UNIVERSITÀ DEGLI STUDI DI UDINE

Facoltà di Ingegneria
Corso di Laurea Specialistica in Ingegneria Elettronica
Dipartimento di Ingegneria Elettrica, Gestionale e Meccanica

Final Thesis

Performance Evaluation of Multi-threading Operating Systems in MPSoCs Generated by ESPAM

Valutazione delle Prestazioni di Sistemi Operativi Multi-threading su MPSoC Generati da ESPAM

Supervisors:

TU Delft, The Netherlands:
Dr. ir. Todor Stefanov
Dr. ir. Georgi Gaydadjiev

University of Udine, Italy:
Prof. dr. ir. Antonio Abramo

Student:

Emanuele Cannella

Academic Year 2007-08

Contents

Acknowledgments

Throughout the work of this Master Thesis I have been supported and guided by Todor Stefanov, not only with technical and scientific advice, but also with motivation and encouragement. He was always very helpful and patient when answering to the several doubts that have arisen during this research project. So, the first special thanks go to him.

I am very grateful also to Georgi Gaydadjiev, who nicely accepted my request of thesis project in the Computer Engineering group, at Delft University of Technology, as an Erasmus-Socrates student.

I also want to acknowledge my Italian supervisor, Prof. Antonio Abramo, who gave me the opportunity of this experience in The Netherlands. He has always shown real interest about my research work, and this has given me additional motivation.

Emanuele Cannella
Udine, Italy
June 17, 2008

Chapter 1

Introduction

This chapter firstly provides a brief description of the background and the premises that have inspired the work. Then, it presents the description of the arising problems and the solution approach, with an overview of the related work. Finally, the research contributions of this thesis are stated.

1.1 Background

A System-on-Chip (SoC) is defined as a single integrated circuit which includes all the components of a computer or other electronic system. A typical System-on-chip is comprised by one or more microprocessor or DSP cores, memory blocks of different types, peripherals like counter-timers and external interfaces.

Nowadays many embedded systems are implemented with a System-on-Chip solution, and this trend is likely to continue in the future. This is because putting a whole system on a single chip gives better performance, in terms of power consumption and area occupation, two of the most important features of an embedded system. Moreover, the packaging of a SoC is simpler, leading to smaller production costs.

Embedded applications are becoming increasingly complex. However, thanks to technology improvement - Moore's law predicts an exponential growth of the number of transistor per chip over time - the computational resources which a designer can exploit are increasing too. The main issue that arises is how to fully capitalize on these resources.

An estimation of this issue was presented in a research made by SEMATECH [3]. The main result is depicted in Figure1.1. The graph shows that the integrated circuit productivity has traditionally grown 58% per year, while the designers productivity grows only 21% per year. This difference leads to the so-called *Productivity Gap*. Closing this gap would mean the full exploitation of technology development. This can be achieved only by improved, new design approaches.

For this purpose, increasing the level of abstraction and reusing components play a

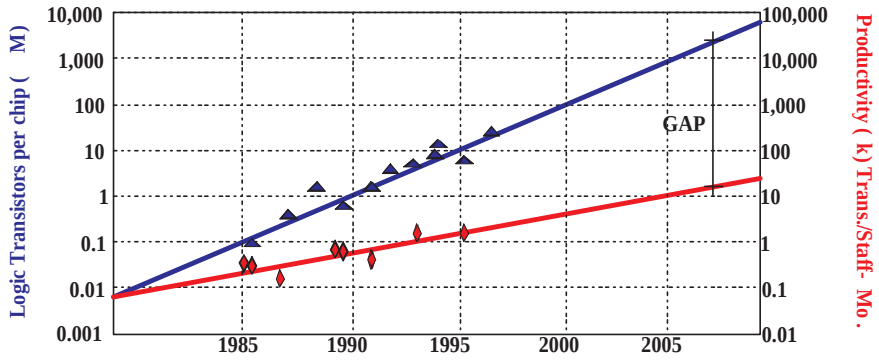

Figure 1.1: The different growth of IC productivity and designers productivity leads to the so-called *Productivity Gap*.

key role. Currently, most of the design methodologies and tools require a Register Transfer Level description of an architecture. This approach is no longer adequate, since architecture complexity of new systems is huge.

Besides, a single-processor embedded system cannot handle the requirements of new applications like high-throughput multimedia and Digital Signal Processing. So, the emerging system on chip architecture are becoming Multi-Processor System on Chip (MPSoC). Designing from scratch a MPSoC with a RTL description is an error-prone and time-consuming process. We believe that an automated way of designing and programming such architectures is essential, in order to decrease the design time and meet the required performance.

The aim of this thesis project is to improve the ESPAM (Embedded System-level Platform Synthesis and Application Mapping) tool, which represents an systematic and automated way of architecture designing and programming. This tool is based on the Kahn Process Network (KPN) [9] model of computation. This research work is focused on the dynamic scheduling of the nodes which comprise the KPN application. This was mainly done by adding an operating system and manage each node of the KPN as a thread.

In the context of automatic application mapping to architecture, several design approaches have been developed by the research community. For instance, Jerraya et al. propose a design flow approach in [4]. This design flow uses a high-level parallel programming model to abstract hardware/software interface in heterogeneous Multiprocessor System on Chip. This work is similar to ESPAM, because both tools generate a multiprocessor system basing on a set of parameterized components as well as communication controllers to connect processors to communication networks. However, the ESPAM tool implements a whole system in much less time, because its design flow is fully automated. By contrast, in [4], many steps of the design have to be performed manually.

Another example of similar design approaches is the Multiflex system, presented in [5]. The target applications of this tool are multimedia and data streaming application. This is also the target of our ESPAM tool. Both tools provide a design space exploration framework, but they are different in the adopted model of computation. While our

tool uses Kahn Process Networks, Multiflex is based on Symmetrical Multi Processing (SMP) model, using shared memory, and distributed system object component (DSOC) object-oriented message passing model. Multiflex does not support at all the automatic derivation of SMP or DSOC, while in our case KPN can be derived in an automated way from a C, C++ code. So, design time includes the manual application partitioning, and it is longer than the ESPAM one.

The Task Transaction Level (TTL), presented in [6], deals with the programmation of embedded multi-processors systems. The ESPAM programming approach is similar, in sense that it works in the context of streaming applications and uses communication primitives. TTL is more flexible because it supports more communication primitives, but the system programmation using TTL is slower because it needs a lot of manual work.

From the comparison with the examples listed above, we might say that the ESPAM tool is more systematic and automated, since it requires less manual modifications. However, developing an automated way of designing and programming a multiprocessor system leads to several problems.

The first one is that most of the applications are specified using a sequential model of computation (e.g. sequential programs written in Matlab or C). This is because, from a programmer point of view, it is easy to develop an application with a monolithic thread of control and a single memory. On the other hand, a sequential specification does not allow to exploit the parallelism available in the application itself.

Currently the porting process, from a sequential complex application to a multiprocessor system, is usually done by hand. This is a very error-prone and time-consuming task, and the final result depends on the designer expertise. To make this porting process in a systematic and automatic way it is essential to use a tool that converts the sequential specification to a parallel one, thus showing the explicit parallelism of the application. For example, the PNGEN tool [11] can convert a sequential code to several concurrent tasks based on the Kahn Process Network model of computation, that is suitable for stream-oriented applications.

Another problem is then how to map automatically this concurrent model of the application in a multiprocessor system. An RTL description can be easily synthesized, in an automatic way, using state-of-art synthesis tools. This is because it is really close to the physical implementation itself. But, as we have discussed above, designing a whole system from scratch with a RTL description is impossible nowadays, because the emerging application have huge complexity, compared to the past. Not only the design of complex systems is error-prone and time-consuming, but also the simulation of such systems is extremely slow and computationally expensive. We believe that the level of abstraction has to be increased, reaching a so-called *System-Level* specification. But this process creates an *Implementation Gap*, between our level of specification and the RTL one.

1.2 Problem description

1.2.1 Closing the implementation gap with ESPAM

ESPAM (Embedded System-Level Platform Synthesis and Application Mapping) is a tool specifically designed to close the *Implementation Gap*, thus converting a System-Level specification to an RTL specification. Capitalizing on this high level of abstraction, a designer can specify an architecture in a short amount of time. The context in which the ESPAM tool is designed is mainly multimedia and stream-oriented applications.

In Figure 1.2 the basic ESPAM design flow is depicted. The desired application has to be described as a Kahn Process Network, that is a network composed by concurrent processes (or nodes) communicating via FIFO channels. This kind of *Application Specification* can reveal the task-level parallelism, and can be derived automatically by using, in our case, the PNGEN tool.

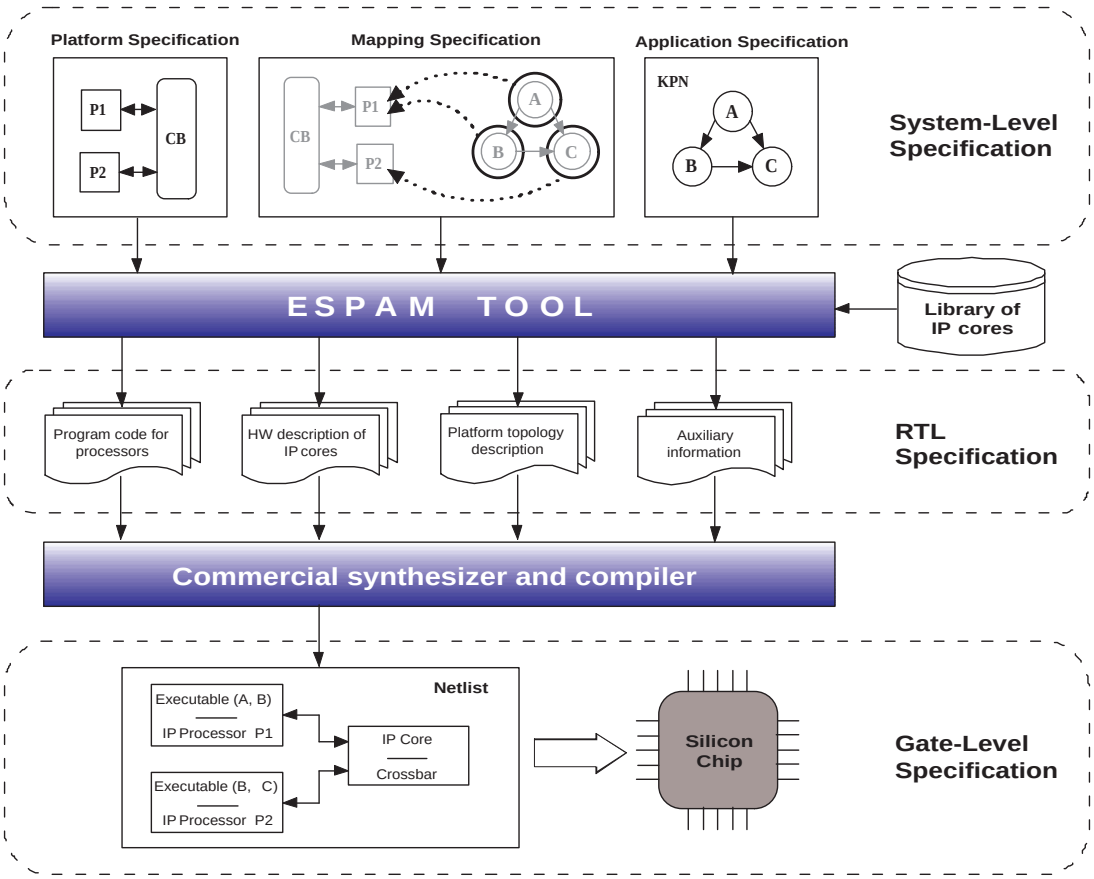

Figure 1.2: ESPAM design flow: starting from a System-level specification of application, platform and desired mapping, ESPAM can generate an RTL description, suitable to be synthesized and compiled with a commercial tool. Finally, the resulting system is downloaded to the target FPGA.

Then, the designer chooses the desired platform structure (with the *Platform Specifica-*

tion file) and maps the nodes of the *Application Specification* to the platform, according to the mapping described in the *Mapping Specification* file. ESPAM allows *one-to-one* mapping (one node per processor) as well as *many-to-one* (more than one node per processor). This process will be described in more details in the next chapter.
When the *Application*, *Platform* and *Mapping Specifications* are provided, the ESPAM tool can automatically generate the RTL specification of the multiprocessor system and the programs that will run on each processor, following these steps:

1. a platform instance is generated, according to the *Platform Specification* file, and checked for consistency (to avoid basic design errors). At this step none of the details of a target physical platform is considered, the platform instance consists of generic parameterized system components;

2. ESPAM refines the platform instance using a library of IP cores, generating a RTL description suitable for the implementation on the target physical board;

3. the program code for each processor is generated, according to the *Application Specification* and *Mapping Specification*.

The output of ESPAM is a RTL specification of the platform, composed by four parts. The *Platform Topology Description* gives a detailed view of the multiprocessor system. The *Hardware Description of IP Cores* contains predefined and custom IP cores used in the Platform Description. The *Program Code for Processors*, for each processor included in the system, in order to execute the application described at System-Level. Finally, the *Auxiliary Information*, which provides tight control on the overall specifications (e.g. precise timing requirements).

This precise RTL description is suitable to be used as input for commercial synthesizer and compiler. In our case, Xilinx Platform Studio [13] is used to convert this RTL description into the bitstream needed to program the target FPGA.

1.2.2 Static scheduling consequences

At this stage of the development, if more than two nodes of the KPN application are mapped on a single computational resource (*many-to-one* mapping on a single processor), the ESPAM tool generates a processor code in which these nodes are strictly **statically scheduled**. For instance, if we consider the scheme depicted in Figure 1.3, nodes 1 and 2 are fired in a precise, fixed order. A possible static schedule can assume that node 1 is fired first (eventually blocking on read/write), then node 2 is fired. This concept can be described with the pseudo-code example shown in Figure 1.4. This scheduling order cannot be changed at run-time.

This static scheduling policy leads to the following consequences:

- It may happen that the execution of one of the nodes is blocked on reading (if the requested input FIFO is empty) or on writing (if no space is available in the desired output FIFO). If another node of the KPN is mapped onto the same

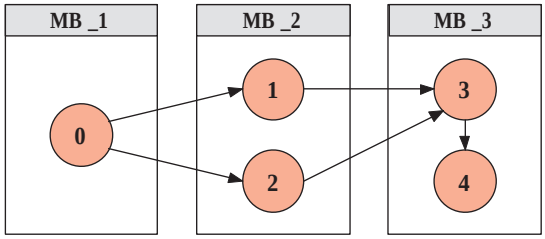

Figure 1.3: Many-to-one mappig example, onto MB_2 and MB_3. The lines connecting different KPN nodes represent FIFO channels, that may lead to read or write blocking.

```
for (n times)
{
   read from node_1 input FIFOs;
   execute node_1;
   write to node_1 output FIFOs;

   read from node_2 input FIFOs;
   execute node_2;
   write to node_2 output FIFOs;
}
```

Figure 1.4: Pseudo-code that describes the static scheduling onto MB_2. Each node is fired n times, but the scheduling order assumes that node_2 must wait the completion of node_1 execution.

processing resource, it could be interesting to find out a method that allows us to implement a dynamic scheduling of KPN nodes. If one of the nodes is blocked, the other one can be fired, reducing in this way the processor time spent on idling. This may also be very useful if we wanted to map more than one application, or multiple instances of the same application, onto one platform.

- In the case of *many-to-one mapping*, if the application is *intrinsically dynamic*, finding a scheduler at compile-time is not possible because the exact order of execution of the nodes is data dependent.

1.3 Solution Approach

The aim of this work is to improve the ESPAM tool by providing a reliable method to implement dynamic scheduling onto our multiprocessor systems. Of course, the case considered is when *many-to-one* mapping is applied (otherwise there is no need of a "local" scheduling). The method we have developed is based on the following steps:

1. **Add operating system to each processor.** We tested a couple of operating systems, in order to make comparisons. The first OS is Xilkernel [15], provided by Xilinx. The second is FreeRTOS [17], which source code and documentation can be found on the internet. We looked for lightweight, simple, and relatively fast operating systems. This is because MPSoCs do not have a lot of memory and sometimes we want real-time behavior.

2. **Divide each processor program code in threads,** that can be executed concurrently. Kahn Process Network semantics is very useful for thread separation because program code is based on intrinsically concurrent processes.

3. **Implement a valid and efficient thread scheduling policy.** Three scheduling policies have been tested: simple interrupt-driven round robin, round robin with yielding on blocking and priority scheduling. Detailed descriptions, implementations and results are presented in the following chapters.

Given a reliable method of dynamic scheduling in a *many-to-one* mapping context, these are the possible new features that a system designer can exploit:

- Intrinsic (such as data-driven) dynamic applications can be implemented onto our multiprocessor platforms.
- Multiple instances of the same application can run in parallel (interleaving). Depending on the application, a higher throughput can be achieved.
- In case of different applications running onto different processors, if one of the applications is significantly more complex than the others, parts (in this case threads) of this application can be transferred onto other processors. In this way the execution time of the different processors is more balanced, leading to better throughput.

1.4 Related work

Our work is mainly related to the previous ones related to the ESPAM tool. Hardware synthesis topics like implementation of heterogeneous and hierarchical architecture and integration of hardware IP cores are presented in [7, 8]. Since applications are getting more and more complex nowadays, some of the tasks of an application have to be executed by dedicated hardware IP cores, because their execution on programmable processors like MicroBlaze or PowerPC could slow down the whole system. Our work is different because it is focused mainly on software improvements.

Furthermore, several works on evaluation of Operating Systems running on MicroBlaze have been presented. A comparison between different operating systems is presented in a master thesis from Swiss Federal Institute of Technology of Lausanne [18], in the context of cryptographic applications running on self-reconfigurable platforms. An evaluation of real-time operating systems such as Xilkernel, Asterix and uClinux is provided in [19], based on interrupt latency and task response time tests. However, none of these works fit to our specific goal, i.e. the evaluation of multi-threading operating systems in the context of MPSoC based on the KPN model of computation.

1.5 Research contributions

These are the main research contributions of this thesis:

- The ESPAM tool has been modified in order to be suitable for dynamic applications and to increase the software design space. We have developed a reliable method to add an operating system to each microprocessor and run different nodes of a KPN application as threads, that can be scheduled dynamically.
- We provide a comparison between two operating systems, namely Xilkernel and FreeRTOS. Several tests have been done, using different applications and platforms, measuring OS footprint and system throughput.

1.6 Equipment

All the experiments have been conducted on the ADM-XRC-II FPGA prototyping board placed in the LIACS laboratory (Leiden Institute of Advanced Computer Science) in The Netherlands. This prototyping board, developed by Alpha Data Parallel Systems Ltd [20], was used to implement the MPSoCs generated by ESPAM. Its scheme is depicted in Figure 1.5. The board is connected to the PCI bus of a Pentium processor. Six Zero Bus Turnaround external memory banks can be accessed from this external processor, with the infrastructures built by Wei [7], in order to set the input data of the multiprocessor platform. Then, the system processes these data and puts the results into the external memory banks. Finally, the external processor reads the content of the destination memory bank and can save/display results to verify correctness.
All memory accesses from/to the external processor use the PCI interface shown in the left side of Figure 1.5.

These are the specifications of the ADM-XRC-II FPGA prototyping board:

- High performance PCI and DMA controllers
- Local bus speeds of up to 66MHz
- Six banks of 256k/512kx32/36 ZBT SSRAM
- User clock programmable between 0.5MHz and 100MHz
- User front panel adapter with up to 146 free IO signals
- User rear panel PMC connector with 64 free IO signals
- Supports 3.3V and 5V PCI signaling levels (VI/O)

As described earlier, the communication with the outside world is provided by the six ZBT (Zero Bus Turnaround) SSRAM memories represented in the right side of Figure 1.5. Zero Bus Turnaround means that zero clock cycles are spent for turnaround, transitions from write to read or viceversa.

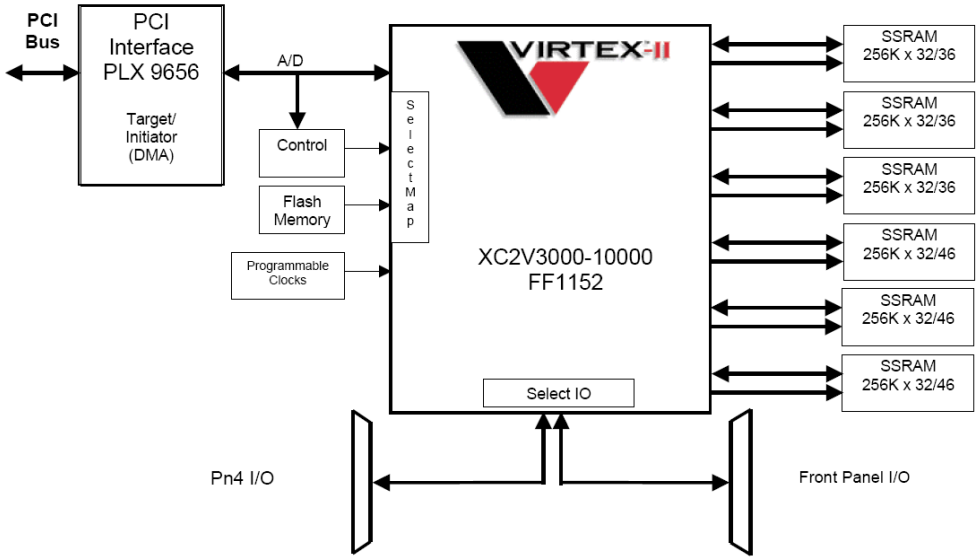

Figure 1.5: Scheme of the ADM-XRC-II FPGA prototyping board.

1.7 Thesis organization

The rest of this thesis is organized as follows:

Chapter 2 describes our system design methodology, focusing on main features and basis of the ESPAM tool. First, this chapter presents the KPN model of computation on which our Application Model is based, and the tool that can derive automatically this specification from a sequential program. Second, it introduces the platform model and synthesis with the ESPAM design flow, followed by the program code generation for each processor. Finally, how Xilinx Platform Studio (XPS) can use the output of our tool, for hardware synthesis and software compilation. The output of XPS is a bitstream that can be directly downloaded to program the target FPGA.

