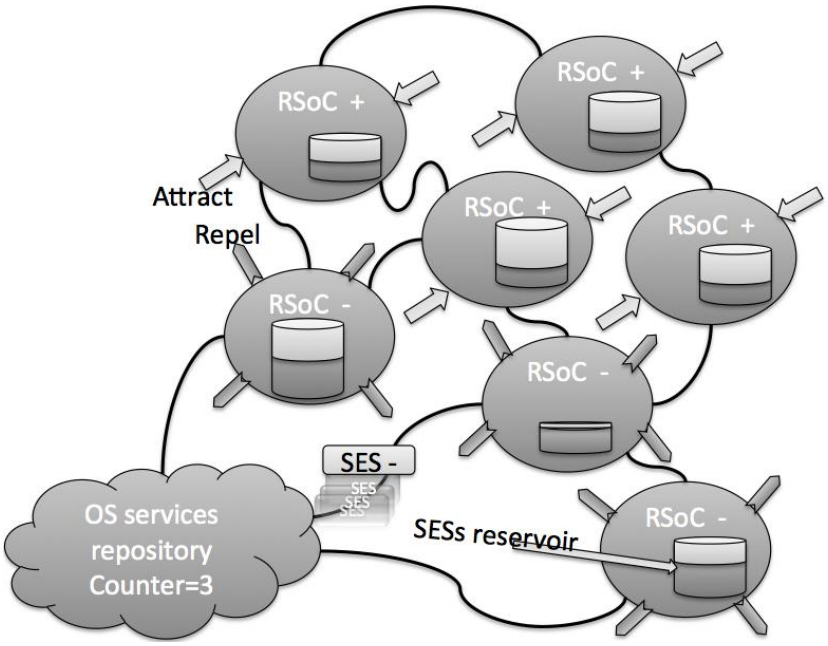


Figure 6.1: Random OS services $SESs$ distribution with load balancing

The counter in the repository is initialized to the number of RSoCs and the polarity of each RSoC is set to be positive. All the $SESs$ are with negative polarity. Each time a SES is sent out from the repository, the counter is decreased. If an RSoC with positive polarity receives a SES, it attracts the SES, stores the SES in its resource reservoir, and changes its polarity to negative. On the other hand, if an RSoC with negative polarity receives a SES, it repels the SES to other RSoCs, see Figure 6.1. A SES is propagated through the RSoCs until it reaches an RSoC with positive polarity. Once the counter in the repository reaches zero, a message is sent to reset the polarities of the RSoCs which still have some free resource reservoir to positive, the counter is set to the number of these RSoCs, and then the operation repeats.

Although this may lead to some load balancing, it is still not fair. Due to RSoCs resource heterogeneity, in some cases one RSoC may have its reservoir totally used while others are barely touched. Moreover, there is no information about the locations of $SESs$. This requires a discovery algorithm, prior to service execution or recovery

process, in order to find the $SESs$ belonging to the desired OS service configuration, see Section 6.3.2.

Another solution is to use a deterministic algorithm. The General Fork Distribution (GFD) algorithm is developed to insure the load balancing of the heterogeneous $SESs$ distribution over the heterogeneous RSoCs. It also provides important information which is used in secondary stages like execution and fault recovery, see Section 6.2.1.

6.2.1 Distribution algorithm

The deterministic GFD distribution algorithm works by determining patterns (each called a fork distribution pattern) with the purpose described in Sections 6.1.1 and 6.1.2. Let V be the set of all RSoCs in the network offering the needed resource and D the set of $SESs$ aka. items of the OS. Further, $c(x)$ is an arbitrary capacity function which describes the context depending on the correspondent entity. The $c(x)$ function denotes the required allocation resource when corespondent to a SES entity, where $c : D \mapsto \mathbb{R}$. In case of an $RSoC$ entity, it represents the available resource capacity, where $c : V \mapsto \mathbb{R}$. In case of a fork distribution pattern fdp, it defines the used resource size, where $c : V^{m+k} \mapsto \mathbb{R}$. Furthermore, we assume that $c(d) << c(v); d \in D, v \in V$ so $SESs$ are much smaller than the OSR or RSoC capacity.

The General Fork Distribution (GFD)

The aim of this algorithm is to partition and to combine a set of limited and fixed resource types of heterogeneous size. The combination of partitions is made to obtain a fork distribution pattern set $FDP \subseteq V^{m+\lambda}$ capable, and to allocate the resources such that $\sum_{v \in V} c(v) = \sum_{fdp \in FDP} c(fdp)$. Further, FDP has to be defined such that for each $fdp \in FDP$, the number of distinct RSoCs is $|fdp| := |\{v, v \in fdp\}| = m + \lambda$. To fulfill this is essential because of the encoding schemes we want to apply, see Section 6.1.2.

The fork distribution pattern set FDP construction is described in two parts. The first part is the construction of $m + \lambda$ frames, Algorithm 6.1, Line 3. This is done by sorting RSoCs decreasingly according to a capacity function $c(x)$ in a list L. This list is then normalized to list M, such that, for each $v_i := L.elementAt(i) \mid i \in \{1, \ldots, |V|\}$, then v_1 owns $[0, r_{v_1}]$, $v_{|V|}$ owns $[1 - r_{v_{|V|}}, 1]$, and $v_i \mid 1 < i < |V|$ owns $[\sum_{j=1}^{i-1} r_{v_j}, \sum_{j=1}^{i} r_{v_j}]$, Algorithm 6.1, Lines 1 and 2. This part describes the order of

subsequent resources within $m + \lambda$ frames in M. Thus, each $x \in M$ belongs to a frame F_j and identifies an RSoC v covering the specific area. Note that a resource can possibly overlap from F_j to F_{j+1}.

The second part describes the construction of FDP, based upon the frames by control variables, Algorithm 6.1, Lines 4 to 15.

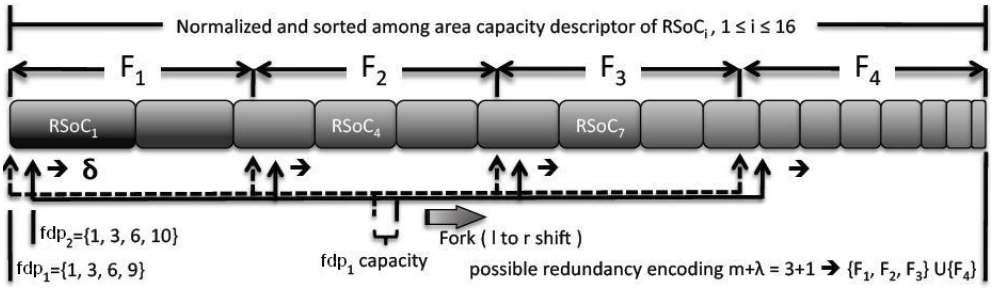


Figure 6.2: An illustration of the General fork distribution algorithm distinct pattern determination

Algorithm 6.1 General Fork Distribution Algorithm

Require: set of RSoCs V, set of $SESs$ V

1: sort $\forall v \in V$ among $c(v)$ decreasingly ordered into a list L
2: define $M := [0, 1]$ to be the ordered capacity in L, where each $v \in L$ owns a range of length $r_v := c(v) \cdot (\sum_{u \in V} c(u))^{-1}$
3: define $F_j \mid j \in \{1, \ldots, m + \lambda\}$ to be the subsequent frame partitioning in M of size $\frac{1}{m+\lambda}$, where F_j covers $[(j-1) \cdot (m + \lambda)^{-1}, j \cdot (m + \lambda)^{-1}]$
4: define a fork with $m + \lambda$ fingers each labeled by f_j, where $1 \leq j \leq m + \lambda$
5: set an offset $\delta := 0$
6: set $f_j := (j-1) \cdot (m + \lambda)^{-1}$, identify the RSoCs $fdp := \{v_1, \ldots, v_{m+\lambda}\}$ covering the finger positions in M
7: set $FDP := \{fdp\}$.
8: **while** $\delta \leq (m + \lambda)^{-1}$ **do**
9: set $fdp' := fdp$
10: increase δ
11: **if** any $f_j + \delta$ identifies a new RSoC, means $|fdp \cap fdp'| > 0$ **then**
12: identify the RSoCs $fdp := \{v_1, \ldots, v_{m+\lambda}\}$ covering the new finger positions in M
13: set $FDP := FDP \cup \{fdp\}$
14: **end if**
15: **end while**
16: At this point we can either return the fork distribution pattern set FDP to be used with an external distribution algorithm or
17: using a proper resource weighing function c, distribute $SESs$ over the the fork distribution patterns in FDP

The described algorithm behaves like a fork with $(m+\lambda)$ fingers of distance $(m+\lambda)^{-1}$ shifted among the range. Each time a finger touches a new RSoC area, the identified RSoCs are added as a new pattern to FDP. Both described construction parts to obtain FDP can be reconstructed by the illustration in Figure 6.2.

Clearly, the number of frames should be defined by the chosen encoding scheme, see Section 6.1.2. This is identified by its fault tolerance λ to be $m + \lambda$ frames. In such a case, the fork would have a number of $m + \lambda$ fingers, and the architecture defines patterns with each of them consisting of distinct $m + \lambda$ RSoCs. Thus, any encoding scheme producing $m + \lambda$ segments can be mapped directly to a pattern and fulfill its computation constraints. However, it must be noted that this can only be guaranteed, if the constraint $\max(\{c(v)\}) \leq \sum_{u \in V} c(u)$ holds. This means, $\forall r_v, r_v \leq (m + \lambda)^{-1}$ is true. If violated by any $v \in V$, the distinct pattern construction is impossible. This can be fixed by including v with less $c(v)$. Eventually, it remains to spread the allocation request evenly among the patterns so that they satisfy the objective function.

One possibility that may occur is to have an RSoC with an overcapacity. This however could be combined with other similar RSoCs to define further patterns. Nevertheless, it may not be possible to combine at least $m + \lambda$ RSoCs at a time. But it may remain adequate to fulfill the objective function by spreading data among the new patterns, which disburdens the others. On the other hand, such overcapacity could be utilized for other duties.

A similar effect occurs if RSoCs are failing or are being removed. If no more than λ RSoCs per pattern are failing, the decoding is not endangered. If patterns should provide the fully specified decoding functionality they must be reorganized. This healing can be done by identifying those patterns with failing RSoCs. Then, with the remaining working RSoCs, we construct a set of new patterns as described. Again, this is only possible if the number of RSoCs is equal or larger than $m + \lambda$. If combining was possible, the data of the new patterns must be reorganized to repair the encoding. Again, overcapacity might remain. In any case, the new additionally defined patterns have in total less capacity than before. This capacity loss and the constraint of the objective function implies data reassignment. The data flow will be from the new patterns among the old ones. So it is similar to the previous one, but in the other direction.

It must be noted that, if in any of both described cases the $m + \lambda$ constraint is not satisfiable and untouched patterns are existing, due to a pattern reservation scheme or a small amount of data distribution, one can involve some of them too. Eventually, it depends on the costs for reorganizing data, because of repairing the data encoding. The more patterns are involved, the less load percentage per RSoC, hence the less

rebalancing is needed. On the other hand, the more the patterns, the more the RSoCs involved in the encoding, hence, the more the costs to repair the encoding.

6.3 The discovery and execution routing stage

After the $SESs$ distribution, the next stage is defining the methods of finding the combination of $SESs$ for a desired OS service configuration. This stage is required to execute an OS service either on one RSoC by $SESs$ migration or on multiple RSoCs by data transferring. In either ways, we need to know where are the $SESs$ belonging to one OS service.

This process, however, is depending on the distribution stage. In case of random distribution, see Section 6.2, there would be no information on where each SES is located. This requires adequate methods to discover and locate $SESs$.

Approaches to discover distributed entities have been proposed in different contexts. To find resources or entities situated on different nodes in a network, the classical link-state protocol [117] maintains a global view of the entire network at each node. To keep this information up-to-date, each node periodically initiates a flooding of the network. Naturally, this method does not scale well. To improve scalability, Perkins and Bhagwat [117] proposed DSDV, which is to a certain extent similar to the link-state approach but only maintains the information about the destination, the next hop towards it, the corresponding distance, and the sequence number. Motivated by the fact that most of the data exchanged by DSDV is not needed (e.g. since a certain entity is of no interest) and changes in the topology cause a network flooding, Perkins and Royer proposed AODV [118]. Being similar to DSDV, it only discovers remote entities and repairs routes when needed, i.e. in an ad hoc manner. As DSDV and AODV discover only a single entity and a single path to such an entity that matches a certain requirement (e.g. a certain node ID), both are not suited for inter-service communication on distributed RSoCs. Protocols that aim to enable this functionality, uniting multicast and multipath methods, such as ADMR [83], however fail to exhibit a behavior that adequately reflects the underlying network properties, e.g., the distance to, the quality of, or the quality of the path towards the remote entity. Several further approaches have been proposed that ignore the network topology and instead focus on the semantic level of service discovery: One of the most prominent examples is Sun Jini [84], which relies on a central service directory, where providers register and clients send queries to, which unfortunately represents a single point of failure. SLP

[138] is in many ways similar to Jini, however, it supports multiple directories (i.e. directory agents).

In our case we have developed two approaches. Our biologically inspired approach, presented in Section 6.3.2, is based on a hop–by–hop reactive routing algorithm [81]. This approach was inspired by the swarm intelligence behavior of ants communicating by changes of their environment [69]. However, this approach is not deterministic, in a scenes that the time needed to discover an entity is hard to be determined prior to the discovery process.

On the other hand, using the deterministic approach, which is presented in Section 6.2.1, much useful information could be obtained seamlessly. These information are utilized to build the "OS service execution routing graph" which is stored in the $START$ entity of the related OS service. An "execution routing graph" contains information about the OS service $SESs$, the communication cost, the execution data, and the location of each related SES in the distributed system. Using these graphs and the algorithms presented in Chapter 5, a desired OS service configuration can be obtained at runtime with minimum effort.

To cope with the deterministic approach, two important records are added to each metadata SES at the distribution stage. The first record contains information about the RSoCs holding alternative $SESs$. These are the $SESs$ with the same behavior as this SES. The size of the alternatives record is $m + \lambda - 1$ and it is used for fast recovery and fault handling procedures.

The second record contains information about the RSoCs holding the direct successors $SESs$ of this SES. The size of this record is $m + \lambda$ and it is used for rebuilding the "execution routing graph" in case the originals were lost. In addition, this record can be used to execute OS services using a greedy algorithm by choosing every time the next SES with minimum resources. As this will provide no prior information on the total execution resources of an OS service, it can be done when no constraints exist.

6.3.1 The building of the execution routing graphs

The seamless building of OS services execution routing graphs can be done using the GFD algorithm. These graphs are used at runtime for routing data, $SESs$ migration, recovery, and execution adaptation in regard to resources availability and requirements. An execution routing graph of an OS service is stored in the $START$ entity of the service. Because of the used encoding, see Section 6.1.2, the $START$ entity exists in $m + \lambda$ copies for each OS service. These, unlike the $SESs$ entities, are merely copies,

with no encoding, to allow quick accessing and processing. An execution routing graph consists of Metadata Distributor Entities (MDE) and directed edges. An MDE holds information about one SES. The directed edges, between two MDEs, represent communication time between successive $SESs$. This is defined as the communication time between $RSoC_i$ and $RSoC_{i+1}$ such that $RSoC_i$ holds SES_i and $RSoC_{i+1}$ holds any immediate successive SES_{i+1} of SES_i, where $1 \geq i \geq n-1$. The mapping between $SESs$ and MDEs is one to one. The information stored in an MDE includes the address of the RSoC where the correspondent SES is located, the correspondent SES ID, and the execution information about the correspondent SES. The SES execution information includes power estimation, $WCET$, and area/utilization requirements.

For instance, let us assume an OS service X with two implementations Im_1 and Im_2. For the sake of discussion, we will assume no special encoding, but a one copy redundancy for each SES. This produces an $m + \lambda = 4$ functionally equivalent $SESs$; two with Im_1 and another two with Im_2. To distribute these $SESs$, the GFD algorithm finds a set of fork distribution patters FDP with each pattern having a set of four distinct RSoCs.

The building of the execution routing graph is done step by step each time a set of four functionally equivalent $SESs$ are chosen for distribution over the RSoCs in a fork distribution pattern. This is to allow on-the-fly-building of the distribution records and the execution routing graph. Otherwise, the communication costs and the locations of the alternative $SESs$ and the successor $SESs$ for each SES have to be collected again. For instance, assume that we have distributed the $SESs$ set $S_{i-1} = \{SES_{Im1,i-1}, SES_{Im1,i-1}, SES_{Im2,i-1}, SES_{Im2,i-1}\}$ of the OS service X to the four RSoCs in fdp_{i-1} and we want to distribute the $SESs$ set $S_i = \{SES_{Im1,i}, SES_{Im1,i}, SES_{Im2,i}, SES$
From the fork distribution patterns found using Algorithm 6.1, Algorithm 6.2 selects a fork distribution pattern fdp_i. This pattern is best suited to distribute the $SESs$ in S_i and keep the system balanced in regard to the balancing criterion, see Section 7.2.1.

Algorithm 6.2 then adds four MDEs to the execution routing graph. Each added MDE is a metadata about one SES in the set S_i. Directed edges are added to each MDE representing a SES in the set S_{i-1} to connect it to every MDE representing a SES in the set S_i.

