Alma Mater Studiorum - University of Bologna

DEIS - DEPARTMENT OF ELECTRONICS, COMPUTER SCIENCE AND SYSTEMS

PhD Course in Electronics, Computer Science and Telecommunications

CYCLE XXIV – Scientific-Disciplinary sector ING-INF /01

MULTI-PROCESSOR SYSTEMS-ON-CHIP WITH CONFIGURABLE HARDWARE ACCELERATION

Candidate:

DAVIDE ROSSI

PhD course coordinator:

PROF. LUCA BENINI

Advisor:

PROF. ROBERTO GUERRIERI

Co-Advisor:

PROF. ELEONORA FRANCHI SCARSELLI

Final examination year : 2012

Keywords:

Multi-Processor Systems on Chip

Reconfigurable Computing

Structured ASIC

HW/SW Co-Design

Digital Signal Processing

List of Abbreviations

PDA	Personal Digital Assistant
ASSP	Application Specific Signal Processor
DSP	Digital Signal Processor
ASIC	Application Specific Integrated Circuit
FPGA	Field Programmable Gate Array
IP	Intellectual Property
CGRA	Coarse Grain Reconfigurable Array
ALU	Arithmetic Logic Unit
NRE	Non Recurrent Engineering (Costs)
MPSoC	Multi-Processor System-on-Chip
SIMD	Single Instruction Multiple Data
VLIW	Very Long Instruction Word
RTOS	Real Time Operating System
RISC	Reduced Instruction Set Computer
CPU	Central Processing Unit
SRAM	Static Random Access Memory
DFG	Data Flow Graph
GPP	General Purpose Processor
PE	Processing Element
LUT	Lookup Table
RC	Reconfigurable Cell
PiCoGA	Pipelined Configurable Gate Array
PAE	Processing Array Element
FIFO	First In First Out
MPI	Message Passing Interface
CUDA	Computer Unified Device Architecture
GP-GPU	General-Purpose Graphic Processing Unit
PPE	Power Processing Element
SPE	Synergistic Processor Element
FLOPS	Floating point Operations Per Second
RTL	Register Transfer Level
VHDL	VHSIC Hardware Description Language

VHSIC	Very High Speed Integrated Circuits
MAC	Multiply And Accumulate
ISA	Instruction Set Architecture
NML	Native Mapping Language
PN	Petri Network
KPN	Kahn Process Network
TCM	Tightly Coupled Memory
DEB	Data Exchange Buffer
CEB	Configuration Exchange Buffer
XR	Exchange Register
NoC	Network on Chip
PLL	Phase Locked Loop
GALS	Globally Asynchronous Locally Synchronous
PCM	Pre-fetch Configuration Manager
FFT	Fast Fourier Transform
DCT	Discrete Cosine Transform
CRC	Cyclic Redundancy Check
GF	Galois Field
ME	Motion Estimation
MC	Motion Compensation
FSM	Finite State Machine
LAN	Local Area Network
GPIO	General Purpose Input Output
SAD	Subtraction and Absolute Difference
RLC	Reconfigurable Logic Cell
CT	Computational Tile
IOT	Input Output Tile
AG	Address Generator
TTM	Time To Market
TLM	Transaction Level Model
CAVLC	Context Adaptive Variable Length Coding
CABAC	Context Adaptive Binary Arithmetic Coding

Contents

List of Figures

List Of Tables

Introduction

During the last few years, the markets for mobile phones, PDAs, portable console, network routers and other specialized high-performance electronic devices have raised explosively. Many of these devices perform computationally demanding signal processing algorithms that are even increasing with the evolution of the applications standards. Moreover, the portability requirements of these devices are growing as well, putting other severe constraints on the energy efficiency demands of such signal processing systems. From the commercial point of view, some major semiconductor industries have proposed many digital signal processors for embedded or portable computing in last few years.

Most of these devices belong to the category of Application Specific Signal Processor (ASSP). They are able to match the computational and energy requirements of the applications thanks to exploitation of powerful Digital Signal Processors (DSP) and hardwired application specific accelerators, usually managed by a standard controller core supporting operating systems in order to ease programmability. Though they form a very large slice of the signal processing market, these devices are not always suited to following the evolution of application standards due to the specificity of their accelerators, so that every time a new standard is deployed, a new device needs to be redesigned. The need for devising specific accelerators for each kernel reduces the possibility of using existing IPs, forcing a large portion of the system to be re-designed and re-verified every time a new application is developed. Moreover, long design and verification times caused may dramatically reduce the market volumes attainable by a given product. A second implication is connected with non-recurrent engineering costs, usually affecting all advanced technologies in general and ASSPs in particular, making production viable only for extremely large market volumes.

One possible solution to extend the life f a product by increasing its flexibility lies in reconfigurable computing. Reconfigurable computing means the capability of a device to exploit spatial computation typical of ASIC design, while maintaining programmability typical of general-purpose processors,

thanks to programmable computational elements cooperating through a configurable interconnect. The main representatives of this class of devices are FPGA devices [7][8]. In several fields of embedded signal processing reconfigurable devices are regarded with interest for their capability to provide ASIC-oriented performance figures while retaining the capability of on-the-fly upgrades of the application portfolio. On the other hand, FPGAs are not suitable to many application domains, due to their inherently redundant structure. As reported in [13], around 90% of the area of commercial FPGAs is occupied by interconnect lines and configuration storage. This leads to significant overheads in area, power and computation throughput that can be inconvenient in some fields and downright unaffordable for battery-operated or portable applications. Another issue closely related to the exploitation of FPGAs is programming productivity: hardware related languages are intrinsically more complex and difficult to use with respect to software oriented imperative languages such as C or C++ regardless of the background of the user. While it is possible to rely on pre-packaged libraries and IPs for standard computation kernels, the development and debugging of the top-level wrapping and synchronization stage of the application becomes a significant slowing factor in the application development time.

Where the application environment allows that, it is possible to trade part of the flexibility offered by Field Programmable Gate Arrays (FPGA) designing computing engines based on coarser computation blocks and simplified interconnect patterns. Coarse-Grained Reconfigurable Architectures (CGRA) are a class of run-time programmable signal processors composed by regular arrays of 4- to 32-bit computation units, typically Arithmetic Logical Units (ALUs) with reduced instruction set in place of standard Look-Up Tales (LUTs). The years 2000-2005 have demonstrated an impressive emergence of CGRA IP solutions covering different flavors of hardware configurability. Each of these companies has boosted the reduction of time to market and of NRE costs as major strong points. On the other hand, the acceptance of these solutions in the signal processing market has been rather slow. The reason for this is probably two-fold: first, CGRAs represent a delicate trade-off between being general purpose and having to make severe assumptions on the application range, so that the user is often struggling to match his applications

specs with the resources offered by the architecture. On the other hand, innovative computation patterns inevitably require specific mapping tools and expertise. Predictably, application developers are reluctant in investing in expertise that is specific only to a given architectural solution and/or computation domain.

A novel computation pattern that has enjoyed lately a moderate success is that of processor arrays, and more in general of Multi/Many Processor Systems and Multi-Processors Systems on Chip (MPSoC). Processor arrays could be described as the "upper bound" of CGRAs, in the sense that they represent reconfigurable architectures of maximum granularity. On the other hand, the exploitation of the processor concept allows for easier application mapping. In most cases computation parallelism is exploited at thread level, rather than at instruction level, which is definitely friendlier from the user/toolset point of view. Even from the interconnect perspective, the exploitation of threads mapped on a processor network allows to capitalize on renowned and established legacy.

More generally, the standard concept of System-on-Chip is slowly but steadily migrating towards Multi-Processor Systems-On-Chip. Once again, the immediate drawback is its redundancy, and the complexity of synchronization of both data and configuration flows in case of complex applications. Moreover, processor-oriented computation obviously cannot match the flexibility of FPGAs in case of bit-oriented computation nor the density of CGRAs in case of massively parallel SIMD computation. From the evaluations above it appears that a Multi-Processor approach brings significant benefits in terms of user friendliness and programmability offering a standardized way to handle thread concurrency and data/control flow synchronization. On the other hand, sheer computational density can be obtained only with the massive parallelism of ASIC or configurable hardware accelerators, but that hardware needs to be matched by the features of the application. Although these devices have been very successful, especially for portable applications, where low power and high performance are essential specifications, they remain very domain specific. Indeed, as technology nodes scale, a clear trend in this category of devices is to substitute bus hierarchies with Networks-on-Chip and augment the number of programmable cores, while reducing the number of ASIC accelerators with the ambition of

widening the application domain. Still, massive highly parallel computation kernels, and bit-level manipulations remain critical aspects that can only be managed with specific ASIC acceleration.

As mentioned above, rapid low-cost design, low production cost, low energy consumption, and high performance are becoming key factors in the embedded electronic market. The approach proposed in this thesis to match all these requirements, is to derive application-specific standard products from customizable multi-core platforms. The software programmability based on multiple processor engines addresses flexibility, although it is not always able to match applications constraints. For this, flexible specialization of processors [5] can be a way to evolve during the life cycle of a product through incremental enhancement of pre-existing engines. In the context of this thesis flexibility of customization can either be provided by run-time configurable (re-configurable) technologies, or design-time configurable technologies, for example based on structured-ASIC solutions such as via-programmable or metal-programmable gate array.

In this scenario, high-level design methodologies are required to support the user in this specialization task, in order to provide easy exploration of the hardware/software co-implementation of applications over the target platform. A specific target of this thesis is to evaluate the application space of multi-processor systems with configurable hardware accelerations, analyzing trade-offs between programming productivity, performance and flexibility of the mapping of applications over multiple cores platforms and the partitioning of kernels between software and different kinds of configuration technologies. Moreover, the analysis will move through the different kinds of configuration technology utilized, being either run-time configurable or based on structured-ASIC technologies analyzing their benefits and overheads in terms of area, power, and manufacture costs.

Chapter 1

1. Overview

The ever increasing requirements of embedded applications push designers to realize electronics systems matching, on one hand performance and energy efficiency, on the other hand fast development time and cost, as well as flexibility and re-usability of the realized platforms. This section, starting from an overview of the solutions proposed over the last few years both in terms of architecture/devices and design/methodology, present the approaches described in this thesis, analyzing the motivations on introducing multi-core platform with configurable hardware acceleration.

1.1 State Of the Art

1.1.1 Application Specific Signal Processors

The term Application Specific Signal Processor (ASSP) implies some kind of hardware specialization of a general-purpose processor that is enabled in this way to match the performance (and energy) requirement of an application, or more in general, of a class of applications sharing similar features. ASSPs have demonstrated during last few years as the most effective way to match the embedded application constraints while guaranteeing to the final customer the user-friendliness typical of general purpose processors due to software abstraction layers that abstracting the utilization of the hardware accelerators. For several applications, especially in the wireless baseband processing, very long instruction word (VLIW) processors were developed to provide high levels of parallelism along with programmability. Other approaches lead to the development of application specific ICs to gain performance during execution of most critical kernels. In these cases, the architecture of such

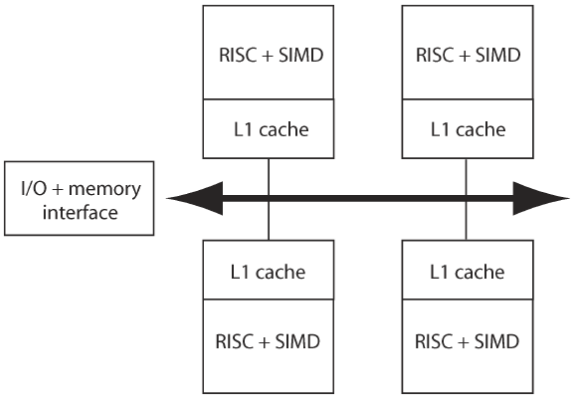


Figure 1.1: *Lucent Daytona Architecture.*

systems often correspond to the blocks diagram of the application for which they were designed, leading to a heterogeneous structure based on multiple processing cores. From the programming point of view these devices achieve their goal levering to the general-purposeness of the standard processor that manage the system, handling control and synchronization of applications with the support of real-time operating systems (RTOS) to ease programmability. On the other hand, the final user is not required to handle execution of the application specific computation intensive kernels of the applications, as they are developed by the hardware providers and encapsulated into pre-packaged software libraries.

One of the firsts MPSoCs with application specific hardware accelerators is the Lucent Daytona [1], shown in Figure 1.1. The main purpose of the Daytona processor is the elaboration of signal processing algorithms typical of wireless base stations, where the identical program-flow is executed for many data channels. Following the specific target of the wireless application, Daytona was realized as symmetric multi-processor architecture with local caches, connected to the external memory interface trough a high-speed bus. The processor architecture is based on the SPARC V8 core, enhanced with application specific functional units to improve efficiency on wireless communication algorithms, such as 16x32 multiplications, division step, and vector coprocessor.

Remaining in the wireless application field, the C5 processor [2] is an embedded processor for packet processing in networks. The C5 architecture

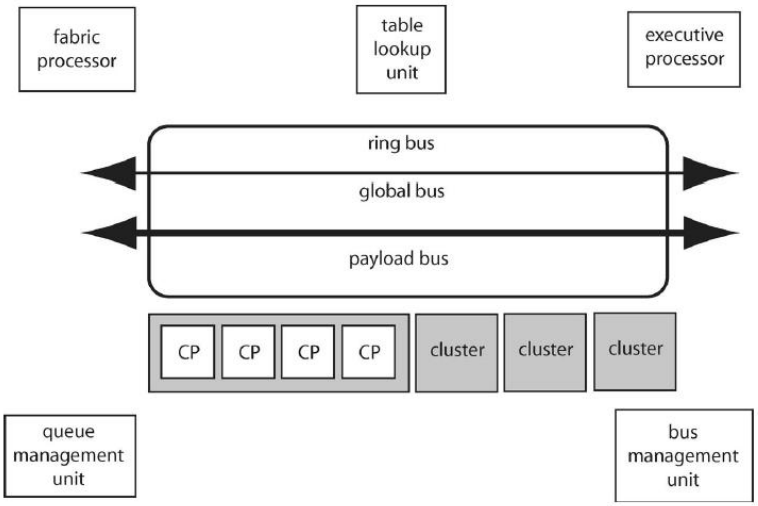


Figure 1.2: *C5 processor Architecture.*

encapsulates a reduced instruction set computer (RISC) managing the system, and several other specialized units connected through a three-layer bus. Packets are handled by 16 channel processors grouped as 4 clusters of 4 processors each. The C5 processor architecture is shown in Figure 1.2.

The processors presented up to now feature homogeneous architectures with dedicated vector units, matching parallelism of wireless application for which they were designed. Contrarily, most recent ASSPs, especially dedicated to multimedia or mobile applications, usually feature similar structures that we can describe as hierarchical heterogeneous MPSoC. One standard processor, drives a multi-layer bus hierarchy comprising IO peripherals, on-chip memory, programmable DSP engines, and a set of specific ASIC accelerators for the computation of the most intensive kernels. The more restrictive are the energy and performance requirements of applications, the more specific are the accelerators.

A further example in the field of multimedia is represented by the Philips Viper Nexperia [3], shown in Figure 1.3. The Viper processor includes two CPUs: a MIPS and a Trimedia. The MIPS acts as manager hosting an operating system, while the Trimedia acts as a signal processing co-processor. The communication is handled by a multi-layer bus, which connect the two processors to the external memory controller and several other ASICs that

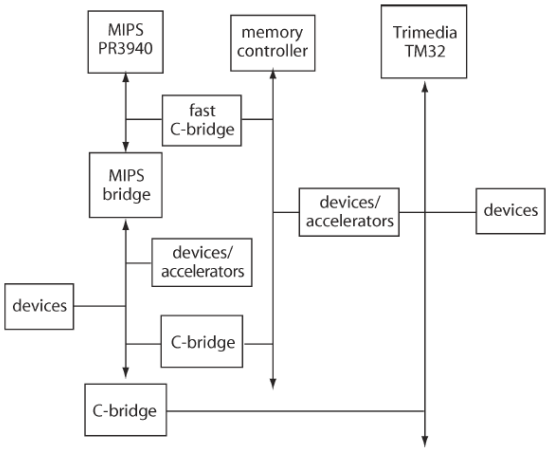


Figure 1.3: *Viper Nexperia processor architecture.*

perform computations such as color space conversion and scaling. The Viper processor allows different mappings of physical memory to address space in order to better match the requirements of the different portions of application executed.

Moving to the mobile area, representative examples in the field of cell phone processing are those of Texas Instruments OMAP [4] and STMicroelectronics Nomadik [5]. The OMAP processor has several implementations. The OMAP 5912 (Figure 1.5) has two CPUs, an ARM9 and a TMS320C55x, where the ARM acts as master processor, while the DSP acts as a coprocessor for execution of several signal processing applications. On ST Nomadik (Figure

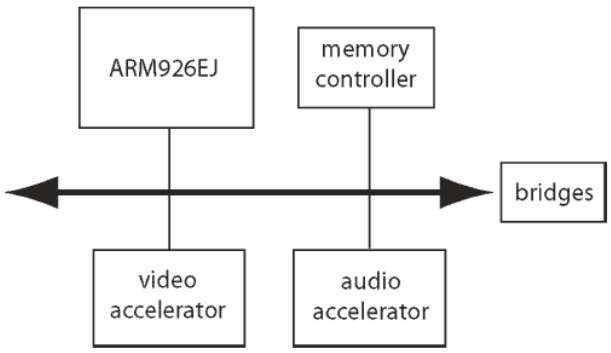


Figure 1.4: *ST Nomadik processor architecture.*

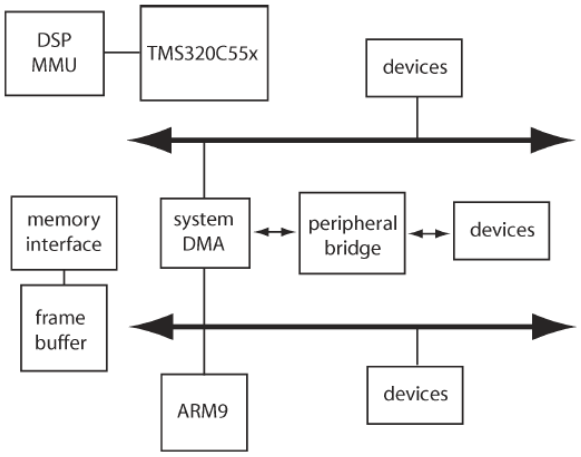


Figure 1.5: *TI OMAP processor architecture.*

1.4), the master processor hosting the operating system is an ARM9. On the contrary the audio and video acceleration units are applications specific accelerators based on the MMDSP+ DSP core. The video accelerator is a heterogeneous multi-core, including the MMDSP+ and application specific accelerators for several important stages of video processing, while the audio processor only leverages on the DSP due to lower computational requirements of audio applications.

1.2 Reconfigurable Devices

Reconfigurable computing is intended to fill the gap between hardware and software by achieving better performance than software, and maintaining a higher level of flexibility thanks to the programmability of its computational elements. Reconfigurable devices, including field-programmable gate arrays (FPGAs), are usually composed of an array of computational elements whose functionality is determined through a set of configuration bits stored in dedicated SRAM distributed among the device. These logic elements are connected together through a set of programmable routing resources. In this way, arbitrary digital circuits can be implemented on the reconfigurable hardware by mapping the logic functions, and using the configurable routing to connect the blocks together to form the required circuit. From the commercial point of view, the most common class of reconfigurable devices is that of FPGA. The two major enterprises producing FPGAs are Altera [7] and Xilinx [8]. The success of FPGA devices is mainly related to their flexibility and ability of upgrading their application portfolio after the fabrication.

The first example of reconfigurable system dates back to 1986. The Programmable Active Memory (PAM) system [9] was composed of a host processor connected to a Xilinx XC3090 device through two unidirectional links Figure 1.6. The main competences of the host processor within the system were the uploading of the configuration bitstream of the FPGA and the execution of non-critical portions of software applications. The reconfigurable

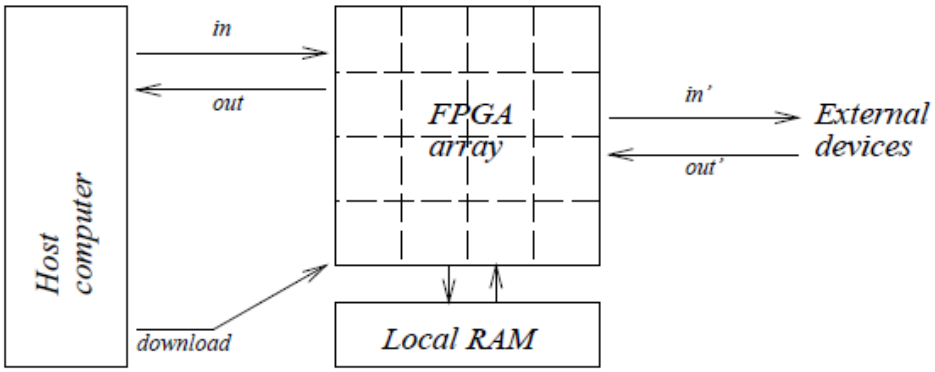


Figure 1.6: *Programmable Active Memory (PAM) system.*

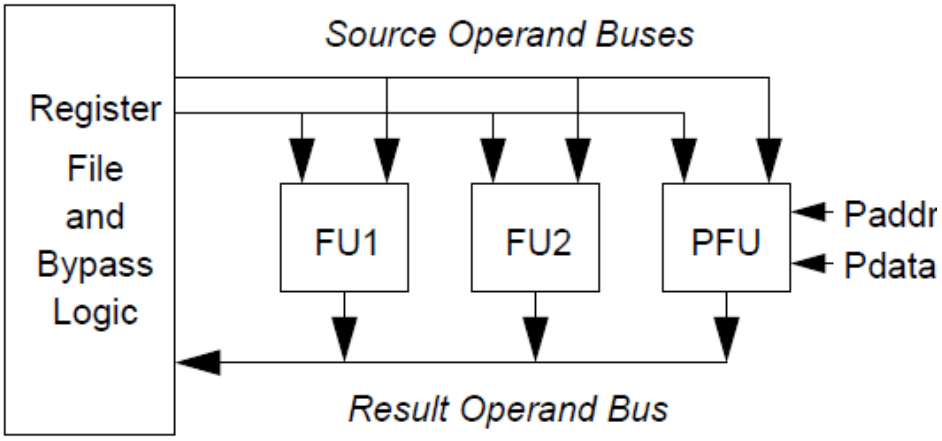


Figure 1.7: *Architecture of PRogrammable Instruction Set Computers (PRISC).*

devices could act as both a stand-alone component or as a system coprocessor, communicating with the host, a local memory or external devices through dedicated data channels. The system was demonstrated to be able to achieve 10 to 1000 speedups on over 10 applications, with respect to the equivalent software implementations.

Razdan and Smith presented a more processor-centric utilization of reconfigurable hardware with the PRISC architecture in 1994 [10]. The PRISC approach formalized the concept of *instruction set metamorphosis* or *adaptive instruction set*. This computational paradigm exploits the reconfigurable device as an application specific hardware-programmable functional unit (PFU) rather than a coprocessor, interfaced to the register file of a RISC processor, as shown in Figure 1.7. As the integration of an external functional unit has a direct impact on the processor micro-architecture, a dedicated compilation flow was realized to preserve the coherency of the executed applications. The PRISC compilation flow assisted the user in the extraction and synthesis of *Execute PFU* instructions (i.e., instruction executed on the PFU) generating both the hardware and software images from the high level application source code and profiling information.

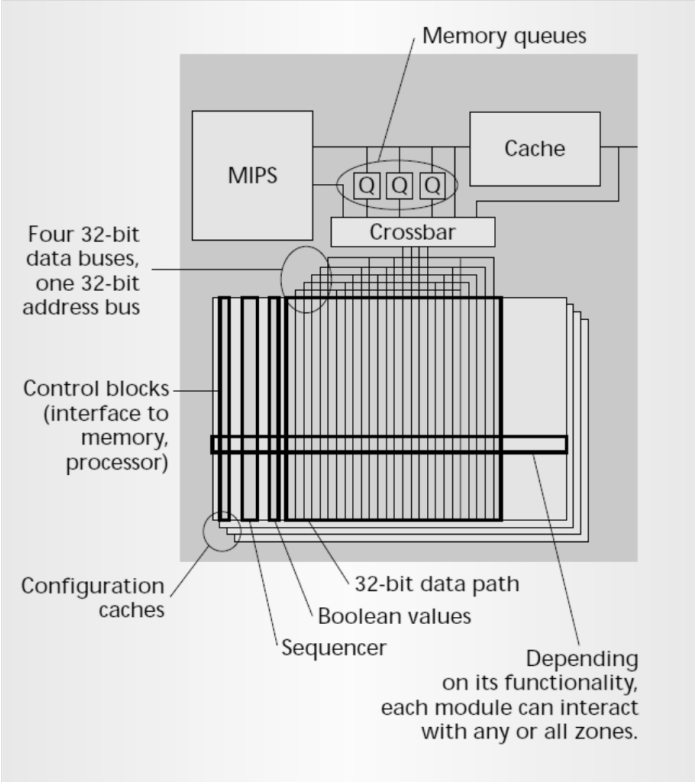


Figure 1.8: *Architecture of the Garp reconfigurable processor.*

One of the most important milestones of reconfigurable computing is the GARP processor, developed at the University of California, Berkeley [11]. GARP couples a MIPS processor with a reconfigurable device organized as a datapath as shown in Figure 1.8. Due to the datapath structure, differently from the previously described architectures based on standard FPGAs, the speed of the clock remains constant for an implementation and doesn't require to be adjusted by an array configuration. In addition, the GARP architecture introduces a caching mechanism in order to speed-up the programming of the reconfigurable data-path, being able to update the array configuration in five clock cycles. The main peculiarity of the GARP approach concerns the applications compilation and synthesis flow. Data flow graphs (DFGs) are automatically extracted from the inner loops of applications; utilizing predication in order to eliminate the need for conditional branches. This way

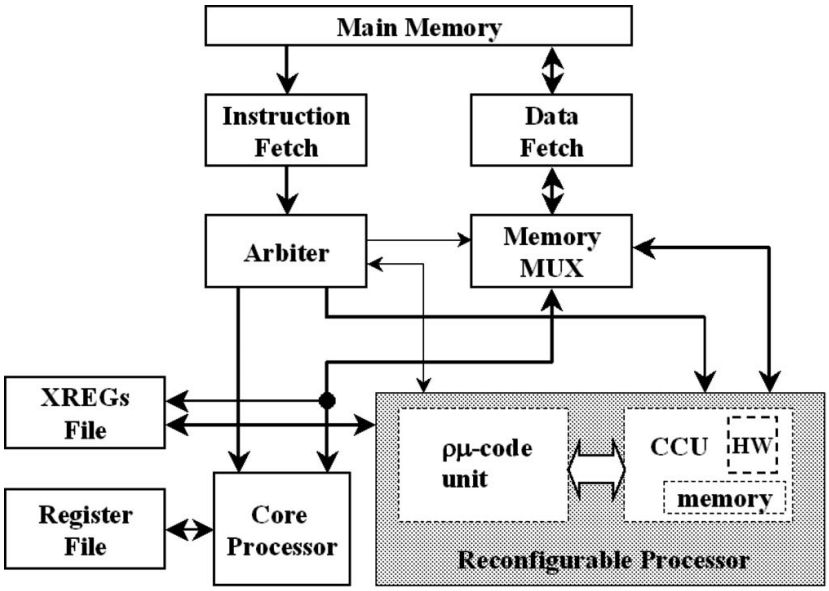


Figure 1.9: *The Molen polymorphic processor.*

it allows to find the optimal granularity of the kernels mapped on the datapath.