Chapter 3 presents the basic implementation concepts on which our solution approach is based. It starts from the usefulness of dynamic scheduling, when dealing with intrinsic dynamic application, multiple application or multiple instances of the same application. Then it provides the definition of thread and the introduction to Pthread standard for thread management and its Application Programming Interface. Finally, it describes the thread scheduling policies we have tested, namely Round-robin, Round-robin with yielding on blocking, and priority scheduling.

Chapter 4 introduces the two application we used to test our dynamic scheduling implementations. The first example is the Sobel edge detection algorithm and the second is an M-JPEG encoder. Both of these applications have been mapped onto a five-processor system, using the standard ESPAM design flow. The obtained results will be used to make performances comparison with the platforms presented in Chapter 5, which are based on dynamic scheduling of the nodes.

The Xilkernel operating system is presented in Chapter 5. This chapter describes also how the scheduling concepts introduced in Chapter 3 are implemented using this OS.

First, the modifications common to all different scheduling policies are listed. Then, for each of the different solutions, a more detailed description of required implementation steps is provided. The chapter finishes with the description of some advanced examples of dynamic scheduling implementations on multiprocessor systems, like M-JPEG and Sobel applications or multiple instances of Sobel application mapped on the same platform.

Chapter 6 deals with the FreeRTOS introduction and implementation details. One of the system examples in Chapter 5 is reproduced using FreeRTOS, in order to make comparisons on performances and memory occupation overhead.

A brief "getting started" tutorial on dynamically scheduled system design is provided in Chapter 6, starting from the C application code to the actual physical implementation of the system. All the PNGEN and ESPAM tool operations, and manual modifications, are listed.

Chapter 8 presents the final conclusions, based on test results. Some future works, that can improve the results obtained in this thesis, are also described.

Chapter 2

Embedded System-level Platform synthesis and Application Mapping

This chapter provides a more detailed description of the ESPAM (Embedded System-level Platform synthesis and Application Mapping) tool. In addition to the information written in the first chapter, we give a simple description of the Kahn Process Network model of computation and the basic ideas of the PNGEN tool, that translates the initial sequential application into a KPN application. Then, we present the main concepts of how the platform and the code for each processor is generated and how Xilinx Platform Studio can be used as back-end of our design flow.

2.1 Derivation of Kahn Process Networks

As Figure 1.2 shows, the ESPAM tool needs an application description different from the sequential one. For the programmer's point of view, using a sequential description (e.g. Matlab or C, C++) is much easier, but this kind of description is extremely hard to map automatically onto a multiprocessor system because the thread-level parallelism is not explicit. Of course a designer can try to map an application by hand, but this process is very slow and can lead to errors in the implementation. We believe that a correct-by-construction process is the way to make platform synthesis and application mapping easier and faster.

For the ESPAM tool, the Kahn Process Network [9] model of computation was chosen because it is parallel and allows distributed memory and distributed control. Furthermore, its operational semantics are simple, but general enough for multimedia or signal processing applications.

2.1.1 Kahn Process Networks description

In Figure 2.1 a simple example of a KPN is depicted. The network has a set of nodes ($P1$, $P2$, $P3$) that represents different processes. The KPN model of computation

assumes that processes are concurrent and autonomous. The communication is done by a point-to-point connection with unbounded FIFO channels.

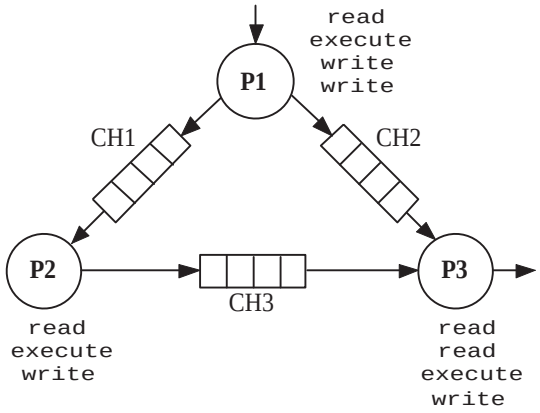

Figure 2.1: Simple example of a KPN. Next to each node the corresponding computation and communication primitives (reading, executing and writing) are presented. These primitives describe the node behavior.

Each node of the network is described by sequential code, executed concurrently to the other nodes. For example, in Figure 2.1 process *P1* first reads a token from its input port, then executes the computation and finally writes data to the other processes, *P2* and *P3*, through *CH1* and *CH2* respectively. In the meanwhile, if *P2* and *P3* have tokens in their input FIFOs, they can continue their execution concurrently. The execution order of the different nodes is determined only by the channel through which the processes communicate. If a node has data to read and free space to write, its execution will continue.

The KPN model of computation offers the following features:

- Regardless to the scheduling order of the different nodes, the final result is the same, i.e. the KPN model is deterministic. This allows us to exploit different scheduling policies when mapping processes to hardware or software.
- The inter-process communication is synchronized by a blocking read. This is a very simple primitive, easily implementable in hardware or software.
- There is no global scheduler, since control is completely distributed to individual processes. So, dividing a KPN over different reconfigurable components is a simple task.
- There is no notion of global memory. Each inter-process communication is distributed over FIFOs, so there is no risk of resource contention.

These features make easy the mapping of a KPN application specification onto a multiprocessor platform and this model of computation matches very well our system design methodology.

2.1.2 The PNGEN tool

Typically, applications are specified with a sequential language such as Matlab or C, C++. This is why this way of application specification is easy for application designers, since there is only one thread of control and a single memory. On the other hand, a sequential specification does not reveal the available parallelism in the application itself. So, a parallel specification is needed if we want to capitalize the resources of a multiprocessor system.

Describing an application directly using a parallel model of computation is a very difficult, time-consuming and error-prone process. The aim of the PNGEN tool [11] is to close this gap between a sequential specification and a parallel model of computation.

The input given to the PNGEN tool must be a SANL (Static Affine Nested Loop) program, made by a set of statements, each possibly enclosed in loops and/or guarded by conditions. All of these conditions and statements must be affine, linear expressions of iterators and parameters. For each function call of the sequential program a node the PNGEN tool generates a KPN node.

The output of the PNGEN tool is a process network specified with XML code, compatible with ESPAM Application Specification. So, this tool can be used as a front-end for ESPAM. In the implementation of the process network, channels must be implemented as bounded FIFOs, thus blocking on write can also occur. To avoid deadlocks, the PNGEN tool determines also FIFO channel minimum sizes. Edges in the process network correspond to variables shared between the function calls. Figure 2.2 depicts an input-output example of the tool.

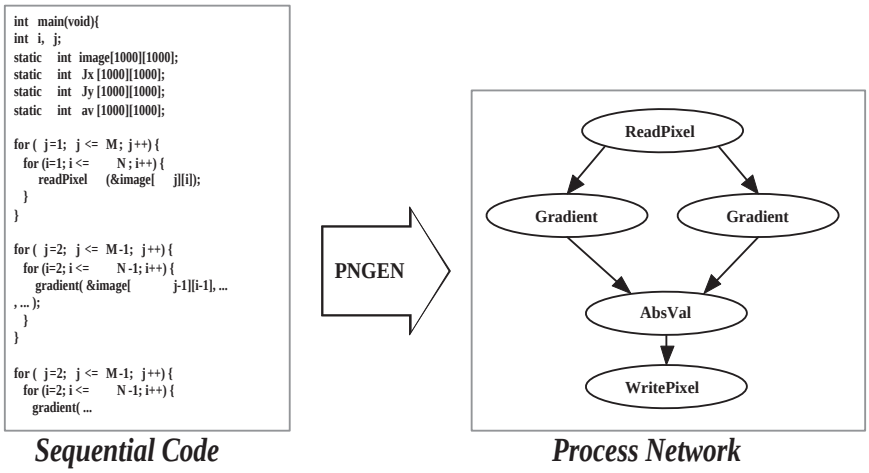

Figure 2.2: Example of input and output of the PNGEN tool. Starting from a sequential program, a KPN specification of the application can be automatically generated. The complete program code of the sequential application is included in Appendix A.

2.2 Platform model

The platform model of our tool is a library of generic parameterized components. This set of components has to be flexible enough to allow a good design space variety. The platform model includes:

1. Processing Components: programmable processors as well as dedicated programmable hardware [7,8] can be used as processing resources of our platforms. In this project the only processing components used are MicroBlaze processors, since we use Xilinx Virtex-II FPGA as physical platform technology.

2. Memory Components: these components are used to describe data and program memories connected to each processor as well as buffers that connect different processing components of the system. The communication between program and data memory and the processor is controlled by a memory controller. The communication memory components are implemented using dual-port memories, and they are organized as one or more FIFO buffers.

3. Communication Components: several ways of communication are included in the ESPAM platform model. The most efficient option for a multiprocessor system is a point-to-point network, but also a crossbar switch and a shared bus are available.

4. Communication Controllers and Links: communication controllers are needed to synchronize the communication between different processing resources, at hardware level. Links are used to connect any two components of our platform model.

For each of these components, many parameters can be set: type, I/O ports, speed for processors; memory sizes and types, etc.

2.3 ESPAM Design flow

The ESPAM design flow, introduced in the first chapter, is depicted in Figure 1.2. The input given to the tool is a System-level description made by three entries, written in XML format:

1. **Application Specification.** As mentioned in Section 2.1, the application is described as a Kahn Process Network, that can be derived automatically by the PNGEN tool.

2. **Platform Specification.** This file describes the platform topology at a very high level of abstraction. The designer does not need to specify memory structures, interface controllers and communication protocols: the tool itself takes care of this task. An example of Platform Specification is shown in Figure 2.3. In the

first part, lines 2-5, four processors are instantiated. The crossbar communication component is included with lines 7-12. Finally, links are specified with lines 14-29. These links are needed to connect each processor to the communication component.

```
1  <platform name ="myPlatform">
     <processor name = "uP1"> <port name = "IO1"/> </processor>
     <processor name = "uP2"> <port name = "IO1"/> </processor>
     <processor name = "uP3"> <port name = "IO1"/> </processor>
5    <processor name = "uP4"> <port name = "IO1"/> </processor>

     <network name = "CB" type = "Crossbar">
       <port name = "IO1"/>
       <port name = "IO2"/>
10     <port name = "IO3"/>
       <port name = "IO4"/>
     </network>

     <link name = "BUS1"/>
15     <resource name = "CB" <port name = "IO1"/>
       <resource name = "uP1"<port name = "IO1"/>
     </link>
     <link name = "BUS2"/>
       <resource name = "CB" <port name = "IO2"/>
20     <resource name = "uP2"<port name = "IO1"/>
     </link>
     <link name = "BUS3"/>
       <resource name = "CB" <port name = "IO3"/>
       <resource name = "uP3"<port name = "IO1"/>
25    </link>
      <link name = "BUS4"/>
        <resource name = "CB" <port name = "IO4"/>
        <resource name = "uP4"<port name = "IO1"/>
      </link>
30  </platform>
```

Figure 2.3: Example of Platform Specification, written in XML format.

Starting from this Platform Specification, the ESPAM tool generates an elaborate platform as follows. First, the tool puts processing and communication components into the design. Second, it attaches program and data memories and respective memory controllers to each processor. The size of these memories is determined by the values set within the Platform Specification file. Third, based on the type of processors considered in the first step, it synthesizes, instantiates and connects all necessary communication controllers and communication memories for data communication between the processors.

An example of such detailed platform is shown in Figure 2.4. Communication Controllers (CC) connect a communication memory (CM) to the communication component - in this example, the crossbar (CB) - and to the processor (uP) it belongs to. Each CC provides the communication between processor and CM for write operations, obeying processor's local bus access protocol. CC are also needed to access the communication component (CB) for read operations. The synchronization between different processors is implemented using blocking read/write on the FIFO buffers included in Communication Memories.

3. **Mapping Specification.** This entry defines which node of the KPN application will be assigned to each processor. Two types of mapping are supported:

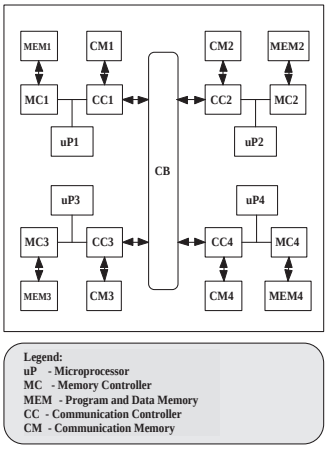

Figure 2.4: Detailed platform, generated by ESPAM using the Platform Specification file of Figure 2.3.

many-to-one or *one-to-one*. As depicted in Figure 2.5, *one-to-one* implies that the number of processors is equal to the number of nodes in the KPN, so each processor will take care of the execution of just one node. *Many-to-one* mapping is required when the number of processors is less than the number of nodes in the KPN. In such a case, the inter-process communication between nodes mapped onto the same processor is still provided by external FIFOs rather than shared memory.

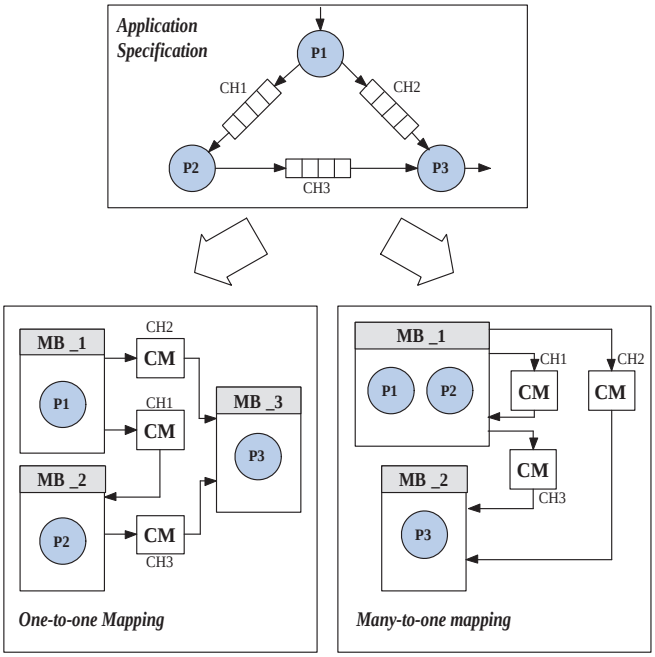

Figure 2.5: Basic example of *one-to-one* and *many-to-one* mapping (data and program memories are not represented).

2.4 Automated programming of multiprocessor platforms

An important step of our ESPAM tool is the automated generation of program code for each processor. Both C and C++ are supported. In our case, we can use MicroBlaze GNU tools such as *mb-gcc* compiler, *mb-as* assembler and *mb-ld* loader-linker to generate ELF executable files from these sources. Each processor must be programmed following this concept: its behavior must match the one of the node(s) it represents.

The program for each processor includes control code and computation code. The computation part performs the specific operations of the mapped KPN node on the received data. The control code, based on *for*-loops and *if*-statements, decides how many times data reading and writing, as well as computation, must be performed. A simple example of program code is shown in Figure 2.6. If more than one node is mapped onto the same processor, the tool determines a valid schedule to avoid deadlocks.

```
1    #include ‘‘primitives.h’’
     #include ‘‘MemoryMap.h’’

     struct myType{
5      bool flag;
       int data[64];
     };
     int N=384;

10   void main(){
       myType in_0;
       myType out_0;

       for (int k=0; k<N; k++){
15       read(p2, &in_0, sizeof(myType));
         compute(in_0, out_0);
         write(p1, &out_0, sizeof(myType));
       }
     }
```

Figure 2.6: Simple example of processor program code. In line 15 the program performs a read request: *sizeof(myType)* bytes are transferred from port *p2* to the variable *in_0*. In line 17 there is a write primitive, following a similar semantic. Data computation is executed in line 16. This sequence of read, execution and write is executed *N* times (see *for*-loop in line 14)

2.4.1 Software communication primitives

The software communication primitives generated by ESPAM basically implement the blocking read/write synchronization. Blocking on write is necessary because in the physical implementation, buffers can not be unbounded. A processor can access a FIFO as two memory locations in its address space. The first is used to read and write data; the second is a “status” location, where a flag is set if the FIFO is full (no more space available for writing) or empty (no data to read). The tool automatically generates correct-by-construction FIFO addresses.

As shown in Figure 2.7 (lines 8 and 17), blocking is implemented by empty *while*-loops, in which the condition is set by the FIFO status. The parameters of the *for*-loops

(lines 6 and 15) determine how many bytes are written to or read from the input/output channels.

```
1   #ifndef _PRIMITIVES_H_
    #define _PRIMITIVES_H_

    void read(byte* port, void* data, int lenght){
5     int *isEmpty = port+1;
      for(int i=0; i<length;i++){
        //reading is blocked if a FIFO is empty
        while(*isEmpty){}
        (byte* data)[i]=*port; //read from a FIFO
10    }
    }

    void write(byte* port, void* data, int lenght){
      int *isFull = port+1;
15    for(int i=0; i<length;i++){
        //writing is blocked if a FIFO is full
        while(*isFull){}
        *port=(byte* data)[i]; //write to a FIFO
      }
20  }
    #endif
```

Figure 2.7: Reading and writing primitives used by ESPAM.

2.5 Xilinx Platform Studio

As depicted in Figure 1.2, the ESPAM tool can generate a very detailed RTL description of a multiprocessor system. This low-level description is suitable to be used as input for a commercial synthesizer/compiler in order to produce the final bitstream that programs the FPGA of the prototyping board. In our examples, this back-end tool is Xilinx Platform Studio (XPS) [13], that will be described briefly in the this section.

Xilinx Platform Studio (XPS) is a graphical user interface that integrates all of the processes from design entry to design debug and verification in the context of the Embedded Development Kit (EDK). EDK is a series of software tools for designing embedded processor systems on programmable logic, and supports the IBM PowerPC hard processor core and the Xilinx MicroBlaze soft processor core. In this thesis we have used only the MicroBlaze soft processor [24], that is a reduced instruction set computer (RISC) optimized for implementation in Xilinx FPGAs. Since MicroBlaze is a soft processor core, the number of this kind of processors in our systems is only limited by the resources of the target FPGA. The MicroBlaze core is highly configurable, allowing users to choose the features that better fit their design.

XPS can be used to create hardware systems from the scratch, and provides a complete library of IP cores. Custom IP cores can be added to a design as well, and system designers can also write applications for each processor of the platform. However, designing and programming a multiprocessor system using only XPS may lead to errors in the implementation and for sure it is a very time-consuming process. Our ESPAM tool takes care of this task, generating all necessary files for a XPS project, starting from the System-level specification shown in Figure 1.2.

2.5.1 XPS project suite generation

An example of the project suite files, generated by ESPAM according to the input requirements of XPS, is presented in Figure 2.8.

```
<PROJECT_SUITE>
|--- system.xmp
|--- system.mhs
|--- system.mss
|--- code/: software program code
|---------- aux_func.h
|---------- MemoryMap.h
|---------- P_1/: program code for processor P_1
|---------------- P_1.cpp
|---------- P_2/: program code for processor P_2
|---------------- P_2.cpp
|--- etc/: optional files for implementation tools
|--------- bitgen.ut
|--------- bitgen_spartan3.ut
|--------- fast_runtime.opt
|--------- download.cmd
|--- data/: UCF files
|---------- system.ucf
|---------- system_ADMXRCII.ucf
|---------- system-default.ucf
|---------- system-zbt.ucf
|--- pcores/: customized IP cores for the EDK project
|------------ buffers_v1_00_a/
|------------ cb_wrapper_v1_00_a/
|------------ fifo_if_ctrl_v1_00_a/
|------------ fin_ctrl_v1_00_a/
|------------ host_design_ctrl_v1_00_a/
|------------ LMB_VB_CTRL_v1_00_a/
|------------ mux_v1_00_a/
|------------ myCLKRST_v1_00_a/
|------------ opb_zbt_controller_v1_00_a/
|------------ VB_Wrapper_v1_00_a/
|------------ zbt_main_v1_00_a/
```

Figure 2.8: XPS project suite automatically generated by our ESPAM tool.

Starting from the top of the project suite of Figure 2.8, these are the description of the included files and folders:

- **system.xmp** The Xilinx Microprocessor Project (XMP) is the top-level project file for an EDK design. It includes options like the version number, the location of MHS and MSS files, the FPGA architecture family and software settings for the project.

- **system.mhs** The Microprocessor Hardware Specification (MHS) file describes which components are used in the project and specifies the connections between these components. A typical entry of an MHS file is shown in Figure 2.9, in which a MicroBlaze processor is instantiated, customized (setting the hardware version and some parameters) and connected to its buses (*DBUS_MB_1* and *PBUS_MB_1*) and to the system clock (*sys_clk_s*). A MHS file defines the configuration of our multiprocessor system, and includes bus architecture, peripherals, processors, connections and address space.

```
BEGIN microblaze
 PARAMETER INSTANCE = MB_1
 PARAMETER HW_VER = 4.00.a
 PARAMETER C_NUMBER_OF_PC_BRK = 1
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
 PARAMETER C_FSL_LINKS = 0
 BUS_INTERFACE DLMB = DBUS_MB_1
 BUS_INTERFACE ILMB = PBUS_MB_1
 BUS_INTERFACE DOPB = mb_opb_1
 PORT CLK = sys_clk_s
END
```

Figure 2.9: Part of an MHS file, in which a MicroBlaze processor is instantiated, customized and connected to relevant signals.

- **system.mss** The Microprocessor Software Specification (MSS) file includes directives for drivers, libraries and operating systems that the project needs. The MSS file is closely related to the MHS file. For example, every periferal instance in the MHS implies a correspondent driver instance in the MSS. An example of an entry defined in an MSS file is presented in Figure 2.10

```
BEGIN OS
 PARAMETER OS_NAME = standalone
 PARAMETER OS_VER = 1.00.a
 PARAMETER PROC_INSTANCE = MB_1
END

BEGIN PROCESSOR
 PARAMETER DRIVER_NAME = cpu
 PARAMETER DRIVER_VER = 1.01.a
 PARAMETER HW_INSTANCE = MB_1
 PARAMETER COMPILER = mb-gcc
 PARAMETER ARCHIVER = mb-ar
END
```

Figure 2.10: Part of an MSS file, related to the hardware instance of *MB_1* MicroBlaze processor of Figure 2.9. The first part defines the OS, which is *standalone* (i.e., no OS is used onto this processor). In the last part driver, compiler and archiver for *MB_1* are specified.