The algorithm then updates the alternative execution record in each SES in the set S_i with the addresses of the RSoCs in fdp_i. It also updates the next execution record in each SES in the set S_{i-1} with the addresses of the RSoCs in fdp_{i-1}. To update the communication edges, the algorithm gathers the communication time between the related RSoCs. This is done by sending the RSoCs in fdp_{i_1} a request to probe each

RSoC in fdp_i for communication time. These communication times are then used to update the edges in the execution routing graph, see Figure 6.3.

Having the execution routing graph built, the algorithm stores it in each $START$ entity of the related OS service and distributes them by choosing another fdp.

Algorithm 6.2 Make Execution Routing Graph

Require: SESs of the OS service X, distribution criterion
1: FDP = get the fork distribution patterns set using Algorithm 6.1.
2: COMM_TIME_SET=NULL
3: $Start$=getStartSES(X)
4: **repeat**
5: S_i = getEncodedSet(X,i) // {get the $m + \lambda$ $SESs$}
6: fdp_i=FDP.getDistributionPattern(S_i, criterion) {get a distribution pattern best suited for $SESs$ in S_i with regard to the distribution criterion}
7: S_i.updateAER(fdp_i) // {Update the "Alternative Execution Record" in the $SESs$ in S_i}
8: **if** i not equal 1 **then**
9: S_{i-1}.updateNER(fdp_i) // {Update the "Next Execution Record" in the $SESs$ in S_{i-1}}
10: COMM_TIME_SET=fdp_{i-1}.GetCommTimeSetOf(fdp_i)
11: **end if**
12: $Start$.makeMDNsAndAddToExecutionRoutingGraph(S_i, fdp_i, i)
13: **until** i is incremented from 1 to n
14: make $m + \lambda$ copies of the $Start$ entity and distribute them using another chosen fpd

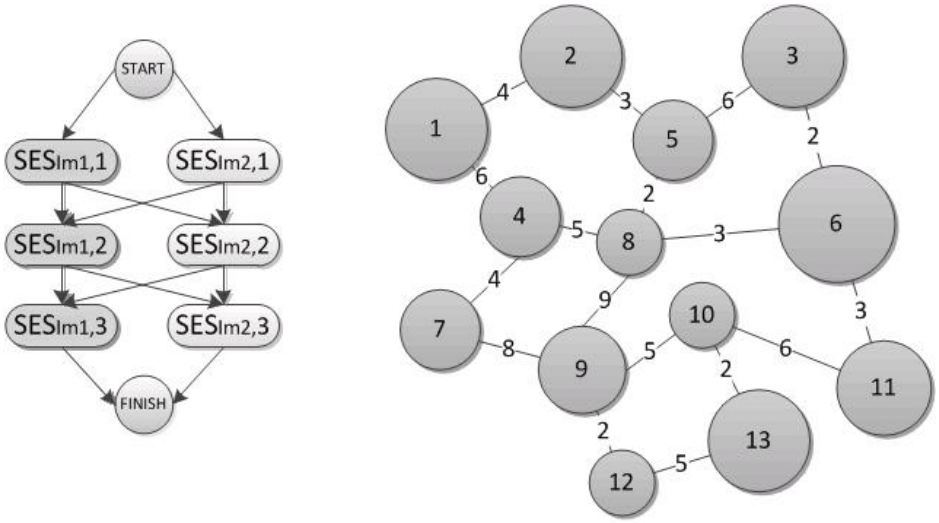
The locations of the $START$ entities can be broadcasted to all the RSoCs. When an RSoC requires an OS service, it acquires the closest $START$ entity of that OS service. The RSoC then uses the algorithms presented in Chapter 5 to identify a suitable configuration for its current resources and constraints.

The recovery of execution routing graphs is possible even if all the $START$ entities of one OS service were somehow lost. This is because each SES is accompanied with the Alternative and Next execution records. By tracing these records, the execution routing graph of one OS service can be rebuilt.

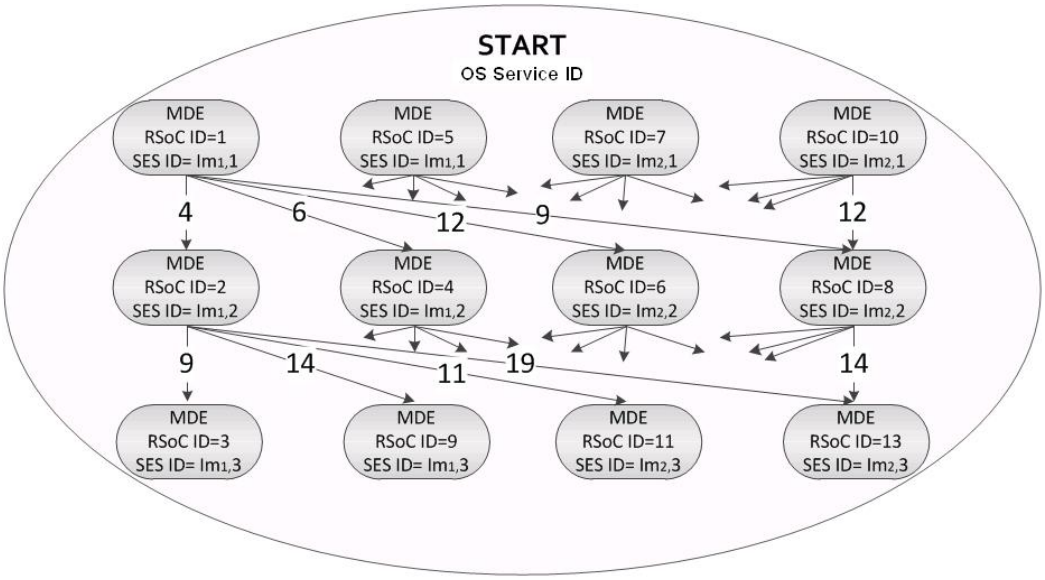
6.3.2 Discovery and execution using self-x routing

In case of random distribution, see Section 6.2, there would be no information on where each SES is located. In such a case only local information can be used. The approach followed is based on the *emergent self–organization* metaphor.

An RSoC which requires the execution of an OS service first needs to find the $SESs$ belonging to that OS service, see Figure 6.4. This is done using an algorithm which was



(a) On the left is a simple OS service with two implementations, each partitioned into three $SESs$. On the right is a network of distributed heterogeneous RSoCs with arbitrary edges denoting communication time.



(b) The execution routing graph of the OS service in 6.3(a) with $m + \lambda = 4$ encoding. Note that each MDE shows only the SES ID and its located RSoC ID, other information, such as the SES execution data, are not shown.

Figure 6.3: An example of an OS service execution routing graph with distribution over heterogeneous RSoCs with arbitrary communication time.

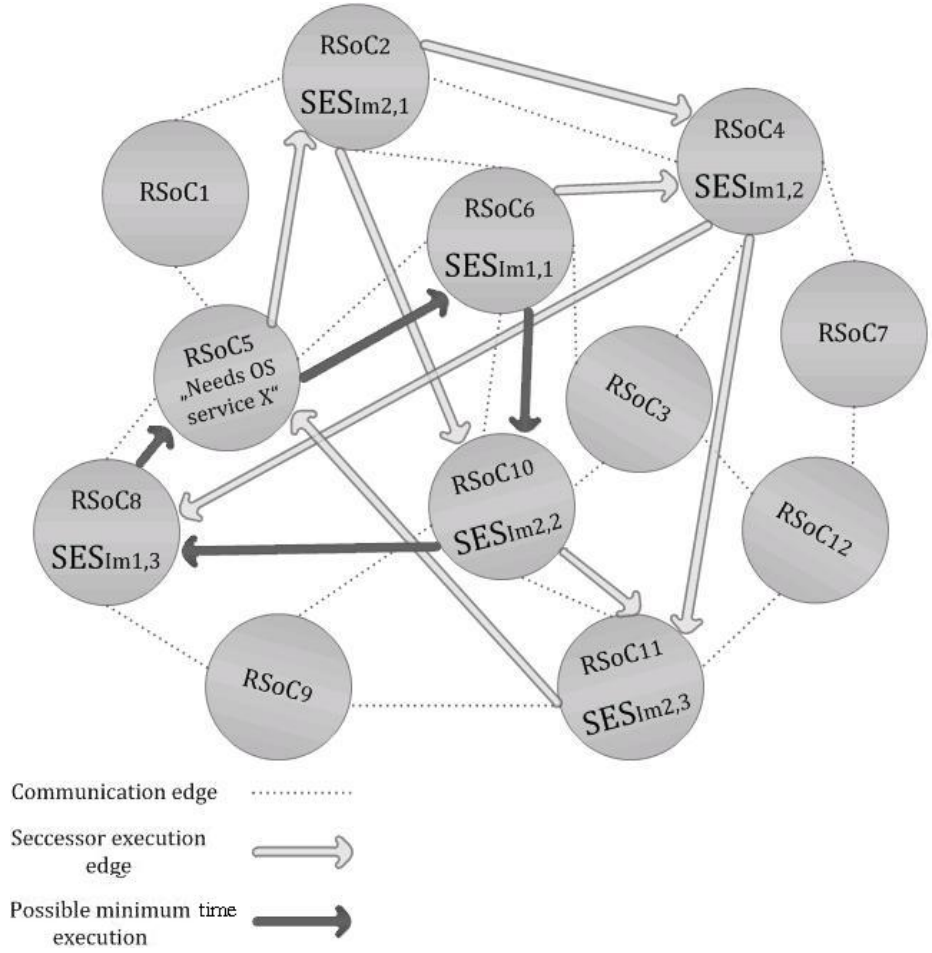


Figure 6.4: An example of finding a desired OS service configuration using only local information

developed with the goal of finding in the network the closest suitable implementation for a given SES. To achieve this goal only local information from the neighbors can be used. This algorithm is based on a hop–by–hop reactive routing algorithm [81] which was inspired by the swarm intelligence behavior of ants communicating by changing their environment [69]. When a food source is found, a chemical substance called *pheromone* is deposited by ants on the paths towards this source. Pheromone can be smelled by other ants and may attract them to the food source. More concretely, during the probabilistic route selection, a route with a higher pheromone value is chosen with a higher probability. Given its physical properties, pheromone evaporates exponentially with time unless a new portion of pheromone is deposited. If a path becomes favored, there will be a higher amount of pheromone deposited contrary to a path which is less favored. Paths which are obsolete vanish as soon as the last amount of pheromone

has evaporated. This particular behavior of ants is useful in terms of two problems which we face in our distributed RSoCs: (i) Finding the shortest path to SES, and (ii) finding a good implementation of the given SES. This is achieved *without* any global knowledge of the topology or central coordination.

In the distributed system, each RSoC stores digital pheromones in a structure called *segment routing table* which has the following format:

> destination segment SES_{id} | next hop ID | implementation im | pheromone $\psi_{SES_{id}}$

Every segment has assigned its identifier SES_{id} which is identical for all instances of the segment, independently of the implementation. With every segment being associated ID of the next hop RSoC and also an amount of pheromone on that path to $\psi_{SES_{id}}$ which has been deposited so far, as well as, the type of implementation which may either be Im_1 or Im_2 if an OS service with two implementations is assumed. Regularly, every time interval τ, values of pheromones in the table are decreased by a pheromone decay coefficient $\psi_\tau \in [0, 1)$:

$$\psi_{SES_{id}}(t + \tau) = \psi_{SES_{id}}(t) \cdot \psi_\tau \tag{6.1}$$

On a given path, if there is no pheromone deposited for a longer period of time, this path becomes less favorable. In order to avoid the creation of loops, each RSoC has to maintain a *recent message table* saving all identifiers (which are unique) of all messages received according to a least-recently-used strategy. The segment discovery algorithm consists of two steps: (i) Availability check of a segment and (ii) migration of data to the found SES.

Availability check

The goal of this step is to check and ensure that there exists at least one path to a requested segment. This is done in two phases: a forward and a backward phase. In the forward phase, links towards the requester are strengthened, or established respectively. The backward phase does the same towards the RSoC with a requested segment. Once an RSoC finishes one SES computation, the RSoC migrates data to another RSoC which is hosting the next subsequent segment. For this reason, the RSoC creates a message called *FDAnt* (forward discovery ant) (Table 6.1).

When an RSoC receives the FDAnt message it proceeds as follows:

Field	Value
Last hop	Requestor ID
Hops	0
Type	FDant
Requestor	ID
Requested SES	SES_{ID}
History	Cleared
Implementation	Im_1 or Im_2

Table 6.1: Fields of FDAnt message

Field	Value
Last hop	SES_{ID} provider
Hops	0
Type	BDant
Requestor	ID
Requested SES	SES_{ID}
History	Cleared
Implementation	Im_1 or Im_2

Table 6.2: Fields of BDAnt message

1. RSoC checks: (i) whether the message exceeds its maximum lifetime, or (ii) the RSoC is already in the *History* field. In those cases, the message is discarded. If the message is contained in the recent messages table, before it is discarded, the message's last hop is recorded in the routing table.

2. If the message is not discarded in the first step, then the following happens:

   - The message is registered in all the relevant tables.
   - If the RSoC contains the required SES, it answers with a BDAnt.
   - The message is propagated using broadcast. Unicast is used if relevant pheromone values are exceptionally high. Moreover, it is used in general for all subsequent messages which transfer payload data between the two endpoints.

A RSoC which can provide the required segment, i.e. SES_{id}, creates a BDAnt (backward discovery ant) message based on the copy of the received FDAnt. In the backward phase, the BDAnt message (Table 6.2) is routed back using links that have been set up by the forward phase. Unicast routing works as follows: A message m, from origin D_{orig} comes from a node with the implementation Im_1, arriving at node i. Before forwarding it further, i alters the corresponding routing table entries:

$$\psi_{D_{orig},Im_1} = \psi_{init} \tag{6.2}$$

if $\psi_{D_{orig},Im_1} < \psi_{init}$, with ψ_{init} being a value used for initialization. Basically, when there is no pheromone or it is below the threshold of ψ_{init}, it is reinitialized to this level. Else, if $\psi_{D_{orig},Im_1} \geq \psi_{init}$,

$$\psi_{D_{orig},Im_1} = \psi_{D_{orig},Im_1} + \psi_{\delta} \tag{6.3}$$

so that a constant amount (ψ_δ) per used link is added to the already existing pheromone level ($\psi_{D_{orig,Im_1}}$), resembling the natural model. Next-hop selection is realized using a probabilistic method. A message heading towards the destination D_{dest}, arriving from a node with the implementation Im_1 at node i is sent to node j, $j \in N_{D_{dest}}$ (i.e. j is in the next-hop table for the destination D_{dest}), with the probability

$$p_{D_{dest,j}} = \frac{\psi_{D_{dest,j}}}{\sum_{k \in N_{D_{dest}}} \psi_{D_{dest,k}}} \tag{6.4}$$

After forwarding a message, the fields $Last\ hop$, $Hops$, and the $History$ are updated. Decay of pheromone is regularly done as described in equation (6.1).

Data migration

After a successful discovery of the required SES_{id} segment, the data is sent using the routing mechanism described above. When data arrives, the hosting RSoC sends back a confirmation to the requesting RSoC about receiving data. The requester waits for confirmation for a certain amount of time. When the timeout expires, the requesting RSoC repeats the segment discovery process in order to find another SES_{id} provider.

6.4 Chapter Conclusion

Situations may exist where there are not enough resources to run an OS service on one RSoC. A solution would be to run the service on another RSoC or to share resources with more than one RSoC in a distributed system. To enable such an option, this chapter has presented heuristic algorithms which allow load balanced distribution of the OS services over the RSoCs.

The distribution algorithms deal with multiple resources with heterogeneous capacities. Two main algorithms were introduced, one deterministic algorithm, namely GFD, and one biologically inspired algorithm. The deterministic algorithm has the advantage of knowing the time cost needed prior to the discovery process. Moreover, it allows the building of records which are used at runtime for fast localization, exploration, and discovery of the suitable configuration for an OS service.

Although the biologically inspired algorithm seams to be not that good, it has a lot of potential, especially because it uses no records and it has more dynamics. However, in order for such an algorithm to work, it has to be initiated before an OS service is being requested. This requires some prediction of the future resources availability

and constraints which may not be always correct. Such an algorithm can be used in a non-realtime distribution system.

CHAPTER 7

Methods Evaluation

This chapter presents the evaluations of the heuristic algorithms introduced in Chapters 5 and 6.

The evaluation is done using the $ShoX$ simulator [94]. $ShoX$ is a special simulator for wireless networks. It can be used for evaluating new network protocols, wireless nodes mobility, signals propagation, nodes energy, and traffic models. In addition, the $ShoX$ can be easily modified to accommodate any special requirement such as modifying the simulated nodes to emulate RSoCs.

Using $ShoX$ as a communication environment, a system of randomly distributed RSoCs is built. The system has a random number, between 60 and 300, of heterogeneous RSoCs. An RSoC has a random amount of limited power, FPGA area, and GPP capability. To support the adaptable OS service requirements, a middleware is developed on every RSoC. The middleware has a control unit, a scheduler unit, and a resource monitoring unit, see Section 3.2.1. These units are required to provide for the novel OS service design which was introduced in Section 4.2.

When necessary, an OSR is made available to the distributed system. An OS is chosen randomly to have between 50 and 150 services. Each OS service has two implementations. These implementations are partitioned into a number n of $SESs$ as described in Section 4.2. The number n of $SESs$ is randomly chosen between 4 and 50 $SESs$ for each OS service. Implementations of different OS services have a different number

of $SESs$. Each SES contains descriptors about its own randomly chosen resource requirements, see Section 8.1.1.