The MOLEN polymorphic processor [12] couples a general-purpose processor with a reconfigurable co-processor enhanced with hardware facilities for synchronization and arbitration as shown in Figure 1.9. The main peculiarity of Molen concerns the formalization of the programming model utilized for the implementation of applications on the system, known as the *Molen Paradigm* [13]. The Molen programming paradigm targets parallel and concurrent hardware execution of single threaded applications. It defines a set of instructions (polymorphic instruction set architecture) that focus on the consistency between functions executed on software and functions executed on the reconfigurable device. The interesting feature of this programming model is that it can be extended to reconfigurable processors whose reconfigurable engine is seen as a co-processor of the general purpose core. The Molen processor was implemented onto a Xilinx FPGA chip, utilizing the PowerPC embedded in the FPGA as General-Purpose Processor (GPP).

In some cases embedded FPGAs can be utilized as on-chip reconfigurable engines. This kind of devices, differently from those described above are

realized with general-purpose CMOS processes, so that they can be integrated as IPs within a more complex System On A Chip. The main target of eFPGAs within more complex systems is the implementation of all those applications which can benefit from bit-level synthesis optimization, usually unsuitable for GPPs. In addition they can possibly be utilized to implement configurable IO peripherals. One example of this category of devices is the Flexeos core developed by Abound Logic [15].

All the reconfigurable architecture presented up to now, feature a general-purpose processor coupled with a fine grain reconfigurable device. Although this kind of devices are characterized by a very good flexibility as they are theoretically capable to implement any kind of logic function, FPGAs early appeared as too big, slow, and power hungry if compared to most of portable application requirements and ASIC-based solutions. The full flexibility offered by the bit-level programmability introduces too much overhead, especially due to the SRAMs utilized to store the configuration bitstream, and redundant interconnect. For many application domains it is possible to trade part of the flexibility offered by fine grained architectures by increasing the granularity of the basic processing elements (PEs) to 4-, 8-, 16- or 32-bit while reducing the overall number of basic elements, thus reducing the impact of interconnect over the overall chip areas. This approach is intended to provide the double advantage of reducing the overhead of both routing and configuration storage, and achieve higher operating frequencies due to the hardwired implementation of standard computational blocks such as adders or multipliers. This class of devices is known as *Coarse Grain Reconfigurable Architecture* (CGRA). Many CGRAs have been proposed from both academia and industry in order to increase the ratio between the granularity of the basic element and the programmable interconnects. In such devices, the computational capability of the basic logic cell raises from the LUT complexity to complete arithmetic logic units (ALUs), while the flexibility of the interconnect drops, for example supporting only the connection of nearest rows or among nearest-neighbors.

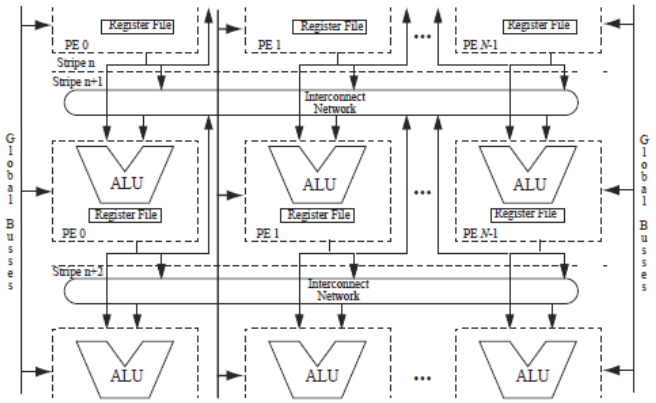


Figure 1.10: *PiPeRench processing elements and interconnect.*

PiPeRench [16], is one of the first and most important CGRAs that appears in literature. The device, introduced as accelerator for multimedia applications, provides reconfigurable pipeline stages named *stripes*. PipeRench consist of 28 horizontal stripes of 32 processing elements composed of register and 4-bit ALUs, implemented as 3-bit LUTs. Each stripe provides facilities for partial dynamic pipeline reconfiguration and automatic scheduling of configuration and data streams. On the other hand, a hierarchical interconnect infrastructure enables communication among processing elements within a stripe (horizontal interconnect) and communication among stripes (vertical interconnect), as shown in Figure 1.10.

MorphoSys [17] is composed of a MIPS-like "TinyRISC" processor with extended configurable instruction set. From the architectural point of view the reconfigurable device is a mesh connected 8x8 reconfigurable array, featuring a frame buffer for intermediate data storage, a context memory for enhanced re-configuration, and a DMA controller (Figure 2.11). The reconfigurable array is divided into four quadrants, each one being composed of 4 by 4 16-bit reconfigurable cells (RCs) each. Each RC features an ALU, a multiplier, a shifter, a register file, and a 32-bit context configuration register. The interconnect network hierarchy is formed of 3 layers: four nearest-neighbor ports, interleaved links, and inter-quadrant buses spanning the whole array.

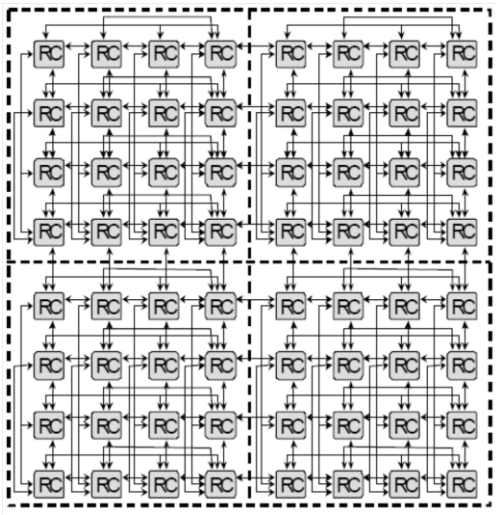


Figure 1.12: *Morphosys array architecture.*

The CHESS [18] array features a chessboard-like structure where rows of ALU and switchbox are alternated as shown in Figure 1.12. Memory requirements of applications are supported by the Embedded RAM areas of the array. In fact, switchboxes can be converted to 16 words by 4 bit RAMs if needed or to a 4-input, 4-output LUT. The interconnect fabrics of CHESS is composed of 4-bit buses of different length. There are 16 buses in each row

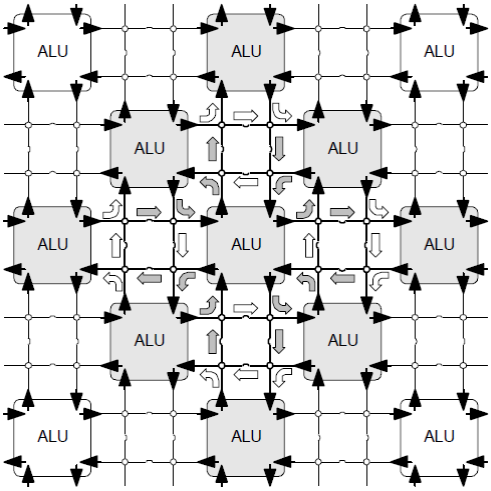


Figure 1.11: *Chess interleaved interconnection scheme*.

and column, with interleaved interconnections of length 1, 2, 4, 8 16. In order to avoid routing congestion, the array also features embedded 256 bytes-size SRAM blocks. The output data of an ALU can feed the configuration input of another ALU. This way it is possible to change its functionality at run-time without uploading the configuration.

The DREAM [19] reconfigurable processor is a mid-grain computation intensive reconfigurable processor mainly targeting signal processing applications featuring iterative computations and irregular data width (Figure 1.13). A RISC processor manages execution of accelerated kernels and reconfiguration. The computational core of the device is the PiCoGA-III [20] (Pipelined Configurable Gate Array) reconfigurable datapath, featuring a matrix of reconfigurable logic cells with 4-bit functionalities and support for multi-context. The local storage consists of a multi-bank memory coupled with the datapath, which provide high bandwidth toward the PiCoGA-III inputs and outputs.

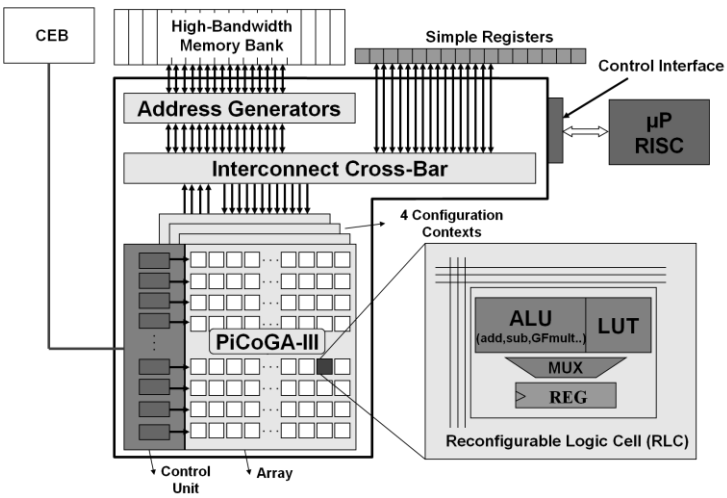


Figure 1.13: *Architecture of the DREAM reconfigurable processor.*

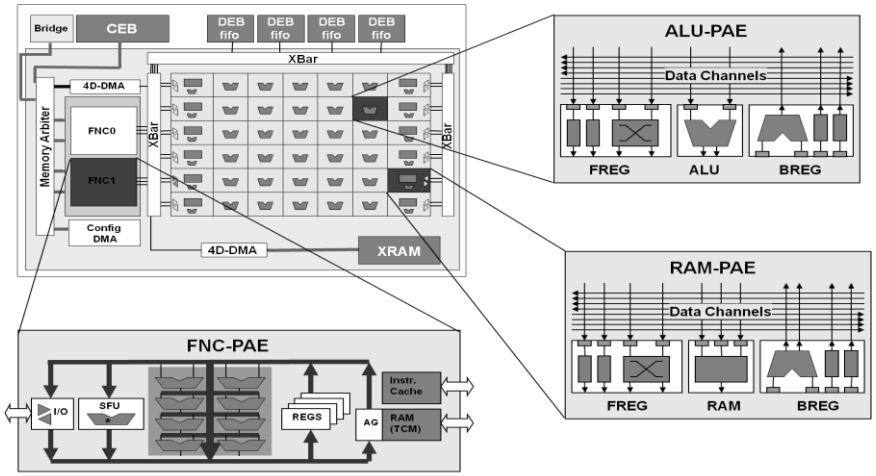


Figure 1.14: *XPP-III reconfigurable array architecture.*

XPP-III [21] is a coarse-grain configurable processor, mainly targeting streaming applications with regular data width and significant computational densities. As shown in Figure 1.14 XPP-III is composed of an array of 16-bit Processing Array Elements (PAEs) and two general-purpose processors (FNC-PAEs) suitable for execution of control-oriented portions of applications. The array features a set of processing (ALU-PAEs) and IO/data storage elements (RAM-PAEs) communicating through a matrix of configurable data channels. Communication with the external world is supported by asynchronous FIFOs, according with its streaming computational models.

Another device proposed in the field of reconfigurable computing is that of BUTTER, developed at Tampere University of Technology [22]. The BUTTER reconfigurable array, mainly targeting FPGA implementations, maintain a structure similar to XPP, provides additional features such sub-word, and floating point capabilities with the ambition of widening its application spectrum. A recent evolution of such architecture is CREMA [23], a coarse grain reconfigurable array with mapping adaptiveness, which allow the designer specify the application characteristics and generate a coarse-grain reconfigurable array optimized for those requirements.

1.2.1 Multi/Many Core Systems

A new class of devices which is emerging in last few years is that of homogeneous multi-many core systems. These devices, rather than exploiting instruction level parallelism or data- level parallelism typical of the previously described approaches leverage to thread level parallelism in order to obtain high performance and high programming legacy typical of software-programmable platforms. The main advantages of this approach with respect to the presented devices are flexibility and programmability. In fact, processor based systems are intrinsically more flexible than ASSPs and easier to program than reconfigurable processors due to high level programming languages (C, C++) and well known programming models (MPI, OpenMP). These devices are usually composed of several general purpose processors (or functional units) arranged as an array or as hierarchical clusters of processors. Communication and memory architecture is also one of more differentiating points among proposed approaches, usually strictly connected with their programming paradigm. Message passing programming models, such as MPI [24], match distributed memory architectures, where connections among processors are usually implemented by a mesh-topology network-on-chip. Within this computational paradigm each processor executes its own task with data and code separate to each other, while synchronization and data communication among cores is achieved by sending messages by addressing a specific core within the system. On the other hand, shared memory programming models, such as OpenMP [25] usually match architectures composed of processor clusters. This computational model leads to exploit parallelism in a homogeneous way, where each task executes the same instructions on a different data-set. More recently, appeared programming models that allow handling mapping of ultra-highly parallel applications on hierarchical architectures of processor clusters. For example the CUDA (Compute Unified Device Architecture) [26] environment was developed by NVIDIA for efficient programming of General-Purpose Graphic Processing Unites (GP-GPU). A standardized evolution of CUDA exploited during last few years is OpenCL [27], which added support for programming of heterogeneous platforms composed of both ultra highly parallel devices, such

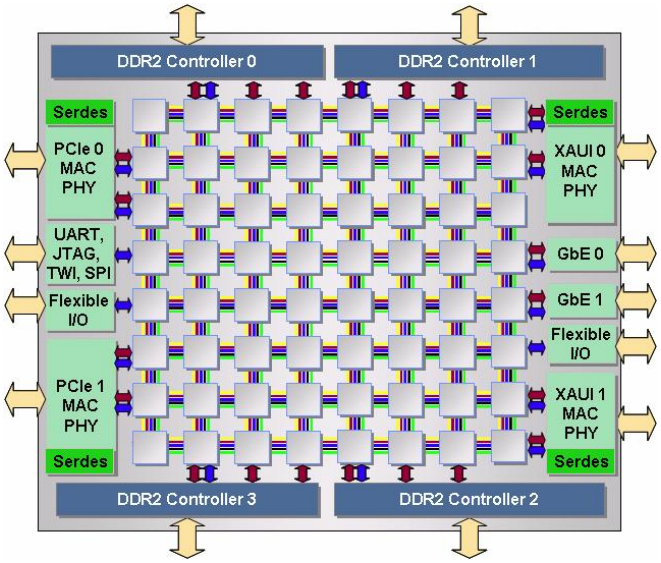


Figure 1.15: *Tile64 architecture.*

as GPUs and other compute devices by supporting both homogeneous data-level parallelism and heterogeneous task-level parallelism.

TILE64 [28] is an array of processors developed by Tilera for advanced networking applications and digital video processing, as well as general-purpose applications. The architecture is based on the RAW processor develop by Massachusetts institute of technology (MIT). Each processor can be programmed utilizing high level languages such as C, or C++ and support execution of operating system. As shown in Figure 1.15, its silicon structure is composed of 64 identical programmable tiles, regularly replicated over the die surface. Each tile includes an 8-pipeline stage MIPS-like processor, tightly coupled with a 4-pipeline stages floating point unit, and a 32-Kbyte data cache and 96 Kbytes of software-managed instruction cache, while communication is achieved through a mesh topology network-on-chip implemented by four routers available within each tile. Two routers are static (routes specified at compile time) and two are dynamic (routes specified at runtime). Each tile only connects to its four neighbors, while communication wires are registered at the input of each tile. This means that the length of the longest wire in the system is no greater than the length or width of a tile, thus ensuring high clock rates, and the continued scalability of the architecture.

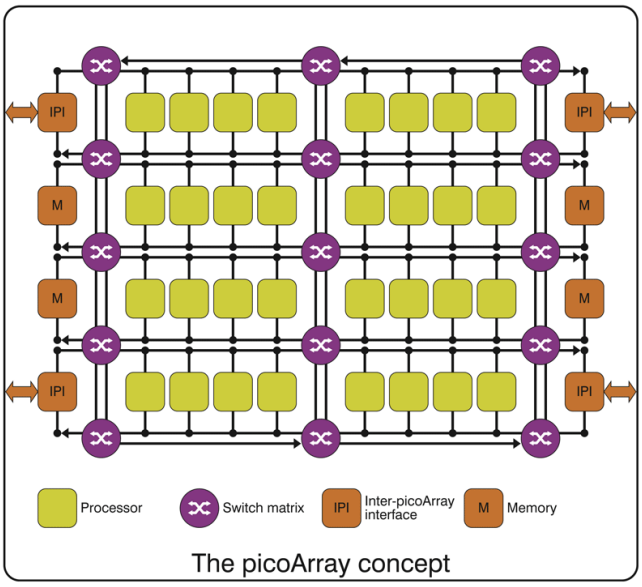


Figure 1.16: *PicoArray.*

The picoArray [29] is a multi-core digital signal processor, integrating hundreds of individual DSP cores within a single die. A picoArray device is composed of 308 processing elements linked together by the picoBus interconnect, as shown in Figure 1.16. The basic cores of the array are three-way Very Long Instruction Word (VLIW) RISC 16-bit processors, each one coupled with local memory. The picoArray core is coupled with a series of coprocessors, such as external interfaces toward eternal devices, and memory interfaces, which can be either asynchronous or synchronous. Each processor is coded independently either in C or assembly languages and can communicate over an any-to-any interconnect mesh. The processor array is integrated with a set of 14 application-specific co-processors called function accelerator units, for a total of 322 processors. The communication infrastructure is composed of a square mesh of 32-bit communications links, which incorporates switch matrix elements at the junctions between its horizontal and vertical lines. The configuration of routing path among processors is computed at compilation time, thus allowing a good predictability of the performance, that making the platform suitable for execution of real-time applications.

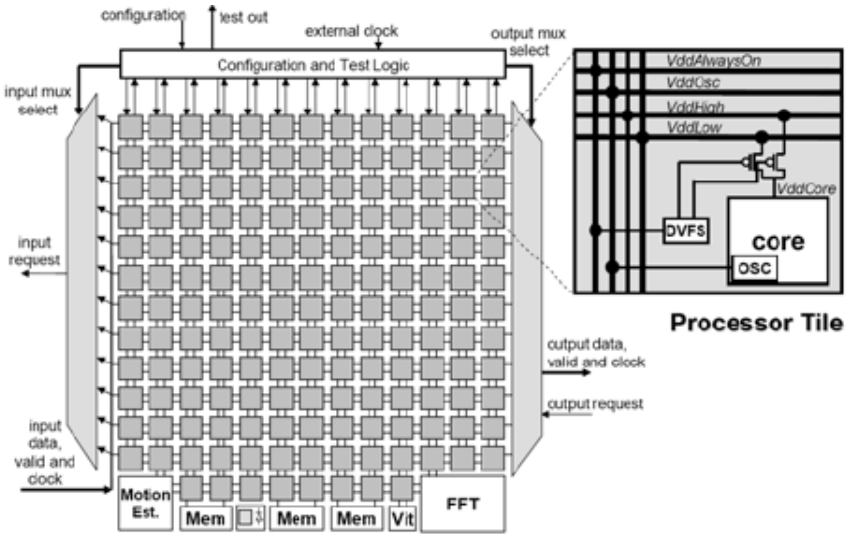


Figure 1.17: *ASAP processor architecture and tile structure.*

The ASAP [30] multi-core system is a computational platform composed of an array of 164 16-bit RISC processors supporting dynamic voltage and frequency scaling, plus three application specific units and three 16-Kb shared memory banks. The ASAP processor is suited for execution of DSP processing as well as wireless and multimedia, and, more in general for all those applications whose block diagrams can be efficiently mapped onto an chain of basic computation blocks. Each tile of the array includes an in-order, single-issue, six-stage RISC processor programmable in both C and assembly executing over 60 basic instructions. In addition, in order to enable dynamic voltage and frequency scaling, the tile includes a local oscillator and three local power domains, allowing the processor to switch to each other depending on the required operating frequency. This technique allows to trade the power consumption of tile of the array with the computational requirements of the related application task, thus achieving high energy efficiency rates. The communication scheme is implemented according to the nearest neighbor policy. Connections are circuit-switched and statically configured, and can be pipelined at each tile to achieve full-rate communication over long distances, or un-pipelined if the distance is short or the source clock's frequency is low.

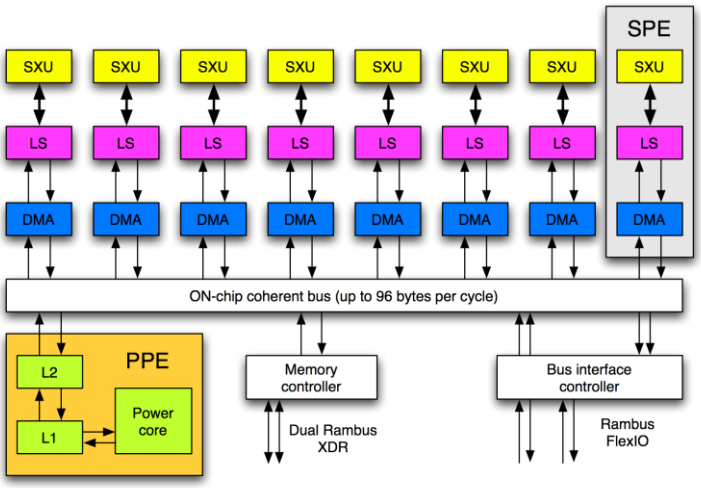


Figure 1.18: *The Cell architecture.*

A new computational paradigm coming from the field of graphical processing or desktop computer is that of the Cell processor and GP-GPUs.

As shown in Figure 1.18, the Cell processor [31] is composed of a power processing element (PPE) supporting universal virtual memory and concurrent double threads which host an operating system, and a set of eight processing elements known as synergistic processing elements (SPE). SPEs are SIMD processing cores aimed at high throughput data processing. They feature a RISC command structure, 128 general-purpose registers of 128 bit, and 256K bytes local storage. The PPE, SPEs, and I/O interfaces are connected by the element interconnect bus, which is built from four 16-B-wide rings. Two rings run clockwise, and the other two run counterclockwise. Each ring can handle up to three non-overlapping data transfers at a time which leads to a 25.6 Gbit/s transmission capacity. The peak computing rate reaches 204.8 GFLOPS.

On the other hand, GP-GPUs [32] provide tremendous computing power of up to 1,3 TFLOPS. As shown in Figure 1.19, GP-GPUs are characterized by a hierarchical architecture composed of an array of Streaming Multiprocessors (SMs) each one featuring up to 32 Streaming Processors, a shared memory, register file and schedulers for handling automatic synchronization of data-parallel threads. A key factor that gained the evolution of such kind of

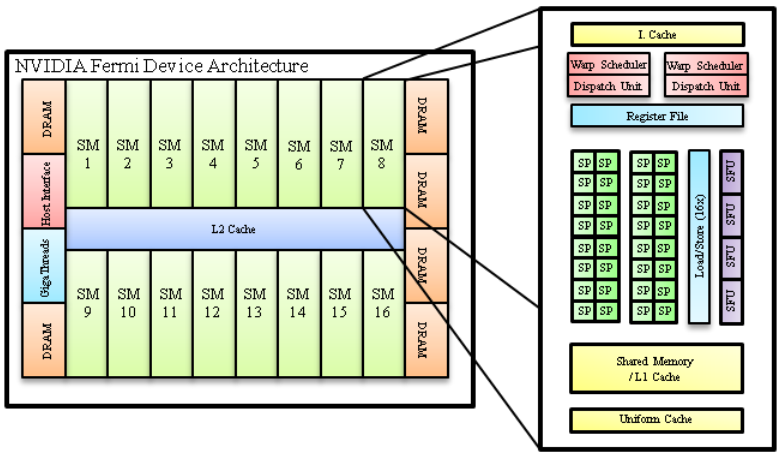


Figure 1.19: *NVIDIA Fermi device architecture.*

architectures is given by the relatively easy to use programming models such as CUDA that allows to exploit data level parallelism, partitioning the applications in thousands of data-parallel threads.

1.3 Design and Specialization of Multi-Processor Systems-On-Chip

Besides the architecture of the proposed computing systems for embedded applications, recent years have seen the growth of strategies at different levels of implementation in order to reduce design effort and related non-recurring engineering costs. These design methodology, which specifically targets design of multi-processor systems on chip, can be applied either at system-level, by directly mapping the platform described with a high level specification, or utilizing a hierarchical approach based on platform based design paradigm. In this last scenario, a common multi-core platform specifies the architectural template at the basis of the system, while the customization of the platform is achieved by tuning the platform parameters according to high-level specifications. On the other hand, the application-specific customization of a platform can be achieved by specializing the processor cores utilizing synthesis of accelerators from high-level languages.

Moreover, the general concept of processor extensions can be directly applied to the programming of the presented reconfigurable processors, where such instruction extensions are mapped on the reconfigurable engines instead of silicon-based structures.

1.4 System-level design of Multi-Processor Systems-On-Chip

The utilization of a Register Transfer Level (RTL) description language as a starting point for complex System-on-Chip design methodologies form a bottleneck. Such methodologies were effective in the past, when systems were based on one single processor or on one processor plus a set of coprocessors. On the other hand, the applications and platforms used in many of today's system designs are based on heterogeneous Multi-Processor System-On-Chip (MPSoCs). Although the RTL system specification has the advantage that the state-of-the-art synthesis tools can use it as an input for its automatic implementation, it is a common thinking that a system should be specified at a higher level of abstraction due to the complexity of today's systems [33]. However, increasing the abstraction level of the system description opens a gap between the specification and the related hardware implementation. Indeed, the RTL system specification is very detailed and close to an implementation, which allows an automated synthesis path from the RTL to the physical implementation. In order to address this issue, during last few years several architectural synthesis flows have been proposed, aimed at the automatic generation of the RTL description of the system starting from high level specifications.

The Compaan design flow [34] uses Kahn Process Networks KPNs as an application model for the automated mapping of applications targeting the FPGA implementations. A KPN specification is automatically derived from a sequential program written in Matlab [35][36] and implemented as a network of dedicated hardware cores on an FPGA [37]. Eclipse [38] defines a scalable architecture template for the design of stream-oriented MPSoCs using KPN model of computation to specify and map data-dependent applications. Jerraya et al. propose a design flow that utilizes a high-level parallel

programming model to abstract hardware/software interfaces in the case of heterogeneous MPSoC design [39][40]. Companies such as Xilinx and Altera provide approaches and design tools that attempt to facilitate the efficient implementations of processor-based systems on FPGAs. These tools are the Embedded Development Kit [41] for Xilinx chips and the System On a Programmable Chip (SoPC) builder [42] for Altera devices. More recently, synthesis flows have been proposed in order to implement applications described utilizing ultra-parallel programming languages typical of GPUs such as CUDA and OpenCL onto FPGA devices [43][44]. These flows take advantage of the common models of computations utilized by these programming languages to ease the parallel mapping of applications on FPGAs.