- **code folder** This folder includes the program code for each processor and two important header files, namely *aux_funct.h* and *MemoryMap.h*. These are files common to all processors. The *aux_funct.h* file provides read/write primitives as well as wrappers for all the function calls of the initial application. The *MemoryMap.h* file defines the addresses of some components of the platform like data and program memories, communication memories and external memory controllers.

- **etc folder** Contains optional file related to system physical implementation.

- **data folder** In this folder several User Constraint Files (UCF) are stored, each one for a different target FPGA board. UCF specifies constraints related to a specific FPGA device, such as pin location, timing, and input/output standards.

- **pcores folder** This directory stores the customized IP cores for the EDK project. These IP cores are included in the library of components shown in Figure 1.2.

The project suite generated by ESPAM and described above is fully compatible with Xilinx Platform Studio. A few manual modifications are needed, as described in Chapter 7. Xilinx Platform studio provides backward compatibility, thus even the latest version of XPS is able to synthesize ESPAM multiprocessor systems. When importing a project, an automatic XPS wizard performs the necessary IP cores upgrade.

Chapter 3

Application of Multi-threading concepts in ESPAM

This chapter describes the solution approach that we have adopted. In case of *many-to-one* mapping on a processor, the program code is divided in concurrent threads, that can be scheduled in a dynamic way. The first section discusses about the usefulness of dynamic scheduling when dealing with an intrinsic dynamic application, multiple application or multiple instances of the same application. Then, the second section introduces the definition of thread and describes the Pthread standard for thread management and its Application Programming Interface. Finally, the third section shows which scheduling policies we have tested, namely Round-robin, Round-robin with yielding on blocking, and Priority scheduling.

3.1 Usefulness of dynamic scheduling

The first consideration that may come in mind is why a dynamic scheduling can be more useful than the simple static scheduling, automatically generated by the ESPAM tool. Some of the possible scenarios, in which a dynamic scheduling can be more useful or even compulsory are listed below.

1. **Intrinsic dynamic application.** In the context of *many-to-one* mapping onto a multiprocessor platform, intrinsic dynamic application can be mapped only if dynamic scheduling is implemented. Let us consider the platform presented in Figure 3.1. If the scheduling is static, the node *P2* is fired n_2 times, then node *P3* is fired n_3 times, and so on, until the end of the computation. If the application is data-dependent, process *P1* does not send data through channels *CH1* and *CH2* in an order known at compile time. Thus, it may happen that process *P2* is blocked on read (there is no data in *CH1*) and process *P1* will not send data through channel *CH1* anymore. In this situation, node *P3* will also be blocked forever, since the firing order cannot be changed. This leads to a deadlock of the system. In case of intrinsic dynamic application, static schedule cannot be found

at compile time.

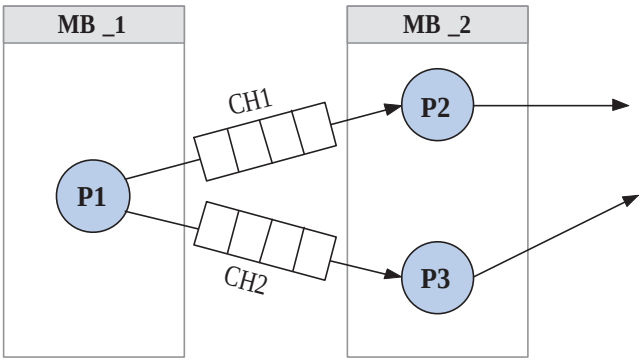

Figure 3.1: Simple *many-to-one* example on processor *MB_2*. Each channel is implemented by one or more FIFO(s) in our systems.

2. **Multiple applications running on the same platform.** With a dynamic scheduling approach, multiple applications can be mapped onto the same platform. In Figure 3.2 a symbolic example of this concept is depicted. Let us consider the execution on *MB_3*. It may happen that *P_3* and *P_4* are blocked in reading, waiting for data from node *P_2*. In this case, processor time is wasted if they are the only nodes mapped onto *MB_3*.

   If, on the same processor, another node of a different application is mapped (in the shown example, *P_7*), we can exploit the processor time spent in idling by executing this additional process. However, determine a static scheduling of the nodes mapped on each processor would be *at least* a vary complicated task, since it would imply a combined dataflow analysis of the two applications. In same cases it could be impossible, for instance when at least one of the applications is dynamic.

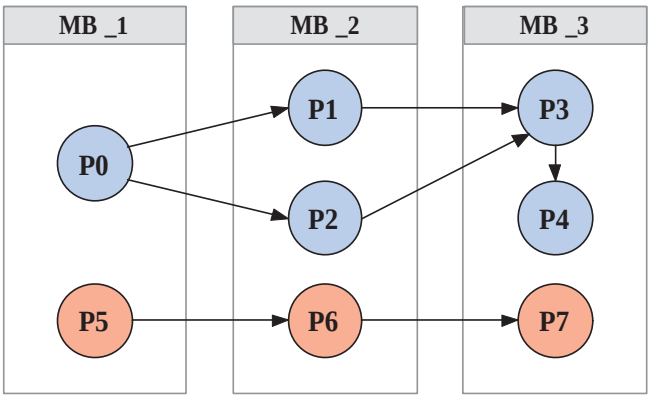

Figure 3.2: Symbolic example of multiple applications running onto the same platform. Each arrow represents one or more FIFO channel(s).

A similar scenario arises when multiple instances of the same application is mapped on the same platform. The aim of this solution is to fully exploit the processor time, removing the time spent in idling.

3. **Execution time balancing of different applications.** Let us consider Figure 3.3(a), where two applications are mapped onto two separate processor. If one of the two applications is much more complicated than the other (naively we can assume that the number of processes is proportional to application complexity), the execution finishes much earlier in the low-loaded processor. Therefore, we can put some of the complexity (in this case, some of the nodes) of the heavy application on the other processor (as shown in Figure 3.3(b)). This improves the throughput of the whole system because both application will finish in less time than the slowest application of Figure 3.3(a). This is why the execution time of the two applications is more balanced.

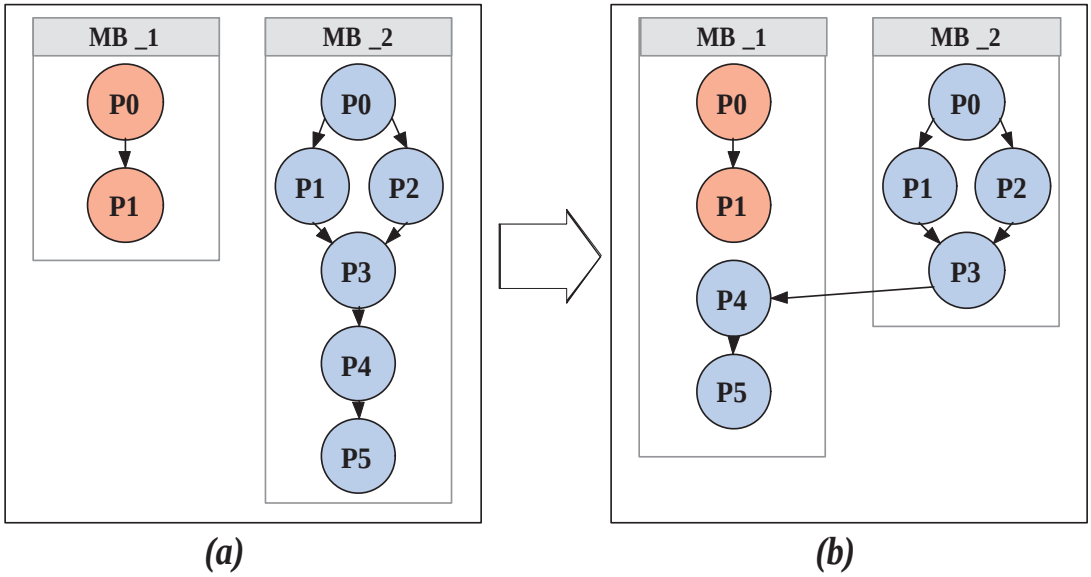

Figure 3.3: Symbolic example of two applications, very different in complexity, running onto two separate processors. The complex application is executed in less time in the second case (b).

3.2 Multi-threading

Since multi-threading is the basis on which we have developed our method of dynamic scheduling for KPN processes, some definitions (taken from [21, 22]) are needed to describe our solution approach.

Threads are generally part of a process, that is an instance of a program sequentially executed by a processor.

Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. From an application developer point of view, a thread can be a procedure that can run independently/concurrently from the main program. Usually a thread is defined within a process, which is created by the operating system and requires a fair amount of overhead, such as information about program resources and program execution state. Some of these informations may

be process ID, user ID, group ID, working directory, program instructions, registers, program counter, stack pointer, heap, file descriptors and much more.

Threads use and exist within these process resources, yet they are able to be scheduled by the operating system and run as independent entities, largely because they duplicate only the essential resources that enable them to exist as executable code: program counter, stack pointer, registers, scheduling properties and few more. This means that managing threads is faster than managing tasks, because of the different overhead. Multi-threading generally occurs by time-division multiplexing ("time slicing") on single-processors systems, while on multi-processors or multi-core systems threads can be executed literally in parallel. Time division multiplexing implies that some of the processor time is spent in context switching, namely saving informations of the running process and restoring the informations of the next process.

Historically, hardware vendors have implemented their own version of threads. To allow portability of threaded applications, a standard programming interface was required. For UNIX systems, this standard is referred to as POSIX threads, or Pthreads. This is the standard used in Xilkernel, the first OS we have tested on our platforms. In this operating system, as in many other lightweight OS, there is no concept of thread groups combining to form a process and scheduling is done at the process context level. So, in this thesis we will use the concept of a thread and of a process as synonyms.

3.2.1 Thread-safeness

Since several threads can share the same resources, for example a piece of global memory, and they can be executed in parallel with time division multiplexing, the design of an application must avoid data corruption or creation of race conditions.

For example, let us consider Figure 3.4. Each of the threads created by the main application calls the same library routine. This routine accesses/modifies a global memory location, so it may happen that all of the threads try to modify this memory location at the same time, leading to data corruption. What this application needs is a way of synchronization/communication between threads, to avoid multiple contemporary access of the same memory location.

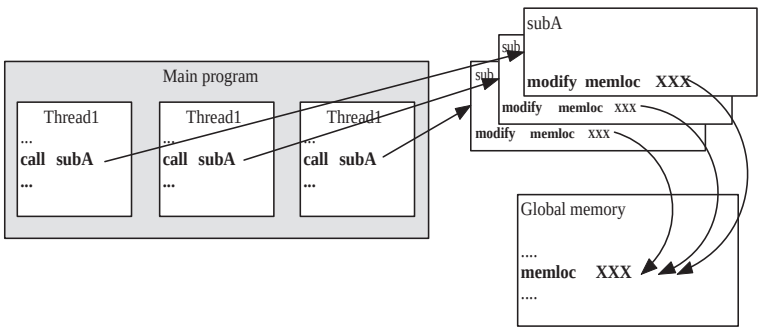

Figure 3.4: Example of data corruption when dealing with threads.

3.2.2 Pthread API

The Pthread Application Programming Interface is a very well-known standard for thread manipulation in UNIX systems and it is also supported by Xilkernel OS. An overview of the subroutine that this API provides is given below, grouped in three classes.

Thread management.
This class of functions works directly on threads. Users can create a thread and set/query thread attributes (joinable, scheduling etc.). Important functions of this class are:

- `pthread_create` creates a new thread and makes it executable. This routine can be called any number of times from anywhere within the code. The arguments passed to this routine are a thread identifier for the new thread, thread attributes, the first function that the new thread will execute once it is created and a single argument that can be passed to the first function called by the thread. The maximum number of threads that may be created is application dependent. Once created, every thread can create new ones.
- `pthread_attr_init` and `pthread_attr_destroy` are used respectively to initialize and destroy the thread attribute object.
- `pthread_exit` is used to explicitly exit a thread, typically when its work is completed.
- `pthread_join` is a way of synchronization between threads. This function blocks the calling thread until the thread specified in the function argument terminates.
- `pthread_yield` forces the calling thread to stop and yield the processor to another thread. The calling thread waits in the ready thread queue before it is scheduled again.

Mutexes. This class of functions manages a kind of synchronization between threads, called mutex, which is an abbreviation for mutual exclusion. A mutex variable acts like a lock, protecting accesses to a shared memory resource. The basic concept is that only one thread can own a mutex variable in a given time. No other thread can own the mutex until the owning one unlocks it. So, threads must take turns to access protected data. Mutexes can protect data corruption, like the one explained in 3.4.

Mutex-related functions can create, destroy, lock and unlock mutexes and also set and modify their attributes.

Condition variables. These variables provide another way of thread synchronization, implemented by controlling the actual value of a data. Without condition variables, users would need to have threads continually polling the data to check if the condition is met. Of course this is a very time-consuming process.

3.2.3 Lightweight multi-threading OS in ESPAM

The solution approach to our problem, stated in Section 1.3, must assure performance, in terms of application execution time. This is because in some cases we want real-time behavior. Also, since usually MPSoCs do not have a lot of memory, we want small overhead on system resources, mainly the memory occupation on the FPGA.

Performances, in the context of a KPN application running on a multi-threading OS, depends mainly on the OS speed in context switch and thread management, and on synchronization between threads. The concepts regarding how we implemented thread synchronization and scheduling are described in the next section. Since we did not modify the kernels of the tested operating systems, context switch time and thread management speed are given and depend only on the operating system design.

The memory footprint of an operating system depends on its design and on its scalability, namely how much an OS can be customized to fit user requirements.

What really helped us in developing multi-threading application for our platforms is the model of computation on which ESPAM is based. In fact, Kahn Process Networks are based on concurrent and autonomous processes, with no notion of a global memory. This means that in our implementation we did not have to use complicated synchronization strategies like mutexes or condition variables, since no data corruption may happen. This is because each KPN node writes to separate FIFO channels and uses its local memory space for data.

Also, since in our platforms different nodes of the same application may be mapped onto different processors, inter-process synchronization cannot be implemented with mutexes or condition variables, because different processors do not share the same global memory.

3.3 Thread scheduling policies

In our solution approach we have tested three methods of thread scheduling, which are largely dependent on the operating system scheduler. Generally, in an operating system, thread entries are stored in queues.

If the adopted scheduling mechanism is Round-robin, there is only one queue for ready (not blocked by the OS) threads, as depicted in Figure 3.5. When the scheduler starts, the first thread in the queue is executed and takes control of the processor. Then, using time division multiplexing, the processor is shared with the other threads. At the end of a certain time slice, the first thread yields the control to the second of the queue, and so on. In the case of a MicroBlaze processor, the end of each time slice is signaled by an interrupt generated by an external timer.

When the processor control is yield from a thread/process to another one, a context switch has to be performed. This means that the state of the first process must be saved somehow, so that, when the scheduler gets back to the execution of the first process, it can restore its state and continue. The state of the thread/process includes all the

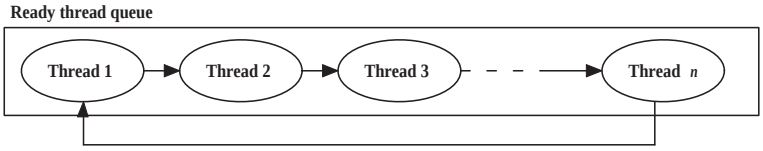

Figure 3.5: Example of ready queue for Round-robin scheduling. The processor control is shared between threads with time division multiplexing.

registers that the thread/process may be using, like the program counter, stack pointer and others. Often all these data are stored in one data structure, called *process control block*.

If the operating system adopts a priority-based scheduling, more than one ready queue is provided, as shown in Figure 3.6. The scheduler decides which thread/process has to be fired basing on the priority associated to each queue. Threads in higher-priority queues are fired first. Threads with the same priority are Round-robin scheduled.

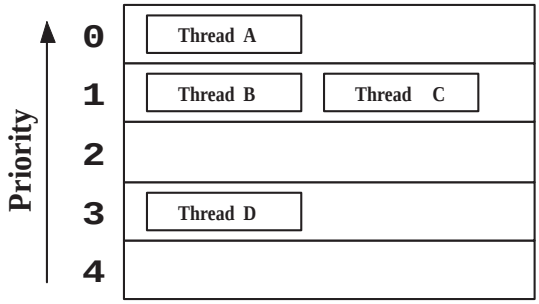

Figure 3.6: Example of ready queues for priority scheduling. Threads in higher-priority queues are always fired first, and threads with the same priority are Round-robin scheduled .

In our solution approach we add a lightweight operating system to our processors, then we derive for each node of the KPN a thread that represent its behavior. Finally, to achieve better performances, we tested three methods of thread scheduling, which are based on the kernel scheduler and on the API of the operating system. These three methods are described below.

1. **Simple Round-robin.** The simple Round-robin scheduling is already implemented by both of the OS we have tested. Switching between threads is driven by an interrupt signal, provided by an external timer. This solution is very simple, and requires no OS enhanced features. As depicted in Figure 3.7, since the interrupt is sent at constant intervals, every process owns the processor control for the same amount of time (the time slice is constant).

   Though this solution is very easy, the behavior of a KPN does not match perfectly this kind of scheduling. It may happen that some of the nodes block much earlier than the end of the time slice. A graphical example is shown in Figure 3.8. From the moment in which the thread is blocked, until the end of the time slice, the processing resources can be wasted, if the condition of input or output channels

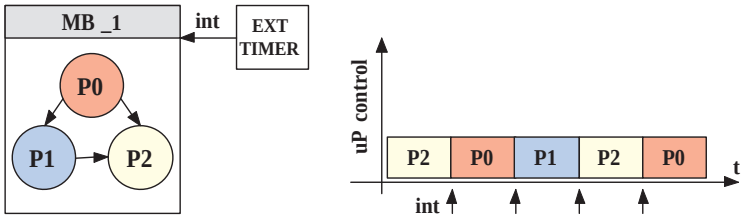

Figure 3.7: Example of Round-robin scheduling of threads. Each thread owns the processor control for the same amount of time. For MicroBlaze processors, the interrupt which drives thread switching is provided by an external timer.

do not change. All this wasted time is represented with shadow areas in the picture.

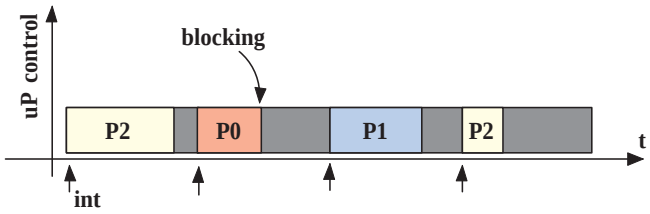

Figure 3.8: Example of Round-robin scheduling of threads that may possibly block. Switching between threads is driven only by the interrupt signal, represented by little arrows in the bottom of the figure. If one thread blocks much earlier than the end of its time slice, a lot of time is spent in idling.

2. **Round-robin with yielding on blocking.** This solution is also based on Round-robin scheduling, but with a little modification. When the current thread is blocked, we force the scheduler to switch directly to the next thread in the ready queue. In this way, if the next thread is not blocked, no processor time is wasted on waiting for data. Also, we give time to the nodes that communicate with the current one to unblock it (filling its input FIFOs with data to be processed, or emptying its output FIFOs). As depicted in the example of Figure 3.9, this solution leads to a higher throughput of the KPN.

   For this solution two mechanisms force thread switch. The first is the blocking of the current thread. The second is the interrupt tick. This last mechanism may not be compulsory for our implementation, but it is useful to avoid that one thread monopolizes the processor time, in case this thread is not blocked for a long time. Actually we want to provide an infrastructure that allows multiple applications to run onto our platforms, and the best performances may result from interleaved execution of these applications. Interrupt-driven switching helps interleaving in the context of multiple applications.

3. **Priority-driven scheduling.** In the actual implementation, some overhead is generated by context switching, when switching from one thread to another. The second solution, namely Round-robin switching with yielding on blocking, may

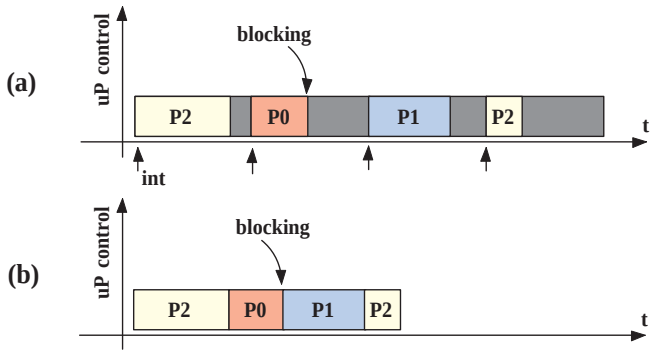

Figure 3.9: Comparison between simple Round-robin scheduling (a) and Round-robin with yielding on blocking (b). In (a), if one thread blocks much earlier than the end of its time slice, a lot of time is spent in idling. In (b), if one thread is blocked, the processor control is passed to the next thread of the ready queue, so no time is spent in idling.

lead to additional overhead. This may happen when one thread blocks, the processor control goes to the next thread in the ready queue, without knowing if it is blocked or not. If it is blocked, a complete context switch is useless, since the next thread will yield immediately.

This concept is explained in Figure 3.10. What may happen with Round-robin scheduling and yielding is presented in Figure 3.10(a). Thread A blocks and thread B is scheduled. This means that the context of thread A is saved and the context of thread B is restored. In this example, thread B is still blocked, in reading or writing, so it must pass control to thread C; another context switch must occur. So, the previous context switch, between thread A and B, is completely useless. The same situation happens in the next transition, between thread C and A.