To ensure the evaluations reliability, each evaluation is repeatedly performed with different random systems. This means a new random number of RSoCs with each one of them having its new random resource capabilities, a new random number of OS services with each one of them being partitioned into a new random number of $SESs$, and new demands and constraints.

7.1 Evaluating OS service configurations

In Chapter 5, we presented two main algorithms. The first one, Algorithm 5.2, is used to find a suitable OS service configuration to run on an RSoC with limited resources without violating any constraint. The second one, Algorithm 5.3, is used to find a Pareto optimal OS service configuration. It assumes a requestor RSoC with abundant resources to run any configuration. In this section, these two algorithms are evaluated.

7.1.1 OS service configurations under limited resources

Algorithm 5.2 finds a suitable configuration to execute an OS service on an RSoC that has scarce resources without violating any constraint. For instance, let us assume that a realtime application, residing on an RSoC, requests an OS service X. Suppose that the application requires the OS service to process some data and provides a response in T_{res} time. In such a case, Algorithm 5.2 has to find a suitable OS service configuration that can deliver the processed data without missing the deadline T_{res}. In addition, the algorithm has to take into account the RSoC's current resource status.

In case of abundant RSoC resources, the algorithm has no problem in finding a suitable OS service configuration. However, we want to find the limits where this algorithm fails to provide a suitable OS service configuration. In order to do that, we enforce each RSoC in the distributed system to have resources which are proportional to the correspondent requested OS service optimal resource requirements.

For instance, let us consider the OS service which has its optimal configurations resource requirements presented in Table 7.1. This OS service has configurations which can be executed with an optimal power consumption of 620 power units, an optimal area usage of 555 area units, and an optimal time of 965 time units.

OS service resource requirements		
OS service configuration	Resource requirements	
Area needed to execute the service totally in software area	725	
Area needed to execute the service totally in hardware area	734	
Configuration optimized for area	Area	555
	Power	796
	Execution Time	1294
Configuration optimized for power	Area	743
	Power	620
	Execution Time	1290
Configuration optimized for execution time	Area	725
	Power	728
	Execution Time	965

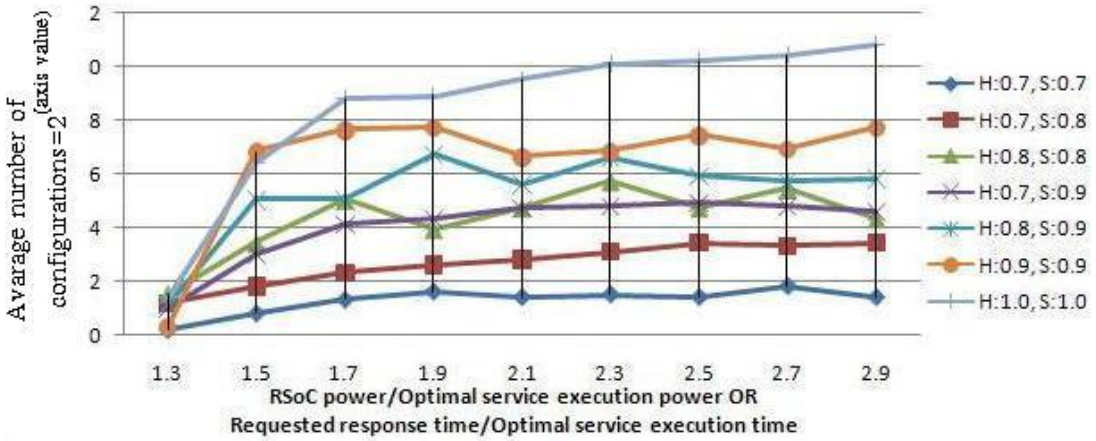
Table 7.1: Arbitrary example of OS service optimal resource requirements

To evaluate our algorithm, we assume that an application, residing on $RSoC_i$, requests this OS service. Then, the power resource value of $RSoC_i$ is forced to be proportional to the power requirement of the requested OS service optimal power configuration which is 620 power units. For example, let us assume that $RSoC_i$ has a power ratio of 1.2 in comparison to the requested OS service optimal power requirement. This makes the RSoC power equal 744 power units. Similarly, the RSoC area is forced to have an amount proportional to the optimal OS service area requirement which is 555 area units. In case of time, we assume that the application demanded the OS service to respond in time T_{res} which is proportional to the OS service optimal execution time which is 965 time units.

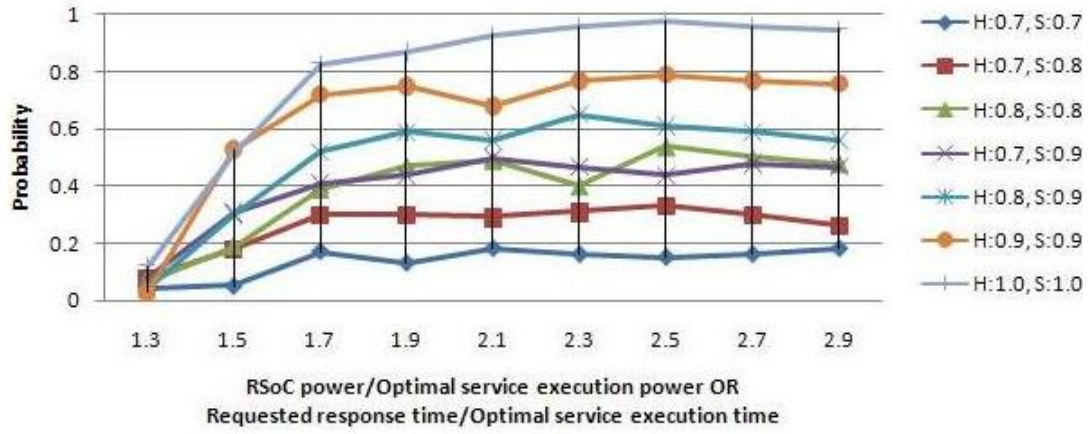
Having these constraints set, Algorithm 5.2 tries to find a suitable OS service configuration which does not violate these constraints. The algorithm is evaluated with different sets of constraints.

To obtain reliable evaluation results, each set of constraints is evaluated with 100 simulation run. Each simulation run has a new randomly generated distributed system and new OS services.

Figure 7.1(a) shows the average number of the obtained OS service suitable configurations for different sets of constraints. The obtained results are averaged over 100 simulation runs. The different enforced power and time ratios are shown by the x-axis. The y-axis shows the exponential part of the number of suitable configurations, where



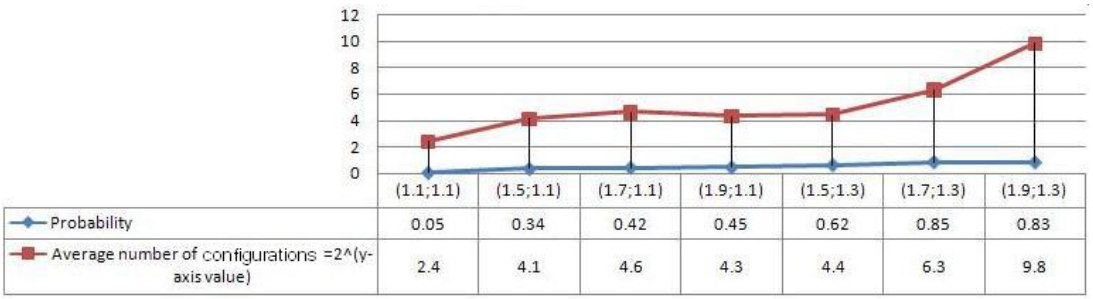
(a) Average number of found suitable OS service configurations for different limited RSoCs resource ratios



(b) The probability of finding the number of suitable OS service configurations in Figure 7.1(a)

Figure 7.1: The evaluation of the found OS service suitable configurations under limited RSoCs resources which are ratio comparable to the required resources of each optimal OS service configuration

the number of suitable configurations equals 2^{y-axis}. The evaluation is carried out with different hardware and software areas for each similar power and time ratios. Figure 7.1(b) shows the probability of getting the number of configurations shown in Figure 7.1(a) for the different sets of constraints. In both charts in Figure 7.1, the time ratio and the power ratio are set to be equal. Figure 7.2 shows the probability of getting a number of suitable configurations for different response time and power ratios. However, the hardware area ratio and the software area ratio are set to be equal at each set of constraints.



	(1.1;1.1)	(1.5;1.1)	(1.7;1.1)	(1.9;1.1)	(1.5;1.3)	(1.7;1.3)	(1.9;1.3)
Probability	0.05	0.34	0.42	0.45	0.62	0.85	0.83
Average number of configurations =2^(y-axis value)	2.4	4.1	4.6	4.3	4.4	6.3	9.8

Figure 7.2: The probability of getting a number of configurations for variable power and response time ratios. The RSoC hardware and software areas are set to to be equal to the optimal OS service required area

Evaluation results

Having 100 different simulation runs for each set of constraints, the algorithm was able to find suitable configurations to execute an OS service with scarce RSoC resources. However, from Figure 7.1 we notice that to get an adequate number of suitable OS service configurations with probability ≥ 0.5, the RSoC has to have an amount of power with a ratio of at least 1.7 of the requested OS service optimal estimated power, 0.8 ratio of hardware and software area to that of the requested OS service optimal area, and 1.7 ratio of demanded response time to that of the requested OS service optimal execution time.

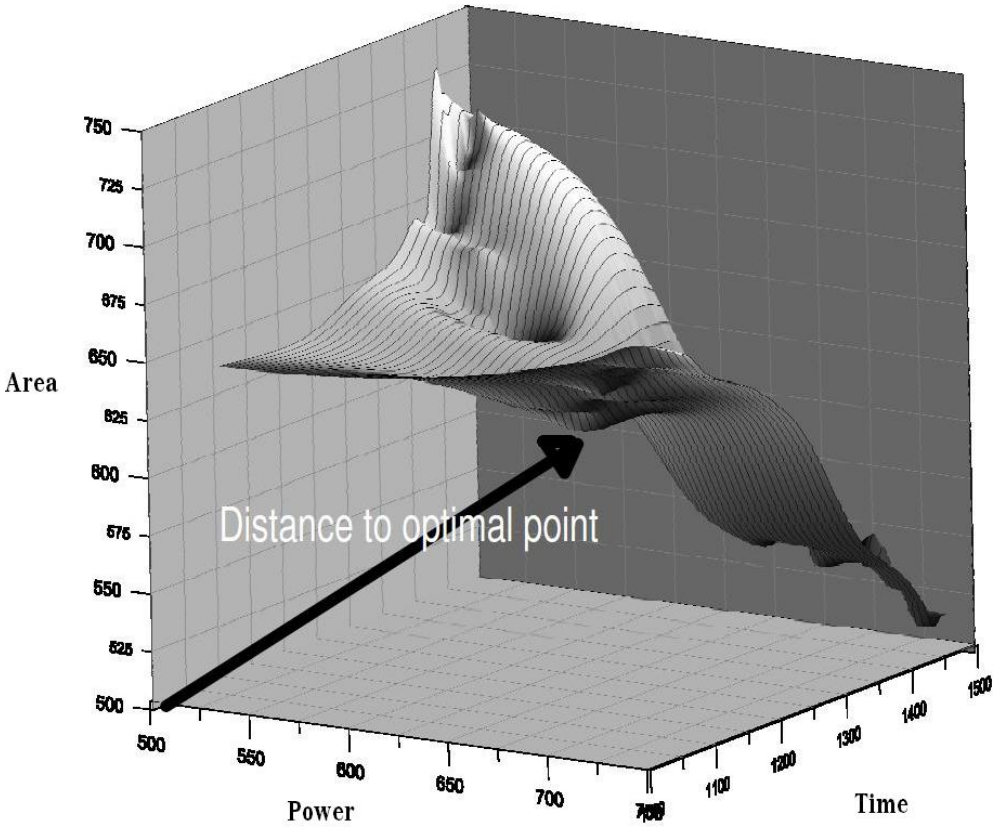
Knowing this, the middleware can decide when to execute an OS service on one RSoC and when to take other measures. These can be sharing resources with other RSoCs or to execute the OS service totally on another RSoC. This result is very important, it helps saving a lot of trial and error calculations. Using the results in this evaluation, a decision can be made depending only on the comparison of the requested OS service optimal values to that of what is currently available of the requester RSoC resources.

7.1.2 OS service Pareto optimal configuration

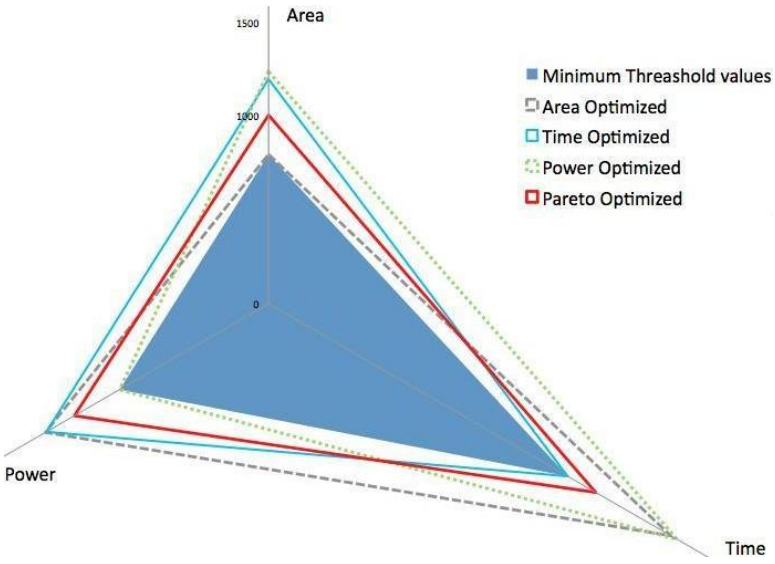
When a requester RSoC has plenty of resources to run any of the requested OS service configurations, our objective becomes finding a Pareto optimal OS service configuration. A configuration which uses as minimal resources as possible, but runs as fast as possible. This is different to what we have in a limited resource environment, as our aim was to find a suitable OS configuration that does not violate any constraint. However, in case of an environment with abundant resources, most of the OS service configurations may happen to be suitable. In such a case, our objective would be finding a configuration which avoids preemption as much as possible. As we do not know which resource may cause preemption, due to application/task demands, we try to minimize the usage of all resources, hence a Pareto optimal OS service configuration. Such a configuration would allocate as much resources as possible to any newly arriving application, therefore, reduces the chance of an application acquiring resources used by the OS service configuration. Hence, minimizing the need of OS service reconfiguration (preemption).

Algorithm 5.3 finds a Pareto optimal OS service configuration. To evaluate this algorithm, we compare the chosen OS service configuration with all the configurations in the search space. Because the search space may be very large, the algorithm limits it by setting maximum and minimum threshold values for each resource, see Section 5.2.3.

There are three resources to be minimized. These are the OS service execution time, the OS service estimated power consumption, and the OS service estimated area usage. The three resources, required by each explored OS service configuration, can be represented as a point in a three dimensional space. Figure 7.3(a), shows a three dimensional chart of the explored OS service configurations. The minimum and maximum values for each resource are shown in Figure 7.3(b). Each explored configuration could be compared with the optimal resources by the minimum threshold as discussed in Section 5.2.3. For example, the three optimal resource values could be considered as an origin point. Each point representing an OS service configuration resource usage is comparable to the other configurations by the distance from that configuration point to the origin point. Figure 7.4 shows the normalized distances of the explored OS service configurations to the OS service original point.



(a) A three dimensional representation of the required resources needed by an OS service explored configurations



(b) The minimum and maximum thresholds which limit the exploration space to find a Pareto optimal OS service configuration

Figure 7.3: An OS service explored configurations and its Pareto optimal configuration

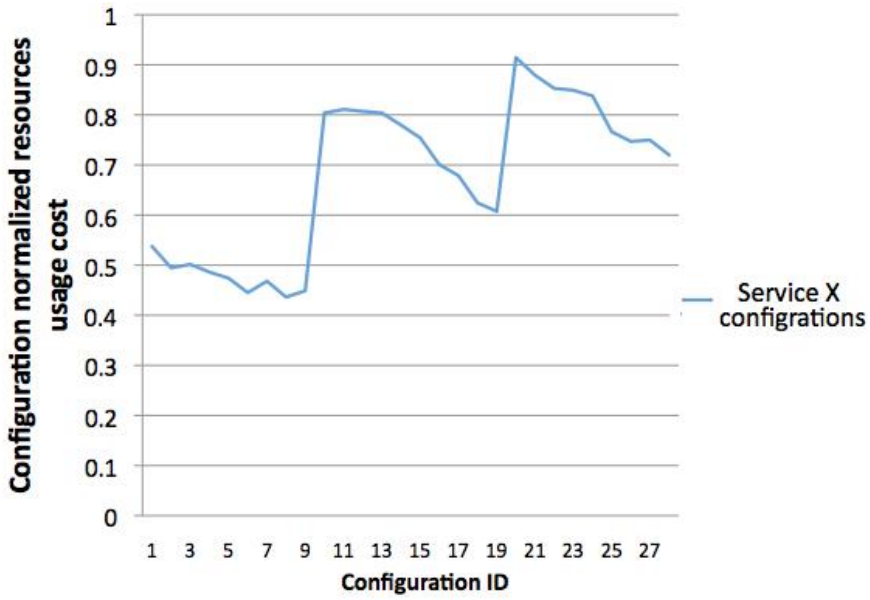


Figure 7.4: The normalized distance between each explored OS service configuration resource requirements and the minimum threshold resource values

Evaluation results

Algorithm 6.1 is able to find a Pareto optimal configuration of an OS service. The evaluation is done repeatedly with many simulation runs. The algorithm always returned a Pareto optimal configuration in comparison with the other configurations. The chosen result shown in this evaluation is randomly selected from these runs. The evaluation is done by comparing each possible configuration resource to the optimal threshold values. A Pareto optimal configuration is chosen from the explored configurations with the property of having its resources as close as possible to the optimal threshold values. Because the algorithm explores all the configurations, the chosen configuration is not a local optimal but a Pareto optimal global configuration.