Although the automatic synthesis of architectures is an attractive way for reducing design costs of complex systems-on-chip, most of these techniques only target FPGA prototyping. Moreover, the automatic architectural optimization of these platforms often targets the implementation of specific applications, still being implemented in most cases with general-purpose components. The implementation of such automatically generated architectures is neither general-purpose nor application-specific. For this reason, neither performance nor market volumes expected by such platforms justify deployment of silicon products based on such design flows.

A step toward a more hardware-centric design methodology, which still allows abstracting the designer for a pure RTL description of the architecture, is that of platform-/component-based design [45][46]. The platform based design paradigm is an attempt of simplifying the system-level design problem by removing one degree of freedom. In platform-based design, the allocation the target system platform consisting of computation and communication components is assumed to be fixed, or at least significantly constrained. Thus, the constraints at the input of the design process consist of a fixed template with a given number of parameters. Such a predefined and predetermined implementation scheme eases the reuse of common design patterns, across the different design instances. Moreover, such an approach allows IP-designers to focus the effort of few configurable blocks, whose RTL implementation can be optimized regarding the physical implementation, exploiting in this way the IP reuse as much as possible. Platform-based design divides system design

into two phases. First, a platform is designed for a class of applications. Then, the platform is adapted for the particular product in that application space. MPSoCs are ideally suited to be used as platforms. CPUs can be used as a way to customize systems in a variety of ways. The platform-based design tends to be software driven, as much of the product customization currently comes from software. Once again, this has the advantage of widening the application domain of a platform, but still, this is often not sufficient to match the strict performance and energy requirements of modern applications. One common way to improve performance of a general-purpose system, is that of configuring and extending processors in order to specialize their functionality for a specific application domain.

1.4.1 Configurable Processor and Instruction Set Synthesis

Instruction sets that are designed for specific applications or domain are commonly used in many embedded systems [47]. As described in the previous sections, the design of customized processors usually requires a relevant amount of work but can result in huge power and area savings. The customization of a processor refers to the tools that generate a RTL description of the processor based on a set of requirements given by the user. Configurable processors are divided into two categories. Those that are based on a pre-existing architecture are enhanced with extensions driven by specifications based on parameter selection and structural choices provided by a processor configuration tool. In other cases, configurable processors create a new instruction set architecture as specified by the user through a more formalized architectural definition language. The configuration of a processor can be of two types:

- Structural configuration of the processor. This implies the presence or absence of a set of interfaces or components associated to the processor. These might include system bus interfaces, local memory interfaces, external memory interfaces or external coprocessor interfaces. The width of the interface and the communication protocols may also be configurable or selectable. Other parametric structural choices may imply the inclusion of special functional units such as

multipliers, dividers, multiply and accumulate (MAC) units, floating point units and shifters. Additional structural parameters may include the presence of on-chip debug, trace JTAG, the register file size, timers, exception vectors, and multi-context register file.

- Extension of the processor instruction set. This implies the integration of the processor ISA with extra instructions, which are mapped directly into the datapath of the processor. The instructions are usually decoded by the processor in the standard way and may even be automatically recognized by the compiler or manually invoked within the processor code. The instruction extensions are usually included in some kind of architectural description language or may be defined by a combination of HDL code and templates for instruction formats, encoding, and semantics.

The architectural optimization of a processor is done by designing or refining the microarchitectural features from high level specifications such as performance or power. This is often performed in conjunction with configurable extensible processors or coprocessors. The optimization flow can either be automated or not, but it is always supported by tools working at various levels of abstraction and sophistication.
The MIMOLA system [48] is one of the first appeared CPU design tool that perform both the architectural optimization (i.e., automatic selection of architectural parameters) and configuration. ASIP Meister [49] is a configuration system that generates processors featuring Harvard architecture. The Synopsys Processor Designer [50] uses the LISA language to describe processors starting from a combination of structural and behavioral features of the desired architecture. From the same description of the architecture, a set of tools associated with the environment enable the generation of both synthesizable RTL code and a compiler for the generated processor. The Tensilica Xtensa processor [50] is a commercial configurable processor that allows the users to configure a wide range of processor parameters, such as the instruction set, feature of the caches, and presence of I/O interfaces. The Toshiba MeP core is a configurable processor optimized for media processing and streaming.
The synthesis of instruction set is a form of architectural optimization that concentrates on instructions. Several commercial approaches that generate application specific instruction set processors or coprocessors from scratch exist. These start with either application source code, such as Synfora PICO [51], based on research from HP or compiled binary code Critical Blue

Cascade [52] and generate a custom highly application-tuned coprocessor. Other commercially affirmed high-level synthesis approaches are the Catapult C [53], which provides synthesis of accelerators starting from a C-level description of an algorithm and permits selection of many synthesis parameters such as pipelining and unfolding. PowerOpt [54] is a high-level synthesis flow that permits the generation of power-optimized hardware accelerators starting from high-level languages such as C, C++ or SystemC.
Some of these tools integrate both the processor configuration and the synthesis of the instruction set extension. The XPRES [55] tool from Tensilica [35] combines the notations of configurable processor, instruction set extensions, and automated synthesis. The processor synthesis flow starts from the user application code and ends up with a configured instruction-extended processor tuned to the particular application. XPRES utilizes optimization and design space exploration techniques that allow the user to select the proper combination of performance improvement, area increase, and energy reduction in order to meet the applications constraints. The STxP70 processor from STMicroelectronics is a configurable and extensible processor for embedded applications that allows the user to handle processor configuration, instruction set extension and automated synthesis of extra instructions into a unified environment. Selection of architectural and micro-architectural parameters and the related generation of RTL are achieved through the graphical user interface. Moreover, the processor is integrated with a set of configurable peripherals and interconnect (i.e., DMAs, Bus) that allow its integration on a complete, configurable sub-system.

1.4.2 Synthesis of instruction set on reconfigurable processors

The general concept of synthesis of instructions set can be naturally applied to reconfigurable processor, leading to the described paradigm of the instruction set metamorphosis. Differently from the synthesis of instruction set extension applied to processors, the implementation of reconfigurable processors instructions is constrained by the specific architecture of the target reconfigurable engine, mainly consisting of the granularity of processing elements, flexibility of the interconnect, and the utilization of either a tightly coupled functional unit approach or a co-processor approach. Milestones of the research on the field of reconfigurable processors, like the GARP processor, and other commercial state-of-the-art reconfigurable processors proposed C-based design environments envisioning the possibility to offer the

end-user the capability of automatic partitioning, and then to co-compile the same source code over both the processor core and the reconfigurable logic. The Nimble compiler [56], targeting the Garp processor, is one of the first tools that tried to automatically move critical kernels from the processor core to the reconfigurable hardware accelerator, selected from the basic blocks of the compiled applications inner loops. Another example is that of PipeRench. It is configured utilizing a single-assignment language with C operators (called DIL, Dataflow Intermediate Language) that is a C-based proprietary language.
Moving the focus on coarse grain reconfigurable processors, direct mapping is probably the most used method, where operators are mapped to the programmable elements that compound the device without a real logic synthesis step. PACT XPP and MorphoSys are effective examples of such an approach. Although they provide a tentative virtualization of the mapping layer using C-based high-level compiler flows [57][58], for the full exploitation of the architecture capabilities assembly-like languages are needed for both of those. PACT XPP is programmed through the Native Machine Language (NML), a structural event-based netlist description language. The MorphoSys architecture is provided with a SUIF-based compiler for the host processor, while the partitioning between hardware and software is performed manually by the programmer. The MorphoASM, a structural assembly-like language, is used to configure each programmable element according to the required functionality. The CREMA architecture is equipped with the Firetool (FIeld programming and REconfiguration management Tool). With Firetool the designer can specify a set of reconfiguration patterns used in the application. The tool generates a VHDL package based on a fixed template, where all the parameters are set accordingly to the specifications, and a set of C header files to manage via software the runtime reconfiguration.

1.5 Bridging the gap between MPSoC design and configurable hardware specialization

The NRE costs associated with the design of complex systems are growing rapidly. More precisely, the design and verification of complex Systems On Chip, and the production of masks and exposure systems are major bottlenecks for the development of such chips. The main goal of the

electronics embedded systems is that of balance the development time and cost, and the production cost with their performance and functionality. As we saw in the previous sections, during the recent past many approaches have been proposed to fill the gaps between the increasing NRE costs of electronics system design and the matching of the requirements of modern applications.

Considering the design time and costs, the deployment of platform-based design provides an effective solution that leverages on the semiconductor manufacturing. A common platform can be manufactured in the large volumes that are required to make chip manufacturing economically viable. Concurrently, it can be specialized for use in a number of products, each of which is sold in smaller volumes. Moreover, the development of standard-based systems encourages the utilization of platform based design methodologies. The standard creates a large market with common characteristics as well as the need for product designers to differentiate their products within the scope of the standard. In this scenario, in order to achieve high performance, a platform vendor may allow a customer to specialize the platform in ways that require new sets of masks, but this negates many of the benefits of platform-based design, due to the still unsolved problem of the manufacturing costs.

One possible solution for successfully extending the application spaces of platform-based MPSoC while exploiting the efficiency of application-specific hardware is that of utilizing reconfigurable logic or structured ASIC solutions as hardware accelerators. In this scenario, the software programmability of processors addresses flexibility, while energy efficiency and performance are addressed by the adoption of powerful configurable or reconfigurable hardware accelerators. The design and programming of such kind of a platform should be supported with design frameworks that assist the user in the customization of the platforms, by providing integrated hardware/software co-design environments that allow the user the implementation of the accelerator engines, and the evaluation of the performance improvement due to software-to-hardware migration starting from the early phases of the development of an application.

The main objective of this thesis is to evaluate the design space of multi-core platform equipped with application specific accelerators realized utilizing design-time configurable and reconfigurable solutions. The evaluation will

flow through the development of two different computational platforms. The Morpheus platform is a heavily heterogeneous multi-core reconfigurable DSP, whose heterogeneity lies in the different flavours and granularities of reconfigurable engines utilized as computational cores. The ManyAC platform is a regular and homogeneous multi-core system specifically addressing high performance, low manufacturing costs, and low time to-market. The main peculiarity of the Manyac platform is that of supporting three kinds of implementation technologies for customization: run-time configurable technology, via-programmable technology and metal-programmable technology. These technologies present different trade-offs between performance, energy efficiency and manufacturing costs, which will be analyzed in the course of this thesis. The thesis is organized as follows. Chapter 3 provides a detailed description of the Morpheus platform, analyzing its programming model, architecture, implementation and providing examples of applications mapping. Chapter 4 describes the Manyac platform, in terms of programming model, architecture, customization technologies and trade-offs that came out from the mapping of applications on the platform depending on architectural choices and the chosen configuration technology. Chapter 5 provides a quantitative evaluation of the developed platforms, with comparison to other state of the art devices, mainly focusing on the applications development time, performance, energy efficiency and manufacturing costs. Finally, Chapter 6 provides final considerations about multi-processor systems with configurable hardware acceleration.

Chapter 2

2 The Morpheus Platform

2.1 Overview

The Morpheus platform can be described as a coarse-grained, heterogeneous MPSoC, which maintains the structure typical of commercial ASSPs, and replaces the application-specific hardware accelerators with a heterogeneous set of reconfigurable engines in order to match the application computational requirements. It is composed of 4 main loosely coupled blocks, each one representing a subsystem featuring local memory and independent, software-programmable clock domain. An ARM9 processor core represents the user interface toward the system ensuring programming legacy typical of software programmable processors. The other computation units in the system are wrapped as auxiliary processor cores, and comprise a 16-bit CGRA (The Pact XPP-III [21]), which is suitable for arithmetic computation such as FFT, DCT, and real time image processing, an embedded FPGA (eFPGA) device

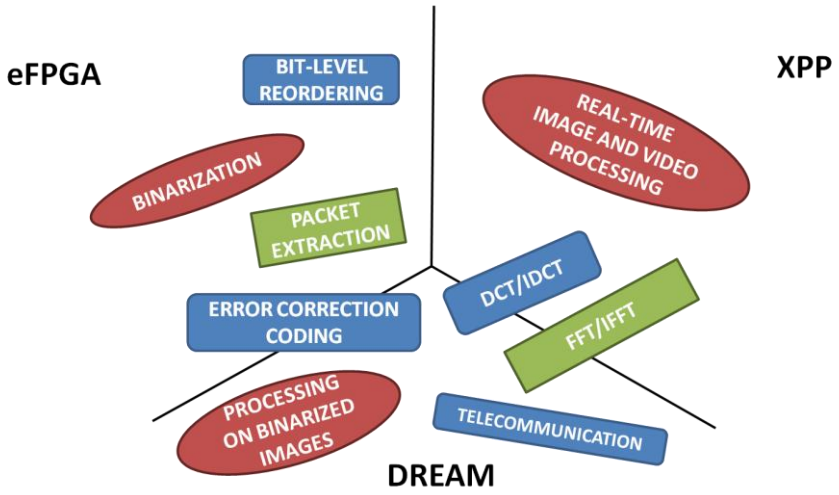


Figure 2.1: *View of the Morpheus application space.*

(the Abound Logic Flexeos core [15]), which can easily handle bit level computation, and a mixed-grain 4-bit reconfigurable datapath (the DREAM reconfigurable processor [19]) which is suitable for a larger set of applications, from error correction coding and CRC to processing of binarized images. The natural application environment for each computation unit is shown in Figure 2.1.

2.2 Computational Model

The computational model of Morpheus is based on the Molen paradigm [14]. The whole architecture is considered as a single virtual processor, where reconfigurable accelerators are functional units providing a virtually infinite instruction set. Tasks (i.e., application kernels) running on the reconfigurable units or on the ARM itself should be seen as instructions of the virtual processor. The configuration bitstream of the reconfigurable engines represent the virtual instructions micro-code, with the added value of being statically or dynamically reprogrammable. According to this paradigm, increasing the granularity of operators from ALU-like instructions to tasks running on reconfigurable engines, the granularity of the operands is forced to increase accordingly. Operands cannot be any more scalar C-type data but become structured data chunks, referenced through their addressing pattern, be it simple (a share of the addressing space) or complex (vectorized and/or circular addressing based on multi-dimensional step/stride/mask parameters). Operands can also be of unknown or virtually infinite length, thus introducing the concept of stream-based computation. From the architectural point of view the Morpheus handling of operands can be described at two levels: Macro-Operand is the granularity handled by extension instructions, x controlled by the end user through the ARM program written in C. Macro-operands can be data streams, image frames, network packets or different types of data chunks whose nature and size depends largely on the application. Micro-Operands are the native types used in the description of the extension instruction, and tend to comply with the native data-types of the specific reconfigurable engines

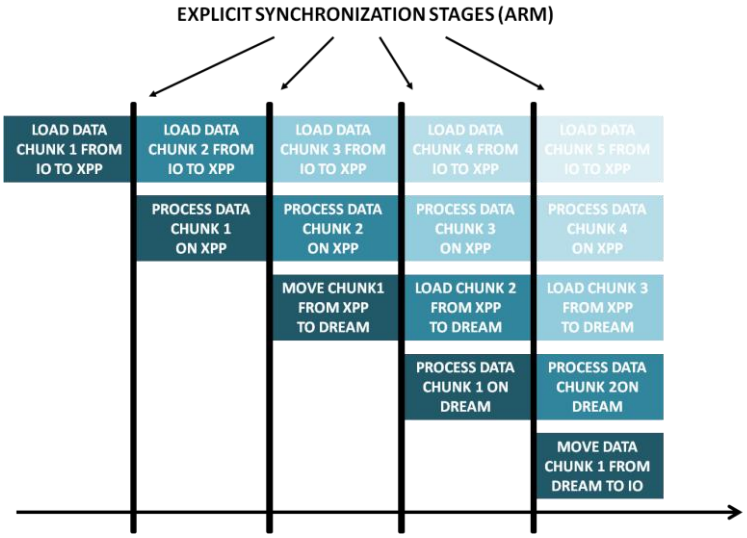


Figure 2.2: *Morpheus computational model.*

entry language. Micro-operands will only be handled when programming the extensions.

As the Morpheus platform is required to process data-streams under given real time constraints the work of user at system level is to schedule tasks in order to optimize the partitioning of the applications computational demands over the available hardware units. The aim of the mapping task should be that of building a balanced pipelined flow in order to induce as few stalls as possible in the data flow in order to sustain the required run-time specifications. The computation should be partitioned on the 3 different reconfigurable engines and the ARM core as much as possible in a balanced way. Figure 2.2 provides a generic example of application mapping, utilizing only two reconfigurable engines for simplicity. It appears evident how the overall performance will be constrained by the slowest stage, where a stage can be either computation or data transfer. The timing budget of each stage is flexible, and can be refined by the user, much depending on the features of his application. The interface between the user and all hardware facilities is the main processor core. Hardware resources are triggered and explicitly synchronized by software routines running on the ARM. In order to preserve data dependencies in the data flow without having to constrain too much size and nature of each application kernel the computation flow can be modeled according to two different design description formalisms: Petri Nets (PN) and Kahn Process

Network (KPN)[59]. In the first case the above described synchronization is made explicit, and each computation node is triggered by a specific set of events. In the second case synchronization is implicit, by means of FIFO buffers that decouple the different stages of computation/data transfer. Generally speaking, the XPP array appears suited to a KPN-oriented flow, as its inputs are organized with a streaming protocol. Unlike XPP, DREAM is a computation intensive engine: input data are iteratively processed inside the reconfigurable engine's local memory. Finally M2K is an eFPGA device programmed in HDL, so that any computation running on it can be modeled according to either formalism. A KPN can be described as a sub-net of a larger PN, while the contrary is not possible: if the target application fits well to the KPN formalism, it appears relatively easy to map it on XPP and eFPGA exploring the local IO buffers as FIFOs, while if the application should exploit DREAM the pattern will have to be extended to a PN with XPP/eFPGA implementing a sub-net organized as KPN. In other cases, a streaming approach cannot be applied as different reconfigurable engine operation may be required to run iteratively on the local buffers to describe a given computation kernel, thus a full PN approach must be applied. The rules of a generic PN can be briefly described as follows: A given node can compute (trigger) when all preceding nodes have concluded computation and all successive nodes have read results of the previous computation. In the context of Morpheus these rules can be rewritten as follows. A given computation can be triggered on a given reconfigurable engine when:

- The Bit-stream for the application was successfully loaded
- All input data chunks have been successfully uploaded to the reconfigurable engine local buffers
- All output data chunks that would be rewritten by the current iteration have been successfully copied from the reconfigurable engine local buffers to their respective destinations

In the case of PN, ARM is required to verify the PN consistency and produce the preceding/successive tokens triggering computation stages. Of course, if data-chunks are large enough, this monitoring will not be required very often. Each reconfigurable engine computation round is applied to a finite input data chunk, and will create an output data chunk. In order to ensure maximum

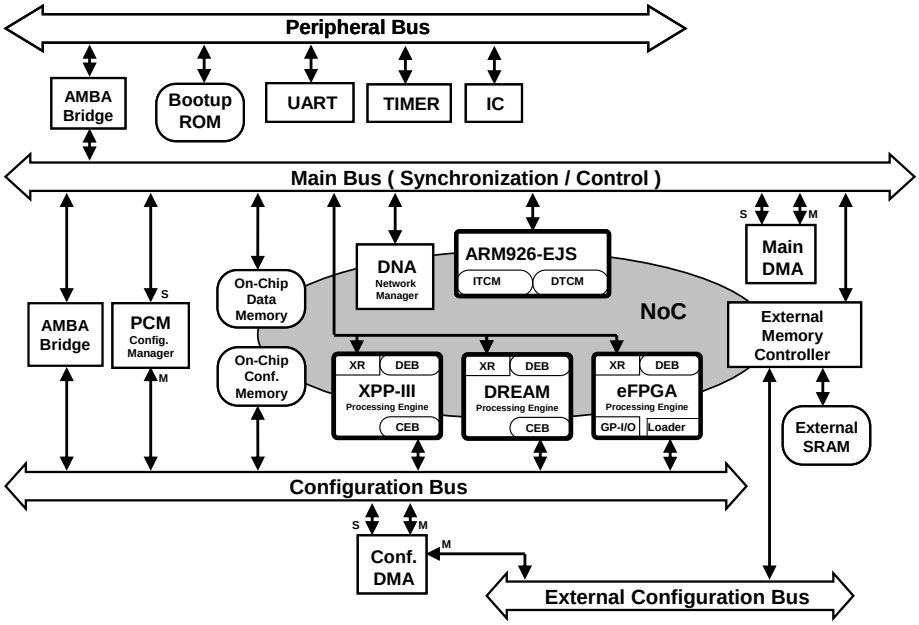


Figure 2.3: Morpheus SoC Architecture.

parallelism, during the reconfigurable engine computation round N the following input chunks N+1, N+2,... should be loaded, filling all available space in the local buffers but ensuring not to cover unprocessed chunks. Similarly, previous available output chunks . . . , N-2, N-1 should be concurrently downloaded ensuring not to access chunks not yet processed. This mechanism is defined ping-pong buffering, and is utilized to provide a sort of processor controlled coarse grained FIFO access.

2.3 Architecture

Figure 2.3 shows the system architecture of the Morpheus platform. As mentioned before, the SoC is built around three heterogeneous, reconfigurable engines which target three different computation styles. These IPs were selected due to their complementary capabilities, introduced in the system as RTL entities and finally implemented and integrated in the design as mix of custom and synthesizable standard cell based macros. An ARM 926EJ-S RISC processor, equipped with 16K I-cache and D-cache, plus 16K software-

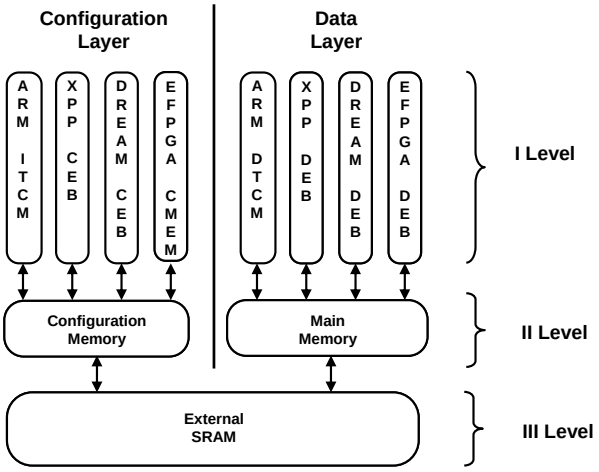


Figure 2.4: *Morpheus SoC Memory Hierarchy.*

managed D- and I- Tightly Coupled Memories (TCM) and a standard set of peripherals connected through a specific AMBA-APB peripheral bus acts as system supervisor.

ARM manages all communication, synchronization, and reconfiguration of the SoC by means of a dedicated "Main" AMBA-AHB bus. All computation, communication and configuration resources in the system are controlled by set of control registers mapped on this bus. The bus is hence critical, but since it carries only control information at computation time, bandwidth is not considered a significant issue. For debugging purposes, the main bus is also capable of accessing all data-storage resources in the system but this feature is not utilized in normal computation.

The SoC memory architecture is organized on three levels of hierarchy, that can in turn be logically divided into a data layer and a configuration layer (Figure 2.4). ARM TCM, and the local buffers of the reconfigurable engines represent the first level of memory hierarchy, local to each functional unit. A second level is composed of 512KB of on-chip SRAM, which is conventionally split into 256 KB data memory and 256 KB configuration memory. The third and last level is represented by the external off-chip memory, which stores both configuration and data. Data are exchanged between each reconfigurable engine and the ARM domain by means of a set of Data Exchange Buffers (DEBs). DEBs are dual port, dual clock memory banks that act as local data storage for reconfigurable engines as well as

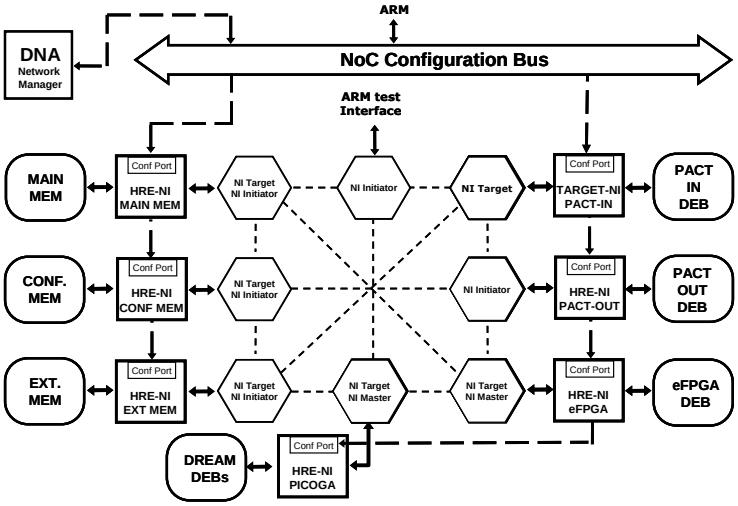


Figure 2.5: *Morpheus Communication Infrastructure.*

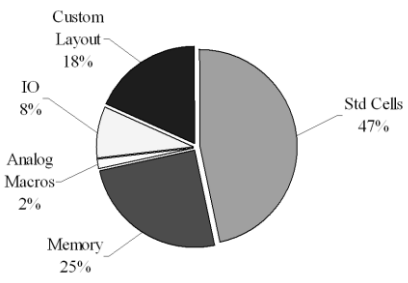
providing safe clock domain crossing. DEBs are seen by ARM and NoC as a single and coherent addressing space. On the other hand, a reconfigurable engine can only address/access data in the local DEBs and has no other visibility of the external world. Data dependencies and computation synchronization between the reconfigurable engines and the ARM domain are resolved by software via a set of exchange registers (XR) mapped in the DEB addressing space. Depending on the nature of the reconfigurable engine and of the features of the application kernels deployed, DEBs can be configured by ARM as FIFOs or Random Access Memories (RAM). Configuration bits are transferred similarly through dedicated Configuration Exchange Buffers (CEBs).

The Morpheus data communication infrastructure is based on a 64-bit, 8-node STNoC [61] included in the design as an RTL IP. The NoC is composed of three basic blocks: the router, the network interface and the physical link. Connections between NoC routers define the topology of the NoC (Figure 2.5). The NoC connects up the computational resources of the SoC (XPP-III, DREAM, eFPGA, ARM) and to the available data storage elements (main memory, configuration memory, external memory). Chip level transactions are handled by a set of two-port DMA engines, each local to a given Network Interface. One port drives the initiator port of the network interface while the secondary port is connected to the reconfigurable engines local buffers. NoC

DMAs are programmed, triggered by ARM via the Main AMBA-AHB bus, and consequently generate traffic on the NoC channels.
Morpheus fully supports dynamic reconfiguration, so that each reconfigurable engine can be reconfigured while the others are computing. With the exception of the eFPGA, reconfigurable engines are also multi-context, meaning that configuration bitstreams can be cached into internal configuration memory and the engines are capable to switch their functionality in one clock cycle. Configuration bit-streams flow through an independent AMBA-AHB "configuration bus".
The reconfigurable engines are encapsulated in independent clock islands, dynamically controlled via software. Frequency synthesis for the three islands is performed by three separate PLLs. This solution has the advantage of allowing fine grain selection of operating frequencies for each of the three computational engines, enabling the user to carefully tune the optimal power versus performance trade-off for each application. The drawback of this solution is that each PLL frequency re-setting requires a 400us locking time, but given the long configuration time of each reconfigurable engine this overhead proves insignificant.