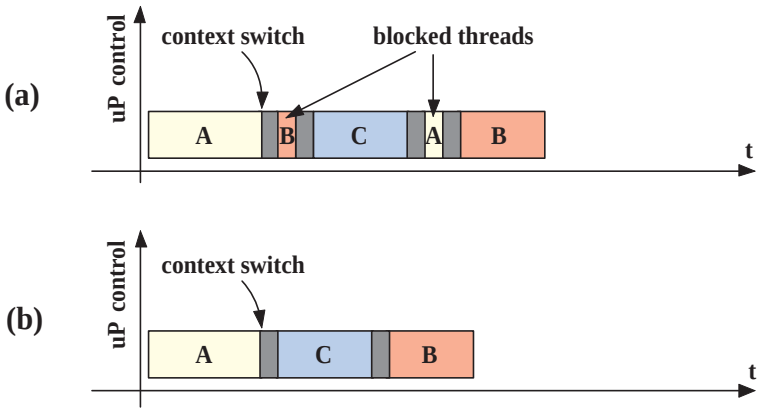

Figure 3.10: In Round-robin with yielding on blocking (a), some useless context switch (shady area) can occur. With priority scheduling (b) this does not happen, since each thread is scheduled only if it is able to run (not blocked).

Each context switch implies some time to be performed (represented in the figure with shady areas), so useless context switches should preferably be avoided. This solution is shown in Figure 3.10(b). The same computation is completed in less time, because in this case a thread is scheduled only if it is not blocked. This leads to better performances, especially if there are a lot of threads to be scheduled and most of them are often blocked. This method can be implemented with *priority scheduling*, giving higher priority to threads that are not blocked. The actual implementation and details will be presented later in this thesis.

Chapter 4

Case Studies

This chapter describes the two applications that have been used to test our method of dynamic scheduling implementation. The first one is the Sobel edge detection algorithm, the second is an M-JPEG encoder. For each of these examples we report the actual implementation on a multiprocessor system generated by ESPAM and corresponding results (in terms of duration of the execution). These results will be compared, in the next chapters, with the ones we got with our dynamic scheduling implementation.

4.1 Sobel algorithm

The Sobel algorithm is a tool used to detect the edges of the shapes represented in a (digital) picture. An input-output example of this application is shown in Figure 4.1.

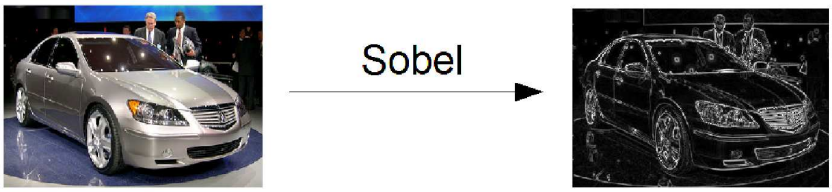

Figure 4.1: Example of input-output relation of the Sobel algorithm.

The KPN specification of the Sobel application is shown in Figure 4.2. This specification was derived using the PNGEN [11] tool. The top node (*ReadPixel*) reads the input image and sends each pixel to the two *Gradient* processes. These processes calculate the gradient along the horizontal and vertical dimension, and send the result to the *AbsValue* node. This one gives as result the sum of the absolute value of the two gradients, calculated for each pixel of the image. If this sum is high, the pixel is likely to be part of the edge of a shape. So, the *WritePixel* node just writes the data produced by *AbsValue* to the output image.

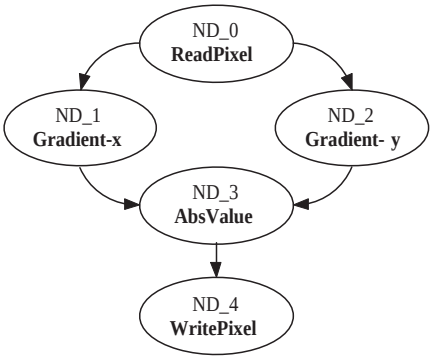

Figure 4.2: KPN specification of the Sobel application. Each arrow represents one or more FIFO channels.

4.1.1 Implementation on a five-processor system

Exploiting the ESPAM design flow, this application was mapped on a five-processor system. Using the *Platform Specification* file, five processors (MB_1, MB_2, MB_3, MB_4, MB_5) are instantiated and connected to the corresponding ZBT memory controllers, used as interface with the outside world. On each of these processors, using the *Mapping Specification* file listed in Figure 4.3, one node of the KPN application is mapped.

```
<mapping name="myMapping">

   <processor name="MB_1">
       <process name="ND_0" />
   </processor>

   <processor name="MB_2">
       <process name="ND_1" />
   </processor>

   <processor name="MB_3">
       <process name="ND_2" />
   </processor>

   <processor name="MB_4">
       <process name="ND_3" />
   </processor>

   <processor name="MB_5">
       <process name="ND_4" />
   </processor>

</mapping>
```

Figure 4.3: The *Mapping Specification* file for Sobel application on five processors. The mapping type is *one-to-one*.

The actual system, generated by ESPAM and run on the target FPGA platform, is depicted symbolically in Figure 4.4. An external processor loads the input image to the ZBT memory connected to MB_1, so that it can read the pixels and send them to the other nodes. The "source" node of this KPN is mapped on MB_1. The "sink" node is *WritePixel*, which is mapped on MB_5. This node writes the final image to its external memory bank. At the end of the execution, an external processor can read the

final result to verify the correctness. However, each processors can write to an external ZBT memory, for debugging purposes and also to communicate the duration of the execution on each processing unit. For this application, we used an option of the PNGEN tool to reduce the number of FIFO channels (*PN-OPTIONS=–no-reuse*), since these components occupy a lot of memory. Memory occupation is not critical in this example, but in the case of multiple applications running on the same platform it is, and this PNGEN option is compulsory. So, to make a fair comparison with the next examples, we used this option also in this case.

The total execution, for an image of 450x275 pixels, requires **29.850 million clock cycles**.

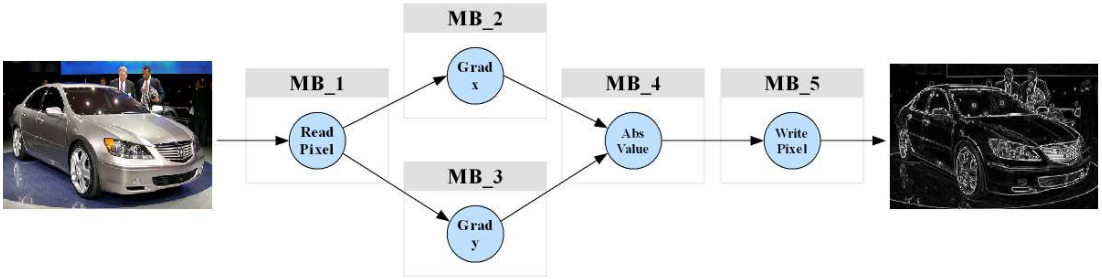

Figure 4.4: Symbolic representation of the five-processor system generated by ESPAM, according to the input specification files.

4.1.2 Implementation on a one-processor system

The same application was mapped on a one-processor system. Only one processor (MB_1) is instantiated, using the *Platform Specification* file, and connected to a ZBT memory controller. All of the nodes which comprise the KPN application are mapped onto MB_1. The ESPAM tool generates an appropriate control code which implements the static scheduling of these nodes.

In this case, the total execution, for an image of 450x275 pixels, requires **109.116 million clock cycles**.

4.2 M-JPEG encoder

The Motion JPEG is a multimedia format where each video frame of a video sequence is separately compressed as a JPEG image. The main C code of this application is listed in Figure 4.5. This M-JPEG encoder processes video data which format is 4:2:2 YUV, so with chroma subsampling.
After the declaration of some variables, the basic computation starts at line 17. For each frame of the video, *initVideoIn* initializes the corresponding header information. Then, the frame is divided into blocks of 8x8 pixels, *mainVideoIn* picks one block per time and sends it to the Discrete Cosine Transform (DCT) of line 22, followed by quantization (Q) and variable length encoding (VLE). Finally, *mainVideoOut* writes the result to the output image, adding the header information to the compressed frame.

```
1    int main(int argc, char **argv)

        int t, j, i;

5       THeaderInfo   hi;
        TQTables        LuminanceQTable;
        TQTables        ChrominanceQTable;
        THuffTablesDC LuminanceHuffTableDC;
        THuffTablesDC ChrominanceHuffTableDC;
10      THuffTablesAC LuminanceHuffTableAC;
        THuffTablesAC ChrominanceHuffTableAC;
        TTablesInfo     LuminanceTablesInfo;
        TTablesInfo     ChrominanceTablesInfo;
        TPackets        stream;
15      TBlocks block;

        for (t = 0; t < NumFrames; t++)
           initVideoIn (&hi);
           for (j = 0; j < VNumBlocks; j++)
20            for (i = 0; i < HNumBlocks; i++)
                 mainVideoIn(&block);
                 mainDCT(&block, &block);
                 mainQ(&block, &block);
                 mainVLE(&block, &stream);
25               mainVideoOut(&hi, &stream);



        return (0);
30
```

Figure 4.5: Main C code of the M-JPEG encoder application.

4.2.1 Implementation on a five-processor system

In order to implement this application on the desired platform, first we use the PNGEN tool to convert this sequential specification to a Kahn Process Network, that matches the ESPAM application specification. The resulting KPN is depicted in Figure 4.6. The diagram shows that the communication between threads is similar to a pipeline. The only thread that does not match this type of communication is *initVideoIn*, but this process is very simple and it is run only once per frame. In this example we used only PNGEN default options, since FIFO channels are not so many.

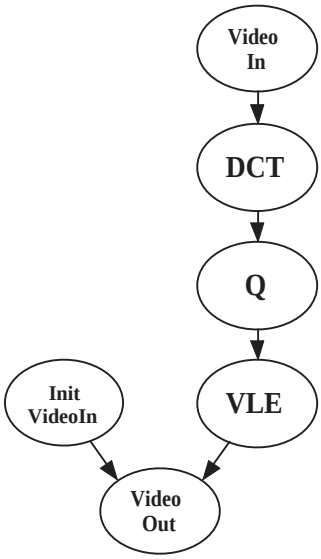

Figure 4.6: Kahn Process Network specification of the M-JPEG encoder, derived via the PNGEN tool.

Then, we generate a platform using the ESPAM design flow, taking care of the memory requirements of this application, because it is much more complex than the Sobel algorithm. Communication with the outside world is done, as in the previous example, via the external ZBT memories. At startup the input image is loaded in the memory connected to the source node of the KPN. At the end of the computation, the output image is read, by the external processor, from the memory connected to the sink node. A symbolic representation of the actual platform is shown in Figure 4.7.

With this system implementation, encoding 2 frames of 128x128 pixels requires **36.863 million clock cycles**.

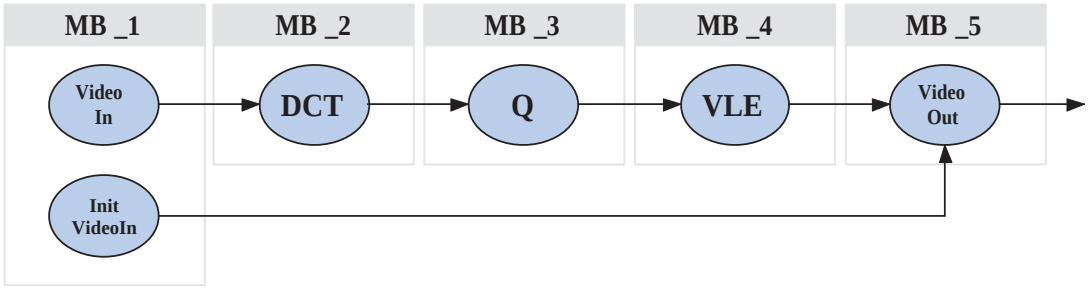

Figure 4.7: Symbolic representation of the M-JPEG encoder, mapped on five processors.

Chapter 5

Implementation using Xilkernel

This chapter describes our implementation of dynamic scheduling using the Xilkernel operating system. Several examples and results are presented, in order to compare the dynamic scheduling approach with the static one.

5.1 Introduction to Xilkernel

Xilkernel [15] is a small, modular operating system, which is highly integrated with the Xilinx Platform Studio framework [13]. We decided to use it firstly for these main features. It allows a high level of customization, so that the designer can choose a trade-off between size (memory occupation) and functionality. It supports Pthread API, though not all of the concepts and interfaces that comprise it are available. Only the most useful concepts and interfaces are implemented to reduce the OS memory footprint. However, Xilkernel programs can run equivalently on desktop OS.

Xilkernel memory footprint on MicroBlaze systems ranges from 7.4 kB to 19.3 kB, numbers obtained with the GCC optimization flag *-O2*. When the memory occupation is just 7.4 kB only the basic kernel functionality is provided, with multi-tasking and multi-threading. The largest memory footprint is required for the full kernel functionality, with all modules included.

Adding a kernel to a system is useful because, generally, embedded applications are comprised of various tasks, that need to be performed in a particular sequence or schedule. As the number of these tasks grows, it gets very hard to organize time-sharing and scheduling. Breaking down tasks as individual applications and implementing them on an OS is more intuitive. Also, a kernel provides common interfaces such as file systems, timers, etc. Xilkernel can provide these services. In our case, we use an operating system because we want to implement a dynamic scheduling method for KPN mapped on multiprocessor platforms.

5.1.1 Xilkernel process model

Xilkernel's units of execution are called process contexts, and scheduling is done at process context level. There is no concept of thread groups combining to form what is conventionally called a process. Threads/processes are manipulated using the POSIX pthreads API.

Each process can be in different states. It can be running, or waiting on the ready queue, or dead; plus other states that manage timeouts and thread synchronization. The only states that we use in our implementations are are shown in Figure 5.1: *PROC_NEW*, which is the state of new created processes; *PROC_RUN*, in which a process owns the control of the processor; *PROC_READY*, when a process is ready to run but waits in the ready queue; finally, *PROC_DEAD*, the state of killed processes. Transitions between *PROC_RUN* and *PROC_READY* are determined by the kernel scheduler.

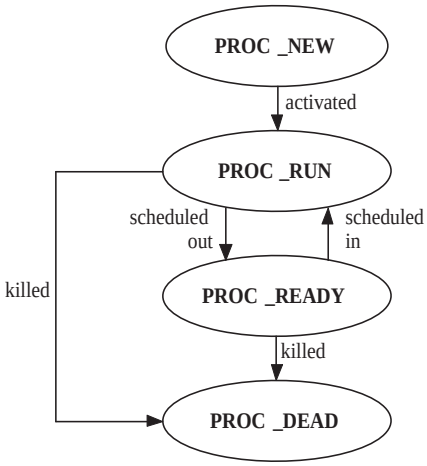

Figure 5.1: Process states of Xilkernel used in our implementations.

5.1.2 Scheduling model

Xilkernel supports either priority scheduling (SCHED_PRIO) or simple Round-robin scheduling (SCHED_RR). This is a global scheduling policy, configured statically at kernel generation time.

In SCHED_RR, there is a single ready queue. Each process executes for a configured time slice, then it yields the execution to the next process in the ready queue. The end of the time slice, in MicroBlaze-based systems, is signaled by an interrupt generated by an external timer.

In SCHED_PRIO, queues are as many as priority levels. Priority 0 is the highest priority in the system and higher values mean lower priority. The maximum number of priority levels is 32. Within the same priority level, the scheduling is Round-robin. When a ready queue level is empty, it is skipped and the next level is checked for ready processes.

5.1.3 Building applications

An application, to run under Xilkernel, must meet these requirements:

- Source C files must include the file xmk.h as the first file among others. Defining this flag makes available definitions and declarations necessary for Xilkernel applications.
- The application must provide a *main()* function, including the kernel invocation *xilkernel_main()*.
- The application must be linked with the Xilkernel library *libxilkernel.a*

5.2 Dynamic scheduling implementation

In order to implement dynamic scheduling of KPN nodes on our platforms, we made some modifications to the Register Transfer Level system specification which is generated by ESPAM, presented in Figure 1.2. These modifications mainly involve the program code for processors and the platform topology description.

The actual dynamic scheduling implementation depends on which scheduling policy is chosen. For further information, see Section 3.3. This section firstly describes the design steps common to all of the scheduling policy, then it focuses on the modifications needed for each different scheduling type.

5.2.1 Common implementation steps

The implementation steps described in this section are common to all of the scheduling policy. They consist of both hardware and software modifications.

Hardware modification

While within PowerPC processors an internal timer can be used for periodic interrupt signal generation, in our MicroBlaze processors this periodic signal must be generated by an external timer. For this purpose, we use the On-Chip Peripheral Bus timer/counter (*OPB_TIMER*). This timer can be connected to the MicroBlaze OPB bus using the Xilinx Platform Studio GUI or, manually, modifying the MHS and MSS files.

For the MHS file, we add the lines listed in Figure 5.2. These lines include timer instantiation, hardware version, the address range for communications with the processor, the connection to the OPB bus and to the interrupt port of the processor. Of course this interrupt port must be included also in the processor declaration lines of the MHS file. The timer OPB address range must be compatible to all the addresses of the other peripherals (no overlapping is allowed).

```
BEGIN opb_timer
 PARAMETER INSTANCE = opb_timer_0
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_BASEADDR = 0xF1000000
 PARAMETER C_HIGHADDR = 0xF100FFFF
 BUS_INTERFACE SOPB = mb_opb_1
 PORT Interrupt = MB_1_INTERRUPT
END
```

Figure 5.2: Lines added to the MHS file to connect an OPB_TIMER to our processor.

The MSS file requires also some little modifications, as shown in Figure 5.3, in order to include the OPB_TIMER driver to the software libraries.

```
BEGIN DRIVER
 PARAMETER DRIVER_NAME = tmrctr
 PARAMETER DRIVER_VER = 1.00.b
 PARAMETER HW_INSTANCE = opb_timer_0
END
```

Figure 5.3: Lines added to the MSS file to include an OPB_TIMER to the software libraries.

For each processor on which Xilkernel is used we have to instantiate and connect an OPB_TIMER. The frequency of the generated interrupt signal can be set within the kernel customization.

Software modifications

The software modifications common to every scheduling policy we have tested are listed below.

Modification of ESPAM code. We have modified ESPAM source code in order to force processors to communicate via their communication controllers. This is why the ESPAM tool uses two kind of inter-processor communication for MicroBlaze: Fast Simplex Link (FSL) and communication controllers. The first kind is faster, and uses the communication primitive shown in Figure 5.4. But in our case, this communication primitive may generate a deadlock. Let us assume that the timer interrupt wants to force a context switch and the processor is blocked on writing, in the primitive written in bold. Before managing the interrupt, the processor waits for the completion of this primitive. But, since this one is an assembly instruction, and completes only if some space is generated in the output FIFO, it will never yield the execution to the next process. So, we force inter-process communication to go through communication controllers, which primitives do not lead to this situation.

```
#define writeFSL(pos, value, len)
    do {
        int i;
        for (i = 0; i < len; i++)
            microblaze_bwrite_datafsl(((volatile int *) value)[i], pos);
    } while(0)
```

Figure 5.4: Communication primitive for Fast Simplex Links.

Adding the Xilkernel OS. The Xilkernel OS can be added to each processor using XPS or modifying the MSS file. The kernel can be highly customized because modules and parameters can be set to suit the user requirements. The modification to the MSS file is listed below, in Figure 5.5. In line 4, the operating system is associated with the desired processor (*MB_1*). In lines 5-6 the OPB_TIMER is specified and the timer interval (which determines the time slice duration) is set, in milliseconds. In lines 7-8 is defined the maximum number of pthreads and the static pthread table, namely the set of pthreads firstly executed when the scheduler is started (in this case *thread_main*). This thread, in our applications, is the one that generates all the others which comprise the KPN.

```
1    BEGIN OS
      PARAMETER OS_NAME = xilkernel
      PARAMETER OS_VER = 3.00.a
      PARAMETER PROC_INSTANCE = MB_1
5     PARAMETER systmr_dev = opb_timer_0
      PARAMETER systmr_interval = 1
      PARAMETER max_pthreads = 8
      PARAMETER static_pthread_table = ( (thread_main,1) )
     END
```

Figure 5.5: Lines added to the MSS file to include the Xilkernel operating system.

Derive threads corresponding to each node of the KPN. In order to derive threads that correspond to each node of the KPN specification of the desired application, we use ESPAM with *one-to-one* mapping onto a fictitious platform, so that each processor program code represents only the behavior of the node mapped on it.

Then, we can copy each of these program codes and create separate threads. We have just to rename appropriately the input and output channels, according to the definitions of the *MemoryMap.h* file (see 2.5.1). Since the actual platform is different from the fictitious one, the channel names are also different. An example of thread derived by processor program code is shown in Figure 5.6. This thread represent the *AbsValue* node of the Sobel algorithm. The node input and output channels are written in bold font. Their definitions have to be changed, according to the new names of the actual platform.

```
void* thread3(void *arg)
{
  int c0, c1;

  // Input Arguments
  tCH_27 in_0ND_3;
  tCH_28 in_1ND_3;

  // Output Arguments
  tCH_29 out_2ND_3;

  for( c0 =  ceil1(3); c0 <=  floor1(M ); c0 += 1 ) {
    for( c1 =  ceil1(3); c1 <=  floor1(N ); c1 += 1 ) {
        read(ND_3_IG_27_CH_27, &in_0ND_3, (sizeof(tCH_27)+(sizeof(tCH_27)%4)+3)/4);
        read(ND_3_IG_28_CH_28, &in_1ND_3, (sizeof(tCH_28)+(sizeof(tCH_28)%4)+3)/4);

        _absVal(in_0ND_3, in_1ND_3, &out_2ND_3) ;

        write(ND_3_OG_29_CH_29, &out_2ND_3, (sizeof(tCH_29)+(sizeof(tCH_29)%4)+3)/4);
    } // for c1
  } // for c0
} //thread3
```

Figure 5.6: Thread derived from the *AbsValue* node of the Sobel application.

Thread creation with Xilkernel API. The Xilkernel API provide a dedicated function for thread creation:

```
int pthread_create (pthread_t* thread, pthread_attr_t* attr,
                    void* (*start_funct), void* param)
```

- Parameters:
  - *thread* is the location to store the created thread's identifier.
  - *attr* is the pointer to thread creation attributes structure. If this pointer is NULL, default settings are used.
  - *start_funct* is the start address of the function from which the thread starts the execution.
  - *param* is the pointer argument to the thread function.
- Returns:
  - Returns 0 and thread identifier of the created thread in **thread*, on success.
  - Different values if an error occurred (*thread* refers to an invalid location, *attr* refers to invalid attributes, or if resources are unavailable to create the thread).