7.2 OS service distribution

In this section we evaluate the distribution of the heterogeneous $SESs$ over the distributed heterogeneous RSoCs as discussed in Section 6.2.1. During the evaluation, we will assume an encoding that produces $m + \lambda = 4$ $SESs$. For that we use a fork with four fingers to distribute each set of four $SESs$, see Section 6.2.1.

7.2.1 Load balancing over fork distribution patterns

Algorithm 6.1 defines a set of fork distribution patterns FDP which are used for distribution. In our case, each pattern $fdp \in FDP$ consists of four RSoCs. In one pattern, each RSoC has a virtual resource capacity. The virtual capacity determines the maximum amount of $SESs$ which can be allocated to one RSoC resources. All RSoCs belonging to one pattern have the same virtual capacity which is defined by δ. However, RSoCs in different patterns have different virtual capacities. The capacity of a pattern $c(fpd)$ is the sum of its four RSoCs virtual capacities. This means that different patterns have different capacities.

Each fds_i has a resource load indicator $\Re\ell_i$ defined as the ratio between the used resource by the allocated $SESs$ and the total capacity of the resource. Every time, four $SESs$ are distributed to the fdp with the maximum positive value φ which is defined by Equation (7.1). Thereby, $\Re\ell_i$ is the resource load of fdp_i or $RSoC_i$ and $\Re\ell_{avg}$ is the average resource load across the all $fdp \in FDP$ or all RSoCs $\in fdp_i$ as given by Equation (7.2).

$$\varphi = \Re\ell_{avg} - \Re\ell_i \tag{7.1}$$

$$\Re\ell_{avg} = \frac{\sum_{i=1}^{N} \Re\ell_i}{N} \tag{7.2}$$

Figure 7.5 shows the load balancing of distributing $SESs$ over a fork distribution

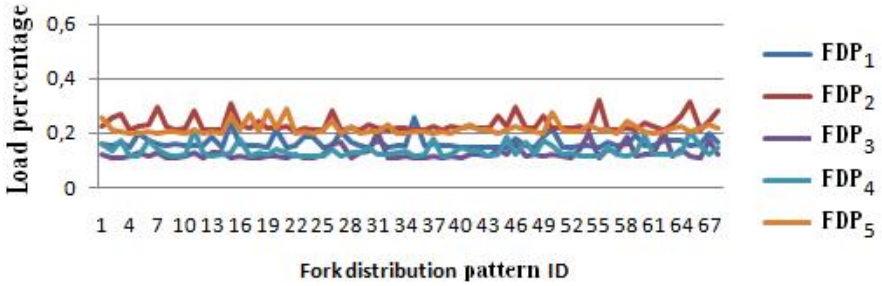


Figure 7.5: Five randomly selected simulation runs showing the distribution load balancing over fork distribution pattern set

pattern set FDP. Although the evaluation shows five randomly simulation runs, the results are similar to all the repeatedly done simulation runs. Each simulation run distributes a set of heterogeneous $SESs$ over heterogeneous RSoCs. These RSoCs are grouped in fork distribution patterns defined by FDP_j. The load balancing criterion was made based on Equation (7.2).

Evaluation results

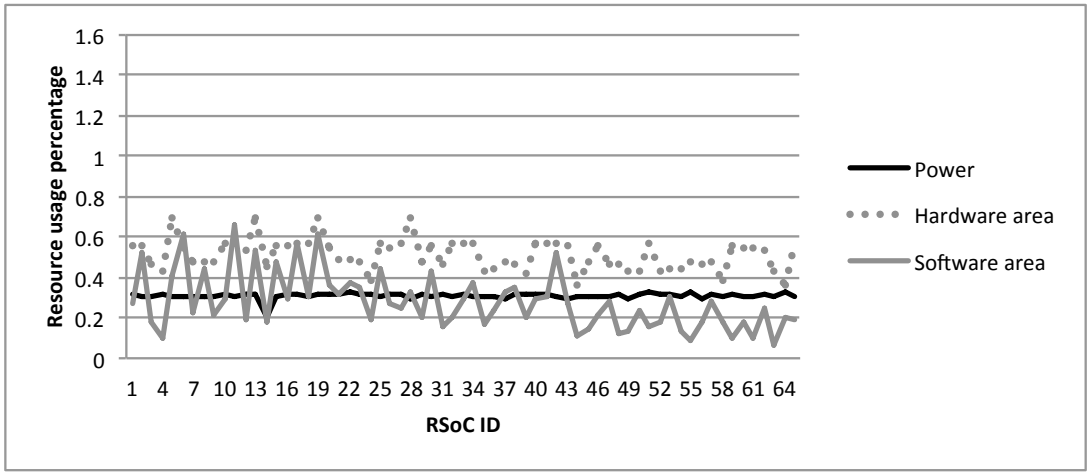
For each fork distribution pattern $fdp_i \in FDP_j$, denoted by the x-axis, the load percentage, denoted by the y-axis, represents the allocated resources capacity for the loaded $SESs$ to the total pattern capacity. As shown in Figure 7.5, the majority of patterns have a similar load percentage. Nevertheless, some patterns have a slightly higher load percentage than the others. This is expected because the distributed $SESs$ are heterogeneous.

The evaluated algorithm results in a load which is well balanced across the majority of patterns with some patterns having an average divination of 6%.

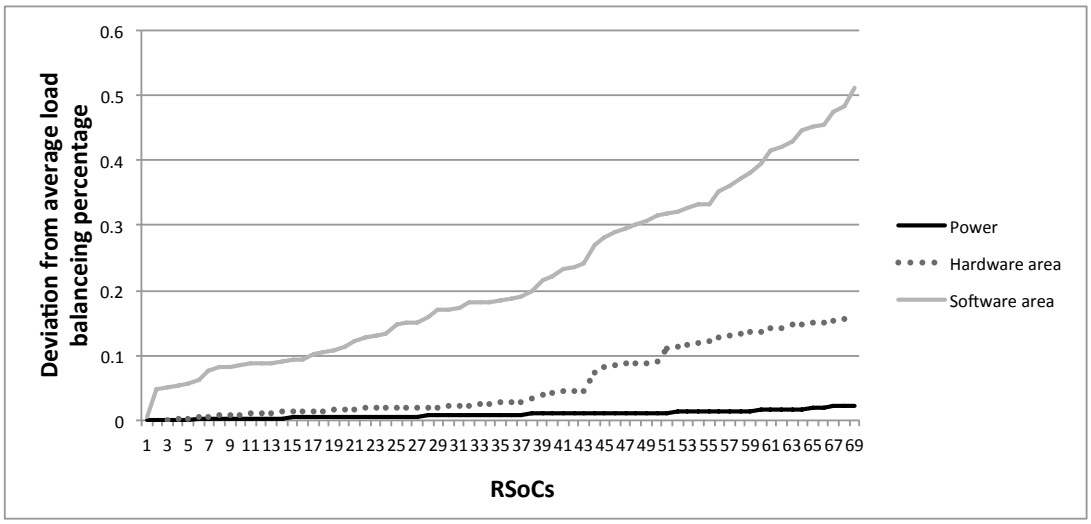
7.2.2 Load balancing over RSoCs

Using Algorithm 6.1, a set of RSoCs may be shared by two or more fork distribution patterns. Each fork distribution pattern containing $RSoC_i$ uses only a partial part of its resources defined by the virtual resource capacity. The distribution is performed by sending a set of $SESs$, containing $m + \lambda = 4$ $SESs$ in our case, to the four RSoCs fork distribution pattern. The choosing of a $SESs$ set, a fork distribution pattern, and the way the $SESs$ are distributed inside a fork distribution pattern depends on our balancing objective. The balancing objective can be concentrated on one resource, like power or hardware area, or generalized to balance with regard to all the resources. Depending on our balancing objectives, many parameters can be tuned to achieve that goal.

Figure 7.6(a) shows the load balance with a load balancing objective concentrated on the power consumption. The load percentage shown by the y-axis is the ratio between the allocated resource amount and the RSoC total resource amount. In this case, the distribution is balanced with regard to power as the highest priority balancing resource. The other resources are also considered for balancing but with low priority. The $SESs$ are distributed over all the RSoCs in a way that makes the power consumption percentage similar on every RSoC. By power percentage we mean the ratio between the allocated power requirements of the $SESs$ on an RSoC and the total power on that RSoC. On the other hand, the hardware and software area usage percentages are not balanced. This can clearly be seen from Figure 7.6(b). The figure shows the deviation of each resource, increasingly sorted, from the average load balance percentage of that resource.



(a) Distributed $SESs$ over RSoCs with power consumption as the priority resource in load balancing



(b) The deviation of resources from the average load percentage when distributing with power resource is chosen as the priority resource for load balancing in Figure 7.6(a)

Figure 7.6: $SESs$ load balancing distribution with one prioritized resource (e.g. Power) and the deviation of each RSoCs load from the average load balance percentage

If we are considering the three resources to be balanced, a weighting function, such as the one shown in (7.3), can be used.

$$W = \sqrt{\rho * (P_{max} - p_x)^2 + \beta * (Hx_{max} - h_x)^2 + \alpha * (Sx_{max} - s_x)^2} \quad (7.3)$$

Here P_{max} is the maximum power consumption allowed across all entities of the same type, p_x is the power consumption for one entity, H_{max} and S_{max} are the maximum total Hardware and Software areas across all entities of the same type, h_x and s_x are the total hardware and software areas for one entity, $\{\rho,\ \beta,\ and\ \alpha\}$ are heuristic real numbers with each to be between 0 and 1. These numbers reflect the priority of one resource in comparison with the others. An entity type can be a SES, an RSoC, or a fork distribution pattern.

Figure 7.7(a) shows the resource load percentage in every RSoC when load balancing is performed using the above weighting function as a cost function. In Figure 7.7(b), the deviation of each RSoC load percentage from the average load percentage is presented. The deviations are sorted increasingly.

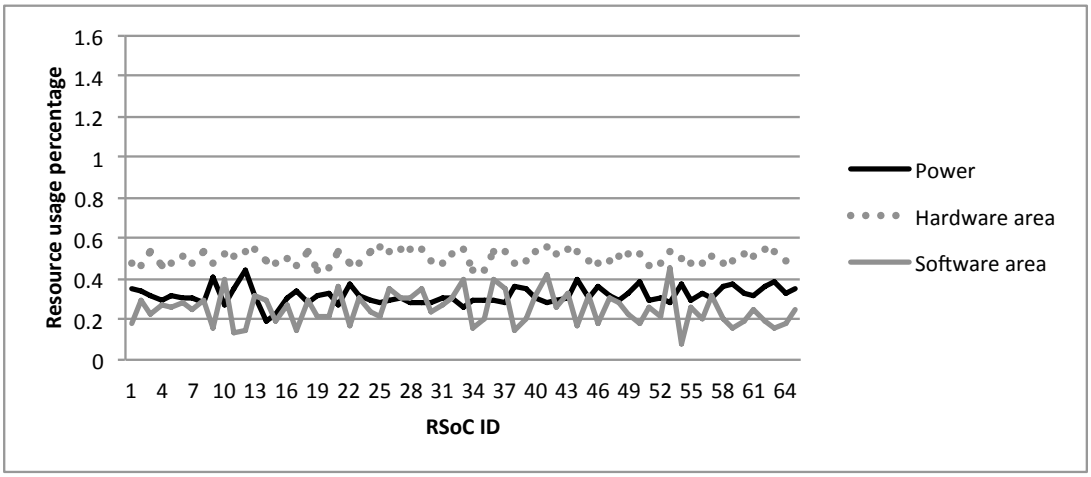
Evaluation results

As seen in Figure 7.6(a), when considering a single priority resource (e.g. power consumption) for balancing, the algorithm successfully balances the resource load allocated for $SESs$ in all the RSoCs. This single priority resource load balancing is very close to optimal. It has a low average deviation of 1% from optimal, see Figure 7.6(b). Nevertheless, the deviation of the other resources differ very much from optimal, with an average deviation from 10% to 30%.

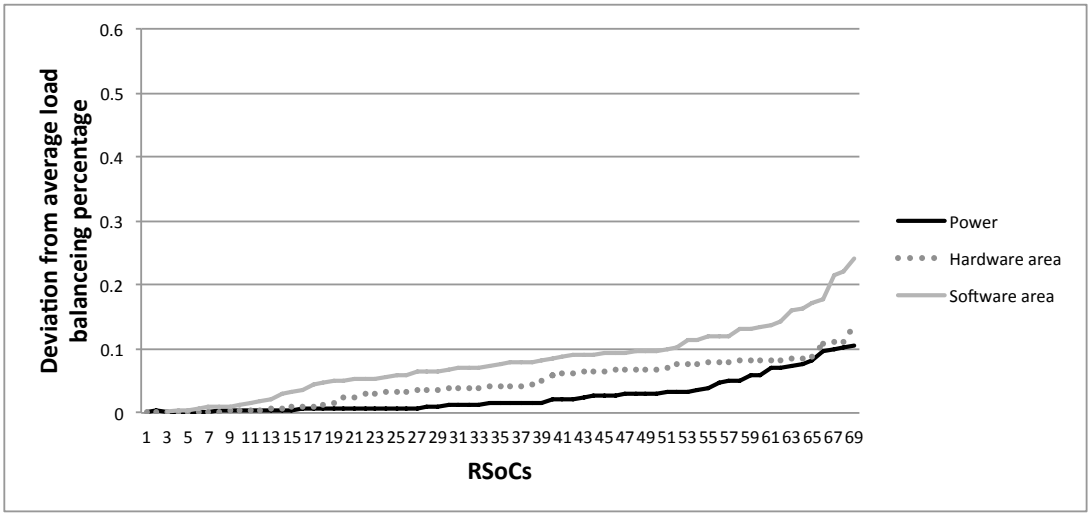
On the other hand, the capacity function (7.3) can be used as a weighting method to balance the three resources over all RSoCs, see Figure 7.7(a). This results in an average deviation from optimal load balancing which varies from 3% to 9% for all the resources.

The choosing of the balancing criterion depends on the system application domain. In any way, the evaluated algorithm can be used to efficiently balance the $SESs$ with little deviation from optimal. At this point, it is important to keep in mind that we are distributing $SESs$ with heterogeneous resource requirements on heterogeneous RSoCs. In many ways, this is very similar to the Knapsack problem which is known to be an NP-hard problem.

The shown results are chosen randomly from many simulation runs. They reflect an average case of load balancing. All the other simulation runs have produced similar, if not better, evaluation results.



(a) Distributed $SESs$ over RSoCs taking into mind the load balancing of all resources



(b) The deviation of each RSoC load from the average load balancing percentage after load balancing as in Figure 7.7(a)

Figure 7.7: $SESs$ load balancing distribution considering all resource and the deviation of each RSoCs load from the average load balance percentage

7.3 Chapter conclusion

The algorithms presented in the previous chapter have been evaluated. The evaluation was carried out using the ShoX simulator. Interesting results have been obtained. Such results help deciding, in early stages, whether to execute an OS service on the requester RSoC, on another RSoC, or on a collaborative RSoCs.

The algorithms used in distributing the heterogeneous OS services over the heterogeneous capacity RSoCs indeed did show very interesting evaluation results. Depending on whether one criterion or multiple criterions are chosen, the algorithms efficiently balance the load of the distributed OS service over the RSoCs.

CHAPTER 8

Case study

In this chapter, we describe the implementation of an encryption/decryption OS service, see Section 8.2. This is provided as a proof of concept to what previously has been presented. For the underlying support, a middleware was developed to synchronize the $SESs$ execution, providing for inter $SESs$ communications, and running the developed algorithms. For managing the underlying computational elements (FPGA/GPP) we have selected the ReconOS run-time environment.

8.1 Reconos

ReconOS [97, 99, 100] supports hardware and software co-design by introducing a uniform thread model for software and hardware. This is done by modifying an $RTOS$, (e.g. eCos [44, 106]), to integrate hardware interface components called the OS Interface ($OSIF$). $OSIF$ allows hardware modules to interact with the operating system in a way similar to software threads. By doing this, ReconOS provides a layer where threads communication can be synchronized across the hardware/software boundary. Whenever a hardware thread is constructed, its user logic is connected to one $OSIF$. This $OSIF$ manages the interaction with the operating system.

The uniform thread model provides a relatively easy hardware/software co-design development. It allows a transparent interaction between software and hardware. The

transparency is done to a level where a thread has no information of whether the other correspondent thread is in hardware or in software. To achieve this transparency, every hardware module has a delegate thread in software. This delegate shows the hardware module as if it would be a software thread, see Figure 8.1. In this way, neither hardware threads nor software threads are preferred. Moreover, this allows the usage of any single processor schedular such as Earliest Deadline First (EDF), or Rate-monotonic scheduling.