2.4 Implementation

The Morpheus SoC is composed of a mixture of custom-designed digital macros (PiCoGA and eFPGA), embedded SRAM memories and standard cell regions, partitioned as described in Figure 2.6. The main characteristics of the Morpheus chip are reported in Table 2.1, while a photograph of the Morpheus chip is shown in Figure 2.8.



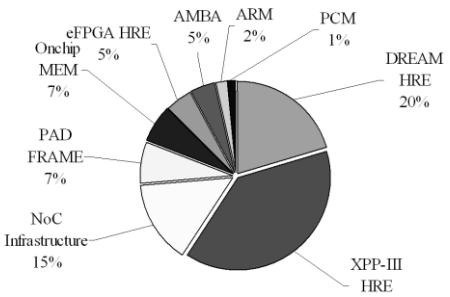


Figure 2.6: *Morpheus Area by design object* . **Figure 2.7:** *Morpheus Area by entity*.

Process Technology	90 nm CMOS90GP Process, 7-metal layers
Power Supply	1,0V for core, 3,3 for I/0
Area	110 mm^2
Transistor Count	44M Logic, 1,1Mbyte SRAM
Pinout	256, 163 I/O
Operating Frequency	ARM, BUS, NoC: 250 MHz XPP : 0 - 160 MHz DREAM : 0 - 200 MHz eFPGA : 0-140 MHz
Power Consumption	Static Power : 235 mW ARM + NoC : 600 mW @ full speed XPP : 1200 mW @ full occupation - full speed DREAM : 420 mW @ full occupation - full speed eFPGA : 112 mW @ full occupation - full speed

Table 2.1: *Morpheus chip characteristics.*

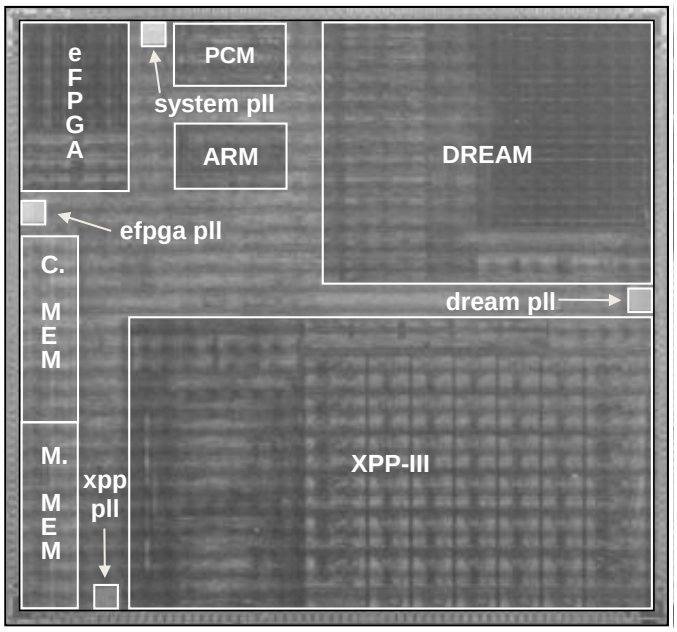


Figure 2.8: *Morpheus Chip photograph.*

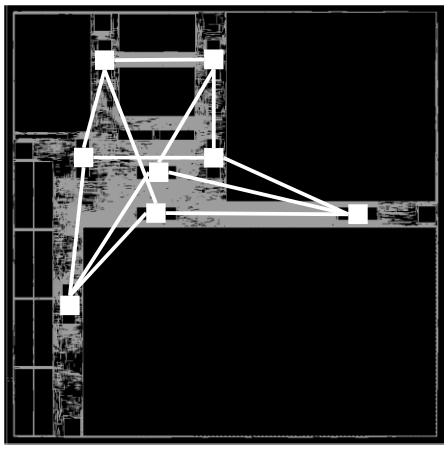


Figure 2.9: *Morpheus NoC topology.*

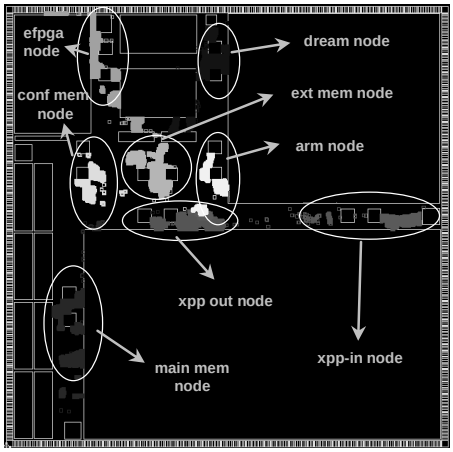


Figure 2.10: *Morpheus NoC Floorplanning.*

The three reconfigurable engines were designed separately, and re-utilized as hard macro-blocks to partition and better organize the physical design effort [63]. The reconfigurable engines are located on three different clock islands and positioned at the chip corners to ease global routing. The XPP-III macro is flipped horizontally so as to better match the input/output ports with NoC topology; it is placed on the bottom-right side of the chip. The DREAM macro is placed top right, the eFPGA in the top left corner of the die, while the ARM processor macro, working at the system clock frequency is placed in the middle of the chip. The PLLs are placed on the four boundaries of the die in order to avoid coupling noise among their analog supplies. Figure 2.7 shows the amount of area occupied by the entities composing the Morpheus platform.

The NoC implementation was realized following the same hierarchical approach of the whole design: the router was implemented separately and included in the course of design as a custom macro during top level implementation. The sites of the routers in the final design were carefully selected in order to constraint the place & route tools for placing the network interfaces cells, and as far as possible to balance the NoC physical link routing, avoiding congestion areas and unduly long wires. Figure 2.9 describes the logical connections between NoC nodes which define the chip layout topology while Figure 2.10 shows the floorplanning of the NoC components. Implementation details of the whole communication infrastructure are reported in Table 2.2.

Considering the clocking scheme, each reconfigurable engine features two

Entity	# of Instances	Std Cells count [Kgates]
Routers	16	292
NoC Initiator NIs	7	202
NoC Target NIs	7	113
AHB to NoC bridges	6	265
NoC to AHB bridges	6	260
DMAs	6	434
DNA	1	65
Other (Bus, mpmc…)		571
Total		2202

Table 2.2: *Details of the NoC implementation.*

clock inputs. One global clock is used to feed the system-side of the synchronization barriers (DEBs, CEBs and XRs) and was properly balanced in order to compensate for insertion delays by the internal clocks. Each reconfigurable engine can be clocked either by the global system clock, or by its private clock, programmed by the ARM setting PLL division factors on specific memory-mapped registers. This mechanism allows one to exploit Globally Asynchronous Locally Synchronous techniques by enabling dynamic frequency scaling on the three auxiliary cores.

2.5 Mapping of applications

The aim of the mapping task on the Morpheus platform should be that of maximizing parallelism and the concurrent execution of computations and data transfers [67]. When possible, it is desirable to partition an application among all available computational cores. In other cases, application kernels can be mapped on a single core. In order to manage the specificity of the reconfigurable engines while preserving a homogeneous interface, the Morpheus mapping strategy enforces a strict separation between management of data flow, synchronization, and control performed by the ARM processor,

and execution of computational kernels performed by the reconfigurable engines. The mapping of accelerations on the reconfigurable engines is library oriented: the user is required to develop it himself using the proprietary tools of the reconfigurable engines and generate the configuration bitstream for the required kernels. The analysis, profiling, and implementation of kernels are supported by specific proprietary tools and languages for the reconfigurable engines (respectively NML [64] for XPP, Griffy-C [65] for DREAM, and VHDL for the eFPGA). On the other hand, application partitioning is performed at compilation time, and driven by an accurate analysis of kernels. This choice is usually driven by the matching between kernels to be implemented and architectural features of the reconfigurable engines. As the Morpheus platform is composed of three granularities of reconfigurable fabrics, a first rough analysis considers both the average data size and complexity of the kernel to be implemented. In addition, other factors could impact the mapping of kernels. The Instruction Level Parallelism (ILP) can play a crucial role in this context. For example, even for 8-bit operand widths, XPP can be the most suitable engine if the applications allow its SIMD capabilities to be fully exploited. On the other hand, bit-level optimizations could be beneficial to achieve better performance when arithmetic optimization involves constants or Data Flow Graphs (DFGs) with feedback arcs, especially if look-ahead technique can be applied. In this case DREAM or eFPGA would be a better choice. In the following, this section describes examples of various signal processing kernels implemented on the reconfigurable engines and the example of an entire application being partitioned among the computational cores.

2.5.1 Kernels Mapping Examples

AES/Rijndael

The Rijndael algorithm [69] is a symmetric key cipher implementing a substitution-permutation network, selected by the National Institute of Standard Technology (NIST) to implement the Advanced Encryption

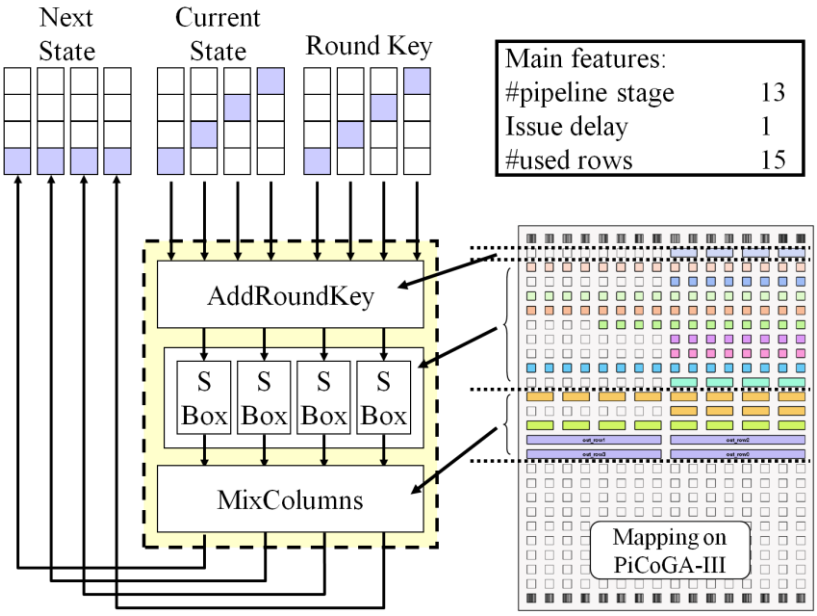


Figure 2.11: *DREAM implementation of the Rijndael algorithm.*

Standard (AES) in 2001. The size of both ciphered block and key, as well as the number of iterations (rounds), depends on the security level required.
The encryption process starts by arranging the block in a matrix form termed State. The AES encryption process is performed by the iteration of 4 routines on the State Columns: SubBytes, ShiftRows, MixColumns, AddRoundKey. The number of iterations depends on the key width and ranges from 10 to 14. Basically, AES is mostly defined by operations on Galois Field arithmetic GF(28). The AES/Rijndael algorithm requires to implement three operations on GF: the sum, the multiplication by constant amount, and the inverse multiplicative. While the sum and the multiplication with constant amount can be written with standard operators (XORs, ANDs and shifts), the inverse multiplicative requires to be implemented over the Galois Field GF(28). Operations over GF(28) can be re-written over the composite field GF((24)2). Thanks to this property the inverse multiplicative can be mapped on two elementary GF(24) operations natively available on the PiCoGA-III RLCs [70] . SubBytes operation elaborates byte-by-byte the input block, without correlation among processed bytes (Figure 2.11). For that reason, the byte-level permutation can be anticipated before SubBytes, thus making possible to

use the modulo addressing provided by the DREAM address generators to implement the ShiftRows stage. In addition, utilizing different memory banks for storing the different rows of the State matrix, PiCoGA is able to load a new State column for each cycle.
The rotation applied by ShiftRows is handled by changing the starting address of each bank, while the different number of columns is handled by setting the address generator end-of-count. The organization by column allows the packing of the MixColumns function in the same PiCoGA operations.
Figure 2.11 shows the corresponding implementation scheme. The four operations are mapped in a single PGAOP utilizing 15 rows of the datapath. The PGAOP computes AddRoundKey, SubBytes and MixColumns on the four current bytes, leaving the addressing engine to handle the ShiftRows for both block and key access. A different set of buffers is used to store PGAOP results, since it is not possible to read-and write a memory bank in the same cycle. This implementation requires 4 PGAOP call in order to accomplish one AES/Rijndael Round, after that we need to re-configure the interconnect cross-bar in order to swap the used I/O buffers.

CRC-32

Cyclic Redundancy Check (CRC) is an error detection coding utilized in many telecommunication protocols such as Ethernet, SONET and Bluetooth in order to verify the consistency of transmitted data. The mathematical background of the CRC algorithm is represented by Linear Feedback Shift registers (LFSR), widely used circuits in modern multimedia and communication devices thanks to their statistical properties. For instance, they are utilized for scrambling purpose in 802.11 (WiFi), 802.15.4 (ZigBee), 802.16 (WiMax) and Digital Audio/Video Broadcasting (DAB/DVB) standards. Furthermore, GSM telephones, Bluetooth devices, and almost all commercially produced DVD-Video discs utilize LFSR as stream cheaper. The serial block diagramof a LFSR is reported in Figure 2.12a, while its utilization as CRC encoder is shown in Figure 2.12b. The CRC input bits are combined with bits flowing in the feedback loop. In this case, as well as for most of the real LFSR applications, we consider feedback loops defined over

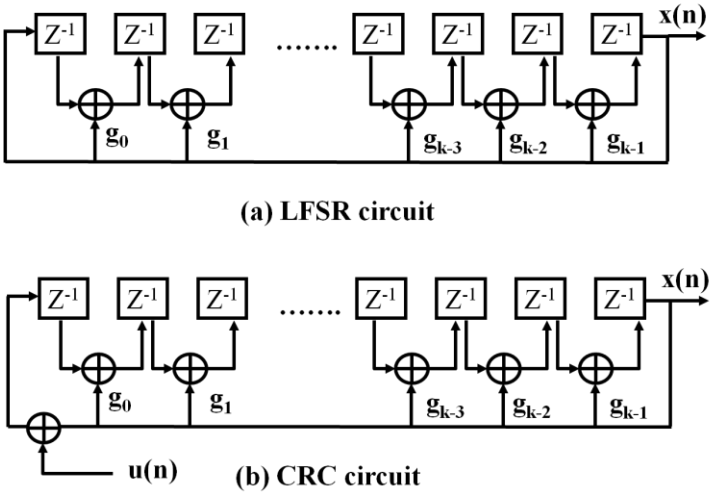


Figure 2.12: *a) LSFR circuits b) CRC circuit.*

an the Galois Field GF(2). This means that the additions necessary in the loop are defined in GF(2) and thus implemented with exclusive-ORs. For that, the DREAM implementation massively uses the 10-bit XOR operation which can be mapped on a single PiCoGA RLC. Furthermore, in order to exploit pipelining on PiCoGA as much as possible, solutions which are not requiring pipeline stalls during the processing flow have been evaluated. The approach proposed by J.H. Derby in [73] was selected. This method allows to parallelizing CRC/LFSR computation without increasing the complexity of the feedback loop. In fact, LFSR can be modeled in a matrix form, where the parallelization is mostly done through matrix exponentiation. Thanks to this property, it is possible to find a transformation, which allows keeping the resulting matrix in a "simple" form. Working on a "transformed" field CRC/LFSR space, we need to call an anti transformation block on the output stage of the CRC. The 32-bit CRC application has thus been partitioned on two PiCoGA operations: the first one implements the transformed status update, while the second one implements the anti-transformation block of the CRC output sequence [72]. The main benefit of this approach is that increasing the available resources allows greater look-ahead factors, hence the number of bits processed per cycle. On the other hand, this partitioning does not decrease performance because the output sequence transformation is required only at the end of the message and it does not break the pipeline

evolution during the status update operation. The status update operation has been generated for different values of M, finding that PiCoGA is able to elaborate up to 128 bits per cycle utilizing all the 24 rows of the array. On the other side the occupation of the output update operation is 10 rows.

Edge detection

The edge detection is a morphological operator widely used in image processing, particularly on motion detection algorithms [71].
Mathematically, it is based on the Sobel Convolution, a discrete differentiation operator computing an approximation of the image intensity function gradient. At each point in the image, the Sobel operator outputs the corresponding gradient vector. From a practical point of view, the Sobel operator is based on the convolution of the image with an integer, hi-pass filter in both horizontal and vertical directions. The following formula shows the mathematical formulation of the operator:

$$\sum_{i=-1}^{i=1}\sum_{j=-1}^{j=1} K(i,j) * p(x-i)(y-i)$$

Being E(x,y) the pixel under elaboration, p(h, k) the pixel in the 3x3 matrix centered in (x,y), and K the Sobel matrix, for horizontal and vertical edge detection, defined as:

$$Kh = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad Kv = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

The resulting gradient is defined as:

$$E = \sqrt{E_h^2 + E_v^2} \cong |E_h| + |E_v|$$

In some cases, as in that of edge detection, the scope of the application is not to detect the magnitude, but the presence of a gradient. For that many motion

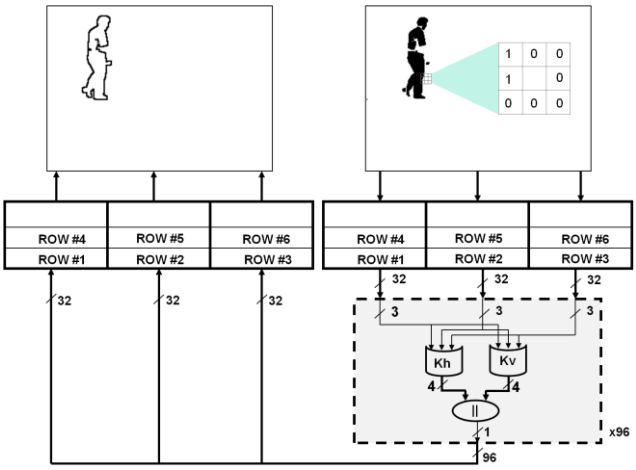


Figure 2.13: *Edge detection implementation on DREAM.*

detection algorithms work on binarized images allowing an easier detection of the edges, thus detection of external agents in the scenario. This feature can be exploited on the DREAM architecture. In fact, inverse-binarized edge detection can be represented as:

$$IB(x,y) = \begin{cases} 0 \; if \; E(x,y) = 0 \\ 1 \; if \; E(x,y) \neq 0 \end{cases} = \begin{cases} 0 \; if \; |E_h(x,y)| + |E_v(x,y)| = 0 \\ 1 \; if \; |E_h(x,y)| + |E_v(x,y)| \neq 0 \end{cases}$$

Since each pixel can be represented by 1 bit, the result of horizontal and vertical Sobel convolution is in the range of [-4, +4] requiring 4 bits. Moreover, it should be noted that given Eh and Ev components, IB(x,y) will be 1 if and only if all the bits of Eh and Ev are zeros, making possible to implement this computation by an 8-input NOR, thus utilizing 2 PiCoGA RLCs per pixel (Figure 2.13). In this case, since each pixel is represented by1 bit, we elaborate 3 * 32 =96 pixels per PiCoGA operation, packing 32 pixels in a single 32-bit memory word stored in the local buffer. The processing is based on this simple operation repeated many times for all the pixels in a frame. It is thus possible to operate concurrently one or more pixels at time unrolling the inner loop of the computation flow. Considering that this operation takes data from three adjacent rows, we use a simple 3-way interleaving scheme in which each row is associated to a specific buffer by the rule buffer_index = #row mod 3. Rows are stored contiguously in each buffer, and the PiCoGA can read one row chunk per cycle. The address generators

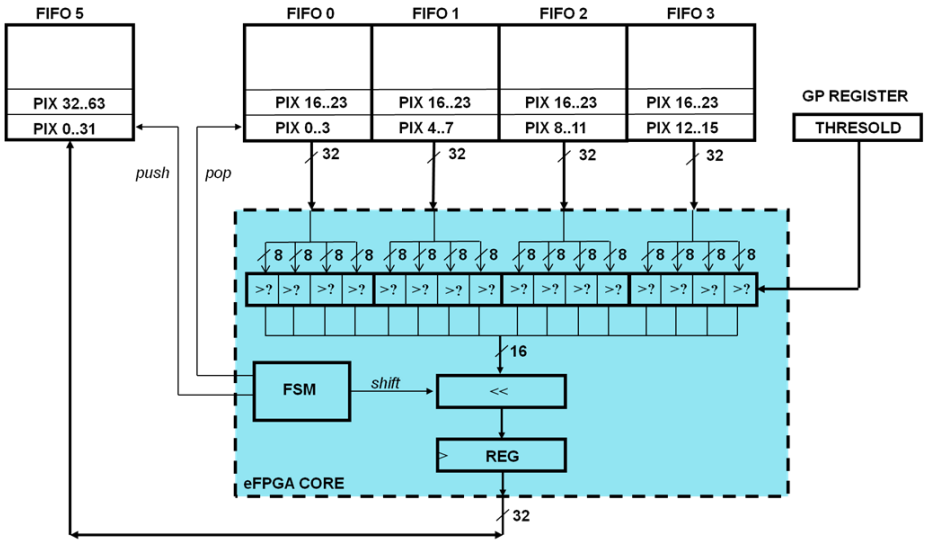


Figure 2.14: *Implementation of the binarization application on the eFPGA.*

are programmed accordingly with the above described access pattern, while programmable matrix is used to switch from one row chunk to another. Boundary effects due to the chunking are handled internally to PiCoGA that can hold the pixels required for the different for the different column elaboration in its internal register, thus avoiding data re-read. The occupation of this operation on PiCoGA is 21 rows and the complete convolution is performed processing the source image with the same PGAOP for both horizontal and vertical edge detection.

Binarization

The binarization or thresholding is a image processing method of image segmentation. From a grayscale image, thresholding can be used to create binary images. During the binarization process, individual pixels in an image are marked as object pixels if their value is greater than some threshold value and as background pixels otherwise. It is utilized in image processing, such as in printers, in order to transform grayscale images to a black and white before printing them on a paper. From the mathematical point of view it is composed by a comparison between each pixel belonging to the target image and a fixed

threshold and a further packaging of the comparison results to 32-bit words. Figure 2.14 shows the implementation of a binarization application on the eFPGA architecture. This application takes benefit from the flexible nature of the eFPGA, utilizing the whole bandwidth of its input buffer, configured for this specific application as FIFOs. 16 8-bit pixels, organized as 4 32-bit words, are concurrently read from the input FIFOs and processed by the logic implemented on the eFPGA core. A finite state machine (FSM) implemented on the eFPGA core detects the presence of data on the input buffers generating the pop signal accordingly. The logic mapped on the core performs 16 concurrent comparisons between the input pixels and the chosen threshold, which is stored in a general purpose register accessed by the ARM processor through the local buffer interface, and connected to the FPGA core I/O. As the number of concurrent processed pixels is 16, while the output binarized image needs to be formatted as 32-bit words the FSM is also responsible for packaging the binarized image storing intermediate results in a register and shifting the binarized vector when necessary. The packaged data is then pushed to the output buffer, configured for this application as output FIFO.

Ethernet

Ethernet protocols refer to the family of local-area network (LAN) covered by the IEEE 802.3 standard. It is a widely utilized communication protocol, almost in every personal computer we can find an Ethernet peripheral. This section describes the implementation of a 10/100 Ethernet Media Access Controller (MAC) on the eFPGA device, showing its capabilities as configurable I/O peripheral.

The main purpose of the MAC is to connect an Ethernet PHY, placed on the circuit board, to the ARM side of the Morpheus system allowing the chip to communicate with the external world utilizing an Ethernet protocol as shown in Figure 2.15 [75].

The MAC core is mainly composed of four blocks: a management module responsible for the configuration of the communication with the PHY, a control module in charge of data flow control, a transmission and a reception module. These modules implement the communication protocol toward the

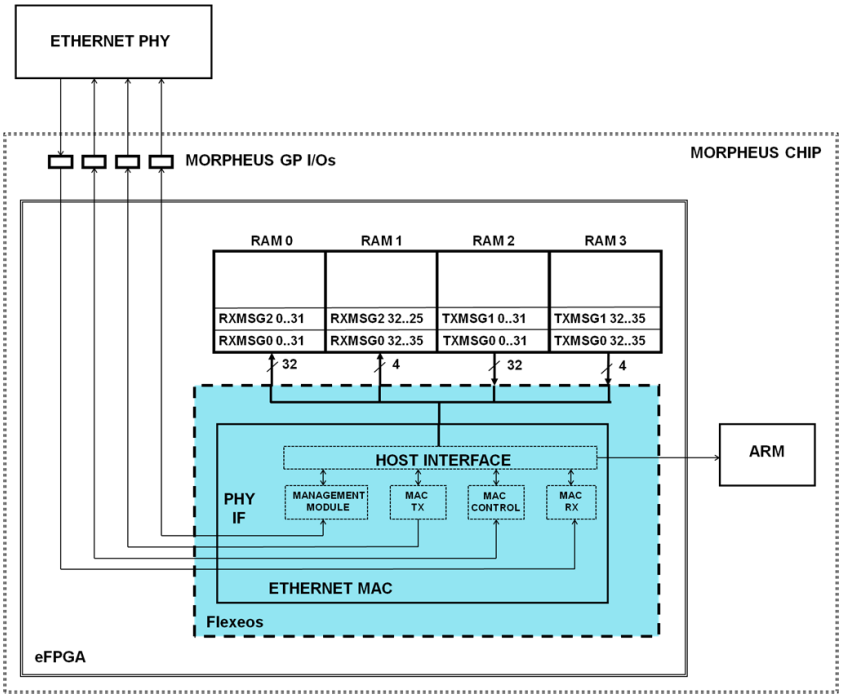


Figure 2.15: *Implementation of an Ethernet MAC on Morpheus.*

PHY. Indeed, the I/O signals of the five modules are connected to the eFPGA interface, and then out of the Morpheus chip thanks to the GPIOs exported to the chip pad frame. On the other side of the MAC, the described modules control through a host interface the data and control ports of the local buffers, configured in this application as random access memories.

As the Ethernet packets are 36 bits wide, buffers 0 and 1 are utilized to store the received packets, while buffers 2 and 3 are utilized for transmission. Notifications of transmitted/received packets are performed by the ARM processor using an interrupt interface and memory mapped control registers accessible through the main bus.