In Figure 5.5 the *PARAMETER static_pthread_table* defines the first thread that the kernel will execute, in this case *thread_main*, with priority 1. This is the only static thread of our implementations. When this thread runs, it creates all the other threads, which belong to the KPN application, with default settings. The program code of this thread is shown in Figure 5.7. There are 5 KPN nodes mapped on this processor in this example, so 5 thread identifiers must be declared (line 4), then the external hardware

clock cycles counter is resetted (line 5). In lines 7-17 the actual thread creations are performed, using the Pthread API described above. After each thread creation, the code performs a check on the returned value of *pthread_create*. If it is not equal to zero, a flag (666) is set in the external memory, so that the error is signaled.

```
1    void* thread_main( void *dummy)
     {
       int i, ret;
       pthread_t threadID[5];
5      int clk_num;
       *clk_cntr = 0;

       ret = pthread_create(&threadID[0], NULL, (void*)thread1, NULL);
       if (ret) *(ZBT_MEM_2+HALF_MEM)=666;
10     ret = pthread_create(&threadID[1], NULL, (void*)thread2, NULL);
       if (ret) *(ZBT_MEM_2+HALF_MEM+1)=666;
       ret = pthread_create(&threadID[2], NULL, (void*)thread3, NULL);
       if (ret) *(ZBT_MEM_2+HALF_MEM+2)=666;
       ret = pthread_create(&threadID[3], NULL, (void*)thread4, NULL);
15     if (ret) *(ZBT_MEM_2+HALF_MEM+3)=666;
       ret = pthread_create(&threadID[4], NULL, (void*)thread5, NULL);
       if (ret) *(ZBT_MEM_2+HALF_MEM+4)=666;

       return 0 ;
20   }
```

Figure 5.7: Thread_main program code. This is a static thread which aim is to create all the KPN threads.

5.2.2 Scheduling policy-dependent implementation steps

The software modifications listed in this section depends on the chosen scheduling policy. Simple Round-robin, Round-robin with yielding and priority scheduling have been tested on the Sobel application running on one MicroBlaze processor. Each node of the KPN application is represented by a thread, created and scheduled using the Xilkernel OS. As depicted in Figure 5.8, all of the inter-process communications go through external FIFOs. There is no notion of shared global memory. At the end of these test we choose the best scheduling solution, that will be the basis for the rest of the experiments.

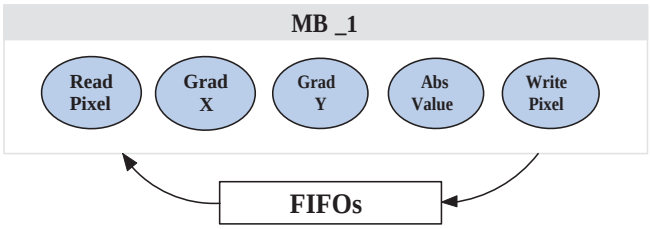

Figure 5.8: Test application: Sobel running on one MicroBlaze only. All of the inter-process communications go through external FIFOs.

5.2.3 Simple Round-robin scheduling

For this kind of scheduling policy (see 3.3) no further modifications are needed, since no enhanced features of the OS are used. Threads are scheduled automatically by the kernel scheduler. A context switch occur when an interrupt is generated by the external timer. The time slice can be set by changing the value *PARAMETER systmr_interval = 1* shown in Figure 5.5.

The clock cycles required to complete execution of the applications are represented in Figure 5.9, in function of the time slice duration. As described in Section 3.3, this kind of scheduling policy does not match the KPN executional semantics, since many threads may block much earlier than the end of the time slice. This is well proofed in the graph, because even with the shortest time slice available (1 ms) the performances are much worse, compared to the ones obtained with Round-robin and yielding, that will be presented later. In the best case, with 1 ms of time slice, the execution time is 199.914 million of clock cycles. The figure also shows that the relation between execution time and time slice is almost linear. This is because, when the time slice is set to larger values, the wasted time in blocking is also increased.

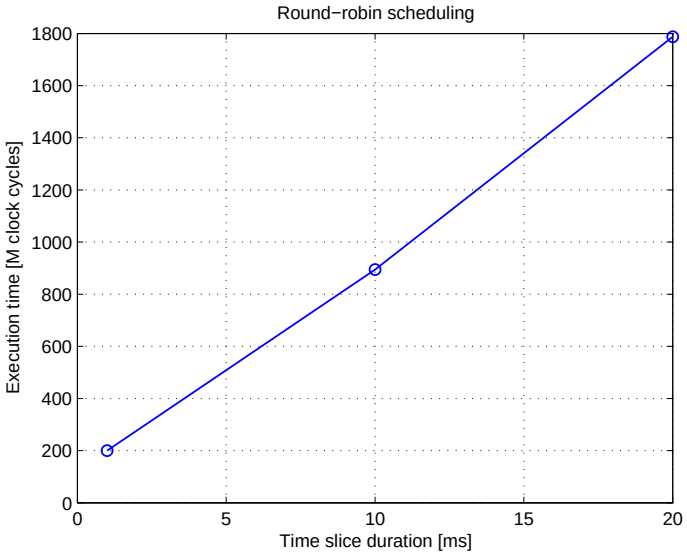

Figure 5.9: Clock cycles measured on MB_1 at the end of the execution of the Sobel application, in function of the time slice duration.

5.2.4 Round-robin scheduling with yielding on blocking

We implemented this solution with a modification to the reading and writing primitives created by ESPAM and included in the *aux_funct.h* file. The modified read primitive is shown in Figure 5.10. The modification is highlighted in bold font. For example, in this primitive, when blocking occurs (the selected FIFO is empty) the thread makes a system call with the *yield()* function, forcing a context switch and yielding the processor control to the next thread. The *yield()* function is made available by defining the

parameters *PARAMETER enhanced_features = true* and *PARAMETER config_yield = true* in the Xilkernel definition included in the MSS file (see Figure 5.5).

```
#define read(pos, value, len)
    do {
        int i;
        volatile int *isEmpty;
        volatile int *inPort = (volatile int *)pos;
        isEmpty = inPort + 1;
        for (i = 0; i < len; i++) {
            while (*isEmpty) { yield(); };
            ((volatile int *) value)[i] = *inPort;
        }
    } while(0)
```

Figure 5.10: Modified read primitive in order to implement Round-robin with yielding. The modification is highlighted in bold font.

The execution time results are shown in Figure 5.11. We can see that, above 2 ms of time slice, the execution time is constant. This is why *yield()* dominates on interrupt-driven context switch with larger time slices. In the best cases, above 2 ms of time slice, the execution time is 92.426 million of clock cycles. This time is less than a half of the simple Round-robin result, shown in Figure 5.9. This scheduling policy leads to a system even faster than the one presented in Section 4.1.2, obtained with ESPAM static scheduling. This may happen because the static scheduling generated by ESPAM may not be optimal. Also, the control code which implements the static scheduling and the nodes communication, in case of many-to-one mapping, generates a larger overhead, compared to the one generated by the context switching.

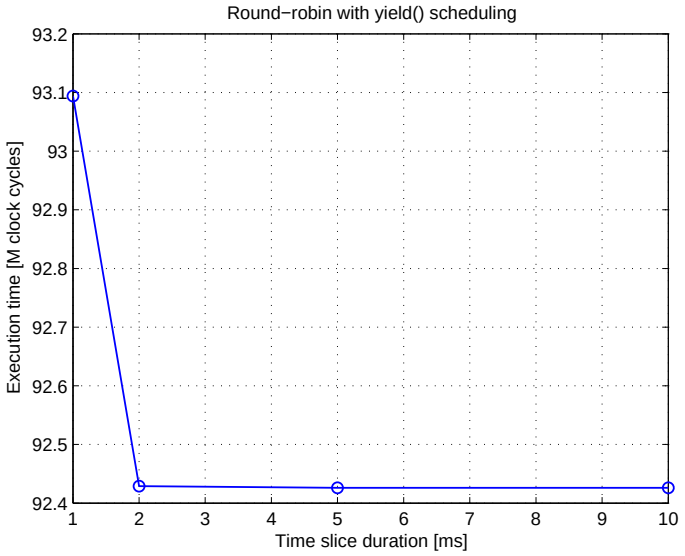

Figure 5.11: Clock cycles measured on MB_1 at the end of the execution of the Sobel application, in function of the time slice duration. The adopted scheduling policy is Round-robin with yielding on blocking.

5.2.5 Priority scheduling

An additional parameter must be included in the MSS Xilkernel definition: *PARAMETER sched_type = SCHED_PRIO* to set the kernel scheduler to priority-based mode. The method we use to exploit priority scheduling is based on the creation of an additional thread, called *control thread*, that is fired every time all the threads of the application are blocked.

Startup. During the startup, as represented in Figure 5.12, the application threads are set to the highest priority level (1) while the control thread is put in the middle-priority level (2). Only the application threads are able to run in this situation, because they have higher priority.

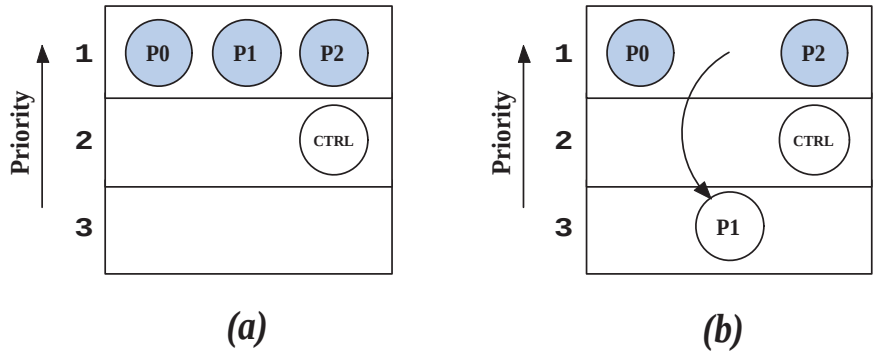

Figure 5.12: Priority assignment at startup (a): application threads have the highest priority. When one thread blocks, its priority is decreased (b).

Application execution. The next steps apply concepts described in Section 3.3, to minimize the context switch number. When one application thread blocks, it first stores the address of the FIFO on which it is blocked. This address will be used by the control thread to check if the thread is able to run or not. Then its priority is decreased (see Figure 5.12(b)), and finally it yields to the next thread in the high-priority level. All of these operations are implemented in the read and write primitives. The modified read primitive is shown in Figure 5.13.

- The address *isEmpty* is stored in the global array *blk_fifo*, accessible from the control thread. The position in the array is determined by the *thread_index*.
- The current thread priority is decreased using the instruction *pthread_setschedparam()* provided by the Xilkernel POSIX API.
- The processor control is yielded with *yield()*.

Control thread execution. The control thread is fired only when all the others are in the low-priority level, as depicted in Figure 5.14(a). The C code of this thread is listed in Figure 5.15. For every alive thread it checks the status of the FIFO which caused the blocking of the thread. If the FIFO is no more empty (in case of blocking read) or no more full (in case of blocking write), it promotes the corresponding thread to the

```
#define read(pos, value, len)
    do {
        int i;
        volatile int *isEmpty;
        volatile int *inPort = (volatile int *)pos;
        isEmpty = inPort + 1;
        for (i = 0; i < len; i++) {
            while (*isEmpty) {
                blk_fifo[thread_index]=isEmpty;
                pthread_setschedparam(threadID[thread_index],0,&lo_prio_spar); yield();
            }
            ((volatile int *) value)[i] = *inPort;
        }
    } while(0)
```

Figure 5.13: Modified read primitive in order to implement priority decreasing in case of blocking.

highest level of priority, as depicted in Figure 5.14(b). With this solution there is no more possibility of yielding the processor control to a thread that is still blocked, an issue described in Section 3.3.

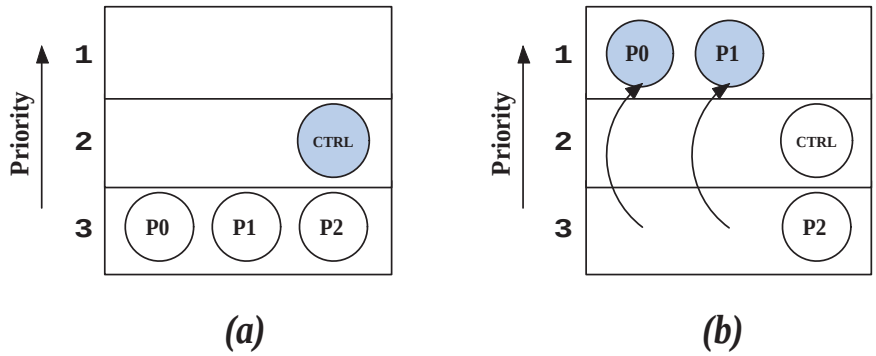

Figure 5.14: Control thread is fired when all of the other threads are in the lowest priority level (a). In (b), control thread promotes only threads that are no more blocked, checking their FIFO status.

```
while(1)
{
  for (i=0; i<5; i++)
    if (alive[i]==1)
      if (*blk_fifo[i]==0)
        pthread_setschedparam(threadID[i], 0, &hi_prio_spar);
  yield();
}
```

Figure 5.15: Control thread code.

Although this seems to be a more efficient way of scheduling implementeation, the test results, on the Sobel application, gave us no improvements, compared to the solution of Round-robin with yielding. The total execution time, in the best case, is 101.388 millions of clock cycles. This result is affected by the intrinsic slowness of priority increasing and decreasing. We measured it and it takes almost 800 clock cycles. What may cause this slowness could be that, in order to change priority, the kernel scheduler must be invoked. This leads to another context switch and time loss. However, this

result is application-dependent. A different application, with a lot of threads, most of them often blocked, may get performance improvements by using this method.

5.2.6 Test result discussion

The test results are summarized in Table 5.1. The *Performance* column shows the total execution time, in millions of clock cycles (M c.c.), for each scheduling method. The *Comparison with static scheduling* is made with the corresponding example, described in Section 4.1.2. The last column shows the OS memory footprint, for each scheduling method.

Scheduling Method	Performance	Comparison with static scheduling	OS footprint
Round-robin	199.914 M c.c.	+90.798 M c.c.	7.836 kB
Round-robin with yield()	92.426 M c.c.	-16.69 M c.c.	9.888 kB
Priority	101.388 M c.c.	-7.728 M c.c.	14.084 kB

Table 5.1: Test results comparison.

With dynamic scheduling, every time the processor control is passed from a thread to another, some time is wasted in context saving and restoring. We measured this context switch time, in the Xilkernel OS example, in 912 clock cycles. Although this context switch overhead occurs, Table 5.1 shows that Round-robin with yield and Priority scheduling are faster than the static scheduling example described in Section 4.1.2. This means that the context switch overhead generated by the dynamic scheduling is overridden by the higher efficiency of the threaded program code. This may be due to the following reasons:

- The static scheduling generated by ESPAM may not be optimal. The dynamic scheduling may find a better order of execution of the KPN nodes.

- The control code for KPN node scheduling and communications is more efficient when the program code is divided in threads.

With this first test of multi-threading dynamic scheduling methods we found out that Round-robin scheduling with yielding on blocking is the most efficient way, and it is quite easily implemented. So, we choose to use this kind of scheduling for the rest of our tests. The memory occupation overhead of the operating system, with this scheduling method implemented, is 9.888 kB. This value is obtained by subtracting the *text* occupation of the compiled application without OS (6.404 kB) from the *text* memory occupation with Xilkernel (16.292 kB). The *text* memory occupation is not dependent on the maximum number of threads supported by the OS. This number is defined in the MSS file modification shown in Figure 5.5. By contrast, for each extra thread 1.2 kB of *stack* memory are reserved. Thus, the total memory occupation is dependent on the maximum number of threads.

5.3 Advanced examples

This section describes some of the several examples we made for our dynamic scheduling approach. The first deals with multiple applications, the second with multiple instances of the Sobel application, running on the same platform.

5.3.1 M-JPEG and Sobel mapped on the same platform

The aim of this test is to verify if dynamic scheduling can lead to performance improvement when multiple applications are mapped on the same platform. System topology and mapping of this example is shown in Figure 5.16. This mapping was chosen in order to balance the load on each processor and prevent contention on communication with external memories (each processor is attached to a single bank of ZBT RAM). Nodes like *VideoIn*, *VideoOut*, *ReadPixel* and *WritePixel* are mapped onto different processors.

Let us focus, for instance, on the MB_3 processor. If the M-JPEG application was the only one mapped onto the system, only the *Q* node would be mapped on it. In this context, when the *Q* node blocks, all the time is spent in waiting new data to process or place to write. The aim of mapping multiple applications on the same platform is that this time can be exploited by an additional thread, in this example *Gradient Y*, so less time is wasted.

As described in Chapter 4, the execution of the M-JPEG and Sobel applications requires respectively 36.863 and 29.850 millions of clock cycles, when each of these applications is mapped alone in a five-processor system. This new platform, where Sobel and M-JPEG applications are mapped together, completes the execution of both tasks in 59.231 millions of clock cycles. If executed in sequence, without interleaving, these two applications would require $(36.863 + 29.850) = 66.713$ M clock cycles. So, this mapping leads to performance improvements (10%) because interleaving occur.

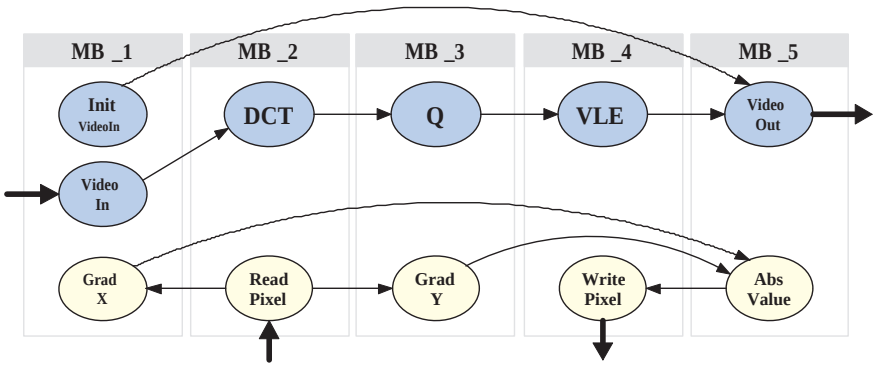

Figure 5.16: System topology and mapping of the M-JPEG an Sobel applications running on the same platform. The thick arrows represent a communication link with external memories. The thin lines represent one or more FIFO channels.

5.3.2 Multiple instances of Sobel application on the same platform

An example similar to the mapping of multiple applications on the same platform is when we consider multiple instances of the same application. In this section we present some implementations of this concept.

3 Sobel instances on 3 processors

The mapping considered for this example is shown in Figure 5.17. It is inspired by pipeline concepts, though we cannot define it as a pipeline. On each processor a source node (*ReadPixel*) is mapped, so that, at startup time, at least one of the nodes of every processor can run. In fact *ReadPixel* just reads data from the external memory, so it cannot be blocked at startup.

This platform processes 3 images of 450x275 pixels in 82.594 M clock cycles (27.531 M per image).

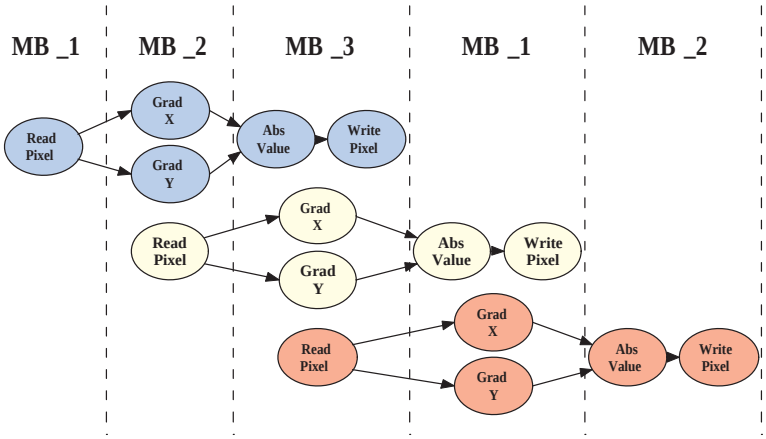

Figure 5.17: Mapping of 3 Sobel instances running on 3 processors. This mapping is inspired by pipeline concepts.

3 Sobel instances on 5 processors

The same concept is applied to this example, strictly related to the previous one. As shown in Figure 5.18, on each processor 3 nodes are mapped. We could not map 5 Sobel instances on this platform due to lack of memory (this system uses almost 100% of FPGA total BRAM). The 3 images are computed in 60.514 M clock cycles (20.171 M per image).

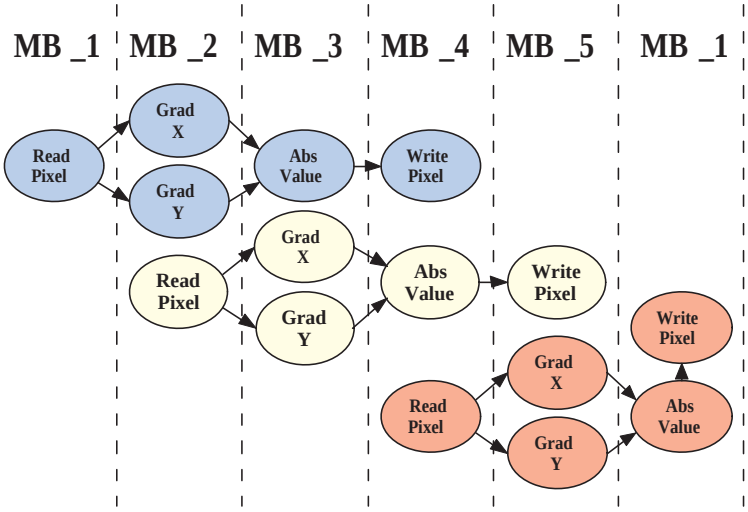

Figure 5.18: Mapping of 3 Sobel instances running on 5 processors.