Although this seems a good choice, the benefits obtained from hardware are ignored at scheduling time. Moreover, there is no possibility for resource management in terms of choosing types of threads for best performance, minimum resource usage, or efficient parallelism utilization.

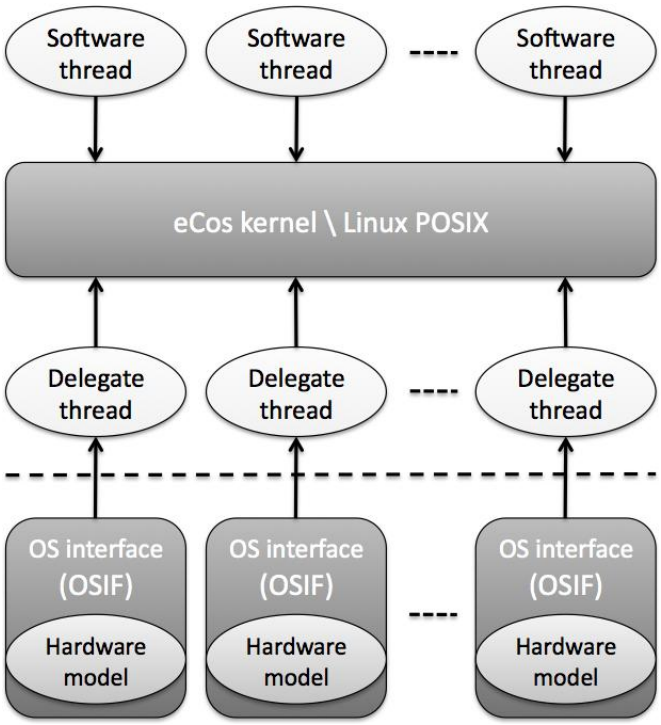


Figure 8.1: The eCos kernel can not differentiate between software and hardware threads. ReconOS uses delegates to allow for unified kernel thread handling.

8.1.1 $SESs$ support

The triple DES case study, see Section 8.2, has been implemented based on the ReconOS library. To allow for the desired $SESs$ execution management, see Section 4, a

simple middleware has been developed. The role of the middleware is to manage and to schedule the $SESs$ and the inter $SESs$ communication. The ReconOS execution behavior has been modified to allow the middleware to take control of scheduling and communication management.

8.1.2 $SESs$ implementation

In this case study, $SESs$ are implemented either in software or in hardware. To enable the inter $SESs$ communication, a uniform interface using the eCos and ReconOS application programming interface (API) has been adopted. The other specific parameters are passed to each SES using memory.

The software implemented $SESs$ are created as thread functions using the eCos API.

```
cyg_thread_create( 16,              // priority, not important
        SES_s1_entry,               // entry point
        ( cyg_addrword_t ) 1,       // entry data
        "SES_s1_ENC",               // SES name
        SES_s1_stack,               // stack
        STACK_SIZE,                 // stack size
        &SES_s1_encryptor_handle,   // SES handle
        &SES_s1_encryptor           // SES object
    );
```

The hardware implemented $SESs$ are created as ReconOS hardware delegate using the ReconOS API.

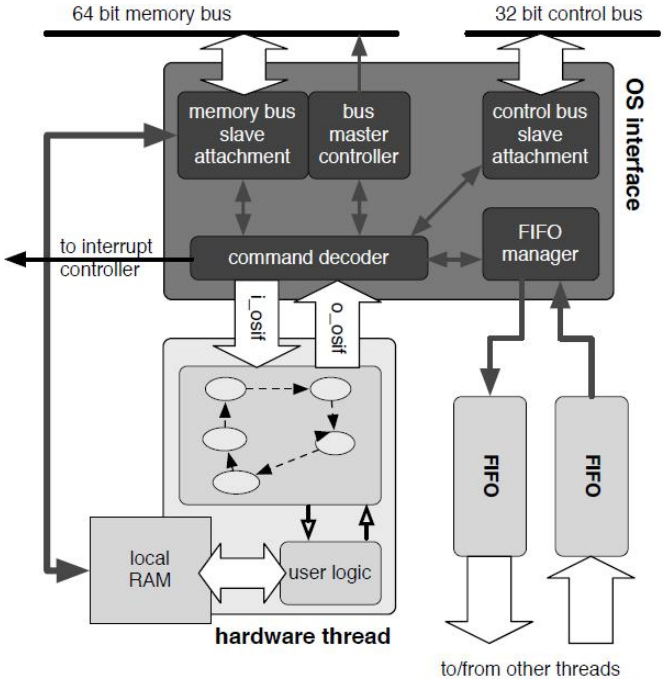
```
reconos_hwthread_create( 16,        // priority, not important
         &SES_h1_encryptor_attr,    // hardware SES attributes
         0,                         // entry data (not needed)
         "SES_h1_ENC",              // SES name
         SES_h1_encryptor_stack,    // stack
         STACK_SIZE,                // stack size
         &SES_h1_encryptor_handle,  // SES handle
         &SES_h1_encryptor          // SES object
     );
```

Each SES waits the $START$ flag, which is triggered by the middleware, to begin processing a memory data chunk which has its pointer sent along the $START$ flag. All the specific parameters are passed as a record at the beginning of the data chunk to be processed. When finishing the execution, a SES sends a $DONE$ done flag. The processed data is saved in the memory data chunk. Note that the priority parameter used in both APIs is not important. This is because the initiation and scheduling of each SES is managed by the middleware.

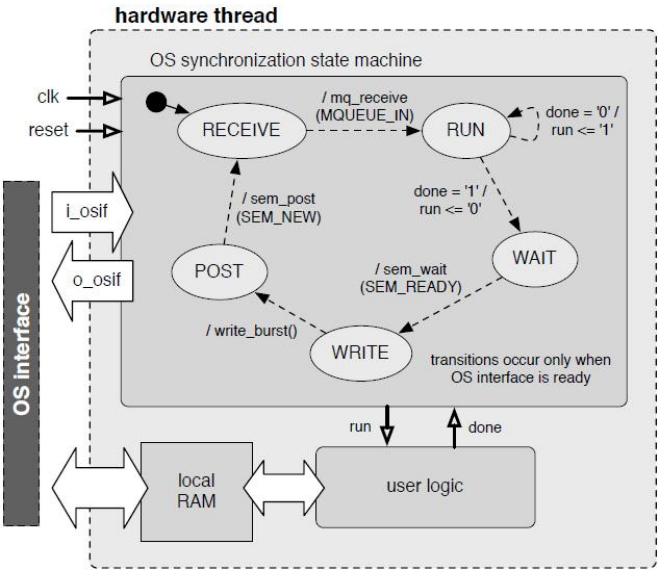
In case of hardware implemented $SESs$, when a ReconOS delegate receives the $START$ flag, it sends a message to the corresponding $OSIF$ to unblock the hardware module. The $OSIF$ consists of several modules such as the $OSIF$ core, a master bus controller and a FIFO manager, see Figure 8.2(a). On the other hand, if the hardware module needs to communicate with the delegate, it will send a command to the $OSIF$. This command is dealt with by the command decoder. The command decoder is a large state machine, with one state for each available command.

In order for hardware implemented $SESs$ to communicate with $OSIF$, each SES hardware module has to be encapsulated within a synchronization state machine, see Figure 8.2(b). The state machine defines five states as shown in the code below. These states are used by $OSIF$ to control the SES hardware module.

```
--other code
encryptor_j: SES_h1_hardware_model
 generic map(
   --generic map
 )
 Port map(
   clk => clk,
   reset=>reset,
   o_RAMAddr =>RAMAddr,      --memory address
   o_RAMData =>o_RAMData,    --memory write data
   i_RAMData =>i_RAMData,    --memory read data
   o_RAMWE => o_RAMWE,       --memory Write/Read
   start =>SES_start,        --start SES processing
   done => SES_done
 };
 -- other code
 --OS synchronization state machine
 state_proc: process(clk, reset)
 -- define any required variables
 begin
```



(a) The OSIF and its modules. The command decoder manages all hardware thread interactions and relays data to the individual bus attachments, external FIFOs, or the CPU.



(b) Example of an OS synchronization state machine. The state transitions are executed under OSIF control, allowing individual OS calls (e.g., sem_wait()) to block. The actual processing of the thread occurs in the user logic

Figure 8.2: The schematic view of OSIF and the synchronization state machine in ReconOS [98]

```
    if reset='1' then
       -- reset reconos and any other signals
    elsif rising_edge(clk) then
       reconos_begin(o_osif, i_osif);
       if reconos_ready(i_osif) then
          case state is
             when STATE_GET=>
               --wait for any message from the delegate
               --if yes go to STATE_READ
             when STATE_READ=>
               --read the memory data chunk to be processed
               --then go to STATE_BEGIN_SES_PROCESSING
             when STATE_BEGIN_SES_PROCESSING=>
               --start SES processing then go to STATE_WAIT
               SES_start<='1';
             when STATE_WAIT=>
               --wait until the SES sets the SES_done flag
               --then go to STATE_WRITE
             when STATE_WRITE=>
               --write the processed data back to memory
               --then go to STATE_PUT;
             when STATE_PUT=>
               --send the DONE message to the delegate
               --then go to STATE_GET;
             when others=>
               -- go to STATE_GET;
             end case;
       end if;
    end if;
  end process;
```

The communication between the hardware module and the delegate can be done either by shared memory or mailboxes. Both communication ways need the bus for communication, see Figure 8.2(a). This may become a bottleneck as all $OSIF$s share the same bus. Another offered possibility is communication via FIFO. This connects every hardware thread to its "right" neighbor. This, however, is only useful if the SES on the right is the immediate successor of the current SES.

8.2 Triple DES

The triple Data Encryption Standard ($3DES$) [80] is an encryption algorithm which is used in many applications such as the electronic payment industry [47]. As a proof of concept, we will assume an OS service which provides the 3DES encryption. The triple DES can be developed as three DES successive blocks [11], each with a different 64-bit key. The data is first encrypted using the first key, decrypted with a second key, and finally encrypted using a third key. The encryption algorithm is $cipher = EK3(DK2(EK1(data)))$.

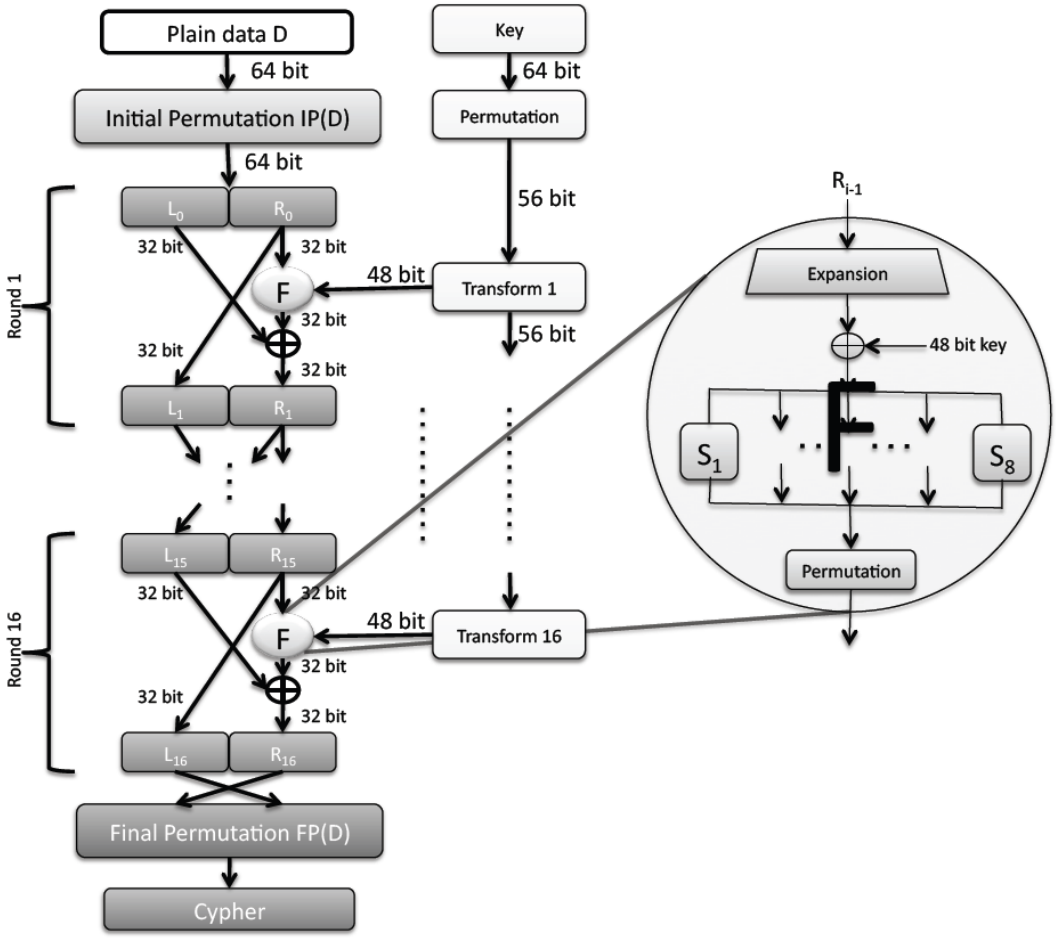


Figure 8.3: The block diagram of the DES encryption algorithm

The DES block diagram is shown in 8.3. It consists of an initial permutation, sixteen main rounds, and a final permutation. The 64-bit key is permutated and transformed to give a new 48-bit key to each of the sixteen main DES rounds. The DES block can be

used for encryption and decryption. The only difference between the two operations is the scheduling of the 48-bit keys. In encryption operation, the first transformed 48-bit key is given to the first round, the second key to the second round, and so on. In the decryption operation, this scheduling is reversed. The first key is given to the sixteenth round, the second key to the fifteenth round, and so on.

As a proof of concept, the 3DES was implemented in both software (runs on PowerPC GPP) and hardware (runs on FPGA). Each implementation is partitioned into six $SESs$ as shown in Figure 8.4. The partitioning is made as described in Section 4.2. This gives us a total number of $2^6 = 64$ configurations. As an underlying platform, the Xilinx Virtex^{TM}II Pro kit was used. It consists of one FPGA containing inside it one dedicated GPP core (PowerPC).

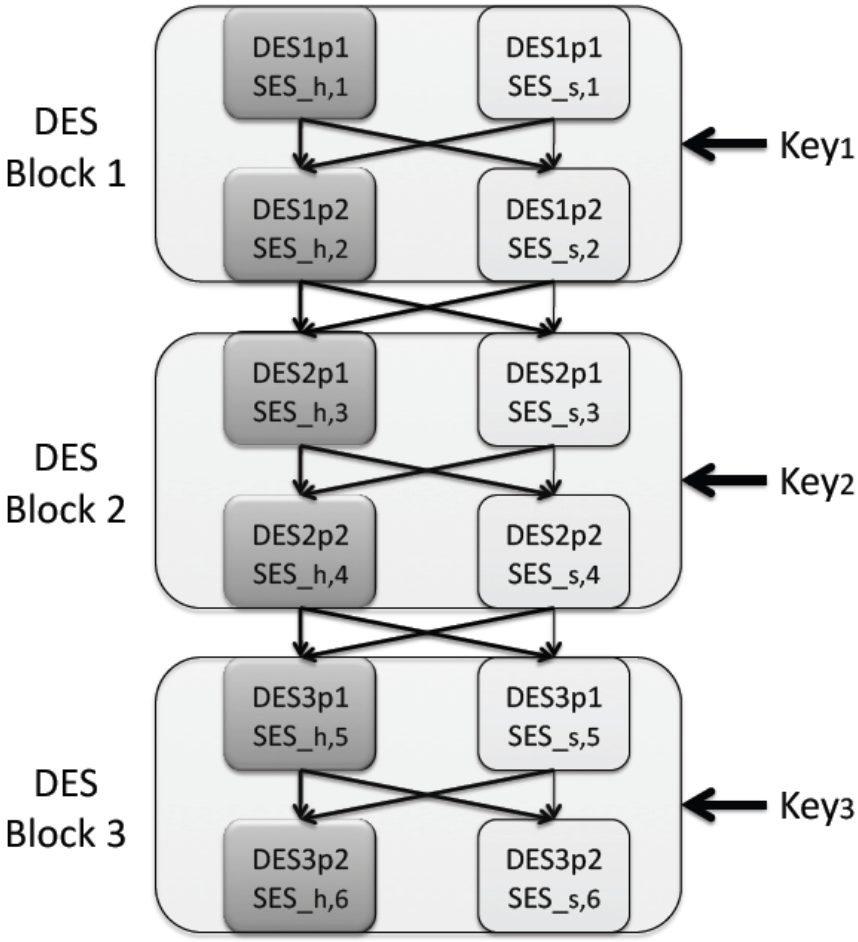


Figure 8.4: The 3DES OS service with two implementations, each partitioned into six $SESs$

Communication type	Time
$SES_{HW} \rightarrow$ Middleware	187 μs
$SES_{SW} \rightarrow$ Middleware	68 μs
Middleware $\rightarrow SES_{HW}$	193 μs
Middleware $\rightarrow SES_{SW}$	76 μs
$SES_{HW} \rightarrow SES_{SW}$	304 μs
$SES_{SW} \rightarrow SES_{HW}$	302 μs
$SES_{HW} \rightarrow SES_{HW}$	422 μs
$SES_{SW} \rightarrow SES_{SW}$	184 μs

Table 8.1: Inter communication overhead

8.3 Results

As shown in Figure 8.4, each DES block is partitioned into two $SESs$. This means that one DES block can be executed in $2^2 = 4$ ways or configurations, two successive DES blocks in $2^4 = 16$ ways, and three successive DES blocks (the $3DES$) in $2^6 = 64$ ways. These configurations can be explored at run time using the algorithms of Chapter 5. The chosen configuration depends on the constraints and criteria of the requester, see Section 5.2.4. Concentrating on one DES block, we have four execution configurations. These are: First and second $SESs$ in hardware (h-h), first and second $SESs$ in software (s-s), first SES in software and second SES in hardware (s-h), or first SES in hardware and second SES in software (h-s).