RGB2YUV conversion

RGB2YUV conversion is an image processing algorithm, which converts pixel data between the common RGB and the YUV color representation,

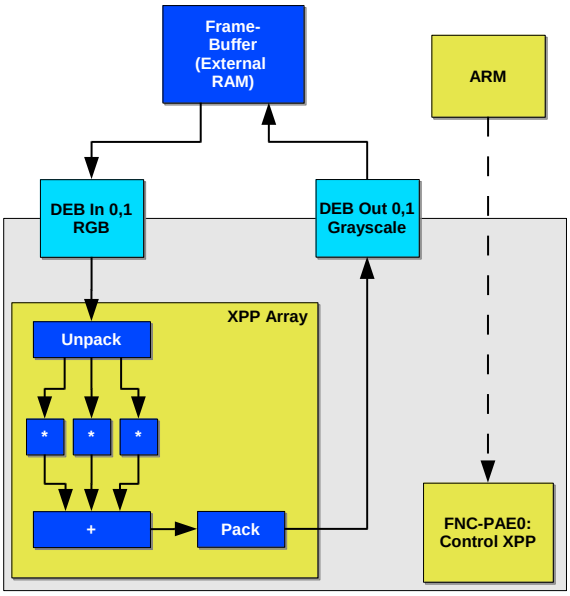


Figure 2.16: *Implementation of RGB2YUV on Morpheus.*

which is utilized in standard definition television formats such as PAL or NTSC. This luminance component is calculated as:

$$Y = 0.299*R+0.587*G+0.114*B$$

As this basic formula contains fractional arithmetic, which is not natively supported by the Morpheus reconfigurable engines, a modified variant following the ITU-R BT 601 standard [68] has been implemented, consisting of integer additions and multiplications as well as shift and rounding operations. Regarding the data format, one RGB pixel is represented as a 32-bit word, which contains 3x10 bits for the color components and two empty bits. This requires additional shift and logical operations for the isolation of the color components before processing. Consequently, the operation granularity of the application varies between 32 and 10 bits, which is beneficial for the comparatively large data word width of the XPP. Figure 2.16 shows the corresponding implementation scheme. As the calculations are performed per pixel, image data can be fetched in a streaming fashion without requiring extra buffering or specific memory access patterns. The application

is implemented using the XPP's Native Mapping Language (NML). The different arithmetic operations are mapped to NML primitives, which are connected via their inputs and outputs. These primitives are then translated to the array objects, which are automatically placed and routed by the XPP tools. Due to the limited complexity of the application, an automated pipeline balancing by the tools was feasible, which results in a perfect pipeline without internal stalls. During pipeline execution, each pixel is read from two parallel incoming FIFOs, processed, packed into a 32-bit output word, and finally output via two outgoing FIFOs. The complete pipeline achieves a throughput of one pixel per clock cycle and has an overall latency of 10 clock cycles, which offers sufficient processing performance for the targeted application domain. The application execution is controlled by one functional processing unit, which initiates the configuration of the XPP array, starts the array execution, and stops the XPP as soon as stopped by the ARM. Data transport for this application is implemented via parallel NoC transfers to and from off-chip memory, which are programmed and controlled externally by the ARM. The XPP's internal 4D address generators are not required in this application.

Motion Estimation

Motion Estimation (ME) detects motion between a reference frame and its preceding and succeeding frames within a movie sequence. The algorithm is based on block matching using the subtraction and absolute difference (SAD) as the decision criterion: the blocks are subtracted pixel-wise and the differences are accumulated for each possible block matching in a given search area. Finally, the matching with the smallest SAD is selected as it shows the best resemblance of reference and search block.
The algorithm is extremely computation intensive, which is due to the large amount of subtractions and accumulations per block matching and the exhaustive search approach that delivers the best results in terms of quality when compared to other block matching techniques such a three step search. For this application, the data word width is 10 bits per pixel, which are packed in larger data words of 16 bits. The image blocks are fetched in a regular

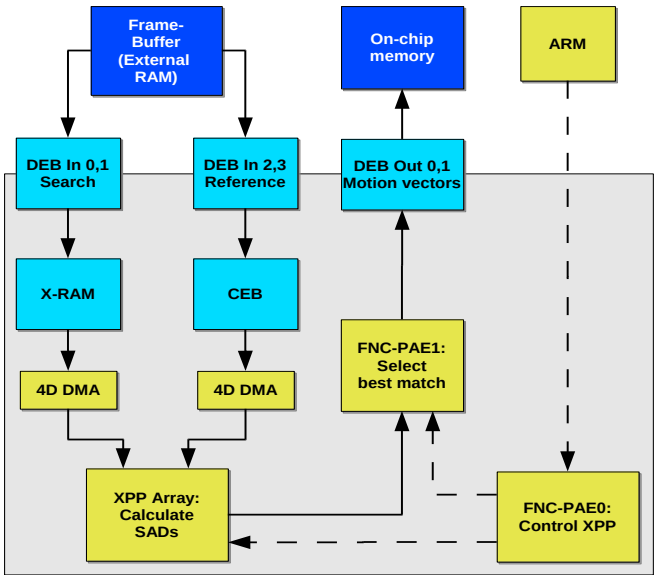


Figure 2.17: *Implementation of Motion Estimation on Morpheus.*

order, which allows predictable memory access patterns and the composition of a gapless pixel stream that matches with the XPP's streaming concept.
Following a specific optimization approach [74], three instead of two different image streams need to be fetched from off-chip memory, and a reordering of pixel data from a row-wise to a column-wise pixel representation is required. For this task, the implementation benefits from the XPP's 4D DMA address generators, which directly enable block based memory access patterns and a reordering of pixel data. As shown in Figure 2.17, the implementation is split into different parts: first data is fetched from all four incoming DEBs and is buffered into the XPP's internal and configuration memory, which has been partly converted into an additional data buffer for this application. Next, the 4D DMA convert the pixel streams and feed them into the XPP array, which performs the calculation of SADs that are then passed to the FNC-PAEs. These units select the best matches, which are finally written to MORPHEUS' on-chip memory via the XPP's crossbars and the outgoing DEBs. This concept fully exploits the XPP's capabilities as it utilizes all available computation, memory and data transport modules. The block matching part is also implemented using NML code. However, due its computation complexity and the corresponding large amount of processing elements, an automated

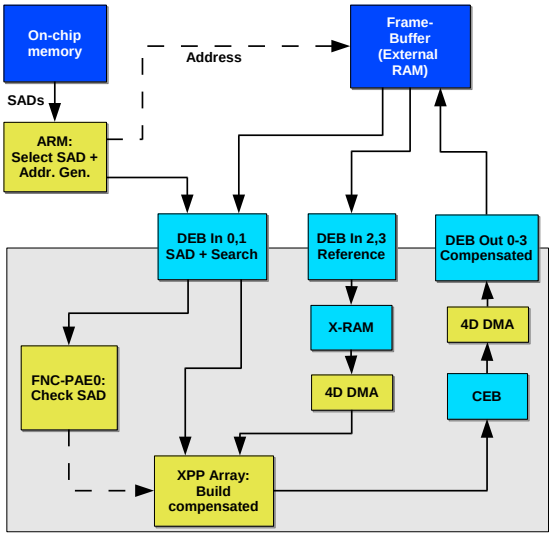


Figure 2.18: *Implementation of Motion Compensation on Morpheus.*

placement and routing was not feasible. Instead, the application has been manually optimized and placed in order to achieve a successful implementation. The FNC-PAE code is implemented as C functions, which also contain the crossbar and array configuration code. Because of the bidirectional execution of the ME, the XPP requires one internal reconfiguration during application execution in order to reinitialize all processing elements. Finally, the 4D DMA engines are controlled by the external ARM processor, which is necessary for the synchronization between on- and off-reconfigurable engine data transfers. External control is again performed by the ARM processor, which triggers and stops the XPP execution and loads the configuration data into the configuration memory.

Motion Compensation

The Motion Compensation (MC) is utilized to remove the detected motion in order to improve the image quality for the final noise reduction step. This is achieved by assembling a compensated image out of image blocks from the two search images. For this task, the MC module mainly executes comparisons, which validate the SAD with different thresholds and check the

compensated image stream for consistency. Depending on the particular best match for each block, the result image consists of image content from the preceding and succeeding images, which results in non-predictable memory access patterns that cannot be pipelined efficiently. This is even aggravated by the motion vectors, which produce unaligned random block offsets inside the off-chip memory. Due to these random memory accesses and the comparatively large data word width of 32 bits per pixel, the application is considered memory-intensive. For the implementation, shown in Figure 2.18, the application is again split into different parts, which are executed by different components. The selection of the matching direction and the off-chip memory address calculation are performed by the ARM, which also initializes the NoC-based data transfers from off-chip memory to the XPP's incoming DEBs and controls the XPP's internal address generators. The SAD is time-multiplexed with the pixel data and is transferred to one FNC-PAE element, which performs the threshold checks. Finally, the image assembly is executed on the XPP processing array. Similar to the RGB2YUV implementation, the array code is automatically balanced, placed and routed by the XPP tools. Reconfiguration control is implemented on the ARM, which starts and stops the XPP via the second FNC-PAE element.

2.5.2 Application Mapping Example

Video Surveillance Motion Detection Application

This section describes the implementation on Morpheus of a video motion detection application used in security and surveillance systems. The aim of the proposed algorithm is to detect the presence of external objects on a video transmitted by a camera framing a fixed background.

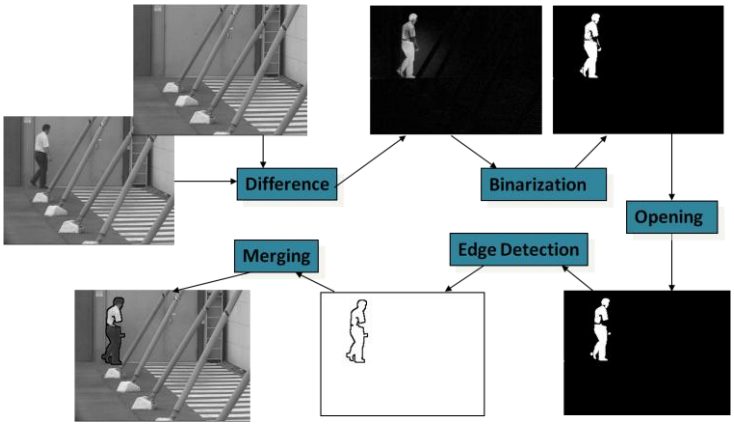


Figure 2.19: *Block scheme of the motion detection application.*

As shown in Figure 2.19, the application is composed of a few main kernels. For each video frame the first algorithmic stage performs the subtraction and absolute value between the current and the background image. The resulting maximum value is extracted and used to calculate the threshold for binarization. Three spatial operators then process the binarized image. Erosion and dilatation implement the opening kernel which de-noise the binarized image, while the edge detection implemented through a bi-dimensional Sobel convolution algorithm, creates the external object boundary. If an external object is detected, the final merge kernel returns the highlighting of that object on the original frame.

Table 2.3 shows the profiling on an ARM 926 EJ-S processor of the proposed application and the kernel mapping on the reconfigurable cores. From such

Kernel	ARM	Computation	Mapping
Sub/Abs/Max	3%	8-bit Arith.	XPP-III
Binarization	2%	Asymm. bit level	eFPGA
Opening	39%	Symm. bit. level	DREAM
Edge Detection	55%	Symm. bit level	DREAM
Final merge	1%	8-bit Arith.	ARM

Table 2.3: *Profiling and partitioning of the motion detection application.*

kernels, data/instruction level parallelism is then extracted and exploited on the chosen configurable fabric. The first stage of the application (SUB, ABS, MAX) is composed of strongly arithmetic operations on 8-bit operands. For this reason, the algorithm is suitable for mapping on the XPP processor, whose implementation can be parallelized up to four times exploiting the XPP-III SIMD capabilities. Similarly, the binarization stage can be efficiently implemented on the eFPGA, being mainly composed of comparisons and packaging, easily fitting the eFPGA device. The core of the computation is implemented by three morphological operators (EROSION, DILATATION, EDGE DETECTION) working on binarized images. The computation is thus composed of a many iterations of these operators on the same data-set. Thus, such an image can be stored in the DREAM local buffers and iteratively processed by the datapath. Moreover, these kernels, which have a native nature of 8-bitwidth arithmetic, can be implemented using bitwise operators thanks to the elaboration on the binarized image, perfectly matching the mid-grain nature of the PiCoGA datapath. Finally, the last merging stage again involves 8-bit arithmetic. Mapping on XPP could be an option, but that would require significant NoC transfers, and some time-multiplexing over the XPP array. In addition, being the last stage in the computation, it requires data packaging for which a RISC processor is more suited. Since the stage is not overly critical, it can be performed on ARM without affecting overall performance.

In order to determine the most suitable balance between computation throughput and data transfer, granularity was determined as 80x60 8-bit pixel image chunks, for which an optimized ANSI-C reference software solution [71] implemented on the ARM 9 processor has a cost of 715 cycles/pixel. A 4-stage coarse grain pipeline managed by the ARM processor through specific synchronization events processes the image chunks. More precisely, the completion of a transfer stage is notified by the communication DMAs, while the completion of a computation stage is notified by the reconfigurable engines through the exchange registers. If the local buffers of the reconfigurable engines are utilized in FIFO mode, computation is transparent and the communication engine notifies the conclusion of a given stage. This is the case with SAD and binarization in this example. If the reconfigurable engines DEBs are programmed in RAM mode, as occurs with

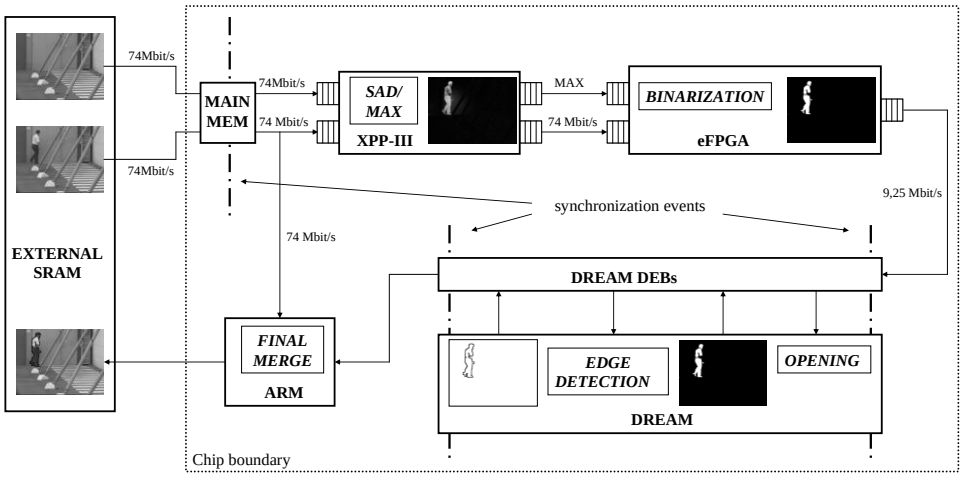


Figure 2.20: *Implementation of a motion detection video surveillance application on the Morpheus platform.*

erosion/dilatation and edge detection, the reconfigurable engine notifies itself the conclusion.

Figure 2.20 describes the implementation of the motion detection application on Morpheus. The kernels partitioned are distributed so as to build a balanced streaming pipeline through the NoC over the various different reconfigurable engines. At the first pipeline stage the reference image and the background image are loaded onto the main on-chip memory. During the second stage, image chunks are processed as a streaming pipe which flows through XPP and the eFPGA, performing sequentially SAD and binarization, and are finally stored on DREAM DEBs. In the third pipeline stage DREAM processes the binarized image chunks, internally iterating erosion, dilatation, Sobel vertical and Sobel horizontal operations. In the last stage the ARM processor merges the reference image with the results of overall computation and stores the final image in the external memory. Configuration management is not necessary for this application, since all the configuration bitstream can be loaded off-line on reconfigurable devices. XPP and eFPGA maintain the same configuration during execution of this whole application, while the DREAM processor can exploit its reconfiguration capabilities, loading each of the four kernel bitstreams onto one configuration context.

Considering a CCTV 640x480, 30 Frames/second, grayscale video, with 8-bit pixels, the real time bandwidth is 9.2 Mpixel/s (74 Mbit/s). Assuming partitioning as described, the critical kernel remains Opening/Edge Detection, which is performed on the DREAM processor at a computational cost of 1.27 cycles/pixel, thus leading to a real-time frequency for the reconfigurable core of 12 MHz. Considering the other computation cores, PACT can perform Subs/Abs/Max kernel @ 0.25 cycle/pixel, while the eFPGA can perform the Binarization kernel @ 0.125 cycle/pixel. Since the two kernels belong to the same pipeline stage, the real-time frequencies for the two devices are respectively 5 and 2.5 MHz. With the described frequency configuration, measurement on the Morpheus test chip showed a power consumption of 600 mW. On the other hand, when working at the maximum computational power, it is possible to process videos coming from up to 7 cameras. The bottleneck in such a case is represented by the external memory controller which is capable of providing 1.6 Gbit/s bandwidth. Removing this limiting factor (accesses to external memory) and using maximum computational power, Morpheus would be capable of processing videos from up to 16 cameras concurrently while consuming a measured power of 1.45W.

2.6 Performance Analysis

This section analyzes main features of the Morpheus platform through a quantitative analysis of application implementations on the 90nm chip prototype. In the first part, a theoretical analysis of the Morpheus platform is provided, based on the characterization of the chip performance and power consumption performed utilizing ad-hoc test vectors appositely realized to stress different parts of the device with well-defined computational loads. The second part of this section gives a detailed view of the Morpheus platform overheads, bottlenecks and benefits through the implementation of kernels implemented on the device.

2.6.1 Characterization of the Morpheus performance

In order to evaluate the Morpheus performance from a theoretical standpoint, the granularity of operations has been classified increasing their granularity, and the affinity of each reconfigurable device composing the system has been analyzed according to this classification. The operations classification starts from logic operation of different output width (1,4), arithmetic operation (sum, sub, shift, comparison.) of increasing operand width (4,8,16,32) up to multiplications with 16 and 32 bits wide operands. Figure 2.21 shows the related performance delivered by each reconfigurable engine. As the most limiting factor of fine-grain devices is routing, the performance of the eFPGA reconfigurable engine was estimated synthesizing logic and arithmetic blocks of different granularities utilizing its proprietary tools. On the other hand, performance of the coarser reconfigurable engines, less sensitive to routing congestion, was approximated to the number of logic and arithmetic blocks of different granularities that can be implemented only considering the available computing elements.
On the other hand, if we consider power as evaluation metrics, the main contributions consists of the working frequency of each clock island, the number of resources utilized (PAE for XPP, RLC for DREAM, LUT for eFPGA), the routing path, as well as the number of concurrent accesses to memory banks or FIFOs. For this reason a characterization of the Morpheus power consumption has been performed by running ad-hoc test vectors which utilize a pre defined number of resources of each reconfigurable engine.

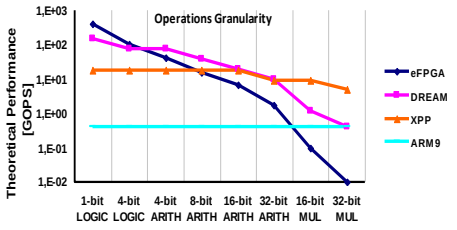


Figure 2.21: *Morpheus performance.*

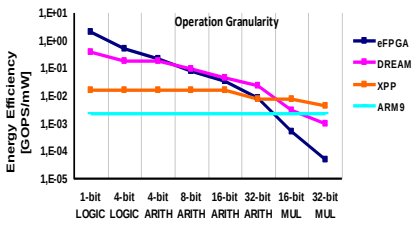
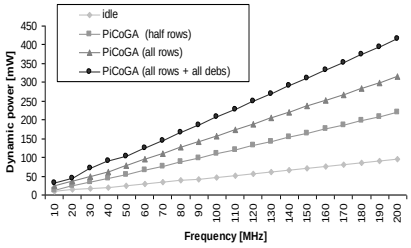


Figure 2.22: *Morpheus energy efficiency.*



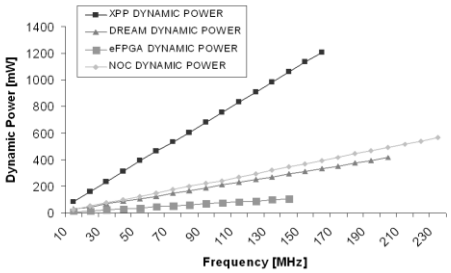


Figure 2.23: *DREAM power consumption.* **Figure 2.24:** *Morpheus component power.*

As a proof of concept, Figure 2.23 shows measurement on the DREAM processor when running in idle mode (RISC processor only), when using half of the available rows fetching data from its internal register, when using all the PiCoGA rows, and when using all 24 rows while accessing all I/O memory banks. These experimental results, which can be extended to the other reconfigurable devices of the system, show significant dependence between the power consumption and the quantity of reconfigurable resources utilized by the devices composing the system.

Figure 2.24 reports dynamic power measurements performed running the power characterization test vectors on the Morpheus prototype. More precisely, the ARM clock island power was measured turning off the clocks of all reconfigurable engines in the system, and programming all DMAs of the system in order to iteratively perform transfers among all the NoC nodes. This number considers power consumed by the ARM, DMAs, the whole NoC, and all the storage elements connected to the NoC. Figures relative to the reconfigurable engines were calculated running the test vectors on the target reconfigurable engines and subtracting from the measured values the idle power consumed by the ARM clock island. In both ARM and reconfigurable engines power calculations the leakage power was considered as an offset subtracted from the measured values. Figure 2.22 show the energy efficiency of the Morpheus platform for the different size and nature of the operations, calculated as the theoretical performance normalized to the power density (mW/MHz) of each reconfigurable engine.

2.6.2 Application-based analysis of the Morpheus platform

This section describes the performance delivered by the Morpheus platform when running a set of application kernels implemented on the Morpheus test chip. The aim of this section is to evaluate the Morpheus capabilities in different application domains analyzing major causes of degradation, with respect to the presented theoretical performance when running real-life applications described in the previous section (Table 2.4).

Occupation of reconfigurable engine resources

The under-utilization of resources is a major cause of overheads in any kind of reconfigurable device. This is usually due to design trade-offs between generality and mapping efficiency, which usually lead to non-optimal matching between device resources and computational patterns of applications. This phenomenon is negligible in applications where inner loops can be parallelized to saturate device resources. In other cases, resource occupation is a major constraint that limits the exploitation of parallelism.

Application	Application field	Target
RGB2YUV	Imaging/Video	2048x1536, 10-bit/pixel, 30 fps
Edge Detection	Imaging	640x480, Grayscale, 30 fps
Binarization	Imaging	640x480, Grayscale, 30 fps
AES/Rijndael	Cryptography	802.16
CRC	Telecom	Ethernet 10/100 Mbps
Motion Estimation	HD Video	2048x1536, 10-bit/pixel, 30 fps
Motion Compens.	HD Video	2048x1536, 10-bit/pixel, 30 fps
Ethernet MAC	Telecom	Ethernet 10/100 Mbps

Table 2.4: *Applications selected for the evaluation of the Morpheus Platform.*

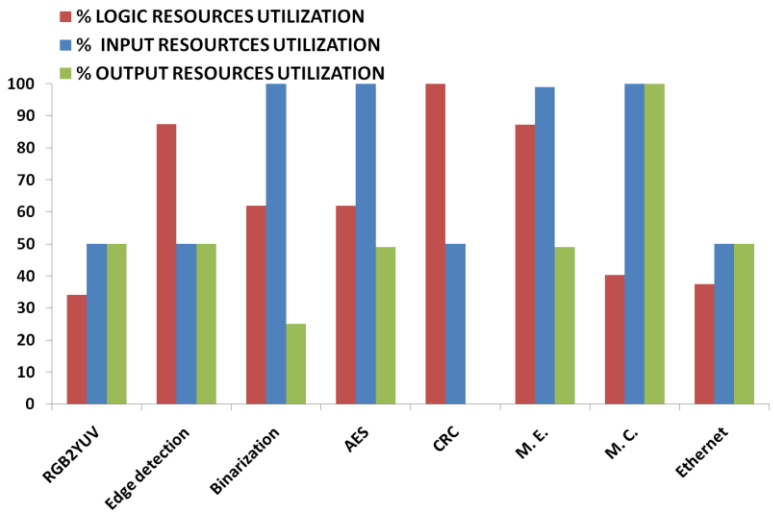


Figure 2.25: *Resources occupation of applications mapped on Morpheus.*

Figure 2.25 shows resource occupation of each application implemented, reported as a percentage of both logic resources and local I/O resources. Results show that CRC employs all 24 PiCoGA rows thanks to the unfolding technique applied to the algorithm. In contrast, since MC is acutely memory-intensive, it saturates the input and output resources of the XPP array, and thus under-utilizes the array-processing elements. Ethernet and RGB2YUV require less than half of the logic resources, while in the other cases utilization of logic resources is more balanced, falling between 60% and 100%. The under-utilization of logic resources is the first factor, which prevents Morpheus to achieve theoretical results. The other factors are reported in the following.

Communication and Memory infrastructure

Considering the Morpheus platform from a system level perspective, we analyzed the overheads caused by the on-chip and off-chip communication infrastructure. For this purpose we run the selected applications initializing their input frames within different levels of the Morpheus memory hierarchy. The experiment was performed considering three different scenarios. In the first one, utilized as reference, input data chunks are initially stored in the reconfigurable engines' local buffers and are output to local buffers after the

elaboration. In the second one, input data chunks are stored in the system on-chip memory, highlighting overheads caused by conflicts on the on-chip network and memory. In the third one, data chunks are stored in the external memory, thus highlighting overheads caused by congestions and latencies introduced by the off-chip memory accesses.
Figure 1.26 shows results of the described scenarios implementations. Data are normalized with respect to the first scenario. The overheads introduced by the on-chip communication network is nearly null, meaning that, is able to sustain the bandwidth of almost all selected applications. In the specific case of the XPP processor, the first two scenarios are necessarily equivalent as they fetch data from FIFOs directly connected to the network on chip. On the other hand, the third column shows that degradation of performance due to access to external memory occurs in most cases. The only applications not matching real-time requirements due to bottlenecks in the off-chip communication infrastructure are ME and MC. In these two cases, the large size of processed images leads to additional swapping between on- and off-chip memory. This swapping breaks the XPP pipeline evolution preventing the device to exploit its full computational power thus leading to a further performance degradation with respect to Figure 2.26.

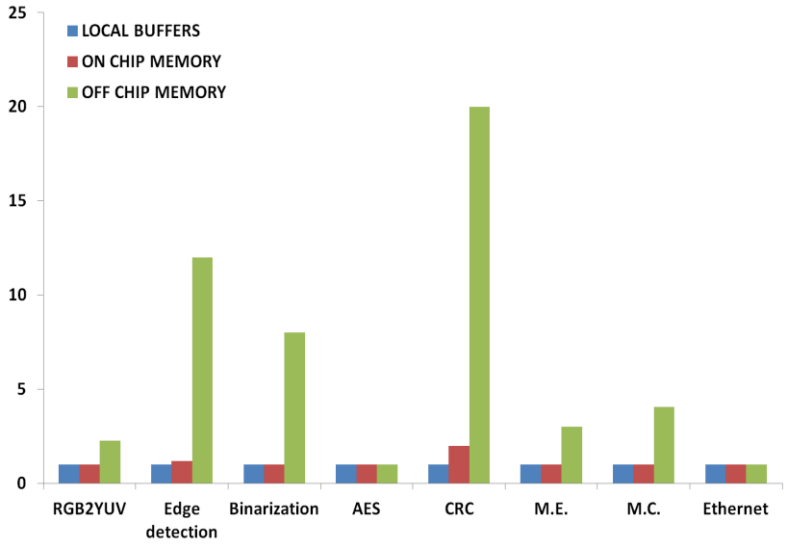


Figure 2.26: *Overhead introduced by on-chip and off-chip communication.*

Application	Off-Chip Mem.	On-Chip Conf. Mem.	Local Conf. Mem.	Context Mem.
RGB2YUV	171491	128652	~1000	n.a.
Edge Detection	23441	8107	441	1
Binarization	44820	14940	n.a.	n.a.
AES	17090	5906	315	1
CRC	34239	7619	714	1
ME	523955	n.a.	~1000	n.a.
MC	312459	n.a.	~1000	n.a.
Ethernet	44820	14940	n.a.	n.a.