Throughput analysis of interleaving applications

In Figure 5.19 the throughput is represented, obtained with our tests and normalized to the smallest value, of the platforms listed below:

- 1 Sobel running on 1 processor (92.426 M c.c. per image) (normalized throughput=1)
- 3 Sobel instances running on 3 processors (27.531 M c.c. per image)(normalized throughput=3.36)
- 3 Sobel instances running on 5 processors (20.171 M c.c. per image)(normalized throughput=4.59)

It is interesting to compare these results with the ones that would be obtained just by instantiating the same platform (1 Sobel running on 1 processor, with operating system) for 1, 3, and 5 times. In this case the applications run independently on separate processors, so the throughput gain is linear. Figure 5.20 shows this concept. Those values are theoretically determined for the platforms listed below:

- 1 x (1 Sobel running on 1 processor) (normalized throughput=1)
- 3 x (1 Sobel running on 1 processor) (normalized throughput=3)
- 5 x (1 Sobel running on 1 processor) (normalized throughput=5)

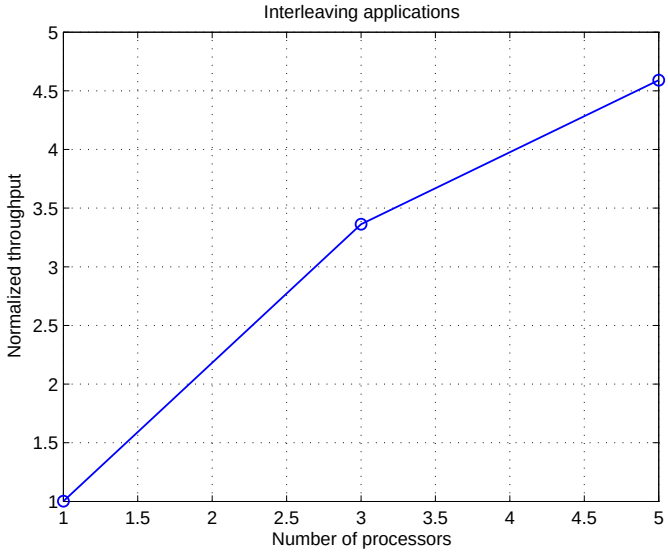

Figure 5.19: Normalized throughput in function of the number of the processors.

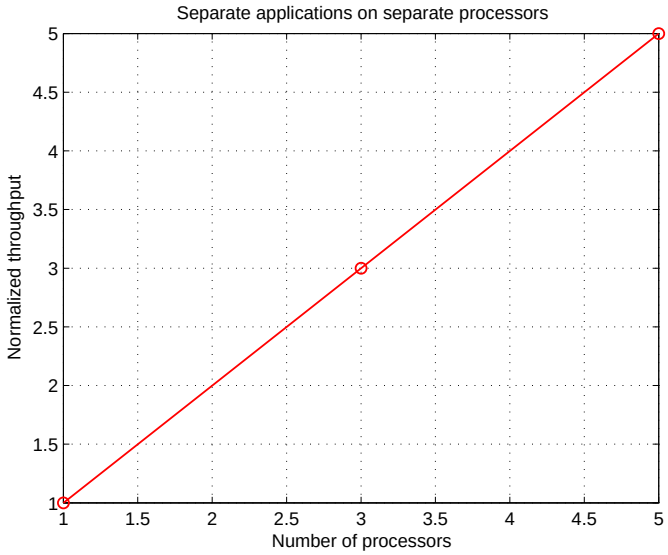

Figure 5.20: Normalized throughput in function of the number of the processors, just by instantiating the platform, which maps 1 Sobel application on 1 processor, for 1, 3, and 5 times.

Combining the two previous graphs, as shown in Figure 5.21, we can see that, in the case of three processors, the interleaving execution leads to throughput improvement (3.36 instead of 3, 12% of speedup). This is not a general result, since speedup, depending on the application, may be higher or lower. For instance, in the case of five processors, interleaving execution is worse than the separate execution. Unfortunately, the comparison between interleaving applications and separate execution is not completely fair in the 5-processor case. This is because, due to lack of memory, we were not able to synthesize 5 threaded Sobel instances running on 5 MicroBlaze. Actually,

in the 5-processor case of Figure 5.21 we compare 5 separate execution Sobel instances with 3 interleaving Sobel instances. This may be the reason why the interleaving execution is slower than the separate one.

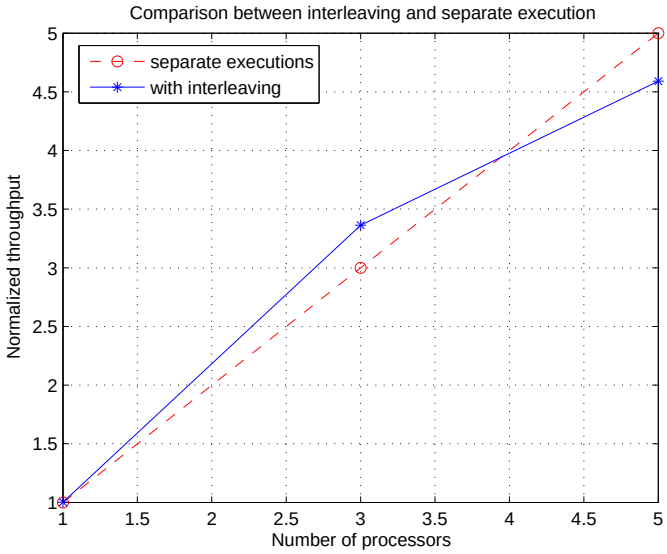

Figure 5.21: Comparison of normalized throughput in function of the number of the processors. The blue line represents interleaving execution, the dashed red one separate execution.

Chapter 6

Implementation using FreeRTOS

FreeRTOS is an alternative operating system that we tested in order to make comparisons with the implementation with Xilkernel. In this chapter we describe the main features of this operating system and we present the obtained results.

6.1 Introduction to FreeRTOS

FreeRTOS.org [17] is an open-source, real-time kernel, free to download. It can be used in commercial applications too. We tested the 4.7.2 version of this operating system. Several ports are available, for many processor architectures and development tools. Most important for our examples, it does exist a port for the MicroBlaze architecture.

FreeRTOS features include:

- Kernel support for preemptive (the scheduler can suspend tasks) or cooperative (tasks must be programmed to yield when they do not need system resources).
- The OS is designed to be small, simple and easy to use.
- The code structure is very portable, predominantly written in C; the source code is contained in only 3 C files.
- It is very scalable.

Although this OS does not support Pthreads API, its API is relatively easy to use and understand. The kernel provides also several kind of inter-process communication, like queues, binary and counting semaphores, mutexes and recursive mutexes. All these features are not used in our implementations.

6.1.1 FreeRTOS fundamentals

FreeRTOS allows a real time application to be structured as a set of autonomous tasks. Only one task within the application can be executing at any point in time and the real time scheduler is responsible for deciding which task this should be. Each task is provided with its own stack. When the task is swapped out the execution context is saved to this stack, so it can be restored safely when the same task is (later) swapped back in.

A task can be in one of the following states:

- **Running.** When a task is actually utilizing the processor, it is said to be in the Running state.
- **Ready.** Ready tasks are those that are able to be executed (not blocked or suspended), but are not currently executing because a different task of higher or equal priority is already in the Running state.
- **Blocked.** A task is blocked when it waits for either a temporal or external event.
- **Suspended.** In this state tasks are also not available for scheduling. Task can enter or exit the Suspended state only with explicit API functions, they cannot wait for temporal or external events.

All of these states and state transitions are depicted in Figure 6.1.

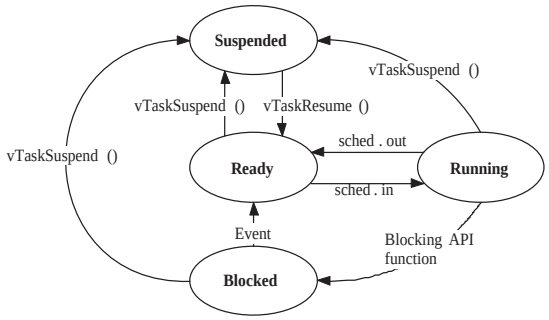

Figure 6.1: Task state diagram supported by FreeRTOS, for alive tasks.

In FreeRTOS, the structure of a task should have the structure shown in Figure 6.2. They are implemented as continuous loops, because they should never return. Finally, task are created using *vTaskCreate()* and deleted using *vTaskDelete()*.

```
void vATaskFunction( void *pvParameters )
{
    for( ; ; )
    {
        -- Task application code here. --
    }
}
```

Figure 6.2: Basic thread structure when using Xilkernel.

6.1.2 Source code description

The basic source code of the FreeRTOS kernel is included in only three files: *tasks.c* includes the core of the kernel scheduler, task creation and deletion, task blocking and suspending, setting thread priority, etc; *list.c* defines list implementation used by the scheduler; finally, *queue.c* implements queues used by the scheduler.

A couple of additional files are needed for porting purposes: *port.c* defines architecture-dependent functions like Timer Interrupt Setup, Stack Initialization, Interrupt Service Routine and Timer Interrupt handling; *portasm.s* includes assembly implementations of some of these functions, to achieve code efficiency. An important example of such a function is presented in Figure 6.3, which shows the processor context saving. Firstly, some space is created in the stack, for the execution context. Then the MSR (Machine Status Register) and all general registers (r30-r1) are saved. Finally, the top of stack is stored in the current TCB (Task Control Block), which is the structure used by the kernel for task handling.

```
.macro portSAVE_CONTEXT
/* Make room for the context on the stack. */
addik r1, r1, -132
/* Save r31 so it can then be used. */
swi r31, r1, 4
/* Copy the msr into r31 - this is stacked later. */
mfs r31, rmsr
/* Stack general registers. */
swi r30, r1, 12
swi r29, r1, 16
swi r28, r1, 20
swi r27, r1, 24
swi r26, r1, 28
swi r25, r1, 32
swi r24, r1, 36
swi r23, r1, 40
swi r22, r1, 44
swi r21, r1, 48
swi r20, r1, 52
swi r19, r1, 56
swi r18, r1, 60
swi r17, r1, 64
swi r16, r1, 68
swi r15, r1, 72
swi r13, r1, 80
swi r12, r1, 84
swi r11, r1, 88
swi r10, r1, 92
swi r9, r1, 96
swi r8, r1, 100
swi r7, r1, 104
swi r6, r1, 108
swi r5, r1, 112
swi r4, r1, 116
swi r3, r1, 120
swi r2, r1, 124
/* Stack the critical section nesting value. */
lwi r3, r0, uxCriticalNesting
swi r3, r1, 128
/* Save the top of stack value to the TCB. */
lwi r3, r0, pxCurrentTCB
sw r1, r0, r3
```

Figure 6.3: Context saving defined in *portasm.s*.

6.2 Implementation example and results

In order to test and evaluate the performances of this operating system in comparison with Xilkernel, we implemented the example of three Sobel instances running on three MicroBlaze. The corresponding mapping is shown in Figure 6.4. The implemented scheduling method is Round-robin with yielding on blocking.

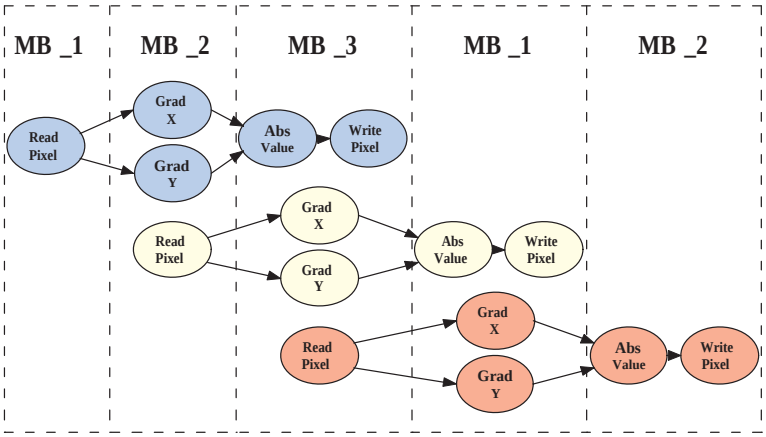

Figure 6.4: Mapping of 3 Sobel instances running on 3 processors.

Some modifications are needed when using FreeRTOS with Round-robin with yielding on blocking, because FreeRTOS is not integrated in the XPS framework. For instance, the time slice duration must be configured using the *FreeRTOSconfig.h* file. In Figure 6.5 2 lines are listed, in which the CPU clock frequency and interrupt tick frequency are defined. During kernel startup, the timer is initialized so that it generates an interrupt every n clock cycles, where $n = CPU\_FREQ/TICK\_FREQ$.

```
#define configCPU_CLOCK_HZ          ( ( unsigned portLONG ) 100000000 )
#define configTICK_RATE_HZ           ( ( portTickType ) 1000 )
```

Figure 6.5: Definition of CPU clock frequency and interrupt tick frequency in *FreeRTOSconfig.h*.

Furthermore, the reading and writing primitives must be compatible with FreeRTOS API. An example of reading primitive is listed in Figure 6.6.

```
#define read(pos, value, len)
    do {
        int i;
        volatile int *isEmpty;
        volatile int *inPort = (volatile int *)pos;
        isEmpty = inPort + 1;
        for (i = 0; i < len; i++) {
            while (*isEmpty) {  TASKyield();  }
            ((volatile int *) value)[i] = *inPort;
        }
    } while(0)
```

Figure 6.6: Modified read primitive in order to implement Round-robin with yielding with FreeRTOS. The modification is highlighted in bold font.

With FreeRTOS, the platform described in Figure 6.4 generates 3 output images in 80.413 M of clock cycles (26.804 M c.c. per image). This result is slightly better than the one obtained with Xilkernel, as shown in Figure 6.7. The FreeRTOS result is represented with a circle. The memory occupation overhead of this operating system is 10.606 kB.

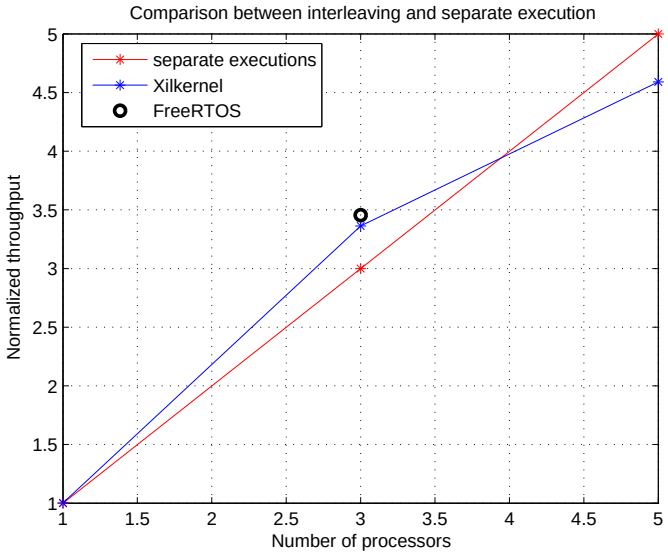

Figure 6.7: Mapping of 3 Sobel instances running on 3 processors.

These results lead to our final conclusions on the FreeRTOS kernel, which are listed below:

+ This operating system is slightly faster (at least, in our examples) than Xilkernel.
+ The source code is very simple and understandable, it can be possibly fit to our aim.
- The memory footprint is almost 10% larger than the Xilkernel one.

Chapter 7

Tutorial on dynamically scheduled system design

This chapter describes step-by-step the implementation of a dynamically scheduled system, starting from the C source code and ending with the final XPS project. For the sake of clarity, this tutorial shows the simplest of our examples, namely the Sobel application running onto only one MicroBlaze. The operating system used in this example is Xilkernel. The described methodology can be simply used also for more complicated cases, with few modifications.

The first section describes how to generate a multiprocessor system using PNGEN and ESPAM tool chain. Then, Section 7.2 includes all of the few manual modifications required for the correctness of the output system. Section 7.3 introduces the MHS, MSS and program code modifications needed for the dynamic scheduling of the KPN application which runs onto our system. The last section describes how to export the modified multiprocessor system in Xilinx Platform Studio and how to generate the final bitstream and get results using a host processor program.

7.1 XPS system generation using PNGEN and ESPAM

The starting point of our tool chain is a folder, as shown in Figure 7.1, which includes the items listed below:

- **sequential folder** includes the sequential specification of the application, and all the functions needed for the correct execution of the program.
- **Makefile** simplifies the running of different tools in the ESPAM design flow.
- **sobel.pla** is the *Platform Specification* used as input for the ESPAM tool (see Figure 1.2).
- **sobel.map** is the *Mapping Specification* file, another ESPAM tool input, which describes the KPN node assignment to each processor of the system.

```
<SOBEL_DEMO>
|--- sequential/: sequential application program code
|---------- car_gray.jpg
|---------- imageIO.c
|---------- imageIO.h
|---------- Makefile
|---------- sobel.c
|---------- sobel_funct.c
|---------- sobel_funct.h
|---------- sources
|--- Makefile
|--- sobel.pla
|--- sobel.map
```

Figure 7.1: Starting folder of the PNGEN and ESPAM tool chain, in the Sobel application case.

The main C source code of the Sobel application can be found in Appendix A. Thanks to the Makefile included in the folder, converting the sequential application to the KPN, parallel specification is a matter of one command:

```
make par FILE_NAME=sobel
```

This command starts the convertion of the sequential application to a KPN, using the PNGEN tool. The specified *FILE_NAME* is the main C source file of the application, in this case *sobel* (without the file extension). Several operations are performed on the sequential program file, such as C parsing in SUIF, dependence analysis (as described in [1]), and finally the convertion from the .yaml PN format to the ESPAM specific XML format. Actually the output (*sobel.kpn*) is a XML file which describes the *Application Specification*, using the KPN model of computation, fully compatible with ESPAM. This output file is copied in the *SOBEL_DEMO* folder.

Once the *Application Specification* is generated, we must adapt the *sobel.pla* and *sobel.map* file to fit the application requirements. Firstly we have to decide the number of processors (in this case, there is only one MicroBlaze) and the system topology. The final *Platform Specification* file is shown in Figure 7.2.

In line 3-5, one MicroBlaze processor is instantiated, with a data memory of 65 kB and a program memory of 32 kB. Its OPB port is named as *OPB_1*. In lines 7-13 we include two ZBT memory controllers in the system, specifying the size of the attached memories (in this case, 1 MB). In lines 15-19 these ZBT controllers are connected to the OPB bus of the processor, using a link. If no communication infrastructure is specified, as in this case, the default choice is a point-to-point network.

When the platform topology is specified, we can decide the mapping of the KPN nodes that comprise the application onto the processors. In this case only one processor is instantiated, so all of the 5 nodes of the application are mapped on it. The *sobel.map* file is shown in Figure 7.3.

Now all the input specifications required by the ESPAM tool are provided, so the next step is running the tool itself. Using the Makefile, this is also a matter of only one command:

```
make espam FILE_NAME=sobel
```

```
<platform name="myPlatform">

  <processor name="MB_1" type="MB" data_memory="65536" program_memory="32768">
      <port name="OPB_1" type="OPBPort"/>
  </processor>

  <peripheral name="ZBT_CTRL_1" type="ZBTCTRL" size="1000000">
     <port name="IO_1" type="OPBPort"/>
  </peripheral>

  <peripheral name="ZBT_CTRL_2" type="ZBTCTRL" size="1000000">
     <port name="IO_2" type="OPBPort"/>
  </peripheral>

  <link name="mb_opb_1">
    <resource name="MB_1" port="OPB_1"/>
    <resource name="ZBT_CTRL_1" port="IO_1"/>
    <resource name="ZBT_CTRL_2" port="IO_2"/>
  </link>

 </platform>
```

Figure 7.2: *sobel.pla*: final *Platform Specification* file for the Sobel application running on only one MicroBlaze.

```
<mapping name="myMapping">

   <processor name="MB_1">
       <process name="ND_0" />
       <process name="ND_1" />
       <process name="ND_2" />
       <process name="ND_3" />
       <process name="ND_4" />
   </processor>

</mapping>
```

Figure 7.3: *sobel.map*: *Mapping Specification* file for the Sobel application. All the nodes are mapped on MicroBlaze processor *MB_1*.

This command implies several operations, as listed in the Makefile:

- The ESPAM tool runs, using the options shown in Figure 7.4:

```
/espam --platform sobel.pla --adg sobel.kpn --mapping sobel.map
       --xps --libxps <ESPAM_LIBXPS> --debugger
```

Figure 7.4: Default ESPAM tool options, set in the Makefile.

  - **platform:** specifies the *Platform Specification* file, in this case *sobel.pla*
  - **adg:** indicates the *Application Specification* file (*sobel.kpn*)
  - **mapping:** refers to the *Mapping Specification* file
  - **xps:** force the tool to generate all the necessary files of a XPS project
  - **libxps:** specifies the path of the library which stores predefined components or files common to all the ESPAM projects, such as custom IP cores, the UCF files and some other optional files required by XPS

- - **debugger:** implies the generation of debugging components (for instance, the hardware clock cycle counters) by the ESPAM tool

- Finally, the *sobel/code/funct_code* folder is generated. All the functions and headers used by the initial sequential application are stored in here.

After running the ESPAM tool, a new directory is created within the original project folder, as shown in Figure 7.5. This new directory includes all the files which comprise the XPS project suite described in Section 2.5.1, such as *system.xmp*, *system.mhs* and *system.mss*.

```
<SOBEL_DEMO>
|--- sobel/: XPS project suite folder
|--- parallel/: contains the files used for parallel application specification
|--- sequential/: sequential application program code
|---------- car_gray.jpg
|---------- imageIO.c
|---------- imageIO.h
|---------- Makefile
|---------- sobel.c
|---------- sobel_funct.c
|---------- sobel_funct.h
|---------- sources
|--- Makefile
|--- sobel.pla
|--- sobel.map
|--- sobel.kpn
```

Figure 7.5: Final project folder, obtained using the PNGEN and ESPAM tool chain, in the Sobel application case.

7.2 Manual modifications

A few manual modifications are compulsory for the correctness of our multiprocessor system execution. They include both hardware and software modifications, but they do not require more than few minutes. The changes on hardware and software parts are listed in the following sections.

7.2.1 Hardware modification

The only required hardware modification is the change of FIFO sizes. The size must be increased to the next upper power of two. This can be achieved by changing the FIFO size parameter in the MHS file, for all of the FIFO channels, as shown in Figure 7.6. The FIFO sizes can also be set to higher values if we want better performances, since larger FIFOs lead to less read/write blocking.