Table 8.1 shows the average inter-communication overhead for these configurations. The inter-communication overhead is maximal when two $SESs$ are implemented in different computational elements (GPP or FPGA). However, in Table 8.1 we notice that the inter communication overhead between two $SESs$ in hardware is more than that of two $SESs$ in different architectures. This is no surprise because the hardware $SESs$ are communicating through the middleware which is implemented in software. The $SES_{HW} \rightarrow SES_{HW}$ is actually $SES_{HW} \rightarrow Software(middleware) \rightarrow SES_{HW}$. This can be improved by either allowing two $SESs$ in one implementation to communicate directly but with a notification to the middleware or by implementing parts of the middleware in different computational elements (e.g. in hardware and software) to allow for fast communication.

The area allocated for each block is shown in Table 8.2. The allocated areas shown include the area allocated for interfaces as well as the SES itself, (e.g. the OSIF interface for hardware implemented $SESs$).

Hardware	LUT	Utilization
DES_SES_1_HW	4008	10%
DES_SES_2_HW	2815	14%
Software	**Memory utilization (bytes)**	
DES_SES_1_SW	8127	
DES_SES_2_SW	6417	

Table 8.2: $SESs$ FPGA and memory utilization

This case study is further used to evaluate the use of pipelining as explained in Section 5.3.2. The middleware is used to explore the four execution configurations of the DES block with and without pipelining. Figure 8.5 shows the execution time for the four DES execution configurations when processing nine memory data chunks. Each data chunk consists of seventeen 64-bits words. Both configurations are executed with and without the use of pipelining.

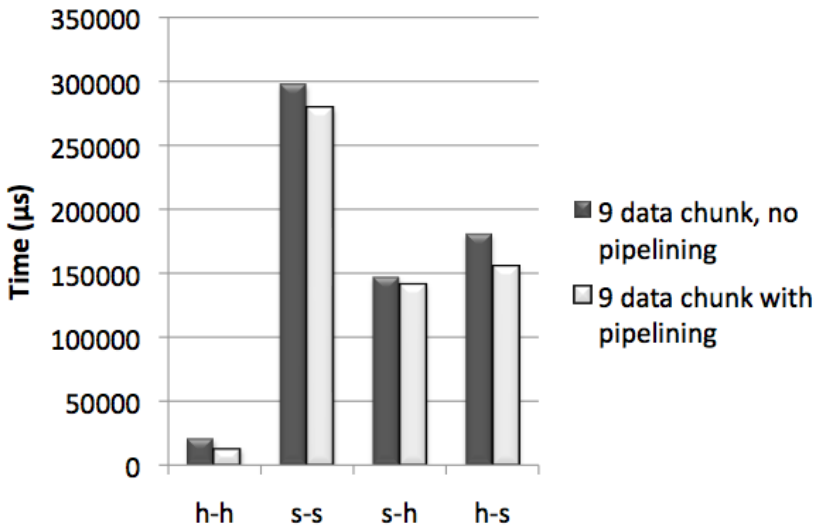


Figure 8.5: A comparison between a non-pipelined execution of the four DES execution configurations and a pipelined execution.

Using pipelining we have been able to improve the execution time of the configurations with a maximum improvement percentage of 37% in the (h-h) configuration and a minimum improvement percentage of 6% in the (s-h) configuration.

The improvement percentage gained by using the pipelining depends on two factors: the level of the parallelism and the difference in execution time between $SESs$. In architectures such as an FPGA, all the hardware implemented $SESs$ can run in parallel without any delay imposed by other applications/tasks running on FPGA. In a

pipelined execution of the (h-h) configuration, the second SES will process the first data chunk while the first SES is processing the second data chunk. The execution of the two $SESs$ is completely parallel. The only delay comes from the fact that the second SES can only process data that has already been processed by the first SES. This delay depends on the execution time of first SES and the communication time between the two $SESs$. From this we can obtain a maximum improvement in execution time using pipelining if the first SES execution time plus communication time equals the second SES execution time. In general, a maximum reduction in a partitioned OS service's execution time can be obtained using pipelining if the $SESs$ of the chosen execution configuration have the ability to execute in parallel on different data chunks and

$$\forall SESs \in configuration,$$

$$T(SES_i) + CT(SES_i, SES_j) = T(SES_j),\ where$$

$T(SES_x)$ is the execution time of SES_x, $CT(SES_x, SES_y)$ is the communication time between the two successive $SESs$, SES_x and SES_y, and $j = i + 1,\ j < n$.

Due to the heterogeneity in execution time between $SESs$ of the DES block we have obtained a maximum reduction in execution time using pipelining in the (h-h) configuration. On the other hand we find that the (h-s) configuration has the maximum reduction percentage in execution time. This is because of both $SESs$ can execute in parallel, and because the execution time of the first SES (hardware implemented) is less than half that of the software one. This allows the processing of two data chunks in the first SES before the second SES finishes one data chunk.

Looking at the (s-h) configuration we notice that it suffers from the minimum reduction in execution time when using pipelining. Although both $SESs$ can execute in parallel, the SES execution time limits any improvements. The first SES executing in software requires time which nearly triples that one of the hardware SES. Because we can not process any data chunk in the second SES unless it has already been processed by the first SES, the pipelining technique shows little improvement.

Comparing the non-partitioned DES with the partitioned DES that uses no pipelining, see Figure 8.6, we get the expected result of a non-partitioned DES being faster in both hardware and software implementations than the partitioned non-pipelined DES due to the added communication overhead. However, the pipelined partitioned DES requires less execution time to process the data chunks compared to the non-partitioned DES in both software and hardware implementations, see Figure 8.7.

Whether pipelining is used or not, or whether we have improvements in execution time or not, the fact remains that partitioning gives us the flexibility in both execution time

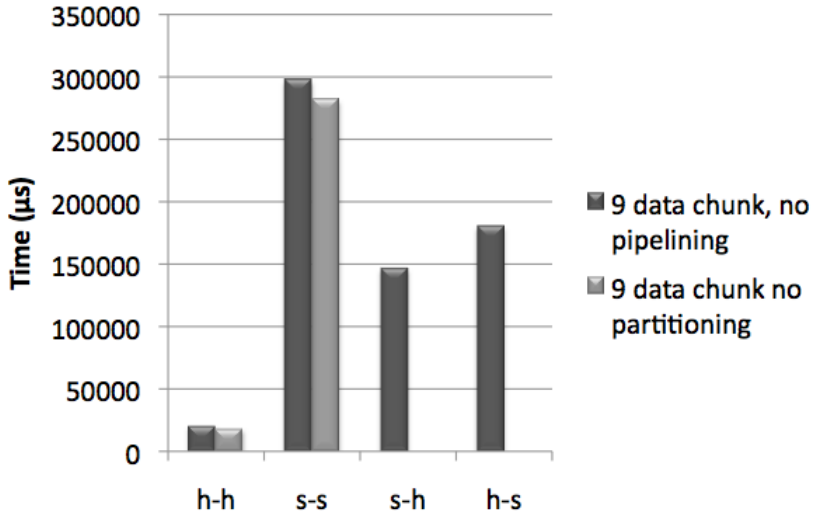


Figure 8.6: A comparison between a partitioned non-pipelined execution of the four DES execution configurations and a DES non-partitioned execution.

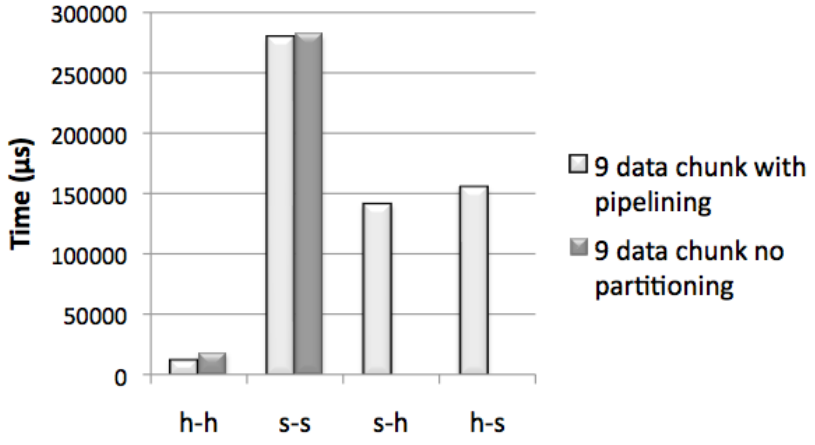


Figure 8.7: A comparison between a partitioned pipelined execution of the four DES execution configurations and a DES non-partitioned execution.

and resource usage. Combined with pipelining, this will give a worst case scenario that has an execution time which is better than that of a non-partitioned service.

8.4 Chapter conclusion

OS services designed for distributed embedded systems can be partitioned and implemented to run on different computational elements. This enables many useful characteristics such as runtime adaptation to the dynamic change in resources and demands. It also introduces an undesired inter communication overhead.

This chapter has presented the usage of the pipelining technique to utilize the parallelism offered by RSoCs embedded systems in favor of reducing the effect of the inter communication overhead. Depending on the level of parallelism and the heterogeneity of the partitions, this may improve the WCET of the service by roughly 40%.

The 3DES encryption standard was introduced as a proof of concept. It is implemented in hardware and software with each implementation partitioned into six $SESs$. This case study shows the different possible configurations to run the $3DES$ and the improvements gained by using the pipelining technique.

The middleware used in the case study was implemented completely in software and on top of ReconOS [100]. This increased the communication overhead as it has been carried out completely through the middleware. To overcome this problem, parts of the middleware could be implemented in hardware. Moreover, successive $SESs$ implemented in the same computational elements could be allowed to communicate directly with notifications sent to the middleware.

CHAPTER 9

Conclusion and future directions

In this chapter, we summarize and conclude the work presented in this thesis. In addition, we outline the directions and the future intended work.

9.1 A brief summary

In a distributed RSoCs system, resources are limited and applications vary. In such an environment, OS services can be designed to adapt themselves by changing their resource requirements at run time. This is achieved by multiple implementations and partitioning into $SESs$, see Section 4.2. This produces a number of possible configurations to run each OS service. An application residing on one RSoC can request an OS service. Depending on the requester's current resources and constraints, an OS service configuration can be chosen. The process is performed using specialized heuristic algorithms, see Section 5.2. Using these algorithms, a suitable OS service configuration can be found even with relatively scarce resources, see Section 7.1. Based on the evaluation and the presented analysis, it is possible to know, even before running the algorithms, whether we can find a configuration for executing on one RSoC or whether a sharing of resources between RSoCs is required.

In case of being able to execute on the requester RSoC, $SESs$ of the chosen configuration will be migrated to that RSoC and the OS service execution is commenced.

In case of distributed execution, the algorithms find the collection of RSoCs to execute within constraints, see Section 6.3. Having an RSoCs collaboration with suitable configuration found, data will be migrated to begin the execution of the selected OS service configuration. This process can briefly be summarized by Figure 9.1.

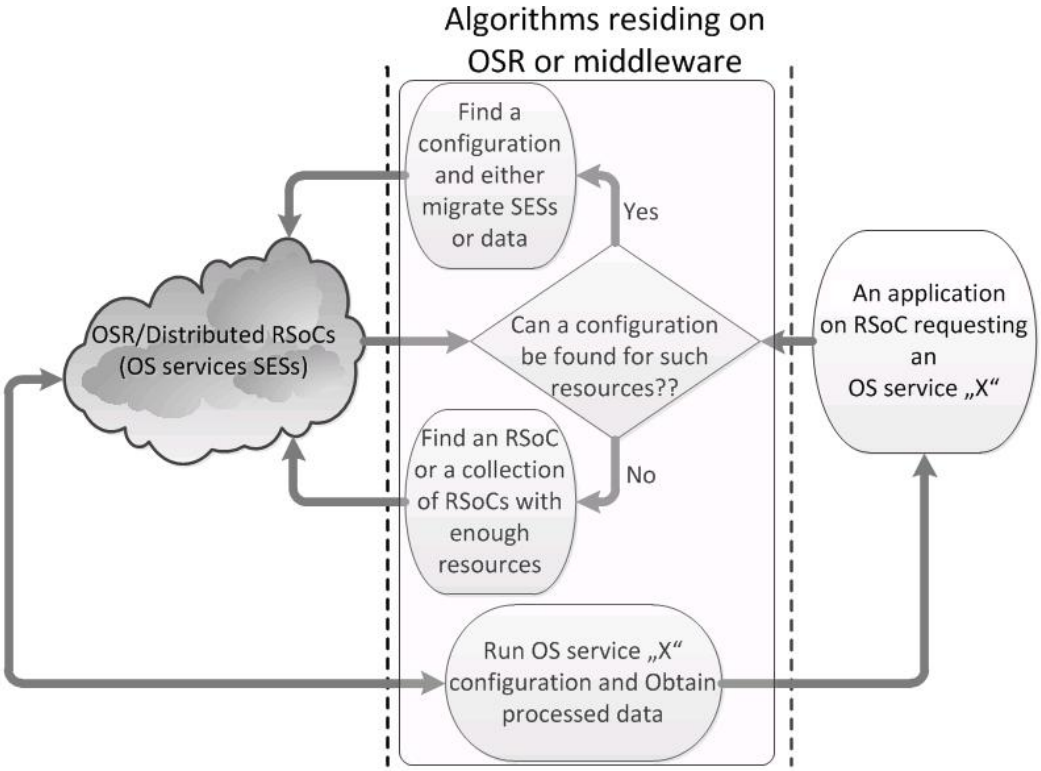


Figure 9.1: A brief summary of finding an adequate OS service configuration

To allow RSoCs collaboration, load balancing, and efficient execution routing, heuristic algorithms were developed to distribute OS services over RSoCs. These algorithms are used for efficient distribution of heterogeneous OS services partitions over the heterogeneous RSoCs. Moreover, they allow building of discovery and execution routing records to efficiently find at runtime an adequate RSoC collaboration with the desired OS service configuration.

To support this novel design of runtime adaptable OS services, each RSoC contains a middleware. This middleware contains the algorithms and the proper units to manage and maintain the RSoC resources, communications, and the OS services adaptation, see Section 3.2.1.

The presented adaptation of OS services enables the full utilization of platforms with heterogeneous computational elements. In addition to the contribution to a new OS ser-

vice design, many of the presented methods and modules are adequate for applications as well. This may even improve other related work such as in [113, 98]. Moreover, this work opens many interesting research directions as discussed in the following section.

9.2 Future directions

The introduced novel design of adaptable OS service depends on multiple implementations which are partitioned into $SESs$ as described in Section 4.2. One approach to automate this partitioning is to modify existing open source (C-to-FPGA) compilers as in [65, 58, 56]. Another approach would be the generation of code using abstract modules as in [28]. This automation of partitioning is with no doubt a key element to ease the transfer of existing OS services to the adaptable design introduced in Section 4.2.

On the other hand, the distribution of heterogeneous $SESs$ over heterogeneous RSoCs, see Section 6.3, introduces another research direction regarding optimization and organization, see Section 9.2.1.

Because RSoCs consist of multiple computational elements, and the new design of OS service provides for recovery techniques, online model checking can be integrated with the middleware as discussed in Section 9.2.2.

9.2.1 Distribution and optimization

After distributing the $SESs$, another stage can be introduced. This stage is performed at run time using profiling information obtained during the execution of services. This profiling information includes statistics about the locations where each OS service is being requested more frequently, and whether there exists a network congestion involving draining resources of one RSoC more than the others. The optimization aims to improve the performance and the efficiency of communication and execution.

Communication optimization

After performing the GFD, see Section 6.2.1. we end up with some kind of permutation network for executing a service. Assuming we have chosen an encoding which provides us with four alternatives for each SES, after executing one SES, the data can be transferred to any of the four successors alternatives, see Figure 9.2.

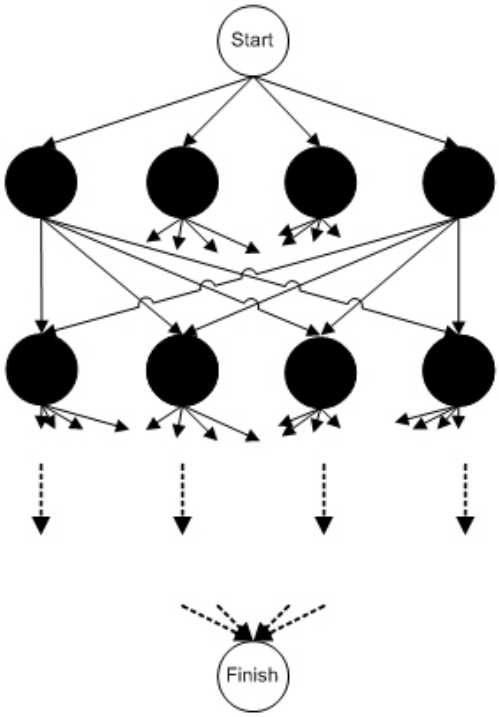


Figure 9.2: Example of the permutation execution network of one OS service

Although one suitable path may have been chosen in the routing and execution stage, see Section 6.3, many other suitable paths may also exist. Following the same path every time has many disadvantages, for example, it may cause the RSoCs across that path to loose a lot of power. The communication optimization stage tries to avoid such problems without violating any constraint.