*Data reported in Off-Chip Mem. and On-Chip Mem. consider a transfer efficiency of 0.3 Transfers/Cycle assuming an average traffic rate on the configuration bus.

Table 2.5: *Reconfiguration latencies of applications implemented on the Morpheus platform (clock cycles).*

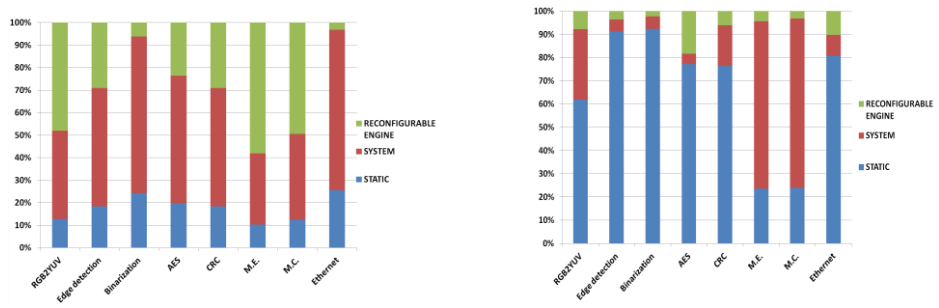
On-the-fly Reconfiguration

When it is not possible to fit a complete kernel into a target reconfigurable engine, or when it is required to time-multiplex more than one application, it became necessary to utilize on-the-fly reconfiguration. In these cases, additional reconfiguration latencies have to be paid in order to re-program reconfigurable engines being involved by this process. In some cases this reconfiguration latency can be hidden by processing data on one reconfigurable engine while the other is being reconfigured, or, in the case of DREAM, utilizing its multi-context capabilities. The reconfiguration latency depends on the reconfigurable engine utilized, the size of the reconfiguration bitstreams for the implemented application, and the configuration memory hierarchy level utilized to store configuration bitstreams. Table 2.5 summarizes reconfiguration times of the selected applications, assuming the configuration bitstream is stored at different levels of the configuration memory hierarchy. Configuration time is the same for all applications implemented on the eFPGA, as the eFPGA bitstream does not depend on the number of resources utilized. On the contrary, the DREAM configuration time is a function of the number of utilized PiCoGA rows, as well as the size of the program and data memories of the RISC processor. Benefits of the

DREAM's multi-context capabilities are exploited in the CRC application. Indeed, the first CRC operation saturates the array resources. Thus, without multi-context support, the second operation that is necessary to accomplish the CRC would require a reconfiguration penalty equal to 714 clock cycles per message. In comparison to the other two reconfigurable engines, the XPP configuration times are significantly larger, which is explained by its large size. Similar to the eFPGA, the local configuration time is identical for all configurations and is estimated to be approximately 1000 clock cycles. The configuration times of the off-chip memory variant show significant differences between the three XPP configurations, which are related to the different sizes of the bitstreams. In combination with the ARM program code, the ME and MC bitstream sizes also prohibited a configuration from on-chip memory as the overall amount of the prototype's on-chip memory has been exceeded. When processing small or medium-sized images, these large bitstreams with their comparatively long configuration times result in a measurable configuration overhead. However, for the targeted class of high-resolution image processing, the configuration overhead can be disregarded.

Frequancy Scaling

In order to demonstrate benefits of frequency scaling within the Morpheus platform, we analyzed its impact on the power consumption when running the presented applications. Most of standards, which implement the described algorithms, do not require the whole Morpheus computational power. In such cases it is possible to tune the frequency of the four clock domains of the platform in order to achieve the lowest possible power consumption for the given application requirement. To perform the analysis we utilized the requirements reported in Table 2.4, and we scaled the frequency of each clock domain (i.e., ARM+NOC, XPP, DREAM, eFPGA) involved in the



(a) (b)

Figure 2.27: *Power breakdown of applications implemented on the Morpheus platform without frequency scaling (a) and with frequency scaling (b).*

	RGB2 YCC	Edge Detection	Binariz.	AES	CRC	ME	MC	Ethernet MAC
XPP freq.	5 MHz	GATED	GATED	GATED	GATED	5 MHz	5 MHz	GATED
DREAM freq.	GATED	5 MHz	GATED	40 MHz	10 MHz	GATED	GATED	GATED
eFPGA freq.	GATED	GATED	10 MHz	GATED	GATED	GATED	GATED	50 MHz
System freq.	40 MHz	5 MHz	5 MHz	5 MHz	20 MHz	250 MHz	250 MHz	10 MHz
Power (no FS)	1835 mW	1278 mW	966 mW	1186 mW	1278 mW	2273 mW	1887 mW	947 mW
Power (FS)	379 mW	258 mW	254 mW	304 mW	307 mW	998 mW	985 mW	291 mW

Table 2.6: *Power consumption of applications implemented on the Morpheus platform.*

computation according to these constraints. Results are reported in Table 2.6. The frequency scaling allows reducing the power consumption by factors that span from 1.9x to 4.9x, depending on the application. It should be noticed that, differently from the other cases, during execution of ME and MC the bottlenecks resides in communications, thus avoiding the down-clocking of the system clock domain which became the major source of power consumption for these applications. Figure 2.27a shows the power breakdown of the applications running on the Morpheus platform. Benefits of frequency scaling are exposed in Figure 2.27b that shows a reduction in the contribution of dynamic power on the overall power consumption from 82% to 34%.

Chapter 3

3 The Manyac Platform

3.1 Overview

Manyac is a modular and customizable multi-core platform aimed at the execution of all those signal processing algorithms whose parallelism can be exploited at thread-level or at data-/instruction-level.
The Manyac platform addresses fast development of multi-core systems for applications requiring high performance and energy efficiency, especially for all those subjects on rapid evolution during the typical life of a product. For this purpose, a high-level design environment allows the user to explore the design space of the platform, customizing its architectural parameters, in order to match the applications specifications. Whenever the exploitation of thread-level parallelism given by the multi-processor approach is not sufficient to match the applications requirements, a second hardware/software development tool enables fast design of custom pipelined hardware accelerators and their automated implementation on configurable areas of the platform. This approach allows one to select the architectural parameters during the design of the platform, and sustain its time life by re-designing or evolving the application specific accelerators implemented on the configurable areas.
From the structural point of view, the Manyac approach exploits regularity at both architectural and layout level in order to address low development and manufacturing costs and time to market. One specific target of the architecture is to focus the effort of users on small silicon areas in order to keep design and verification costs as small as possible. For this reason, as shown in Figure 3.1, the regularity of the platform is achieved at the architectural level by the replication of two basic entities: the *IO tile*, and the *computational tile*. The implementation of both entities is based on design-

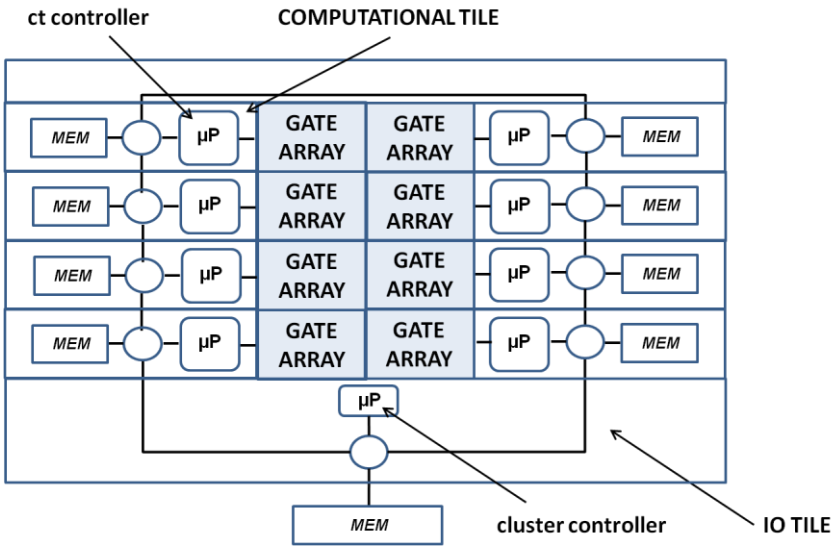


Figure 3.1: *Overview of the Manyac architecture and physical structure.*

time configurable, parametric IP components, whose customization within the platform do not require to the final user specific knowledge of hardware description languages. On the other hand, the application specific areas, designed starting from a C-level description language, are considered from both architectural and structural point of view as "pluggable", stand-alone components, allowing their customization regardless of the specific architecture implementation. The aim of the flow is the generation of a hard macro IP, and its integration in a more complex System-On-Chip or its implementation as a stand-alone component.

A specific peculiarity of the Manyac platform is that of supporting three different kinds of customization technologies for the implementation of the application-specific accelerators, which represent different trade-offs between the performance of the platform and its flexibility. Those are a run-time configurable gate array, a via-programmable gate-array and a metal-programmable gate array.

3.2 Computational Model

The Manyac computational model leads to the exploitation of both thread-level and instruction-/data- level parallelism from a wide class of signal processing applications. For this reason the Manyac platform supports two different execution models: a data parallel model and a task parallel model (Figure 3.2).

The data-parallel execution model can be exploited in all those application whose parallelism can be explicitly described utilizing the OpenCL data-parallel programming model [27]. When a data parallel kernel is submitted for execution, an index space is defined. Each instance of the kernel executes a point in this index space, which is called *work-item*. Each *work-item* executes the same code, but the specific execution pathway through the code and the processed data can vary per *work-item*. *Work-items* are organized into *work-groups*, which provide a coarse-grained decomposition of the index space. Synchronization between *work-items* in a single *work-group* is done using a *work-group barrier*. All the *work-items* of a *work-group* must execute the barrier before any are allowed to continue execution beyond the barrier. Work-items executing a kernel can access three separate memory regions. The *global memory* permits read/write access to all *work-items* in all *work-groups*. The *local memory* is utilized to allocate variables that are shared by all *work-items* within a *work-group*. The *private memory* defines the region of memory private to a *work-item*.

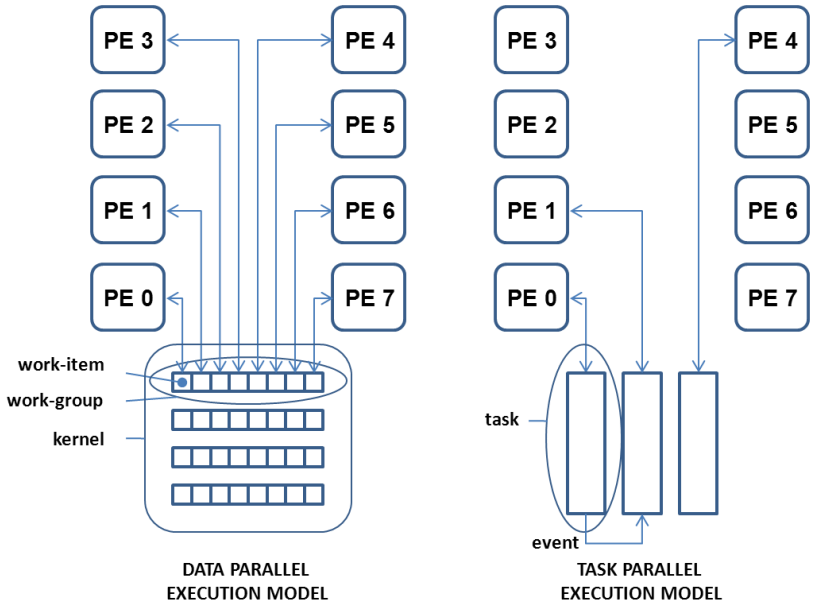


Figure 3.2: *Data-parallel and task-parallel execution models.*

Whenever applications or portions of applications do not show this kind of explicit parallelism, a task parallel model can be utilized for execution. Within the task parallel execution model, a single instance of a kernel is executed independently of any index space. In this case, tasks, whose allocation is selected at compilation time, execute on different computational tiles, while the consistency of processed data is guaranteed by specific synchronization events handled by software.

Both data-parallel kernels and tasks can handle the execution of hardware functions in order to take advantage of data and instruction level parallelism provided by the application specific accelerators. For many applications, especially when dealing with hardware accelerators, memory transfers utilize a relevant portion of the whole computation time. In order to hide memory transfer latencies it is often desirable to overlap data transfer and computation phases in a pipelined stream. In this scenario, a hardware/software partitioning of kernels can be utilized in order to balance as much as possible the area of hardware accelerators with the throughput achievable by this pipeline. This is achieved, on the Manyac platform, utilizing the index space decomposition provided by the OpenCL programming model as shown in Figure 3.3. *Work-items* execute concurrently on different computational tiles,

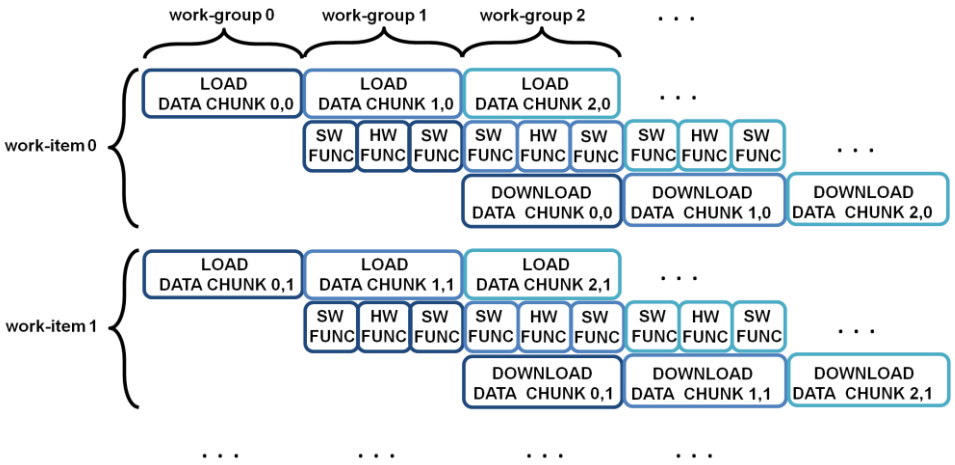


Figure 3.3: *Parallel execution of work-items and pipelined execution of work-groups within data parallel kernels.*

while *work-groups* execute in a pipeline fashion, whose computation stages are achieved with throughput-balanced sequences of software and hardware accelerated functions.

3.3 Architecture

As shown in Figure 3.1, the Manyac architecture is composed of a scalable cluster of computational elements called *computational tiles* connected together by an on-cluster communication interconnect sharing data through a multi-bank, distributed local memory.

3.3.1 System level architecture

At cluster level, a RISC processor implemented within the IO tile is responsible for executing sequential code of applications, configuring and launching parallel and hardware-accelerated kernels, as well as handling synchronization among cores.

The global synchronization mechanism is achieved through a set of distributed, memory mapped synchronization registers. When parallel threads running on *CT controllers* reach a synchronization barrier, they set a bit of their local synchronization register and stop their execution waiting for an acknowledgment by the *cluster controller*. The *cluster controller* collects all synchronization requests, and acknowledges the execution to the *CT controllers* as soon as all threads within a work-item reach the barrier. A dedicated hardware mechanism was realized to improve the synchronization during the execution of data parallel kernels. On the other hand, custom synchronization mechanisms can be implemented by software utilizing memory mapped registers accessible by the *cluster controller* in order to improve flexibility.

The communication infrastructure of the platform is based on a ring topology state of the art Network-on-Chip (NoC): the STNoC [61]. The regular geometries of the chosen topology allow to hard-wire each node of the NoC within a computational tile accordingly with the Manyac implementation philosophy. A peculiar feature of the STNoC exploited in the architecture is that of programmable addressing spaces and routing paths. Depending on the number of computational tiles implemented on the architecture, this feature allows configuring routing paths and addressing spaces accordingly, preserving the regularity of the computational tile component down to layout, as well as the scalability of the platform.

According with the Manyac computation model, each thread running on a *CT controller* must be able to access its *private memory* space, a *local memory* space, and a *global memory* space. The private memory space is implemented by the tightly coupled memories of the *CT controllers*, as well as by the computational tile's local buffers. The *local memory* hierarchy level is implemented by a set of multi-bank single-port memories connected to the NoC within each computational tile. Finally, the *global memory* space is implemented at system level outside of the multi-core cluster

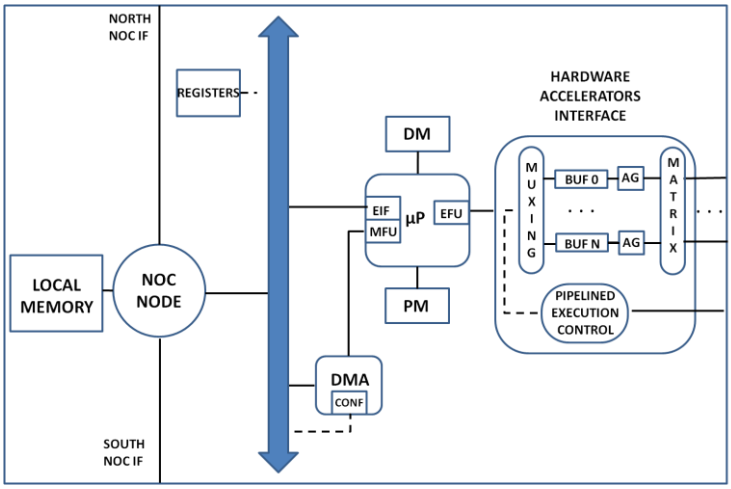


Figure 3.4: *Computational tile architecture.*

3.3.2 Computational Tile Architecture

The computational tile architecture is shown in Figure 3.4 Each computational tile provides the software programmability by the presence of a GP processor, while exploiting benefits of the application specific customization through a set of programmable hardware facilities.

In order to take advantage of the high parallelism typical of hardware accelerators, it is necessary to provide a data communication mechanism capable to sustain the available computation bandwidth. For this reason, each computational tile is equipped with a set of buffers that feed the hardware accelerators with highly-parallel data. In order to minimize the overheads caused by memory transfers, the buffers can be concurrently accessed at both system side and hardware accelerators side, thus allowing a user to create a pipelined exchange of data-chunks from/to hardware accelerators by overlapping upload, download and computation phases as shown in Figure 3.3.

A second key factor to consider when handling with hardwired acceleration unit, especially when implemented on structured ASIC platforms, is that of flexibility. In order to achieve high computational densities, one key target of

the proposed methodology is to implement computation patterns of kernels within the hardware accelerators, while leaving the software programmable components to handle data feed, addressing and specialization of hardware functions. In order to improve the hardware/software cooperation, a specific multi-ported register file was included as local data repository, whose specific purpose is that of exchange data between successive issues of hardware accelerated functions and software instructions executed on the processor. As hardware accelerators feature function-specific latencies, a hardware register locking logic was added to sustain access consistency, generating stalls to preserve the correct program flow. Moreover, the programmable address generators (AG) connected to all local buffers provide two-dimensional and circular addressing capabilities typical of many signal processing applications. Finally, a programmable matrix coupled with a cache, able to store up to 64 configurations, allows connecting all the available buffers or registers to each input or output of the hardware accelerators. A specific added value of the programmable matrix is that of implementing complex addressing patterns not provided by the AGs, such as transpositions or zig-zag scans, by calling successive hardware accelerated functions with different matrix configurations.
On the system side, local buffers can be accessed by both the processor, being mapped on its private addressing space, and a programmable direct memory access controller (DMA) providing asynchronous, bi-dimensional memory accesses to the shared and global memory space.

3.4 Configurable Accelerators

As described in previous sections, each computational tile is equipped with a set of customizable hardware accelerators, which specialize the platform, whose flexibility depends on the specific kind of customization adopted. The configuration technology used as the silicon platform for the implementation of application specific accelerators are a run-time programmable gate array, a via-programmable gate array, and a metal-programmable gate array.

3.4.1 Architecture

In order to improve the flexibility and portability of the approach the gate arrays coupled with each computational tile were specifically designed to be plugged to the system as stand-alone components. For this purpose, they integrate the datapath, which implements the computations, a dedicated interface toward the system, which handles control, and synchronization toward the system and a control unit, which schedules the execution of the pipelined dataflow.

From the architectural point of view, the datapaths are composed of 24 rows, each implementing a possible stage of a customized pipeline. The run-time programmable and via programmable gate array feature a fixed template structure composed of an array of Reconfigurable Logic Cells, as shown in Figure 3.5. Each row is composed of 16 RLCs and a configurable horizontal interconnect channel. Each RLC includes a 4-bit ALU, that allows to efficiently implement 4-bitwise arithmetic/logic operations, and a 64-bit look-up table in order to handle small hash-tables and irregular operations hardly describable in C and that traditionally benefit from bit-level synthesis. Each RLC is capable of holding an internal state (e.g. the result of an accumulation), and provides fast carry chain propagation through a datapath row. On the other hand, in the metal programmable gate array, the equivalent logic and arithmetic functionalities are implemented as a set of VHDL functions automatically instantiated by customization the flow, which implement a separate datapath for each hardware accelerated operation (Figure 3.6). This VHDL netlist is further synthesized on a library of metal-programmable cells.

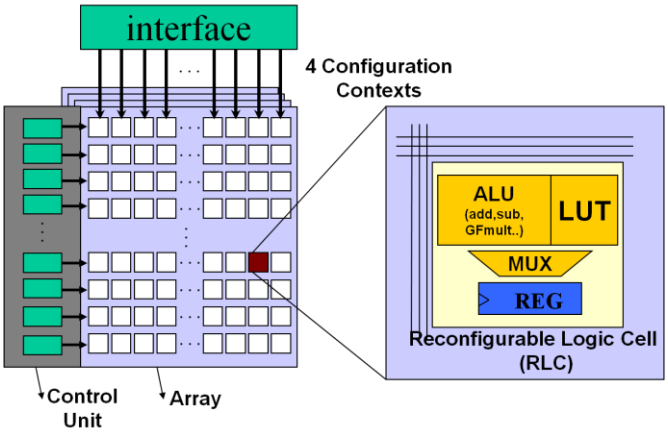


Figure 3.5: *Simplified view of the run-time programmable and via-programmable gate array architectures.*

In order to improve the working frequency, the gate array supports the direct implementation of Pipelined Data-Flow Graphs (PDFGs), avoiding the signals which control the pipelined execution of the datapath to be shared with the computations. The main role of the control unit is to handle the pipeline evolution, while eliminating unnecessary dynamic power consumption. For this reason it handles the execution of accelerated operations mapped on the datapath, activating each row according to the scheduled execution flow and gating the clock of all those rows not involved in the computation.

In order to enable the easy integration and efficient management of the pipelined datapath, it is equipped with a dedicated control interface. The main role of the interface is to translate and synchronize the subsystem signals addressing the pipelined datapath. Utilizing this approach just a few system-side control signals are necessary to handle the configuration and execution toward the datapath, while all timetables and relative stalls are handled internally on the interface. This way, it decreases the external systems' computational load necessary to manage timings and exceptions related to datapath signals.

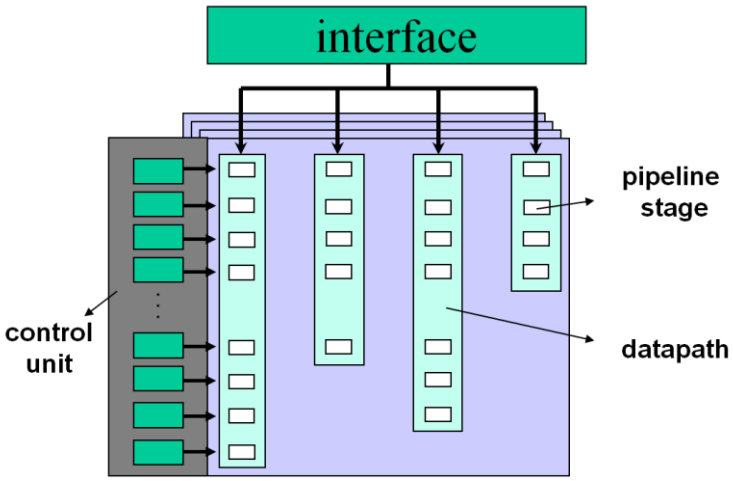


Figure 3.6: *Simplified view of the metal programmable gate array(a).*

3.4.2 Implementation and Customization Strategies

From the implementation point of view, most peculiar differences between the three approaches reside in their physical structure and customization philosophy, being achieved by configuration of SRAM cells, configuration of input signals utilizing the VIA4 layer, and synthesis and place & route over a library metal-programmable gate array cells. The layout structure of both run-time and via programmable gate arrays macro is fixed, this meaning that the placement of cells and the routing paths are frozen and they can't be changed for customization. All possible functionalities are already available inside the RLC, and the required behavior can be obtained by either driving a multiplexer or programming a LUT. In both cases, the configuration can therefore be achieved by forcing 0/1 to specific input signals. An important aspect to consider is that of the approaches utilized to realize the skeleton template of the realized gate arrays. Each approach exploits a different trade-off between aspects related with fixed costs (Design, Verification) which have an impact on TTM, and aspects connected with manufacturing costs (area) and performance (application throughput, power consumption). In order to make effective performance of the more flexible solutions (run-time and via configurable), advanced implementation approaches were utilized. For this reason, within the run-time programmable datapath, the implementation of the

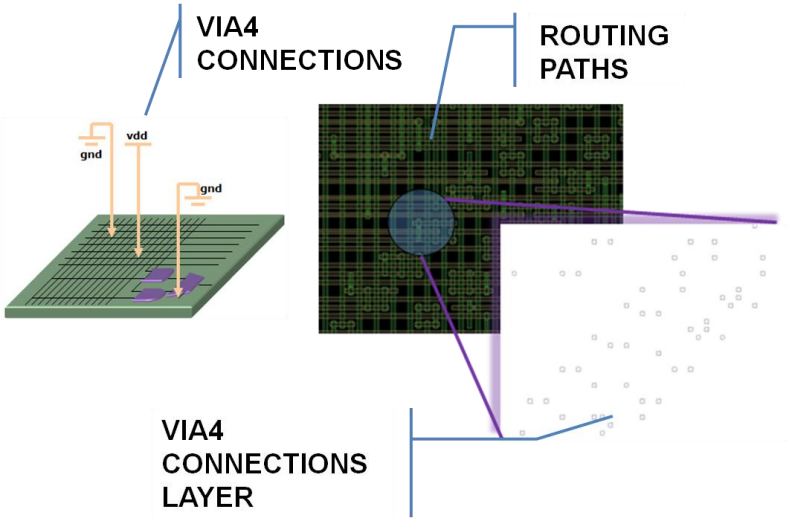


Figure 3.7: *Via programmable datapath customization strategy.*

RLCs and of the control unit was realized utilizing a full-custom design approach. Contrarily, the via-programmable gate array RLCs were realized by synthesizing their architecture over a standard cell library, while place and route was performed manually. In both cases, the datapath implementations take advantage of the regularity of the approach as each RLC is replicated over the 24x16 array, and instances of the control unit replicated per row.