7.2.2 Software modifications

Some software modifications are also required, and they involve mainly the input and output functions. In this aspect the behavior of sequential application compared to our

```
BEGIN fsl_v20
 PARAMETER HW_VER = 2.00.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_1
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END
```

Figure 7.6: How to increase FIFO sizes to the next upper power of two.

multiprocessor systems is very different. A sequential application, for instance, can read the input data from a file. In our embedded systems there is no notion of files and the input data is read directly from the external ZBT memories.

In the case of the Sobel application, the *ReadPixel* function is modified as shown in Figure 7.7. The commented instructions, in lines 2-12, represent the reading from a file, used by the sequential application. In lines 14 and 15, highlighted in bold font, the reading is implemented via a pointer that access to the external memory, where the starting image has been initialized. The modifications of the *WritePixel* function is very similar. This methodology is general, and all the input and output functions must be implemented using pointers that access data put in the external memories. In our case, these functions are implemented in the *sobel_funct.c* file. Therefore, this file must include *MemoryMap.h*, in which the definition of ZBT memory address and all the other addressable components of the system are provided.

```
1     void readPixel( int *output ) {
        /*
        int pp;
        static FILE *fh = NULL;
5
        if (fh == NULL) {
           fh = mrOpen("car_gray.B");
        }

10      pp = mGetc(fh);
        *output =  pp;
        */

        static int addr = 0;
15      *output = *(ZBT_MEM_1+(addr++));
      }
```

Figure 7.7: Modifications of the readPixel function.

After these few modifications, the system is ready to be synthesized and the program is ready to be compiled. At this step of the design, the scheduling of KPN nodes is set at compile time by the ESPAM tool. In the following section we will describe the further modifications which are required to implement dynamic scheduling.

7.3 Modifications for dynamic scheduling implementation

In order to implement the dynamic scheduling of the KPN nodes which comprise the application, some further modifications are needed. They include MHS, MSS and program code modifications, as listed below:

- **MHS modifications.** For the sake of clarity, the final MHS file of this example is provided in Appendix B. As described in Section 5.2.1, in lines 704-711 an OPB timer is instantiated. This timer is required for the periodic interrupt tick generation. Its bus interface is connected to MicroBlaze 1 (MB_1) OPB bus and its output signal, *MB_1_INTERRUPT*, is linked to the *INTERRUPT* port of MB_1 (see line 91).

- **MSS modifications.** The final MSS file is included in Appendix C.

  In lines 4-14, the Xilkernel operating system is instantiated, and its parameters are set:

  - The processor which uses the OS is *MB_1*.
  - The system timer device is *opb_timer_0*.
  - *Enhanced_features* and *Config_yield* are required to use the *yield()* function.
  - The system timer interval is set to 500 ms, a huge value, so that the switching between threads is done only by the *yield()* function. This is the best solution for this application.
  - The static thread table is defined. The only static thread, created at kernel startup, is *thread_main*.

  Lines 284-288 are added to the MSS file to generate the driver libraries corresponding to the OPB timer.

- **Program code modifications.** We need to modify the main program code generated by ESPAM and the communication primitives, which are included in the *aux_funct.h* file.

  The final main program code is reported in Appendix D. This file is structured as follows:

  - Some header files are compulsory included, such as *xmk.h*, *os_config.h*, *pthread.h*. These headers are related to the kernel library generation.
  - The code of threads which represent the KPN nodes of our application is listed. For further information on how we derive the code of each KPN node, see Section 5.2.1. We must must be sure that the input and output FIFOs have the correct name, corresponding to the actual generated platform.

- The *thread_main* is declared. This thread is the only static one, and it is created at kernel startup. The aim of *thread_main* is to create all of the other threads which comprise the KPN application.
- The *main* program only starts the kernel scheduler, using the *xilkernel_main()* instruction.

The communication primitives, included in *aux_funct.h*, are modified using the concepts explained in Section 5.2.4. For instance, when the reading primitive is used and blocking occur, the code must invoke the *yield()* function, in order to pass immediately the processor control to the next thread in the ready queue.

7.4 Generate the bitstream and collect results

7.4.1 Bitstream generation

The system stored in the *sobel* directory of Figure 7.5 is ready to be imported into XPS. We just have to double-click on the *system.xmp* file. Since ESPAM was designed basing on XPS version 6.3, a simple upgrading wizard will be automatically started if the XPS used version is higher (as in our case, since we used XPS version 9.1). All of the default options within this upgrading wizard are set properly.

Since the Xilkernel OS is used in our project, the user must link Xilkernel libraries with the application during the compilation. This is done by a simple modification of the compiler options. Go to the "Project Information Area" on the top-left corner of the screen, select "Applications". Then double-click on "Compiler Options". The corresponding window will pop up. Click on "Paths and Options", then in "Libraries to link against (-l)" write *xilkernel*. Now the project is ready to be synthesized and compiled.

The synthesis process includes the steps listed below:

- **Generate netlist.** The netlist is generated using the command *Hardware - Generate Netlist*. This forces the XPS Platform Generator (*Platgen*) to read the design platform information included in the MHS file along with the the IP attribute settings available from the respective Microprocessor Peripheral Definition (MPD) files. Platgen produces as output a Hardware Description Language (HDL) file and a system netlist file in NGC format.
- **Generate Bitstream.** The FPGA-programming bitstream is generated using the NGC netlist file as input by the *Xflow* tool. The bitstream is stored in the *sobel/implementation* folder.

All these commands can be found in the menu option **Hardware** or in the tool bar.

The compilation process is done using these commands, placed in the menu option **Software** or in the tool bar:

- **Generate Libraries.** This command uses the library building tool (Libgen), that reads the corresponding MSS file and generates device drivers, libraries, input-output configuration, and interrupt handlers.

- **Build All User Applications.** Using the cross-compiler *mb-gcc*, this command generates one ELF file for each processor in the system. Each ELF file is the result of the program code compilation.

Finally, the hardware and software flow must be merged. This is done using the command *Update Bitstream*, which can be found in the menu option **Device Configuration**. However, if the above commands have not been executed, this command will invoke them one by one.
The final bitstream is stored in *sobel/implementation/download.bit*.

7.4.2 Using a host processor program to get results

In order to download the final bitstream and check the output of the FPGA system, we use a software program that runs on an outside host processor. This software program is a Microsoft Visual C++ 6.0 project. It uses the ADM-XRC-II Application Programming Interface to initialize the device and control the input-output of the system. Before running the program, the actual *sobel/implementation/download.bit* file has to be copied in the C++ program folder. For the Sobel application, the result correctness may be simply verified by viewing the jpeg output image.

The main host processor program is shown in Figure 7.8. The bitstream downloading step is omitted.

- In lines 15-25 the buffer (which will be loaded in the external memory space) is initialized with 0 and the input image is put in the location corresponding to the first memory bank. This is because the “source” node of the KPN of this example is linked to this memory bank.

- In lines 27-40 the buffer is actually loaded in the external ZBT memories, so that the input data is made available to the FPGA. This step completes the system initialization.

- In lines 50-60 the program checks if the FPGA system has finished the processing phase. When this condition is true, it reads the clock cycle counter value.

- In lines 65-75 the content of ZBT_MEM 2, which represents the FPGA system output, is saved in the buffer.

- In lines 77-89 the execution time, measured by the clock cycle counter connected to MB_1, is displayed. Then, the output image is stored in the *car_sobel.raw* file, and converted to a jpeg image using an external tool.

```
void FPGA::MY_FUNCTION()
{
    UINT bank_6 = 5*bankSize;
    UINT bank_5 = 4*bankSize;
    UINT bank_4 = 3*bankSize;
    UINT bank_3 = 2*bankSize;
    UINT bank_2 = 1*bankSize;
    UINT bank_1 = 0;

    //-------------------------------------------------------------------------------------
    // Initialization
    system("convert car_gray.jpg -interlace partition RGB:car");

    fh1 = mropen("car.B");

    // All unused memory banks will be initialized with 0
    for (int n=0; n<6*bankSize; n++) {
      rambuf[n] = 0x00;
    }

    // The input data to be processed
    for (n=0; n<450*275; n++) {
      rambuf[bank_1+n] = (DWORD)bgetc(fh1);
    }

    mclose(fh1);

    // write data into to the memory Banks of the FPGA board
    fpgaSpace[COMMAND_REG] = cmd_Initialize; // initialise memory mode + banks access from host (Pentium)
    fpgaSpace[COMMAND_REG];

    //------------------     src            dest    size      mode
    status = writeSSRAM(rambuf            , bank_1, bankSize, dma);
    status = writeSSRAM(rambuf+bankSize   , bank_2, bankSize, dma);
    status = writeSSRAM(rambuf+2*bankSize , bank_3, bankSize, dma);
    status = writeSSRAM(rambuf+3*bankSize , bank_4, bankSize, dma);
    status = writeSSRAM(rambuf+4*bankSize , bank_5, bankSize, dma);
    status = writeSSRAM(rambuf+5*bankSize , bank_6, bankSize, dma);
    if (status != ADMXRC2_SUCCESS) {
      printf("exiting n");
      exit(0);
    }
    // Initialization DONE

    //=============================================================================
    // Execution Steps

    WORD temp;

    // Check whether the system has finished processing and
    // read the counter register (execution time)
    while(1){
      temp = fpgaSpace[STATUS_REG];
      if (temp == stat_Finished) {              // read status
        DWORD clock_num = fpgaSpace[COUNTER_REG]; // read the counter
        printf("Clk cycles measured in ZBT_MAIN = %d n", clock_num);
        break;
      }
    }
    // Execution Steps Done

    //=============================================================================
    // read data from the FPGA board

    fpgaSpace[COMMAND_REG] = cmd_Read; // read memory mode + banks access from the host (Pentium)
    fpgaSpace[COMMAND_REG];

    // memory bank 2 is moved to rambuf array
    //---------------- dest            src     size     mode
    status = readSSRAM(rambuf+bankSize, bank_2, bankSize, dma);

    if (status != ADMXRC2_SUCCESS) {
      printf("Error: failed to read SSRAM n");
      exit(1);
    }
    // Memory read DONE
    //=============================================================================
    printf("CC measured in MB1 = %d n", rambuf[bank_2 + HALF_MEM]);

    // Store the raw image
    fh4 = mwopen("car_sobel.raw");

    for (int k = 0; k < 448*273; k++) {
      bputc(rambuf[bank_2+k],fh4);
    }

    mclose(fh4);
    system("convert -depth 8 -size 448x273 gray:car_sobel.raw car_sobel.jpg");
    return;
}
```

Figure 7.8: Host processor main program code.

Chapter 8

Conclusions and future work

8.1 Conclusions

This thesis project was focused on the development of a dynamic scheduling method for Kahn Process Networks nodes onto multiprocessor systems generated by ESPAM, in the context of *many-to-one* mapping. Such a method can be useful when we want to map intrinsic dynamic applications, multiple applications or several instances of the same application in the design.

Firstly we introduced the ESPAM tool design methodology, and how it can close the *Implementation Gap* between System-level specification and RTL specification automatically. Our tool allows efficient and systematic mapping of streaming and multimedia applications on a multiprocessor system. The aim of this project was to improve the ESPAM tool, allowing system designers to exploit advanced software solutions, based on dynamic scheduling of the KPN nodes which comprise the application(s).

The project goal was achieved following the basic steps listed below:

- Adding an operating system to each processor, if more than one node is mapped on it (*many-to-one* mapping).
- Extract program threads that represent all of the KPN nodes which comprise the application(s).
- Create, run and schedule these threads, using operating system API (three scheduling strategies have been tested).

Performing several tests, we found out that the Round-robin scheduling with yielding on blocking is the one that matches more efficiently the KPN behavior. Altough it is very simple to implement, it is effective for dynamic KPN nodes scheduling.

The second aim of this project was the performances comparison between two different operating systems, namely Xilkernel and FreeRTOS. From the implementation results, we can summarize the different OS features as follows.

- **Xilkernel**
  - + With the kernel configuration we adopted, the memory footprint is only 9.9 kB. Actually, the Round-robin with yielding scheduling policy requires only one advanced feature (the yield function). All the rest are default settings, that lead to this small memory footprint size.
  - + This operating system is very easy to implement on our systems because it is highly integrated with the Xilinx Platform Studio framework. Adding and setting this OS is a matter of few lines in the MSS project file.
  - - Custom kernel modifications are less easy than with FreeRTOS.
  - - The actual test performances are slightly worse than the FreeRTOS ones.
- **FreeRTOS**
  - + Three source code files contain all the basic kernel functionalities. They are predominantly written in C and are well understandable, so custom kernel modifications are easy to make.
  - + This operating system performance is slightly better than the Xilkernel.
  - - The memory footprint is almost 10% larger than the Xilkernel one.
  - - Adding this operating system in our projects is more tricky because FreeRTOS is not integrated in Xilinx Platform Studio framework.

These Operating Systems features are summarized in the following table:

Operating System	Performance	Complexity	Kernel customization	Memory footprint
Xilkernel	slightly slower	lower	harder	9.888 kB
FreeRTOS	slightly faster	higher	easier	10.606 kB

Table 8.1: Operating Systems features comparison.

So, when coosing an operating system to add to a processor, a trade-off decision has to be made. Xilkernel is easy to implement and gives a slightly smaller overhead in terms of memory occupation, while FreeRTOS is more customizable and slightly faster.

8.2 Future work

This work on dynamic scheduling implementation can be continued in several aspects.

The first one can be a custom kernel modification in order to fit our goal. The solution we presented in Section 5.2.5 to avoid useless context switches is based on priority scheduling and an additional control thread, but the performances are not optimal, because of the slowness in priority increasing and decreasing. Instead of using a control thread that checks if threads are able to run, the global kernel scheduler can include a similar function. Before scheduling a thread, it can scan if this thread is still blocked on a FIFO.

This kernel modification is not based on priorities, thus we can assume that it could be faster than our implementation. On the other hand, this customization surely adds some complexity to the global kernel scheduler, thus the scheduling operation could be slower. Performance improvement can occur if the time saved in “clever” scheduling dominates the scheduling complexity overhead.

Another aspect that can improve this work is the complete automation in processor code and MHS, MSS files generation. All of the modifications of the ESPAM-generated systems, described in this thesis, were made by hand. Changes in the MHS and MSS files are needed to add an operating system to a processor and the external timer that generates a periodic interrupt. Furthermore, program code modifications have to be made in order to start the global scheduler, create threads and modify the scheduling order.

All this work can be fully automated, in order to make dynamic scheduling solutions on our multiprocessor systems easier to use. This can lead to a faster design space exploration, a fundamental concept in contemporary system design.

Appendix A

Main program code of the Sobel application

```
#include "sobel_func.h"

int N = 450;
#pragma parameter N 450 1000
int M = 275;
#pragma parameter M 275 1000

int main(void)
{
   int i, j;

   static int image[1000][1000];
   static int Jx[1000][1000];
   static int Jy[1000][1000];
   static int av[1000][1000];

   for (j=1; j <= M; j++) {
      for (i=1; i <= N; i++) {
         readPixel(&image[j][i]);
      }
   }

   for (j=2; j <= M-1; j++) {
      for (i=2; i <= N-1; i++) {
         gradient( &image[j-1][i-1], &image[j][i-1], &image[j+1][i-1], ...
                   &image[j-1][i+1], &image[j][i+1], &image[j+1][i+1], &Jx[j][i] );
      }
   }

   for (j=2; j <= M-1; j++) {
      for (i=2; i <= N-1; i++) {
         gradient( &image[j-1][i-1], &image[j-1][i], &image[j-1][i+1], ...
                   &image[j+1][i-1], &image[j+1][i], &image[j+1][i+1], &Jy[j][i] );
      }
   }

   for (j=2; j <= M-1; j++) {
      for (i=2; i <= N-1; i++) {
          absVal( &Jx[j][i], &Jy[j][i], &av[j][i] );
      }
   }

   for (j=2; j <= M-1; j++) {
      for (i=2; i <= N-1; i++) {
          writePixel( &av[j][i] );
      }
   }

   return (0);
}
```