Execution configurations optimization

After executions of a service, it may be found that one SES is located apart from other $SESs$ belonging to that service, or the requests to a service are coming from one place and the service is located on a different one. This may be minimized by migrating $SESs$ of a service to the place where they are frequently requested. This is to be done taking into account load balancing and updating the alternative and successor records in each related SES, see Section 6.3.

9.2.2 Online model checking and recovery

Due to the expected dynamic changes in applications/tasks, the resources or the constraints may change accordingly at runtime. This may lead to change/adjustment in OS service configurations. Due to the sensitivity of the process, as this may be a service requested by a real time application/task, dependability and fault tolerance of the underlying operating system is highly expected. Due to the huge time and space complexity, it is impossible to verify the safety properties of all the m^n different implementations of an OS service at the systems development phase. Therefore, online model checking [1], [155] might be a good choice to improve the dependability and fault tolerance of our distributed RSoC with adaptable OS services.

This online model checking can be done by generating a sufficiently precise abstract model from safety-critical $SESs$ code using approaches like in [68]. The abstract model is an over-approximation of the concrete system. For every concrete state sequence, there exists a corresponding abstract state sequence. The relations between concrete states and abstract states are defined by means of two functions: an *abstraction* function α maps every set of concrete states to a corresponding abstract state; a *concretization* function γ maps every abstract state to a set of concrete states that it represents.

In this way, for each safety-critical SES_i of a service, we can get the corresponding abstract model $\widehat{SES_i}$ as well as the abstraction function α_i and concretization function γ_i.

Model Checking Paradigm

Online Model Checking (MC) runs on an RSoC in parallel with the $SESs$ to be checked. The abstract model of the checked SES is explored with respect to the given safety property. Since model checking is done online, the current (concrete) states of the $SESs$ execution can be monitored and reported to the model checker from time to time. By mapping these concrete states to the corresponding abstract ones at a model level through the abstraction function, the online model checker needs only to explore such a partial state space reachable from these abstract (current) states. If this partial abstract model is checked *safe* against the given property, then we have more confidence to the safety of the actual execution trace. If an error is detected within the partial state space, a recovery process will be triggered.

Recovery Process

A recovery process is initiated by sending a recovery message from the model checker to the RSoC middleware which coordinates processing of OS service execution, see Figure 9.3. This can be the same RSoC where the model checker is being executed or a different RSoC in a distributed system.

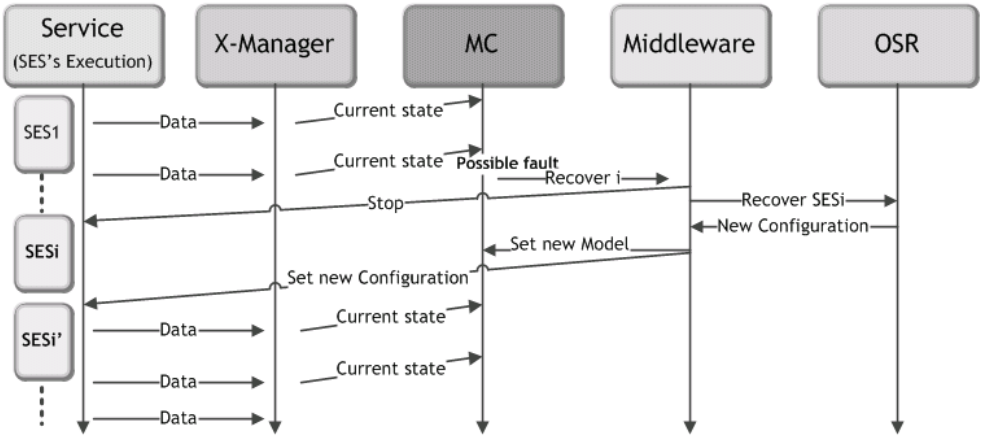


Figure 9.3: Recovering a possible fault in OS service execution

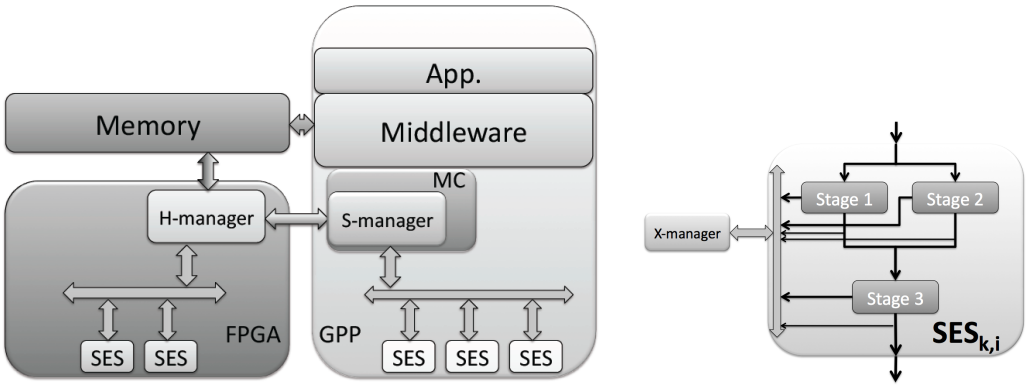
The recovery message contains information about the possible incorrect SES and the current state of this SES. The middleware then evaluates a new suitable configuration and if necessary acquires any missing SES from the OSR. Meanwhile, it raises a stop signal to the incorrect service. The middleware afterwards initializes the new configuration and redirects the further execution of the service to it. The middleware also sends to the model checker a message to set a new model associated with the new SES to be executed if necessary.

SESs, middleware, and model checking integration

For mapping abstract states to concrete states, the online model checker needs to communicate with $SESs$ running on FPGA as well as on GPP. However, $SESs$ with the same behavior in one OS service have different implementations, hence they are distinct in terms of execution time and resource usage. This makes it unrealistic to synchronously schedule the communication between the model checker and a SES to be checked, e.g., every T time. Fortunately, an event driven approach is a suitable solution.

At the design phase, some triggering point can be defined at which the communication is initiated. The triggering points depend on the functional behavior and the states of each OS service. This can be obtained by analyzing the source code. When the communication is triggered, a decoded message with the required information is sent to the model checker. These normally contain global data and conditional values.

Usually, the model checker is running on GPP. This eases data transferring between $SESs$ running on GPP and the model checker. Such communication can be achieved by coping/accessing the address space of the SES or using shared memory. Where $SESs$ running on FPGA are to be considered for checking, another component (the X-manager) could be introduced. The X-manager works as a complementary part to the model checker. It consists of two parts: the H-manager which works on FPGA and the S-manager which runs with the model checker (MC) on GPP, see Figure 9.4(a). All the $SESs$ communications involving reading or writing memory are done through the X-manager. In doing so, the X-manager can monitor $SESs$ and report back to the model checker.



(a) An overview of an RSoC with one FPGA and GPP and the communications between $SESs$, MC, and memory.

(b) An insight of the SES stages and the communication with the X-manager.

Figure 9.4: $SESs$, middleware, and model checker integration

The communication between a SES and the X-manager can be triggered by defining check points after specified execution stages, see Figure 9.4(b). A stage is a logical grouping of a SES code after which the MC is triggered. Any access to the memory from every stage is monitored by the X-manager. At the end of each stage, an event will be sent to the X-manager. On receiving the event, the X-manager will provide the model checker with a snap shot of the memory just modified by the stage, so far, executed of the SES to be checked. This minimizes the time needed to trans-

fer data to the model checker. Thus, the model checker might have more chance to run in pre-checking mode, i.e., looking ahead in the near future at the model level. This is important as this allows the recovery process to happen without violating any constraints.

Author's Publications

[1] Franz J. Rammig, Yuhong Zhao, and Sufyan Samara. On-line model checking as operating system service. In *SEUS '09: Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, pages 131–143, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] Sufyan Samara. Partitioning granularity, communication overhead, and adaptation in os services for distributed reconfigurable systems on chip. In *The 13th IEEE International Conferences on Computational Science and Engineering (CSE 2010) / The 8th IEEE/IFIP International Conferences on Embedded and Ubiquitous Computing (EUC 2010)*. IEEE Computer Society, 2010.

[3] Sufyan Samara and Fahad Bin Tariq. Os service optimization in a heterogeneous distributed system on chip (soc). In *Workshop on Distributed Computing in Ambient Environments (DiComAe) within the KI2009 Conference*, 2009.

[4] Sufyan Samara, Dalimir Orfanus, and Peter Janacik. Towards biologically inspired decentralized self-adaptive os services for distributed reconfigurable system on chip (rsoc). *ACM SIGBED Rev.*, 6(3):1–4, 2009.

[5] Sufyan Samara and Gunnar Schomaker. Self-adaptive os service model in relaxed resource distributed reconfigurable system on chip (rsoc). *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, Computation World, IEEE*, 0:1–8, 2009.

[6] Sufyan Samara and Gunnar Schomaker. Real-time adaptation and load balancing aware os services for distributed reconfigurable system on chip. *Computer and Information Technology, IEEE*, 0:1743–1750, 2010.

[7] Sufyan Samara, Fahad Bin Tariq, Timo Kerstan, and Katharina Stahl. Applications adaptable execution path for operating system services on a distributed reconfigur-

able system on chip. In *ICESS '09: Proceedings of the 2009 International Conference on Embedded Software and Systems*, pages 461–466, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Sufyan Samara, Yuhong Zhao, and Franz Rammig. Integrate online model checking into distributed reconfigurable system on chip with adaptable os services. In *Distributed, Parallel and Biologically Inspired Systems*, volume 329 of *IFIP Advances in Information and Communication Technology*, pages 102–113. Springer Boston, 2010.

References

[9] Jason Agron and David Andrews. Building heterogeneous reconfigurable systems with a hardware microkernel. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 393–402, New York, NY, USA, 2009. ACM.

[10] Ali Ahmadinia, Christophe Bobda, Dirk Koch, Mateusz Majer, and Jürgen Teich. Task scheduling for heterogeneous reconfigurable computers. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, pages 22–27, New York, NY, USA, 2004. ACM.

[11] L. Ali, N.A.M Yunus, H. Jaafar, R. Wagiran, and E. Low. Implementation of triple data encryption algorithm using vhdl. In *IEEE International Conference on Semiconductor Electronics (ICSE)*, 2004.

[12] Altera. www.altera.com.

[13] David Andrews, Douglas Niehaus, Razali Jidin, Michael Finley, Wesley Peck, Michael Frisbie, Jorge Ortiz, Ed Komp, and Peter Ashenden. Programming models for hybrid fpga-cpu computational components: A missing link. *IEEE Micro*, 24(4):42–53, 2004.

[14] Péter Arató, Zoltán Ádám Mann, and András Orbán. Algorithmic aspects of hardware/software partitioning. *ACM Trans. Des. Autom. Electron. Syst.*, 10(1):136–156, 2005.

[15] Atmel. www.atmel.com.

[16] S. Baskiyar, Ph. D, and N. Meghanathan. A survey of contemporary real-time operating systems. informatica 29:233Ű240, 2005.

[17] Th. Benner, Rolf Ernst, Ingo Könenkamp, Ulrich Holtmann, P. Schüler, H.-C. Schaub, and N. Serafimov. Fpga based prototyping for verification and evaluation in hardware-software cosynthesis. In *FPL '94: Proceedings of the 4th International Workshop on Field-Programmable Logic and Applications*, pages 251–258, London, UK, 1994. Springer-Verlag.

[18] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemys law Pardyak, Stefan Savage, and Emin Gün Sirer. Spin—an extensible microkernel for application-specific operating system services. *SIGOPS Oper. Syst. Rev.*, 29(1):74–77, 1995.

[19] Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. The pure family of object-oriented operating systems for deeply embedded systems. In *ISORC '99: Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 45, Washington, DC, USA, 1999. IEEE Computer Society.

[20] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: a system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst.*, 16(4):321–366, 1998.

[21] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, 2005.

[22] Christophe Bobda. *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*. Springer Publishing Company, Incorporated, 2007.

[23] Scott A. Brandt and Gary J. Nutt. Flexible soft real-time processing in middleware. *Real-Time Syst.*, 22(1/2):77–118, 2002.

[24] G. Brebner. The swappable logic unit: a paradigm for virtual hardware. In *FCCM 97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, page 77, Washington, DC, USA, 1997. IEEE Computer Society.

[25] Gordon Brebner. A virtual hardware operating system for the xilinx xc6200. In Reiner Hartenstein and Manfred Glesner, editors, *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, volume 1142 of *Lecture Notes in Computer Science*, pages 327–336. Springer Berlin / Heidelberg, 1996.

[26] André Brinkmann, Sascha Effert, Friedhelm Meyer auf der Heide, and Christian Scheideler. Dynamic and redundant data placement. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007)*, Toronto, Canada, 25 - 29 June 2007.

[27] André Brinkmann, Kay Salzwedel, and Christian Scheideler. Compact, adaptive placement schemes for non-uniform distribution requirements. In *Proc. of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 53–62, Winnipeg, Manitoba, Canada, 11 - 13 August 2002.

[28] Lisane B. Brisolara, Marcio F. S. Oliveira, Ricardo Redin, Luis C. Lamb, Luigi Carro, and Flavio Wagner. Using uml as front-end for heterogeneous software code generation strategies. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 504–509, New York, NY, USA, 2008. ACM.

[29] Dov Bulka and David Mayhew. *Efficient C++: performance programming techniques*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[30] Roy H. Campbell, Nayeem Islam, David Raila, and Peter Madany. Designing and implementing choices: an object-oriented system in c++. *Commun. ACM*, 36(9):117–126, 1993.

[31] Rita Yu Chen, Robert M. Owens, Mary Jane Irwin, R. S. Bajwa, and Raminder S. Bajwa. Validation of an architectural level power analysis technique. In *DAC '98: Proceedings of the 35th annual Design Automation Conference*, pages 242–245, New York, NY, USA, 1998. ACM.

[32] Pong P. Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Wiley-IEEE Press, 2006.

[33] Antoine Colin and Guillem Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *ECRTS '02: Proceedings of the 14th Euromicro Conference on Real-Time Systems*, page 50, Washington, DC, USA, 2002. IEEE Computer Society.

[34] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[35] Klaus Danne and Sven Stuehmeier. Off-line placement of tasks onto reconfigurable hardware considering geometrical task variants, 2005.

[36] B.P. Dave, G. Lakshminarayana, and N.K. Jha. Cosyn: Hardware-software co-synthesis of heterogeneous distributed embedded systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 7(1):92 –104, mar. 1999.

[37] Robert P. Dick and Niraj K. Jha. Cords: hardware-software co-synthesis of reconfigurable real-time distributed embedded systems. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 62–67, New York, NY, USA, 1998. ACM.

[38] Robert P. Dick and Niraj K. Jha. Cowls: Hardware-software co-synthesis of distributed wireless low-power embedded client-server systems. In *VLSID '00: Proceedings of the 13th International Conference on VLSI Design*, page 114, Washington, DC, USA, 2000. IEEE Computer Society.

[39] R.P. Dick and N.K. Jha. Mogac: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(10):920 –935, oct. 1998.

[40] Florian Dittmann. *Methods to Exploit Reconfigurable Fabrics*. PhD thesis, University of Paderborn, 2007.

[41] Carsten Ditze. *Towards Operating System Synthesis*. PhD thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, Paderborn University, 1999.

[42] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[43] Klaus Ecker, David Juedes, Lonnie Welch, David Chelberg, Carl Bruggeman, Frank Drews, David Fleeman, David Parrott, and Barbara Pfarr. An optimization framework for dynamic, distributed real-time systems. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 111.2, Washington, DC, USA, 2003. IEEE Computer Society.

[44] eCos. http://ecos.sourceware.org/. visited on, August 2010.

[45] Giovanni De Micheli (Editor), Rolf Ernst (Editor), and Wayne Wolf (Editor). *Readings in hardware/software co-design*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[46] Petru Eles, Zebo Peng, Krzysztof Kuchcinski, and Alexa Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems*, 2:5–32, 1997. 10.1023/A:1008857008151.

[47] EMV. *EMV 4.2 Specifications*, volume Book 2 of *version 4.2*. EMV, security and key management edition, June 2008.

[48] Frank Engel, Ihor Kuz, Stefan M. Petters, and Sergio Ruocco. Operating systems on socs: A good idea?, 2004.

[49] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM.

[50] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. VDM Verlag, Saarbrücken, Germany, Germany, 2008.

[51] Rolf Ernst, Jorg Henkel, and Thomas Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Des. Test*, 10(4):64–75, 1993.

[52] Anand Eswaran, Anthony Rowe, and Raj Rajkumar. Nano-rk: An energy-aware resource-centric rtos for sensor networks. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 256–265, Washington, DC, USA, 2005. IEEE Computer Society.

[53] Gui-Liang Feng, Robert H. Deng, Feng Bao, and Jia-Chen Shen. New efficient mds array codes for raid part i: Reed-solomon-like codes for tolerating three disk failures. *IEEE Trans. Comput.*, 54(9):1071–1080, 2005.