Considering the customization, the run-time configurable gate array, is programmed through a set of configuration SRAM cells which allow to specifying the functionalities implemented by each RLC and the related configurable interconnections. Within the via programmable gate array, the layout is arranged in advance in order to draw configuration input signals in M3 so run under both ground and power M4 nets, thus obtaining the customization by placing the VIA4 on specific places. Similarly, for routing resources all possible configurable paths are already available on the skeleton layout, designed in M3/M4 and disconnected by default. Then the enabling of a specific path only requires the placement of a single VIA4 in the inserction of lines as shown in Figure 3.7.

On the contrary, the approach utilized for the metal programmable gate array is quite different, leading to achieve regularity at the silicon level rather than at architectural/circuit level. The customization process starts from a high level model of the macro that could be afterwards mapped on a configurable

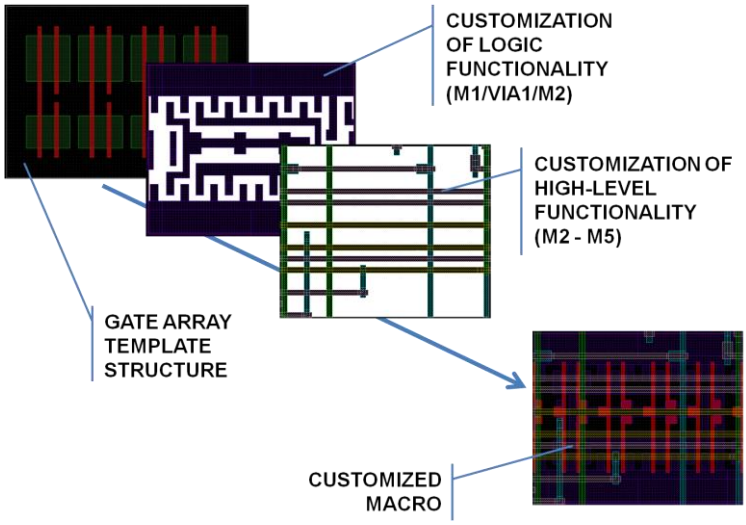


Figure 3.8: *Metal programmable gate array customization strategy.*

library based on a gate-array structure. This model is automatically built using VHDL language, starting from a top structural block and then instantiating behavioural sub-blocks which link some base functions gathered in a separate library. These base functions represent atomic operators such as adder/comparator/multiplier that can be brought back to the available operators inside the reconfigurable and via-programmable gate arrays operational units. Because of the pipelined structure of the mapped architecture, a separate library contains the VHDL model of the control unit. Routing is not represented by VHDL blocks, but it is transposed by logically connecting correct blocks together via specific signals. Thus, we can consider customization as achieved at two levels as shown in Figure 3.8. On the first level logic the logic functionality of each cell is achieved by utilizing only M1, VIA1 and M2, which realize the metal programmable library. On the second step, the implementation of the datapath entities, which realize the equivalent RLC operators, is achieved utilizing the remaining metal layers (M2 to M5). For this reason, the overheads of the metal programmable gate array with respect to the standard cell based approach only reside in the regular implementation of the metal programmable cells.

3.5 Implementation Flow

The Manyac framework consists of an integrated hardware/software environment that assists the user in the implementation of signal processing applications on the described multi-core platform. A Global view of the Manyac framework is shown in Figure 3.9. The framework is logically divided in four layers tightly integrated to each other, allowing the final user to explore, for a given application, many platform implementations at different levels of abstraction. Those are the software layer, a transaction level modeling (TLM) layer, a register transfer level (RTL) layer and a physical layer.

Within the software layer, a dedicated compilation flow allows to partition the application extracting the host code that runs on the cluster controller and kernels running on each instance of the CT controller, according with the OpenCL programming model. The compiler extracts from the source code the definitions of the host functions (defined as standard C functions) and kernels (defined with the _kernel function qualifiers). Moreover, it allocates variables defined within kernels on the proper memory region, depending on the address space qualifier specified during their declaration (_global for global variables, _local for local variables, _private for private variables). Allocation of work-items and work groups, even if explicitly assigned during the definition of kernel functions calls, is handled at run-time, as well as the management of memory transfers and hardware accelerated functions. The TLM model of the platform enables one to explore applications described with OpenCL from a high-level standpoint, and to select the architectural parameters of the platform, which are reported in Table 3.1, in order to match the applications constraints with and without hardware acceleration. With respect to a cycle accurate model, which describes all low-level details of the communication protocol, the TLM model achieves higher performance in terms of instructions per second, while it is still able to highlight congestion situations due to multiple concurrent accesses on the same bus or memory. Contrarily, the RTL model is used for cycle accurate simulations, for the validation of the application implementations and as an entry point for the physical implementation process. Once the architectural parameters of a

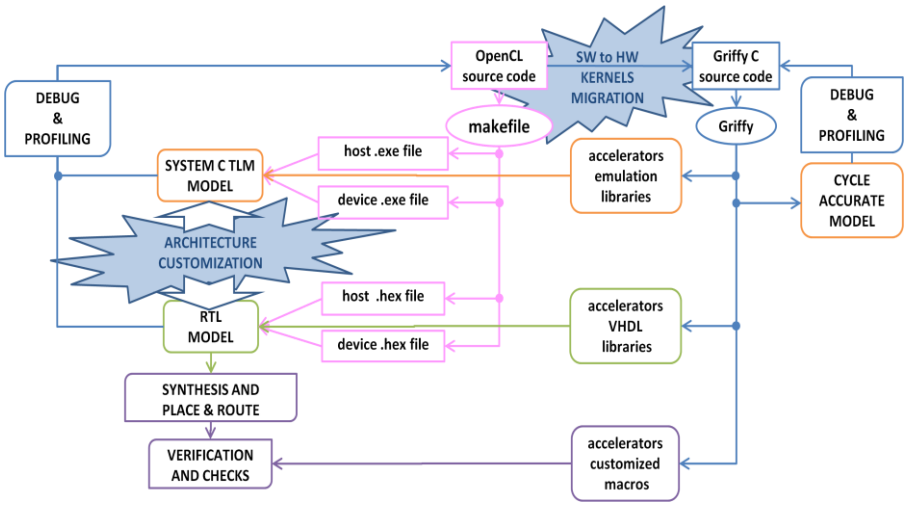


Figure 3.9: *Overview of the Manyac design flow.*

platform's implementation are selected, two separate place & route flows are performed separately for the IOT and CT. Many instances of the CT are further regularly replicated over the cluster and are merged together with the IOT, leading to the implementation of a customized multi-core hard macro IP.

3.5.1 The Griffy environment

The main target of the Griffy environment is the design space exploration and implementation of the application-specific hardware accelerators which form the customization of each computational tile. This environment was initially developed to configure a run-time programmable gate array [65] and recently extended to configure the via-programmable gate array and lightly-pipelined synthesizable RTL code.

The language used to implement pipelined hardware accelerators on the gate array is called Griffy-C. Griffy-C is based on a restricted subset of ANSI C syntax enhanced with some extensions to handle variable resizing and register allocation. Differences with other approaches reside primarily in the fact that

	Supported Values
Number of computational tiles	2,4,6,8
Network on chip data width	32,64,128,256
Computational tile local data width	32, 64, 128, 256
CT local memory size	2K,4K,8K,16K,32K,
CT program and data memory size	2K,4K,8K,16K,32K
Number of computational tile PGA	2,4,8,16
Number of computational tile PGA	2,4,8,16,24
PGA buffers and registers width	8, 16, 32

** Number of buffers + Number of Register cannot exceed 32*

Table 3.1: *Manyac Platform Main Configuration Parameters.*

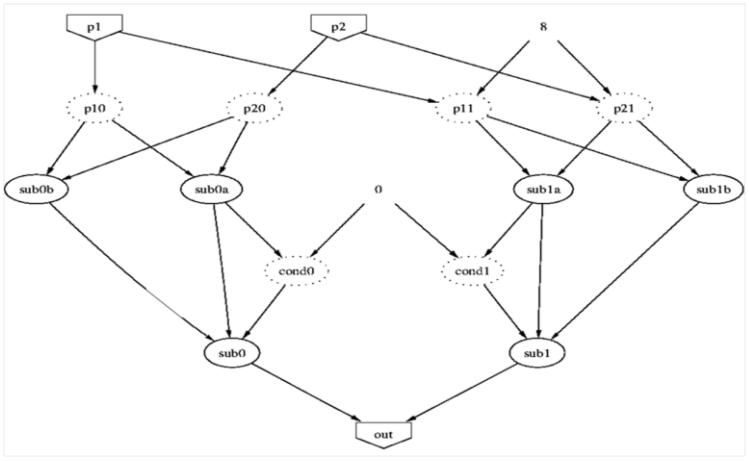


Figure 3.10: *Example of PDFG implemented utilizing the Griffy environment.*

Griffy is aimed at the extraction of a Pipelined Data Flow Graphs (PDFGs) from standard C to be mapped over a datapath pipelined by explicit stage enable signals. Griffy-C is used as a friendly format in order to configure customizable macros using hand-written behavioral descriptions of DFGs. Restrictions essentially refer to supported operators (only operators that are significant and can benefit from hardware implementation are supported) and semantic rules introduced to simplify the mapping into the datapath. Three basic hypotheses are assumed. (1) No control flow statements are supported, as the embedded control unit manages the pipeline evolution (DFG-based description). Only conditional assignments are supported and are implemented on standard multiplexers. (2) Each variable is assigned only once, avoiding hardware connection ambiguity (single assignment). (3) Only single operator expressions are allowed (manual dismantling). Besides standard C operators, special intrinsic functions are provided in the Griffy-C environment in order to allow the user to instantiate non-standard operations, such as for example the multiplier module.

The Griffy environment permits a graphical visualization of the realized PDFGs, which highlights data dependencies between nodes and the pipeline stages computed by the compiler as well as the parameters which describes the timing behavior of the developed accelerator: latency and issue delay. An example of PDFG dump of an adder tree is given in Figure 3.10.

Moreover, the environment provides a cycle-accurate simulator which enable easy exploration of the hardware/software co-design space. The simulator models the sub-system composed of the processor, the subsystem which implements the interface toward the datapath (buffers, matrix interconnect, and address generators) and the datapath, too. This way it guarantees the consistency of both functionality and cycle count with the hardware of the realized customizations in a user-friendly hardware/software co-design environment.

The customized datapaths designed and validated utilizing the cycle accurate simulator can be further exported as a functional emulation library. Differently to the stand-alone cycle accurate model, the key target of the functional emulation libraries is that of integration with the system-level TLM simulator. Although this model does not guarantee the cycle-accurate count, it provides the highest simulation performance (i.e., instructions/second) as well as guarantees the consistency of provided results. For these reasons, the functional emulation libraries form, together with the system-level simulation model, an essential step for both validation and further exploration of the hardware/software design space at a higher level of abstraction.

The implementation of design-dependent customizations for the provided silicon structures are performed by multi-target design flows, which integrate custom tools specifically designed to generate the a general-purpose bitstream for the configuration of the run-time and via programmable gate arrays, and the equivalent VHDL netlist for the implementation over the metal programmable gate array. The "general-purpose" description of the functionality is further processed by three design-specific tools aimed at the generation of the configuration bitstream for the run-time configurable gate array, the VIA4 customization layer for the via-programmable gate array and the metal-programmable gate array layout. For what concerns the via-programmable and the metal-programmable flow, a specific target is that of providing to the user a way to generate a customized, fully-verified layout, ready for integration as hard IP macros into standard digital design flow. For this reason, the proposed flows include a mix of design-specific tools and commercial tools, integrated within a .csh environment, which allow the user to generate the customized macros by executing just one command. Given the structure of the via-programmable gate array, most of verification are already

performed on the skeleton template of the gate array. For this reason, the overall customization and verification process is very fast and takes less than 24 hours on a standard quad-core server. On the other hand, the customization flow for metal-programmable gate array performs synthesis, place&route over the metal-programmable library. For this reason, its execution time depends on the complexity of the implemented accelerators and cannot be estimated with accuracy.

3.6 Mapping of Applications on the Manyac Platform

The aim of the application mapping on the Manyac platform is that of exploiting thread/task level parallelism through an appropriate mapping on the computing elements of the platform and data/instruction level parallelism by implementing configurable hardware accelerators.
Considering the high level partitioning of an application, the data parallel model best fits all those applications executing the same code over large data sets, such as image processing. On the other hand, the task parallel model better fits all those computations characterized by the execution of many successive kernels which process relatively small amount of data. In other cases, such as video processing both models can be utilized, and an accurate analysis of the application parameters can be helpful whenever data-locality or parallelism is the best way to match the application computational patterns.
The implementation of applications on the Manyac platform leads to the exploitation of the best trade-off between two leading factors. The first one is represented by the balancing between the amount of configurable area utilized for the implementation of the accelerators and the required performance. The second is comprised of the architectural parameters of the platform. As we will see in the course of this section, the more the hardware accelerator power is exploited, the more performance becomes sensitive to the architectural parameters of the platform, such as the amount of local memory, or the width of the network-on-chip. In the following, a series of application implementations will be explained highlighting the trade-offs discussed.

3.6.1 Implementation of pipelined accelerators

The design of application specific accelerator is often a critical task that aims at the exploitation of data-level parallelism and instruction-level parallelism of the target applications. Data-level parallelism is usually exploited in all those kernels whose code is identically executed for many sets of data, utilizing, in the easiest case the Single Instruction Multiple Data (SIMD) execution. In other cases, such parallelism is not explicitly described in the application kernels, but mathematical transformations, such as the look-ahead technique can be applied in order to exploit the data-level parallelism. In both cases, the data-level parallelism is exploited replicating the hardware resources necessary to accomplish the computation by a specific *unfolding* level, which represents the number of parallel instances of the application-specific basic unit. Contrarily, the exploitation of instruction-level parallelism leads to an accurate analysis of data dependencies among instructions executed within a kernel. The instruction level parallelism of an application can be efficiently described utilizing the DFG formalization. DFGs can either be pipelined (PDFG) as those generated by the Griffy design flow or not. In the second case, the time required to accomplish the computation of a DFG is given by the sum of the computation time of all the operators present on the longest DFG path. This implies that the working frequency of the accelerator is limited by the overall depth of the DFGs implementing the application-specific accelerators causing a low computation throughput. When dealing with PDFG the throughput of the computation depends on the longest pipeline stage within the graph, generally leading to higher computation throughputs. On the other hand, PDFGs introduce other parameters that can affect the performance of the overall computation, if not appropriately taken into account during the design of the accelerators: the *latency* and the *issue delay*. The latency represents the number of cycles which elapse between the accelerated function call and the first output data is available, and depends on the depth of the PDFG pipeline. The issue delay represents the minimum number of clock cycles between two successive hardware accelerated functions fetch, and it usually depends on the balance of the PDFG. Although the issue delay of PDFGs can be usually reduced to one, by inserting retiming stages (i.e. pipeline registers) on the graph, the latency remains as an

irremovable constraints of accelerators implemented with the Griffy flow, as the DFG pipelining is automatically extracted from the source code describing the graph. When the granularity of the pipelined accelerators is sufficiently large, and the pipelining structure of the accelerators is exploited by processing a sufficiently large data chunk, the latency becomes negligible with respect to the throughput of the computation. Coarser accelerators imply a larger area, and less flexibility, as the functionalities hardwired within the accelerators can only be utilized by a few kernels. Contrarily, reducing the granularity of the accelerators increases their flexibility, as the same micro-kernel can be utilized in more processing steps, for example alternating hardware-accelerated and software functions within a loop. In this case, the latency of the accelerator could become a limiting factor in the overall application throughput, requiring to unfold the loop to avoid stalls. The loop unfolding is a technique commonly used to hide the latencies of multi-cycle functions, but unrolling loops causes the growth of program memory requirements. The implementation of pipelined hardware accelerators plays on delicate trade-offs among performance, the amount of customizable area, and the amount of program memory required to accomplish a computation. Next section provides a quantitative example of this trade-off through the implementation of significant kernels extracted from the H.264 standard.

3.6.2 Accelerator implementation examples

H.264/AVC macroblock residual transform and quantization

The transform and quantization of the H.264/AVC process the residual data coming from the difference of reference images before the entropy coding performed utilizing CAVLC or CABAC encoder, depending on the utilized standard profile. Each macroblock is organized as one 4x4 matrix of 4x4 blocks containing the luminance residual data (*Y*) and two 2x2 arrays of 4x4 blocks containing the chrominance data (*Cb* and *Cr*, respectively). A 4×4 integer transform is applied to the residual data from either intra or inter prediction procedures. If the macroblock is encoded utilizing the 16x16 intra prediction the DC components of the luminance and chrominance blocks are

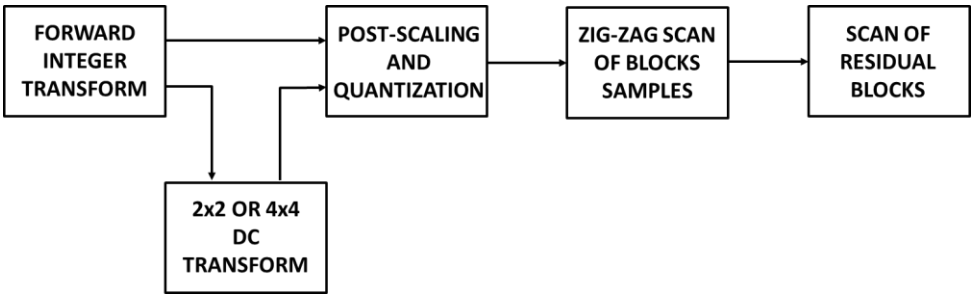


Figure 3.11: *Flow diagram of residual data transform and quantization in a H.264/AVC encoder.*

transformed again utilizing a 4x4 or 2x2 Hadamard transform, respectively. The transformed samples within each block are further quantized and *zig-zagged*. The quantized blocks are finally re-ordered and transmitted to the entropy encoder. The H.264/AVC encoder uses three different transforms. The forward 4×4 integer transform is performed for all macroblock modes on the 4x4 blocks. The integer transform first operates on each 4×4 block X and produces a 4×4 block Y as follows:

$$Y = C_i x C_i^T$$

Where:

$$C_i = \begin{bmatrix} 1 & 1 & 1 & 0{,}5 \\ 1 & 0{,}5 & -1 & -1 \\ 1 & -0{,}5 & -1 & 1 \\ 1 & -1 & 1 & -0{,}5 \end{bmatrix}$$

The DC coefficients of 4x4 luminance blocks are further transformed utilizing a 4x4 Hadamard transform:

$$Y = H_f x H_f^T$$

where:

$$H_f = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix}$$

The DC coefficients of all chrominance blocks, are transformed utilizing the 2x2 Hadamard transform:

$$Y = H_t x H_t^T$$

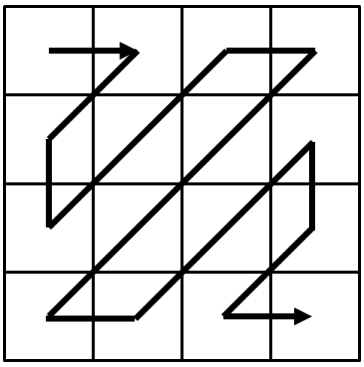


Figure 3.12: *Zigzag scan of blocks in H264/AVC.*

where:

$$H_t = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The H.264/AVC standard adopts two different quantization procedures for residual data from 4 × 4 integer transform and DC coefficients from 4 × 4 or 2 × 2 Hadamard transform. Each 4 × 4 block *Y* can is individually quantized as follows:

$$Z_{ij} = round\ (Y_{ij} \cdot \frac{PF_{ij}}{Q_{step}})$$

where Yij is a coefficient of the transform described above, Qstep is a quantization step size, Zij is a quantized coefficient, and PFij is a scaling factor from the transform stage. In H.264/AVC, 52 Qsteps are stored in a table indexed by a quantization parameter (QP) (0–51). In order to avoid division operations, the above equation can be simplified as follows

$$Z_{ij} = round\ \left(Y_{ij} \cdot \frac{MF}{2^{Q_{bits}}}\right)$$

where

$$\frac{PF_{ij}}{Q_{step}} = \frac{MF}{2^{Q_{bits}}}$$

And

$$Q_{bits} = 15 + floor\left(\frac{Q_p}{6}\right)$$

The above equations can be further simplified in integer arithmetic as follows:

$$|Z_{ij}| = \left(Y_{ij} \cdot MF_{ij} + f\right) \gg qbits$$

$$sign(Z_{ij}) = sign(Y_{ij})$$

A *zig-zag* for the 2x2 or 4x4 is further performed for each blocks within a macroblock, together with the checks of blocks featuring all samples equal to zero. Figure 3.12 provide an example of zig-zag scan for 4x4 blocks. All the blocks within a macroblock are finally transmitted to the entropy encoder according with the ordering described in Figure 3.13.
The implementation of the H264/AVC transform and quantization on the Manyac platform allow to describe trade-offs among area of accelerators, performance and flexibility exploited utilizing different implementation strategies.

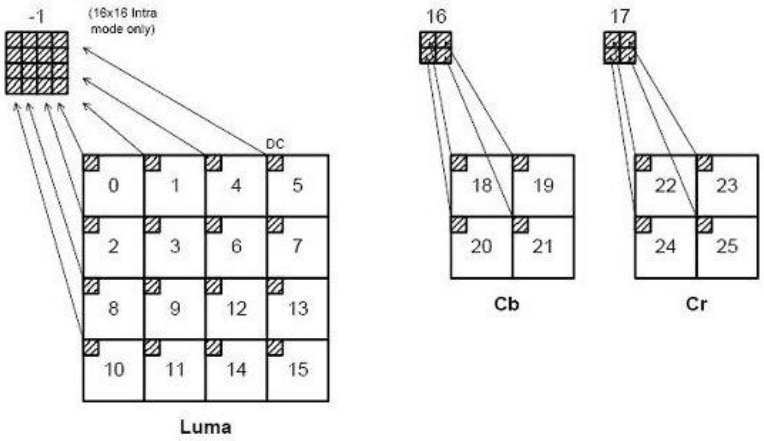


Figure 3.13: *Scanning order of residual blocks within a macroblock.*

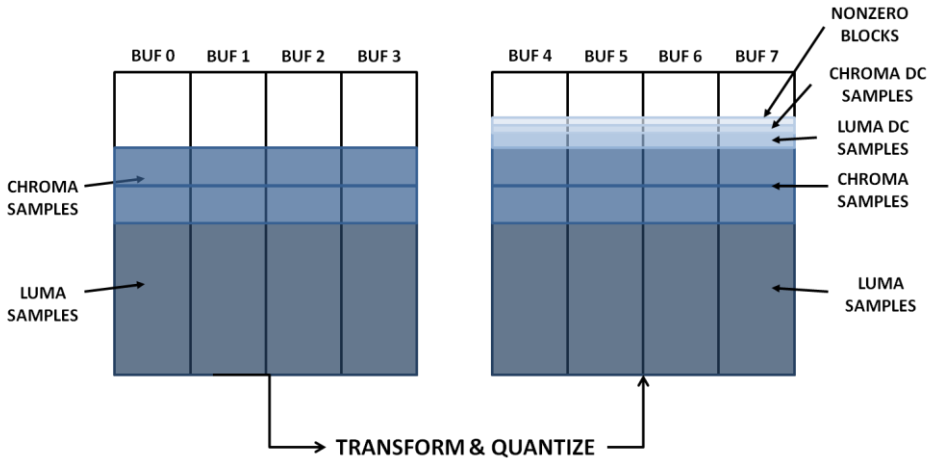


Figure 3.14: *Scanning order of residual blocks within a macroblock.*

As we saw in the previously in this section, the H264 utilizes three forward transforms: the forward integer transform for the 4x4 residual blocks, the Hadamard transform for the 4x4 luminance DC residual blocks and the Hadamard transform for the 2x2 chrominance residual blocks. The memory organization of the residual blocks coming from the previous H.264 step (i.e. calculation of residual between the reference and the estimated macroblock) is structured as an array of 16 4x4 luminance residual blocks, and two arrays of 4 4x4 chrominance residual blocks. Considering the computational tile configuration, the instruction-level parallelism can be massively exploited in this application by processing each row of the 4x4 residual blocks at once. The memory organization of each macroblock within the computational tile buffers is reported in Figure 3.14. The size of samples is 16 bits, then the optimal width and number of buffers results in 8x16bits, partitioned as 4 input buffers (BUF0, BUF1, BUF2, BUF3) and 4 output buffers (BUF4, BUF5, BUF6, BUF7).

The first approach for implementing hardware accelerators for the H264/AVC transform kernels consists of designing a custom pipelined hardware accelerator for each one of the described transforms. For each transform, the accelerator performs a unidimensional transform applied to each row, the transposition of the intermediate matrix, and a second unidimensional transform applied to each row of the transposed matrix. The number of

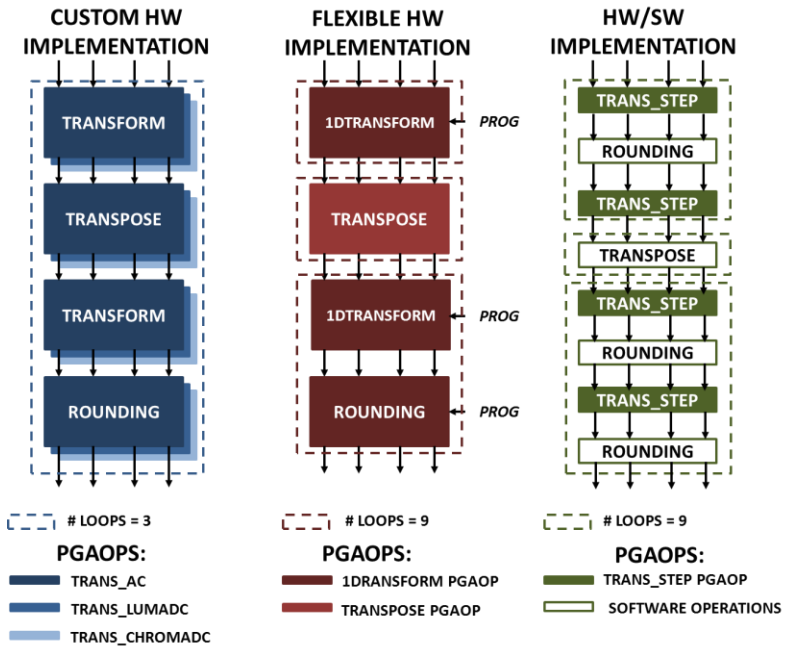


Figure 3.15: *Implementation strategies for the hardware accelerators of the H264/AVC transform.*

accelerated functions included into each computation loop is equal to the number of rows to process for each component. The block diagram on the left of Figure 3.15 shows the described approach for the implementation of the accelerator for the forward integer transform. Utilizing the proposed approach, the overall H264 transform computation can be achieved in three steps. The first processing step consists of the transform of the luma and chrominance blocks resulting in the transformed blocks stored in the output buffers according to the same data organization described for the input buffers. The second processing step transforms the DC components of the transformed luminance blocks. To complete this process, the 16 DC components of the luminance blocks need to be extracted from the output buffers and temporary stored in the register file. This operation can be achieved by programming the address generators with a vectorized addressing pattern, utilizing 16 matrix configurations in order to fill the register file. The extracted dc block is further processed by the luminance DC accelerator and stored on the output buffer. Utilizing the same approach, the DC components of the chrominance block are finally extracted and stored on the output buffer.