Appendix B

MHS File for Sobel application mapped onto a one-processor system

```
1     PARAMETER VERSION = 2.1.0

      PORT lclk = lclk, DIR = I
      PORT mclk = mclk, DIR = I
5     PORT ramclki = ramclki, VEC = [1:0], DIR = I
      PORT ramclko = ramclko, VEC = [1:0], DIR = O
      PORT lreseto_l = lreseto_l, DIR = I
      PORT lwrite = lwrite, DIR = I
      PORT lads_l = lads_l, DIR = I
10    PORT lblast_l = lblast_l, DIR = I
      PORT lbterm_l = lbterm_l, DIR = IO
      PORT ld = ld, VEC = [31:0], DIR = IO
      PORT la = la, VEC = [23:2], DIR = I
      PORT lreadyi_l = lreadyi_l, DIR = O
15    PORT lbe_l = lbe_l, VEC = [3:0], DIR = I
      PORT fholda = fholda, DIR = I
      PORT ra0 = ra0, VEC = [19:0], DIR = O
      PORT rd0 = rd0, VEC = [31:0], DIR = IO
      PORT rc0 = rc0, VEC = [8:0], DIR = O
20    PORT ra1 = ra1, VEC = [19:0], DIR = O
      PORT rd1 = rd1, VEC = [31:0], DIR = IO
      PORT rc1 = rc1, VEC = [8:0], DIR = O
      PORT ra2 = ra2, VEC = [19:0], DIR = O
      PORT rd2 = rd2, VEC = [31:0], DIR = IO
25    PORT rc2 = rc2, VEC = [8:0], DIR = O
      PORT ra3 = ra3, VEC = [19:0], DIR = O
      PORT rd3 = rd3, VEC = [31:0], DIR = IO
      PORT rc3 = rc3, VEC = [8:0], DIR = O
      PORT ra4 = ra4, VEC = [19:0], DIR = O
30    PORT rd4 = rd4, VEC = [31:0], DIR = IO
      PORT rc4 = rc4, VEC = [8:0], DIR = O
      PORT ra5 = ra5, VEC = [19:0], DIR = O
      PORT rd5 = rd5, VEC = [31:0], DIR = IO
      PORT rc5 = rc5, VEC = [8:0], DIR = O
35

     BEGIN lmb_v10
      PARAMETER INSTANCE =      PBUS_MB_1
      PARAMETER HW_VER = 1.00.a
40    PARAMETER C_EXT_RESET_HIGH = 0
      PORT SYS_Rst = net_design_rst
      PORT LMB_Clk = sys_clk_s
     END

45   BEGIN lmb_v10
      PARAMETER INSTANCE = DBUS_MB_1
      PARAMETER HW_VER = 1.00.a
      PARAMETER C_EXT_RESET_HIGH = 0
      PORT SYS_Rst = net_design_rst
50    PORT LMB_Clk = sys_clk_s
     END

     BEGIN opb_v20
      PARAMETER INSTANCE = mb_opb_1
55    PARAMETER HW_VER = 1.10.c
      PARAMETER C_EXT_RESET_HIGH = 0
      PORT SYS_Rst = net_design_rst
      PORT OPB_Clk = sys_clk_s
     END
60
     BEGIN fin_ctrl
      PARAMETER INSTANCE = fin_ctrl_P1
      PARAMETER HW_VER = 1.00.a
      PARAMETER C_BASEADDR = 0xf9000000
65    PARAMETER C_HIGHADDR = 0xf900000f
      PARAMETER C_AB = 8
      BUS_INTERFACE SLMB = DBUS_MB_1
      PORT Sl_FinOut = net_fin_signal_P1
     END
70
     BEGIN clock_cycle_counter
      PARAMETER INSTANCE = clock_cycle_counter_P1
      PARAMETER HW_VER = 1.00.a
      PARAMETER C_BASEADDR = 0xf8000000
75    PARAMETER C_HIGHADDR = 0xf8000003
      BUS_INTERFACE SLMB = DBUS_MB_1
      PORT LMB_Clk = sys_clk_s
     END

80   BEGIN microblaze
      PARAMETER INSTANCE = MB_1
      PARAMETER HW_VER = 4.00.a
      PARAMETER C_NUMBER_OF_PC_BRK = 1
      PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
85    PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
      PARAMETER C_FSL_LINKS = 0
      BUS_INTERFACE DLMB = DBUS_MB_1
      BUS_INTERFACE ILMB = PBUS_MB_1
      BUS_INTERFACE DOPB = mb_opb_1
90    PORT CLK = sys_clk_s
      PORT INTERRUPT = MB_1_INTERRUPT
     END
```

```
BEGIN zbt_main
 PARAMETER INSTANCE = host_zbt_main
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE HOST_BUFF_0_PORT = buff_rd_0
 BUS_INTERFACE HOST_BUFF_1_PORT = buff_rd_1
 BUS_INTERFACE HOST_BUFF_2_PORT = buff_rd_2
 BUS_INTERFACE HOST_BUFF_3_PORT = buff_rd_3
 BUS_INTERFACE HOST_BUFF_4_PORT = buff_rd_4
 BUS_INTERFACE HOST_BUFF_5_PORT = buff_rd_5
 BUS_INTERFACE HOST_MUX_PORT = mux_to_host
 PORT lclk = lclk
 PORT mclk = mclk
 PORT ramclko = ramclko
 PORT ramclki = ramclki
 PORT lreseto_l = lreseto_l
 PORT lwrite = lwrite
 PORT lads_l = lads_l
 PORT lblast_l = lblast_l
 PORT lbterm_l = lbterm_l
 PORT ld = ld
 PORT la = la
 PORT lreadyi_l = lreadyi_l
 PORT lbe_l = lbe_l
 PORT fholda = fholda
 PORT CLK_out = sys_clk_s
 PORT RST_out = sys_rst_s
 PORT COMMAND_REG = net_command
 PORT DESIGN_STAT_REG = net_design_status
 PORT PARAMETER_REG = net_parameter
END

BEGIN host_design_ctrl
 PARAMETER INSTANCE = host_design_controller
 PARAMETER HW_VER = 1.00.a
 PARAMETER N_FIN = 1
 PARAMETER PAR_WIDTH = 16
 PORT RST = sys_rst_s
 PORT COMMAND_REG = net_command
 PORT STATUS_REG = net_design_status
 PORT      PARAMETER_REG = net_parameter
 PORT RST_OUT = net_design_rst
 PORT FIN_REG_0 = net_fin_signal_P1
END

BEGIN mux
 PARAMETER INSTANCE = multiplexer
 PARAMETER HW_VER = 1.00.a
 PARAMETER N_MUX = 2
 BUS_INTERFACE MUX_BUFF_PORT = buff_to_mux
 BUS_INTERFACE MUX_DESIGN_0_PORT = mux_design_0
 BUS_INTERFACE MUX_DESIGN_1_PORT = mux_design_1
 BUS_INTERFACE MUX_HOST_PORT = mux_to_host
 PORT ra0 = ra0
 PORT ra1 = ra1
 PORT ra2 = ra2
 PORT ra3 = ra3
 PORT ra4 = ra4
 PORT ra5 = ra5
 PORT rc0 = rc0
 PORT rc1 = rc1
 PORT rc2 = rc2
 PORT rc3 = rc3
 PORT rc4 = rc4
 PORT rc5 = rc5
 PORT RST = sys_rst_s
 PORT CNTRL = net_command
END

BEGIN buffers
 PARAMETER INSTANCE = buff
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE BUFF_MUX_PORT = buff_to_mux
 BUS_INTERFACE BUFF_RD_0_PORT = buff_rd_0
 BUS_INTERFACE BUFF_RD_1_PORT = buff_rd_1
 BUS_INTERFACE BUFF_RD_2_PORT = buff_rd_2
 BUS_INTERFACE BUFF_RD_3_PORT = buff_rd_3
 BUS_INTERFACE BUFF_RD_4_PORT = buff_rd_4
 BUS_INTERFACE BUFF_RD_5_PORT = buff_rd_5
 PORT rd0 = rd0
 PORT rd1 = rd1
 PORT rd2 = rd2
 PORT rd3 = rd3
 PORT rd4 = rd4
 PORT rd5 = rd5
END

BEGIN opb_zbt_controller
 PARAMETER INSTANCE = ZBT_CTRL_1
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf0000000
 PARAMETER C_HIGHADDR = 0xf00fffff
 PARAMETER C_EXTERNAL_DLL = 1
 PARAMETER C_ZBT_ADDR_SIZE = 20
 BUS_INTERFACE SOPB = mb_opb_1
 BUS_INTERFACE DESIGN_BUFF_PORT = buff_rd_0
 BUS_INTERFACE DESIGN_MUX_PORT = mux_design_0
END

BEGIN opb_zbt_controller
 PARAMETER INSTANCE = ZBT_CTRL_2
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xf0100000
 PARAMETER C_HIGHADDR = 0xf01fffff
 PARAMETER C_EXTERNAL_DLL = 1
 PARAMETER C_ZBT_ADDR_SIZE = 20
 BUS_INTERFACE SOPB = mb_opb_1
 BUS_INTERFACE DESIGN_BUFF_PORT = buff_rd_1
 BUS_INTERFACE DESIGN_MUX_PORT = mux_design_1
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_1
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_2
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_3
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_4
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_5
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_6
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END
```

```
BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_7
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_8
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_9
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_10
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_11
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_12
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_13
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_14
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_15
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_16
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_17
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_18
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_19
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_20
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_21
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END
```

```
BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_22
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_23
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_24
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_25
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_26
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_27
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_28
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fsl_v20
 PARAMETER HW_VER = 2.10.a
 PARAMETER INSTANCE = FIFO_MB_1_Out_29
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_ASYNC_CLKS = 0
 PARAMETER C_IMPL_STYLE = 1
 PARAMETER C_USE_CONTROL = 0
 PARAMETER C_FSL_DWIDTH = 32
 PARAMETER C_FSL_DEPTH = 1024
 PORT FSL_Clk = sys_clk_s
 PORT SYS_Rst = net_design_rst
END

BEGIN fifo_if_ctrl
 PARAMETER INSTANCE = CTRL_MB_1_FIFOs
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xc0000000
 PARAMETER C_HIGHADDR = 0xc0ffffff
 PARAMETER C_AB = 8
 PARAMETER C_FIFO_WRITE = 29
 PARAMETER C_FIFO_READ = 29
 BUS_INTERFACE FIFO_READ_1 = FIFO_MB_1_Out_1
 BUS_INTERFACE FIFO_READ_2 = FIFO_MB_1_Out_2
 BUS_INTERFACE FIFO_READ_3 = FIFO_MB_1_Out_3
 BUS_INTERFACE FIFO_READ_4 = FIFO_MB_1_Out_4
 BUS_INTERFACE FIFO_READ_5 = FIFO_MB_1_Out_5
 BUS_INTERFACE FIFO_READ_6 = FIFO_MB_1_Out_6
 BUS_INTERFACE FIFO_READ_7 = FIFO_MB_1_Out_7
 BUS_INTERFACE FIFO_READ_8 = FIFO_MB_1_Out_8
 BUS_INTERFACE FIFO_READ_9 = FIFO_MB_1_Out_9
 BUS_INTERFACE FIFO_READ_10 = FIFO_MB_1_Out_10
 BUS_INTERFACE FIFO_READ_11 = FIFO_MB_1_Out_11
 BUS_INTERFACE FIFO_READ_12 = FIFO_MB_1_Out_12
 BUS_INTERFACE FIFO_READ_13 = FIFO_MB_1_Out_13
 BUS_INTERFACE FIFO_READ_14 = FIFO_MB_1_Out_14
 BUS_INTERFACE FIFO_READ_15 = FIFO_MB_1_Out_15
 BUS_INTERFACE FIFO_READ_16 = FIFO_MB_1_Out_16
 BUS_INTERFACE FIFO_READ_17 = FIFO_MB_1_Out_17
 BUS_INTERFACE FIFO_READ_18 = FIFO_MB_1_Out_18
 BUS_INTERFACE FIFO_READ_19 = FIFO_MB_1_Out_19
 BUS_INTERFACE FIFO_READ_20 = FIFO_MB_1_Out_20
 BUS_INTERFACE FIFO_READ_21 = FIFO_MB_1_Out_21
 BUS_INTERFACE FIFO_READ_22 = FIFO_MB_1_Out_22
 BUS_INTERFACE FIFO_READ_23 = FIFO_MB_1_Out_23
 BUS_INTERFACE FIFO_READ_24 = FIFO_MB_1_Out_24
 BUS_INTERFACE FIFO_READ_25 = FIFO_MB_1_Out_25
 BUS_INTERFACE FIFO_READ_26 = FIFO_MB_1_Out_26
 BUS_INTERFACE FIFO_READ_27 = FIFO_MB_1_Out_27
 BUS_INTERFACE FIFO_READ_28 = FIFO_MB_1_Out_28
 BUS_INTERFACE FIFO_READ_29 = FIFO_MB_1_Out_29
 BUS_INTERFACE FIFO_WRITE_1 = FIFO_MB_1_Out_1
 BUS_INTERFACE FIFO_WRITE_2 = FIFO_MB_1_Out_2
 BUS_INTERFACE FIFO_WRITE_3 = FIFO_MB_1_Out_3
 BUS_INTERFACE FIFO_WRITE_4 = FIFO_MB_1_Out_4
 BUS_INTERFACE FIFO_WRITE_5 = FIFO_MB_1_Out_5
 BUS_INTERFACE FIFO_WRITE_6 = FIFO_MB_1_Out_6
 BUS_INTERFACE FIFO_WRITE_7 = FIFO_MB_1_Out_7
 BUS_INTERFACE FIFO_WRITE_8 = FIFO_MB_1_Out_8
 BUS_INTERFACE FIFO_WRITE_9 = FIFO_MB_1_Out_9
 BUS_INTERFACE FIFO_WRITE_10 = FIFO_MB_1_Out_10
 BUS_INTERFACE FIFO_WRITE_11 = FIFO_MB_1_Out_11
 BUS_INTERFACE FIFO_WRITE_12 = FIFO_MB_1_Out_12
 BUS_INTERFACE FIFO_WRITE_13 = FIFO_MB_1_Out_13
 BUS_INTERFACE FIFO_WRITE_14 = FIFO_MB_1_Out_14
 BUS_INTERFACE FIFO_WRITE_15 = FIFO_MB_1_Out_15
 BUS_INTERFACE FIFO_WRITE_16 = FIFO_MB_1_Out_16
 BUS_INTERFACE FIFO_WRITE_17 = FIFO_MB_1_Out_17
 BUS_INTERFACE FIFO_WRITE_18 = FIFO_MB_1_Out_18
 BUS_INTERFACE FIFO_WRITE_19 = FIFO_MB_1_Out_19
 BUS_INTERFACE FIFO_WRITE_20 = FIFO_MB_1_Out_20
 BUS_INTERFACE FIFO_WRITE_21 = FIFO_MB_1_Out_21
 BUS_INTERFACE FIFO_WRITE_22 = FIFO_MB_1_Out_22
 BUS_INTERFACE FIFO_WRITE_23 = FIFO_MB_1_Out_23
 BUS_INTERFACE FIFO_WRITE_24 = FIFO_MB_1_Out_24
 BUS_INTERFACE FIFO_WRITE_25 = FIFO_MB_1_Out_25
 BUS_INTERFACE FIFO_WRITE_26 = FIFO_MB_1_Out_26
 BUS_INTERFACE FIFO_WRITE_27 = FIFO_MB_1_Out_27
 BUS_INTERFACE FIFO_WRITE_28 = FIFO_MB_1_Out_28
 BUS_INTERFACE FIFO_WRITE_29 = FIFO_MB_1_Out_29
 BUS_INTERFACE SLMB = DBUS_MB_1
END
```

```
BEGIN bram_block
 PARAMETER INSTANCE = BRAM1_MB_1
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTA = BUS_DCTRL_BRAM1_MB_1
 BUS_INTERFACE PORTB = BUS_PCTRL_BRAM1_MB_1
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = DCTRL_BRAM1_MB_1
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x0000ffff
 BUS_INTERFACE SLMB = DBUS_MB_1
 BUS_INTERFACE BRAM_PORT = BUS_DCTRL_BRAM1_MB_1
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = PCTRL_BRAM1_MB_1
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x0000ffff
 BUS_INTERFACE SLMB = PBUS_MB_1
 BUS_INTERFACE BRAM_PORT = BUS_PCTRL_BRAM1_MB_1
END

BEGIN bram_block
 PARAMETER INSTANCE = BRAM2_MB_1
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTA = BUS_DCTRL_BRAM2_MB_1
 BUS_INTERFACE PORTB = BUS_PCTRL_BRAM2_MB_1
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = DCTRL_BRAM2_MB_1
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00010000
 PARAMETER C_HIGHADDR = 0x00017fff
 BUS_INTERFACE SLMB = DBUS_MB_1
 BUS_INTERFACE BRAM_PORT = BUS_DCTRL_BRAM2_MB_1
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = PCTRL_BRAM2_MB_1
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_MASK = 0xff000000
 PARAMETER C_BASEADDR = 0x00010000
 PARAMETER C_HIGHADDR = 0x00017fff
 BUS_INTERFACE SLMB = PBUS_MB_1
 BUS_INTERFACE BRAM_PORT = BUS_PCTRL_BRAM2_MB_1
END

BEGIN opb_timer
 PARAMETER INSTANCE = opb_timer_0
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_BASEADDR = 0xF1000000
 PARAMETER C_HIGHADDR = 0xF100FFFF
 BUS_INTERFACE SOPB = mb_opb_1
 PORT Interrupt = MB_1_INTERRUPT
END
```

Appendix C

MSS File for Sobel application mapped onto a one-processor system

```
PARAMETER VERSION = 2.2.0


BEGIN OS
 PARAMETER OS_NAME = xilkernel
 PARAMETER OS_VER = 3.00.a
 PARAMETER PROC_INSTANCE = MB_1
 PARAMETER enhanced_features = true
 PARAMETER systmr_dev = opb_timer_0
 PARAMETER config_yield = true
 PARAMETER max_pthreads = 8
 PARAMETER systmr_interval = 500
 PARAMETER static_pthread_table = ((thread_main,1))
END


BEGIN PROCESSOR
 PARAMETER DRIVER_NAME = cpu
 PARAMETER DRIVER_VER = 1.01.a
 PARAMETER HW_INSTANCE = MB_1
 PARAMETER COMPILER = mb-gcc
 PARAMETER ARCHIVER = mb-ar
END


BEGIN DRIVER
 PARAMETER DRIVER_NAME = opbarb
 PARAMETER DRIVER_VER = 1.02.a
 PARAMETER HW_INSTANCE = mb_opb_1
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = fin_ctrl_P1
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = clock_cycle_counter_P1
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = host_zbt_main
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = host_design_controller
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = multiplexer
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = buff
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = ZBT_CTRL_1
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = ZBT_CTRL_2
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_1
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_2
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_3
END
```

```
BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_4
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_5
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_6
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_7
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_8
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_9
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_10
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_11
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_12
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_13
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_14
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_15
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_16
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_17
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_18
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_19
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_20
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_21
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_22
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_23
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_24
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_25
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_26
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_27
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_28
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = FIFO_MB_1_Out_29
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = CTRL_MB_1_FIFOs
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = bram
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = DCTRL_BRAM1_MB_1
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = bram
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = PCTRL_BRAM1_MB_1
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = bram
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = DCTRL_BRAM2_MB_1
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = bram
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = PCTRL_BRAM2_MB_1
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = tmrctr
 PARAMETER DRIVER_VER = 1.00.b
 PARAMETER HW_INSTANCE = opb_timer_0
END
```

Appendix D

MicroBlaze program code for the multi-threaded Sobel application

```
#include "xmk.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <os_config.h>
#include <sys/process.h>
#include <sys/timer.h>
#include <pthread.h>
#include "MemoryMap.h"
#include "aux_func.h"

#define HALF_MEM 125000

void* thread1(void *arg)
{
    int c0, c1;
    // Input Arguments
    // Output Arguments
    tCH_24 out_0ND_0;

      for( c0 =  ceil1(1); c0 <=  floor1(M ); c0 += 1 ) {
        for( c1 =  ceil1(1); c1 <=  floor1(N ); c1 += 1 ) {

            _readPixel(&out_0ND_0) ;

            ...writing to output FIFOs is omitted...

        } // for c1
      } // for c0
} // thread 1

void* thread2(void *arg)
{
    int c0, c1;
    // Input Arguments
    tCH_18 in_0ND_2;
    tCH_20 in_1ND_2;
    tCH_23 in_2ND_2;
    tCH_25 in_3ND_2;
    tCH_26 in_4ND_2;
    tCH_26 in_5ND_2;
    // Output Arguments
    tCH_28 out_6ND_2;

      for( c0 =  ceil1(3); c0 <=  floor1(M ); c0 += 1 ) {
        for( c1 =  ceil1(3); c1 <=  floor1(N ); c1 += 1 ) {

            ...reading from input FIFOs is omitted...

            _gradient(in_0ND_2, in_1ND_2, in_2ND_2, in_3ND_2, in_4ND_2, in_5ND_2, &out_6ND_2) ;

            ...writing to output FIFOs is omitted...

        } // for c1
      } // for c0
}  // thread2
```

```
void* thread3(void *arg)
{
    int c0, c1;
    // Input Arguments
    tCH_27 in_0ND_3;
    tCH_28 in_1ND_3;
    // Output Arguments
    tCH_29 out_2ND_3;

      for( c0 =  ceil1(3); c0 <=  floor1(M ); c0 += 1 ) {
        for( c1 =  ceil1(3); c1 <=  floor1(N ); c1 += 1 ) {

            ...reading from input FIFOs is omitted...

            _gradient(in_0ND_1, in_1ND_1, in_2ND_1, in_3ND_1, in_4ND_1, in_5ND_1, &out_6ND_1) ;

            ...writing to output FIFOs is omitted...

        } // for c1
      } // for c0
} // thread3


void* thread4(void *arg)
{
    int c0, c1;
    // Input Arguments
    tCH_27 in_0ND_3;
    tCH_28 in_1ND_3;
    // Output Arguments
    tCH_29 out_2ND_3;

      for( c0 =  ceil1(3); c0 <=  floor1(M ); c0 += 1 ) {
        for( c1 =  ceil1(3); c1 <=  floor1(N ); c1 += 1 ) {

            read(ND_3_IG_27_CH_27, &in_0ND_3, (sizeof(tCH_27)+(sizeof(tCH_27)%4)+3)/4);
            read(ND_3_IG_28_CH_28, &in_1ND_3, (sizeof(tCH_28)+(sizeof(tCH_28)%4)+3)/4);

            _absVal(in_0ND_3, in_1ND_3, &out_2ND_3) ;

            write(ND_3_OG_29_CH_29, &out_2ND_3, (sizeof(tCH_29)+(sizeof(tCH_29)%4)+3)/4);

        } // for c1
      } // for c0
} //thread4

void* thread5(void *arg)
{
   int c0, c1;
   // Input Arguments
   tCH_29 in_0ND_4;

   for( c0 =  ceil1(3); c0 <=  floor1(M ); c0 += 1 ) {
     for( c1 =  ceil1(3); c1 <=  floor1(N ); c1 += 1 ) {

       read(ND_4_IG_29_CH_29, &in_0ND_4, (sizeof(tCH_29)+(sizeof(tCH_29)%4)+3)/4);

       _writePixel(in_0ND_4) ;

     } // for c1
   } // for c0

   *(ZBT_MEM_2+HALF_MEM) = *clk_cntr;
   *FIN_SIGNAL = (volatile long)0x00000001;
} // thread5

void* thread_main( void *dummy)
{
        int i, ret;
        pthread_t threadID[5];
        int clk_num;
        *clk_cntr = 0;

        ret = pthread_create(&threadID[0], NULL, (void*)thread1, NULL);
        if (ret) *(ZBT_MEM_2+HALF_MEM)=666;
        ret = pthread_create(&threadID[1], NULL, (void*)thread2, NULL);
        if (ret) *(ZBT_MEM_2+HALF_MEM+1)=666;
        ret = pthread_create(&threadID[2], NULL, (void*)thread3, NULL);
        if (ret) *(ZBT_MEM_2+HALF_MEM+2)=666;
        ret = pthread_create(&threadID[3], NULL, (void*)thread4, NULL);
        if (ret) *(ZBT_MEM_2+HALF_MEM+3)=666;
        ret = pthread_create(&threadID[4], NULL, (void*)thread5, NULL);
        if (ret) *(ZBT_MEM_2+HALF_MEM+4)=666;

        return 0 ;
}

int main ()
{
    xilkernel_main();
}
```

Bibliography

[1] Hristo Nikolov, Todor Stefanov, Ed Deprettere. “Multi-processor System Design with ESPAM”. *CODES+ISSS’06*, October 22-25, Seoul, Korea.

[2] Hristo Nikolov, Todor Stefanov, Ed Deprettere. “Systematic and Automated Multi-processor System Design, Programming, and Implementation”. June 01, 2007. Leiden Institute of Advanced Computer Science, Leiden, The Netherlands.

[3] K.Keutzer, R.Newton. SEMATECH, 1997.
http://www.sematech.org/public/home.htm

[4] A. Jerraya et al., “Programming Models and HW-SW Interfaces Abstraction for MultiProcessor SoC”, in *Proc. DAC*, July 2006.

[5] P. Paulin et al., “Parallel Programming Models for a Multiprocessor SoC Platform Applied to Networking and Multimedia”, *IEEE Trans. on VLSI Systems*, vol. 14, no. 7, July 2006.

[6] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink, “Design and Programming of Embedded Multiprocessors: an Interface-Centric Approach”, in *Proc. CODES+ISSS*, Sept. 2004.

[7] Wei Zhong. “Embedded System-level Platform Synthesis and Application Mapping for Heterogeneous and Hierarchical Multiprocessor Systems”. Master thesis, 2006, Leiden Institute of Advanced Computer Science, Leiden, The Netherlands.

[8] Ying Tao. “Heterogeneous Multiprocessor System Design with ESPAM: Integration of Hardware IP Cores”. Master thesis, 2006, Leiden Institute of Advanced Computer Science, Leiden, The Netherlands.

[9] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[10] Paul Feautrier. “Parametric Integer Programming”. September 1988, Laboratoire MASI, Institut Blaise Pascal.

[11] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. “PN: A Tool for Improved Derivation of Process Networks”. EURASIP Journal on Embedded Systems Volume 2007, Article ID 75947.

[12] Bryan H. Fletcher. “FPGA Embedded Processors. Revealing True System Performance”. Embedded System Conference, San Francisco, 2005.

[13] Platform studio user guide: Xilinx, inc.
http://www.xilinx.com/ise/embedded/edk_docs.htm

[14] Embedded Systems Tools Reference Manual, Embedded Development Kit, EDK 9.1i
http://www.xilinx.com/ise/embedded/edk91i_docs/est_rm.pdf

[15] Xilkernel 3.00a description
http://www.xilinx.com/ise/embedded/edk91i_docs/xilkernel_v3_00_a.pdf

[16] EDK OS and Libraries Reference Manual
http://www.xilinx.com/ise/embedded/edk91i_docs/oslib_rm.pdf

[17] The FreeRTOS.org Project
http://www.freertos.org/

[18] Arnaud Lagger. “Self-Reconfigurable platform for cryptographic application”. Master Thesis, 2006. School of Computer and Communication Sciences, Swiss Federal Institute of Technology Lausanne.

[19] Anders Rönnholm. “Evaluation of Real-Time Operating Systems for Xilinx MicroBlaze CPU”. Final thesis, 2006, Mälardens University, Department of Computer Science and Electronics.

[20] Alpha Data. ADM-XRC-II, PCI Mezzanine Card, User Guide, Version 1.5. Alpha Data Parallel Systems Ltd, 2002.

[21] POSIX Threads Programming, Lawrence Livermore National Laboratory.

[22] http://en.wikipedia.org/wiki/Thread_%28computer_science%29

[23] Andrew S. Tanenbaum, “Modern Operating Systems”, 2nd Edition. Prentice Hall, February 2001.

[24] MicroBlaze Processor Reference Guide
http://www.xilinx.com/support/documentation/ip_documentation/microblaze.pdf

[25] Fast simplex link (fsl) bus (v2.00a), Xilinx, inc.
http://www.xilinx.com/bvdocs/ipcenter/data sheet/FSL_V20.pdf.

[26] OPB Timer/Counter documentation.
http://www.xilinx.com/products/ipcenter/OPB_Timer_Counter.htm