[54] David Fleeman, M. Gillen, A. Lenharth, M. Delaney, L. Welch, D. Juedes, and C. Liu. Quality-based adaptive resource management architecture (qarma): A corba resource management service. *Parallel and Distributed Processing Symposium, International*, 3:116b, 2004.

[55] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit: a substrate for kernel and language research. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51, New York, NY, USA, 1997. ACM.

[56] FPGAC. http://sourceforge.net/projects/fpgac/, visited September 2010.

[57] L. Fernando Friedrich, John Stankovic, Marty Humphrey, Michael Marley, and John Haskins. A survey of configurable, component-based operating systems for embedded applications. *IEEE Micro*, 21(3):54–68, 2001.

[58] Jan Frigo, Maya Gokhale, and Dominique Lavenier. Evaluation of the streams-c c-to-fpga compiler: an applications perspective. In *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 134–140, New York, NY, USA, 2001. ACM.

[59] Wenyin Fu and Katherine Compton. An execution environment for reconfigurable computing. In *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 149–158, Washington, DC, USA, 2005. IEEE Computer Society.

[60] Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. The pebble component-based operating system. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 20–20, Berkeley, CA, USA, 1999. USENIX Association.

[61] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. *Specification and design of embedded systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[62] D.D. Gajski, F. Vahid, S. Narayan, and Jie Gong. Specsyn: an environment supporting the specify-explore-refine paradigm for hardware/software system design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 6(1):84 –100, mar. 1998.

[63] Philip Garcia, Katherine Compton, Michael Schulte, Emily Blem, and Wenyin Fu. An overview of reconfigurable hardware in embedded systems. *EURASIP J. Embedded Syst.*, 2006(1):13–13, 2006.

[64] Samuel Garcia and Bertrand Granado. Ollaf: A fine grained dynamically reconfigurable architecture for os support. *EURASIP Journal on Embedded Systems*, 2009, 2009.

[65] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented fpga computing in the streams-c high level language. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 49, Washington, DC, USA, 2000. IEEE Computer Society.

[66] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[67] Marcelo Götz. *Run-time Reconfigurable RTOS for Reconfigurable Systems-on-Chip*. PhD thesis, University of Paderborn, 2007.

[68] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83, London, UK, 1997. Springer-Verlag.

[69] Plerre-P Grassé. La reconstruction du nid et les coordinations interindividuelles chezbellicositermes natalensis etcubitermes sp. la théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs. *Insectes Sociaux*, 6(1):41–80, March 1959.

[70] Jan Gustafsson and Andreas Ermedahl. Experiences from applying wcet analysis in industrial settings. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 382–392, Washington, DC, USA, 2007. IEEE Computer Society.

[71] Christian Haubelt, Dirk Koch, and Jürgen Teich. Basic os support for distributed reconfigurable hardware, 2003.

[72] Christian Haubelt, Dirk Koch, and Jürgen Teich. Reconet: Modeling and implementation of fault tolerant distributed reconfigurable hardware, 2003.

[73] Scott Hauck and Andre DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[74] Hayden Kwok hay So, Borph An, Operating System, Fpga based Reconfigurable, and Hayden Kwok hay So. Borph: An operating system for fpgabased reconfigurable computers. Technical report, UNIVERSITY OF CALIFORNIA, BERKELEY, 2007.

[75] Johannes Helander and Alessandro Forin. Mmlite: a highly componentized system architecture. In *EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 96–103, New York, NY, USA, 1998. ACM.

[76] Dan Hildebrand. An architectural overview of qnx. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Berkeley, CA, USA, 1992. USENIX Association.

[77] Mike Hinchey, Bernd Kleinjohann, Lisa Kleinjohann, Peter Lindsay, Franz J. Rammig, Jon Timmis, and Marilyn Wolf. *Distributed, Parallel and Biologically Inspired Systems*. Springer, 2010.

[78] George J. Y. Hsu, Pao long Chang, and Tser yieth Chen. Various methods for estimating power outage costs : Some implications and results in taiwan. *Energy Policy*, 22(1):69 – 74, 1994.

[79] Xilinx Inc. *Virtex-II Pro / Virtex-II Pro X Complete Data Sheet (All four modules)*. San Jose, CA, USA, 4.7 edition, 2007.

[80] ISO/IEC. 18033-3:2005 information technology Ů security techniques Ů encryption algorithms. Part 3: Block ciphers, 2008.

[81] Peter Janacik, Odej Kao, and Ulf Rerrer. An approach combining routing and resource sharing in wireless ad hoc networks using swarm-intelligence. In *Proceedings of the 7th ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2004)*, 2004. Poster session.

[82] Jbed. Jbed whitepaper: Component software and real-time computing. Technical report, Oberon Microsystems, 1998.

[83] Jorjeta G. Jetcheva and David B. Johnson. Adaptive demand-driven multicast routing in multi-hop wireless ad hoc networks. In *MobiHoc '01: Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, pages 33–44, New York, NY, USA, 2001. ACM.

[84] Jini.org. online, accessed September, 2010.

[85] A. Kalavade and P.A. Subrahmanyam. Hardware/software partitioning for multifunction systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(9):819 –837, sep. 1998.

[86] Asawaree Kalavade and Edward A. Lee. A hardware-software codesign methodology for dsp applications. *IEEE Des. Test*, 10(3):16–28, 1993.

[87] Heiko Kalte and Mario Porrmann. Context saving and restoring for multitasking in reconfigurable systems. In *15th International Conference on Field Programmable Logic and Applications*, pages 223–228, August 2005.

[88] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 654–663, El Paso, Texas, 4–6 May 1997.

[89] Felipe Klein, Guido Araujo, Rodolfo Azevedo, Roberto Leao, and Luiz C. V. Dos Santos. On the limitations of power macromodeling techniques. *VLSI, IEEE Computer Society Annual Symposium on*, 0:395–400, 2007.

[90] P.V. Knudsen and J. Madsen. Pace: a dynamic programming algorithm for hardware/software partitioning. In *International Workshop on Hardware-Software Co-Design*, pages 85 –92, mar. 1996.

[91] Tim Kogel and Heinrich Meyr. Heterogeneous mp-soc: the solution to energy-efficient signal processing. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 686–691, New York, NY, USA, 2004. ACM.

[92] Fabio Kon, Roy H. Campbell, M. D Mickunas, and Klara Nahrstedt. 2k: A distributed operating system for dynamic heterogeneous environments. Technical report, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1999.

[93] Fabio Kon, Ashish Singhai, Roy H. Campbell, Dulcineia Carvalho, and Robert Moore. 2k: A dynamic, component-based operating system for rapidly changing environments. Technical report, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1998.

[94] Johannes Lessmann, Tales Heimfarth, and Peter Janacik. Shox: An easy to use simulation platform for wireless networks. In *UKSIM '08: Proceedings of the Tenth International Conference on Computer Modeling and Simulation*, pages 410–415, Washington, DC, USA, 2008. IEEE Computer Society.

[95] Zhiyuan Li and Scott Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 187–195, New York, NY, USA, 2002. ACM.

[96] S. M. Loo, B. E. Wells, and R. K. Gaede. Exploring the hardware/software continuum in a computer engineering capstone design class using fpga-based programmable logic. In *MSE '01: Proceedings of the 2001 International Conference on Microelectronic Systems Education (MSE'01)*, page 36, Washington, DC, USA, 2001. IEEE Computer Society.

[97] E. Lubbers and M. Planner. Reconos: An rtos supporting hard-and software threads. In *In Proceedings of the 17th International Conference on Field-Programmable Logic and Applications*, pages 441 –446, aug. 2007.

[98] Enno Lübbers. *Multithreaded Programming and Execution Models for Reconfigurable Hardware*. Phd thesis, University of Paderborn, 2010.

[99] Enno Lübbers and Marco Platzner. Communication and Synchronization in Multithreaded Reconfigurable Computing Systems. In *Proceedings of the 8th International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'08)*. CSREA Press, July 2008.

[100] Enno Lübbers and Marco Platzner. Reconos: Multithreaded programming for reconfigurable computers. *ACM Trans. Embed. Comput. Syst.*, 9(1):1–33, 2009.

[101] Roman Lysecky and Frank Vahid. A study of the speedups and competitiveness of fpga soft processor cores using dynamic hardware/software partitioning. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 18–23, Washington, DC, USA, 2005. IEEE Computer Society.

[102] J. Madsen, J. Grode, P.V. Knudsen, M.E. Petersen, and A. Haxthausen. Lycos: the lyngby co-synthesis system. *Design Automation for Embedded Systems*, 2:195–235, 1997. 10.1023/A:1008884219274.

[103] Alessandro Di Marco, Giovanni Chiola, and Giuseppe Ciaccio. Using a gigabit ethernet cluster as a distributed disk array with multiple fault tolerance. In *LCN '03: Proceedings of the 28th Annual IEEE International Conference on Local Computer Networks*, page 605, Washington, DC, USA, 2003. IEEE Computer Society.

[104] T. Marescaux, V. Nollet, J.-Y. Mignolet, A. Bartic, W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Run-time support for heterogeneous multitasking on reconfigurable socs. *Integr. VLSI J.*, 38(1):107–130, 2004.

[105] Peter Marwedel. *Embedded System Design*. Kluwer Academic Publishers, 2003.

[106] Anthony Massa. *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002.

[107] Clive Maxfield. *The Design Warrior's Guide to FPGAs*. Academic Press, Inc., Orlando, FL, USA, 2004.

[108] Mario Mense and Christian Scheideler. Spread: An adaptive scheme for redundant and fair storage in dynamic heterogeneous storage systems. In *19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, San Francisco, California, USA, 20.-22. Febr.,, January 2008.

[109] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[110] David Naylor and S. Jones. *VHDL: A Logic Synthesis Approach*. Chapman & Hall, Ltd., London, UK, UK, 1997.

[111] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable soc. In *Proceedings of the RAW'03 workshop*, 2003.

[112] null Bo Zhou, null Weidong Qiu, null Yan Chen, and null Chenglian Peng. Shum-ucos: A rtos using multi-task model to reduce migration cost between sw/hw tasks. *International Conference on Computer Supported Cooperative Work in Design*, 2:984–989 Vol. 2, 2005.

[113] Simon Oberthür. *Towards an RTOS for Self-ptimizing Mechatronic Systems*. PhD thesis, University of Paderborn, 2009.

[114] L. Parker. Current research in multirobot systems. *Artificial Life and Robotics*, 7:1–5, 2003. 10.1007/BF02480877.

[115] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, New York, NY, USA, 1988. ACM Press.

[116] Wesley Peck, Erik Anderson, Jason Agron, Jim Stevens, Fabrice Baijot, and David L. Andrews. Hthreads: A computational model for reconfigurable devices. In *FPL*, pages 1–4, 2006.

[117] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. *SIGCOMM Comput. Commun. Rev.*, 24(4):234–244, 1994.

[118] Charles E. Perkins and Elizabeth M. Royer. Ad-hoc on-demand distance vector routing. *Mobile Computing Systems and Applications, IEEE Workshop on*, 0:90, 1999.

[119] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[120] James S. Plank. The raid-6 liberation codes. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[121] Nachiketh R. Potlapally, Michael S. Hsiao, Anand Raghunathan, Ganesh Lakshminarayana, and Srimat T. Chakradhar. Accurate power macro-modeling techniques for complex rtl circuits. *VLSI Design, International Conference on*, 0:235, 2001.

[122] S. Raaijmakers and S. Wong. Run-time partial reconfiguration for removal, placement and routing on the virtex-ii pro. In *International Conference on Field Programmable Logic Applications*, pages 679 –683, aug. 2007.

[123] R. K. Raval, C. H. Fernandez, and C. J. Bleakley. Low-power tinyos tuned processor platform for wireless sensor network motes. *ACM Trans. Des. Autom. Electron. Syst.*, 15(3):1–17, 2010.

[124] Adi Mallikarjuna V. Reddy, A.V.U. Phani Kumar, D. Janakiram, and G. Ashok Kumar. Wireless sensor network operating systems: a survey. *Int. J. Sen. Netw.*, 5(4):236–255, 2009.

[125] Javier Resano, Daniel Mozos, Diederik Verkest, and Francky Catthoor. A reconfiguration manager for dynamically reconfigurable hardware. *IEEE Des. Test*, 22(5):452–460, 2005.

[126] Achim Rettberg. *Low-power Driven High Level Synthesis for Dedicated Architectures*. PhD thesis, Paderborn University, January 2007.

[127] M. Saldana, D. Nunes, E. Ramalho, and P. Chow. Configuration and programming of heterogeneous multiprocessors on a multi-fpga system using tmd-mpi. *Reconfigurable Computing and FPGAs, International Conference on*, 0:1–10, 2006.

[128] Tom Saulpaugh and Charles Mirho. *The Java OS Design and Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[129] Christian Schindelhauer and Gunnar Schomaker. Weighted distributed hash tables. In *Proc. of the 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), to appear*, 2005.

[130] Roy Shea, Simon Han, and Ram Rengaswamy. Motivations behind sos. Technical report, University of California Los Angeles, 2004.

[131] Seng Lin Shee and Sri Parameswaran. Design methodology for pipelined heterogeneous multiprocessor system. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 811–816, New York, NY, USA, 2007. ACM.

[132] Amit Sinha and Anantha Chandrakasan. Dynamic power management in wireless sensor networks. *IEEE Des. Test*, 18(2):62–74, 2001.

[133] Christoph Steiger, Herbert Walder, and Marco Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Trans. Computers*, 53(11):1393–1407, 2004.

[134] David B. Stewart. Measuring execution time and real-time performance. In *In: Proceedings of the Embedded Systems Conference (ESC SF*, pages 1–15, 2002.

[135] David B. Stewart and Gaurav Arora. A tool for analyzing and fine tuning the real-time properties of an embedded system. *IEEE Trans. Softw. Eng.*, 29(4):311–326, 2003.

[136] Wind River Systems. *VxWorks ProgrammerŠs Guide*. Alameda, CA 94501-1153, USA, 5.5 edition, 2002.

[137] Jean-Charles Tournier. A survey of configurable operating systems. Technical report, University of New Mexico, 2005.

[138] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service location protocol.

[139] Herbert Walder and Marco Platzner. Reconfigurable hardware operating systems: From design concepts to realizations. In *In Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA*, pages 284–287. CSREA Press, 2003.

[140] Herbert Walder and Marco Platzner. A runtime environment for reconfigurable hardware operating systems. In *in Proceedings of the 14th Field Programmable Logic and Applications (FPLŠ04*, pages 831–835. Springer, 2004.

[141] Colin Walls. *Embedded Software: The Works*. Newnes, 2005.

[142] Catherine Lingxia Wang, Bo Yao, Yang Yang, and Zhengyong Zhu. A survey of embedded operating systems. Technical report, UCSD, Computer Scince and Engineering department, 2001.

[143] Christof Wehner. *Tornado and VxWorks*. BoD, 2006.

[144] G. Wigley, D. Kearney, and M. Jasiunas. Reconfigme: a detailed implementation of an operating system for reconfigurable computing. *Parallel and Distributed Processing Symposium, International*, 0:218, 2006.

[145] Grant B. Wigley and David A. Kearney. Research issues in operating systems for reconfigurable computing. In *In Proceedings of the International Conference on Engineering of Reconfigurable System and Algorithms(ERSA*, pages 10–16. CSREA Press, 2002.

[146] Grant B. Wigley, David A. Kearney, and David Warren. Introducing reconfigme: An operating system for reconfigurable computing. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 687–697, London, UK, 2002. Springer-Verlag.

[147] Tim Wilmshurst. *An Introduction to the Design of Small-Scale Embedded Systems*. Palgrave Macmillan, 2001.

[148] Wayne H. Wolf. An architectural co-synthesis algorithm for distributed, embedded computing systems. *IEEE Trans. Very Large Scale Integr. Syst.*, 5(2):218–229, 1997.

[149] Xilinx. www.xilinx.com.

[150] Xilinx. Xbrf 014: A simple method of estimating power in xc4000xl/ex/e fpgas. Technical report, Xilinx, 1997.

[151] Xilinx. *Differencing Method for Partial Reconfiguration*, xapp290 (v2.0) edition, December 3 2007.

[152] Xilinx. *Partial Reconfiguration User Guide*, ug702 (v 12.2) edition, July 23 2010.

[153] J. Yannakopoulos and A. Bilas. Cormos: a communication-oriented runtime system for sensor networks. pages 342 – 353, jan. 2005.

[154] Pavel G. Zaykov, Georgi K. Kuzmanov, and Georgi N. Gaydadjiev. Reconfigurable multithreading architectures: A survey. In *SAMOS '09: Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 263–274, Berlin, Heidelberg, 2009. Springer-Verlag.

[155] Yuhong Zhao and Franz Rammig. Model-based runtime verification framework. *Electron. Notes Theor. Comput. Sci.*, 253(1):179–193, 2009.

[156] Bo Zhou, Weidong Qiu, and Chenlian Peng. An operating system framework for reconfigurable systems. In *CIT '05: Proceedings of the The Fifth International Conference on Computer and Information Technology*, pages 788–792, Washington, DC, USA, 2005. IEEE Computer Society.

[157] R. Zimmermann and S. Ghandeharizadeh. Hera: Heterogeneous extension of raid, 1998.

[158] Richard Zurawski, editor. *Embedded Systems Handbook, Second Edition: Networked Embedded Systems*. CRC Press, 2009.