IMPLEMENTATION		Software	Hardware/ software	Flexible hardware	Custom hardware
NUMBER OF CLOCK CYCLES	*AC*	4606	1207	439	164
	LUMA DC	305	182	62	77
	CHROMA DC	47	48	49	58
	OVERALL	4958	1437	550	299
PMEMORY BYTES		1216	5408	2104	1916
DMEM BYTES		832	32	36	44
BUFFERS BYTES		0	1632	1632	1632
ACCELERATORS KGATES		0	7	29	44

Table 3.2: *Implementation of the H264 transform.*

In order to increase the re-use of the hardware blocks, while reducing the area of the accelerators, it is possible to partition the accelerators in a few smaller kernels, still being able to cover the whole computation, but with more processing computation loops. For example, the transposition of 4x4 blocks can be implemented separately from the 1D transforms, as shown by the central block diagram of Figure 3.15. Moreover, all the transforms feature similar computation patterns, that could lead to a single, programmable hardware accelerator that performs all transforms, being customized by a configuration bits stored on the register file by the processor. This approach requires two hardware accelerators and 9 computation loops (three for each transform), as intermediate data needs to be stored on the local buffers before and after the transposition.

A further decomposition of the hardware accelerators can be achieved by reducing the amount of operations implemented in hardware. This approach results in a hardware/software computation, where common patterns of the 1D transform stages are executed by the hardware blocks, and the processor executes the peculiar rounding operation required by each processing step, as shown on the right block diagram of Figure 3.15. It still requires 9 computation loops, but only one hardware accelerator, which performs a portion of the 1D transform, common to all computations. Moreover, the matrix transposition can be performed in a fully flexible way without the need of hardware accelerators. More precisely, it is performed by storing the intermediate results of the row processing of each 4x4 transform on the register file, and utilizing the programmable matrix to address the samples

		Software	Hardware/ software	Flexible hardware	Custom hardware
NUMBER OF CLOCK CYCLES	*Q LUMA AC*	4096	1427	122	122
	Q CHROMA AC	261	117	38	38
	Q LUMA DC	2088	728	77	77
	Q CHROMA DC	522	74	31	31
	NZ LUMA AC	176	158	158	95
	NZ CHROMA AC	12	13	13	10
	NZ LUMA DC	88	72	72	50
	NZ CHROMA DC	24	33	33	17
	ZZ LUMA AC	1008	201	201	97
	ZZ CHROMA AC	67	17	17	48
	ZZ LUMA DC	536	97	97	6
	ZZ CHROMA DC	134	13	13	40
	OVERALL	9012	2950	872	632
PMEM BYTES		1784	3328	2312	2036
DMEM BYTES		7632	1324	1312	1308
BUFFER BYTES		0	1696	1696	1696
ACCELERATORS KGATES		0	12	28	48

Table 3.3: *Implementation of quantization, zig-zag-scan, and non-zero detection blocks algorithms.*

according to the required addressing pattern. Results of the implementation of the hardware accelerators are exposed in Table 3.2, reporting the number of cycles, the area of the hardware accelerators, the program and data memory bytes and the bytes of buffers required for each of the proposed implementations.

The same analysis has been performed for the implementation of the second kernel, which can be divided into 3 sub-kernels that are quantization, detection of non-zero blocks and zig-zag scan. The structure of the quantization allows easy exploitation of data parallelism, as the same operations are performed over the blocks within each macroblock. On the other hand, the implementation of "programmable" hardware accelerators for this kernel is mandatory, as the multiplication coefficients utilized by the quantization process differs per macroblock components (CHROMA AC, CHROMA DC, LUMA AC, LUMA DC) and per macroblock, depending on the quantization parameter. The other kernels composing the computation are the detection of non-zero blocks and the zig-zag scan. The detection of non-

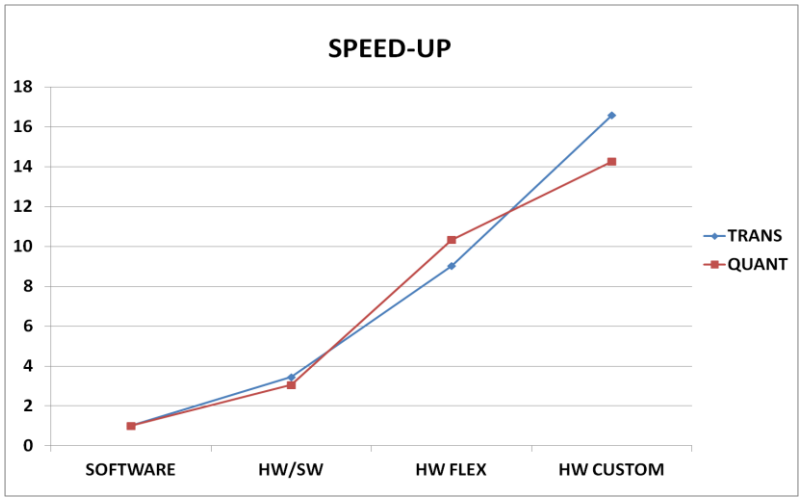
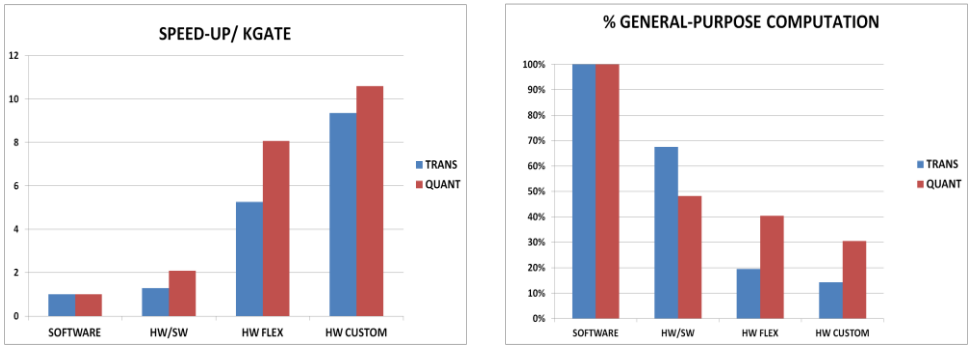


Figure 3.16: *Speed-ups of transform and quantization (plus non zero block detection and zig-zag scan) kernels with respect to the software implementation. Data refer to the elaboration of one macroblock.*

zero blocks benefits from the data-level parallelism, mainly consisting of comparisons of all the samples within a block performed by rows. The results of the computation for each macroblock is then temporary stored in the register file and finally arranged in the output buffers. Finally, the blocks are zig-zag scanned.

Trade-offs between flexibility and performance mainly reside in this application in the implementation of the zig-zag scan and the quantization. The zig-zag scan can be implemented, similarly to the matrix transposition, either with a dedicated hardware accelerator (4 cycles per block), or utilizing the register file and the programmable matrix (8 cycles per block). Contrarily, the quantization can be either implemented as a monolithic hardware accelerator implementing the whole computation or as a two-step hardware accelerator interleaved with software functions. In this last case first hardware function implements the absolute value and the sign extraction of four samples, a software function implements the multiplication with the quantization multiplication factors (MF) and the second hardware operation performs the output rounding as well as the sign insertion. Implementation results of the different algorithm portions are summarized in Table 3.3 together with the resources utilized.



(a) (b)

Figure 3.17: *(a) Speed-ups/Kgate ratio of transform and quantization kernels. Data are normalized with respect to the software implementation. (b) % of the overall computation time involved in general-purpose processing.*

Figure 3.16 put in relation the different implementations of the realized hardware accelerated functions in terms of speed-ups with respect to the software solution. Obviously, the approach providing the best performance for both transform and quantization is the custom hardware implementation, as each accelerator targets the specific kernel, and the related data needs to be processed by the accelerators only once for each computation step. Nevertheless, analyzing data reported in the tables it is possible to notice that utilization of pipelined hardware accelerators is useful especially when large data chunks require to be processed as in the case of AC transforms or quantization (28x and 33x respectively) while other cases, such as transform of DC components or detection of non-zero blocks achieve smaller speed-ups, or no speed-ups at all. In these pathological cases, the time spent for configuration of the address generators, the programmable matrix, and the latency of the accelerators almost reach the time required for the software elaboration, hiding benefits of pipelined hardware processing. Thus, the situation can only improve by processing more than one macroblock at time, which requires larger size of the buffers, or exploiting the data-level parallelism, for example processing two or more blocks concurrently.

In order to explore the proposed trade-off between specialization and general purposeness from a quantitative point of view, the area efficiency and the flexibility of the proposed solutions have been analyzed. Figure 3.17 shows the speed-ups achieved normalized by the area of the resources utilized for

each implementation. Each proposed implementation targets a specific utilization of the data memory, program memory and buffers that have to be considered in this analysis as well as the area of the hardware accelerators. For this reason, the hardware resources considered in this analysis include, in addition to the area of the hardware accelerators, the overall amount of the platform resources involved in the computation by the different implementation (i.e., the processor, the utilization of program and data memory, and the utilization of buffers). For instance, the software implementation of the quantization utilizes lookup tables with replicates of the multiplication factors over the blocks to avoid indexes calculation, while the hardware implementations only utilize a small portion of such tables, resulting in a smaller utilization of the program memory resource in the hardware implementations with respect to the software implementation. Another example is that of the hardware/software implementations, where the need of unrolling the loops to avoid stalls (described previously in this section) forces a large utilization of program memory, with respect to a pure hardware implementation. Results of Figure 3.17a show that in general hardware solutions provide a better silicon utilization. Thus, the area spent for realizing the application specialization is well spent even considering the overheads introduced by the metal programmable approach utilized for the proposed analysis. Of course, the more customized solution results in a less flexible implementation. The flexibility of the different implementations are shown in Figure 3.17b, which report the amount of the overall computation time involving general purpose computations (i.e. processor instructions). This time is 100% when dealing with a fully flexible processor, while it decreases while handling with specialized units down to 10% in the case of the transform.

The results evidence that, for the analyzed kernels, the utilization of specialized accelerators coupled to the general purpose processor can improve the performance up to 20x and area efficiency of the overall computational structure up to 10x with respect to the software solution. On the other hand, when dealing with complete applications, other factors can impact these choices, such as the actual requirements of the application, the overall number of functions to implement and the impact of each function on the overall computation time. For this reason, implementation of less specialized and

more flexible accelerators could be a preferable choice to improve the accelerator reuse over a larger set of functions, while the utilization of homogeneous multithreading could allow one to further improve performance to meet the required target.

3.6.3 Application mapping example

Video Surveillance Motion Detection Application

This section discusses the implementation of the motion detection application described in the previous chapter on the Manyac platform. The last section focused on the trade-offs exploited by the implementation of the hardware accelerators. Contrarily, this section focuses on the high-level partitioning of the application, analyzing the benefits and overheads in the utilization of data-parallel or task parallel computational model. In order to focus on the computational model, we assume in this context to fix the number of computational tiles to four.
The partitioning of the application among the available computational units starts from the profiling of the separate kernels that implement the application. Table 3.4 shows the main features of the implementations of the application kernels accelerated with metal programmable arrays using the

KERNEL	*THROUGHPUT [CYCLES/PIXEL]*	*AREA OF ACCELERATORS [KGATES]*	*PMEM UTILIZATION [BYTES]*
SUB/ABS	0,11	7	256
MAX	0,09	11	156
BINARIZATION	0,45	7	484
EROSION	0,42	4	828
DILATATION	0,42	4	828
EDGE DETECTION	0,43	20	828
FINAL MERGE	0,17	8	178

Table 3.4: *Implementation results of motion detection video surveillance application accelerators.*

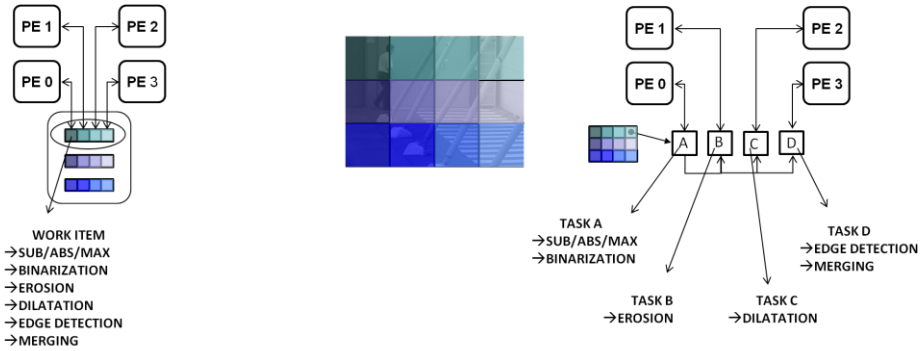


Figure 3.18: *Partitioning of the Motion Detection application over four computational tiles of the Manyac platform utilizing a data parallel computational model (left) and a task parallel computational model (right).*

Griffy environment. These features are the throughput in terms of cycles/pixel, the area of the metal-programmable area utilized for implementing the accelerators and the number of required program memory bytes.

Concerning the computational model, two solutions are possible that are explained in Figure 3.18. The utilization of a data-parallel model leads to homogeneous execution of the application among the 4 available cores. In this scenario, each core executes sequentially the kernels composing the application, but processes different data. This computational model explicitly exploits the spatial parallelism provided by the application, which executes

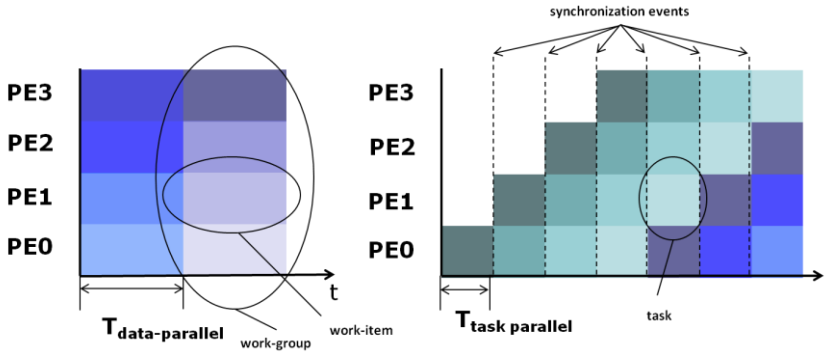


Figure 3.19: *Temporal scheduling of work-groups and tasks on the Manyac computational tiles (PE) when utilizing the data parallel computation model (left) and the task parallel computation model (right).*

the same computation on the different portions of the source image. On the other hand, when utilizing the task-level parallelism, the kernels composing the application have to be partitioned among the available cores. Each task, which executes sequentially one or more basic kernels, is allocated to a specific computational tile. Utilizing this computational model, the parallelism is achieved in a heterogeneous way, where the different tasks allocated to the computational tiles compute different portions of the source image, scheduled by synchronization events explicitly handled within the host program.

Figure 3.19 shows the temporal scheduling of the image portions execution, when the data parallel and the task parallel programming models are utilized.

When utilizing the data parallel programming model, the index-space decomposition provided by OpenCL provides a subdivision between work-items that are executed concurrently on the available processing elements of the platform, and work-groups that execute sequentially. The execution time of each work-item executing on each computational tile (or each work-group executing on the platform) can be calculated as:

$$T_{work-group} = \sum_{i=1}^{i=\#kernels} T_{kernel\_i}$$

Where the #kernels is the number of kernels executed by each work-item (6 in this case) and they are those reported in Figure 3.19. The overall computation throughput can be calculated as:

$$Troughput_{data-parallel} = 4/T_{work-group}$$

Each work-group executes concurrently 4 work-items. Considering the kernels composing the motion detection application, the computation time per pixel is equal to 2,09 cycles, thus the throughput achieved by the platform for this application is 1,91 pixel/cycle for an overall speed-up of 1365x with respect to the sequential software implementation.

Contrarily, the execution time for each task executed on the platform can be calculated as:

$$T_{task} = \max\left(\sum_{i=1}^{i=\#kernels} T_{kernel_i}\right)$$

Where #kernels is the number of kernels executed by each task, and they are sub/abs/max and binarization for the task A, erosion for the task B, dilatation for the task C, and edge detection and merging for the task D. In other words, when adopting a heterogeneous computation model, the overall computation time is constrained by the slowest stage of the pipeline implemented through the synchronization events. The overall throughput is calculated in this case as:

$$Troughput_{task-parallel} = 1/T_{work-group}$$

as the computation chain produces one block for each stage of the software pipeline. In this scenario, the slowest stage is that of task A, whose computation time per pixel is equal to 1,53 cycles leading to a throughput of 0,65 pixels/cycle resulting in a speed-up of 464x with respect to the sequential software implementation.

Besides the throughput the trade-off between data parallel and task parallel computation involves different utilization of resources. These are the area of the hardware accelerators and the utilization of program memory. When utilizing the data parallel programming model, the program memory within each computational tile should be able to contain the code of all the application kernels. Contrarily, with the task parallel programming model only the program required to execute the kernels allocated to each task needs to be stored on the related computational tile. The same discussion can be afforded for what concerns the hardware accelerators, that require to be implemented for each computational tile when utilizing a data parallel computation model, while they can be equally partitioned among the computational tile areas when utilizing the task parallel model.

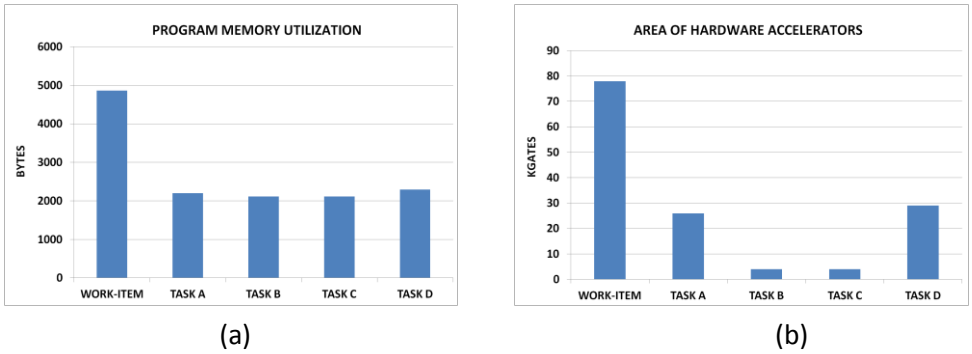


Figure 3.20: *Program memory (a) and area of hardware accelerators (b) utilized for implementing the work-items and tasks for the motion detection application.*

Figure 3.20 summarizes the resource occupation of work items, when utilizing the data parallel programming model, and of each task utilized with the task-parallel implementation. The program memory of each task and work item is composed of the code required for the related kernels plus an "offset" which includes the initialization code, and the runtime libraries required to handle the DMA channels, hardware accelerators and OpenCL primitives. The results show that the data parallel computation, even providing a greater throughput requires more than double program memory and four times the area necessary to implement the hardware accelerators, with respect to the task parallel computation.

The analysis performed on the motion detection application evidences the benefits and overheads of the task-parallel and data parallel computational models, but assumed the cost of memory transfer completely overlapped to the computation, and no synchronization overhead. Next section introduces problems related with the overheads introduced by synchronization and data transfers; assuming that the computation/data transfers overlapping is not possible. The impact of these overheads on the performance is analyzed when modifying architectural parameters of the platform.

3.7 Performance Analysis

In order to evaluate the performance of the proposed platform from a quantitative standpoint, we discuss now the implementation results of signal processing applications on the Manyac platform. The applications selected for the experiments are the edge detection (part of the motion detection application), two granularities of FFT (64, 1024), the H264 discrete cosine transform and quantization described previously, and the YCC2RGB color space conversion. A specific target of the proposed analysis is the understanding of the performance sensitivity with respect to the architectural parameters of the platform, and the main differences between multi-processor acceleration and acceleration based on multiple application specific hardware. The parameters chosen as most representative for the Manyac platform have been identified as:

- Hardware accelerators
- Size of buffers
- Number of cores
- Width of the NoC

In order make the discussion more general and independent of the specific trade-offs applicable to complete applications when choosing a task-parallel computational model, weeassume in this analysis the utilization of the data-parallel programming model applied to kernels instead of complete applications. As we saw in the previous sections, the presence of hardware accelerators is the first main factor that impacts the performance of a given implementation of an application. Depending on the features of the application and the granularity of the accelerators it is possible to speed-up a kernel by up to three magnitude orders. Figure 3.21 shows the speed-ups achieved by the hardware accelerated implementation of the applications, calculated considering a single processor equipped with a set of hardware accelerators. Speed-ups reported in Figure 3.21 refer to the implementation of an “elementary” data chunk for each application which are an 80x60 binarized

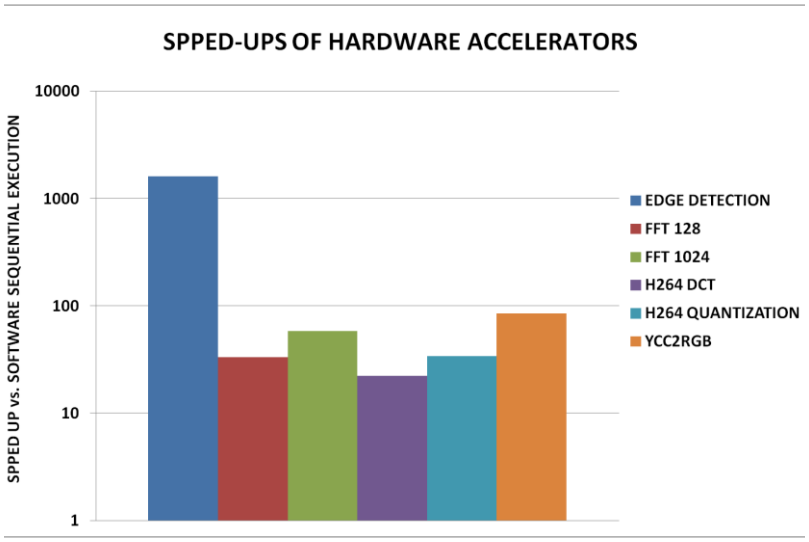


Figure 3.21: *Speedups of application implemented with hardware accelerators with respect to the software sequential implementation.*

image for the edge detection, a macroblock for the H264 DCT and quantization, a 1024 pixel image chunk for the YCC2RGB.

The dependency between the features of the applications and speed-up can be approximated as two-sided. The first feature that affects the performance is strictly related with the exploitation of the instruction-level parallelism, and concerns the computational density of the applications, defined as the number of operations per bit. This means that applications requiring more operations on the same data set would reach higher performance when implemented with hardware accelerators. This case is well represented by the two granularities of FFT implemented, showing that the 1024-point FFT reach higher speed-ups than the 128-point FFT, only due to its higher computational density (that scales according to the number of points).

The second point concerns the granularity of the operands. Given a fixed bandwidth toward the hardware accelerators, the smaller the operands, the more data-level parallelism can be exploited. The application that shows main benefits from this point of view is the edge detection, which computes on binarized images, thus being able to process an enormous number of parallel pixels concurrently. On the other hand, the H264 transform and quantization

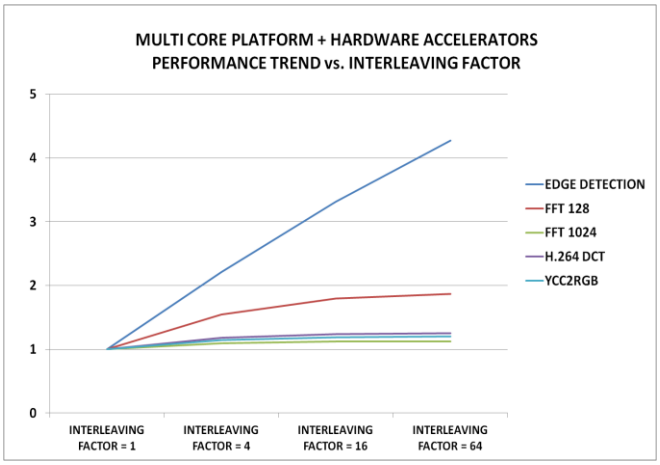


Figure 3.22: *Speed-ups of applications implemented on the Manyac platform when varying the interleaving factor of elementary data chunks processing.*

feature relatively small computational densities and 16-bit operands width, thus limiting their speed-ups to ~20x.

Moving the architectural analysis to the computational tile parameters, we evaluated the impact of increasing the size of the local buffers, which impacts the interleaving factors with which data chunks can be processed by the hardware accelerators. This technique allows to exploit the pipelined behavior of the hardware accelerators over larger data chunks thus amortizing the time spent for control, configuration of the address generators and the configuration matrix, and the setup of the hardware accelerators. Figure 3.22 shows the trend of the platform performance when increasing the interleaving factor, for a fixed NoC width (in this case 128). The reported curves are normalized to the speed-ups of applications processing a single data chunk. Speed-ups are calculated with respect to the sequential software implementation. The curves show a saturating trend whose saturation point depends on the computational density and the size of the basic data chunk processed by each application. The saturation is caused by the global communication that emerges as main bottleneck when the exploitation of hardware accelerators reduces the computation components of the algorithms.

Moving to the system-level, we analyze as the first parameter the number of cores composing the platform. In particular, the speed-ups of the selected applications have been analyzed on both a multi core platform and a hardware

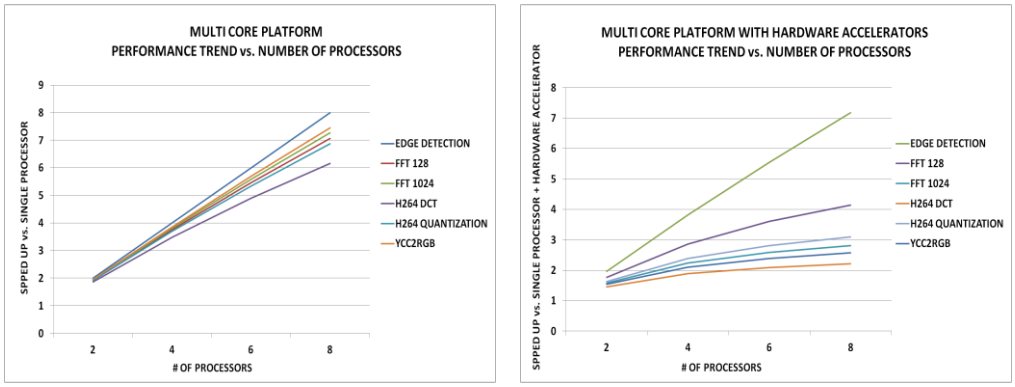


Figure 3.23: *Speed-ups of applications implemented on the Manyac platform without hardware accelerators (a) and with hardware accelerators (b). Speed-ups are normalized with respect to the single processor implementation without and with hardware acceleration, respectively.*

accelerated multi core platform. Figure 3.23 shows results of the analysis, performed on 2, 4, 6, 8 processors, keeping the interleaving factor to 1 and the NoC width to 64-bits. Results related to the parallel software implementations of the applications are reported in Figure 3.23a, as they are very close to the ideal results (speed-up equal to the number of cores). This happens because the most of the overall applications time remains related to the computation, while only a small part of the time is spent for data transfers (which represents the portion of time not reducible with parallelism). On the other hand, when handling with a multi-core platform with distributed hardware accelerators, the computational portion of the application is already reduced by the hardware accelerators, and the data transfer time remains as a major contribution, not eliminable with further parallelism exploitation. Still, applications with higher computational densities show more benefits from the thread-level parallelization, due to the higher ratio between time utilized for computation and time utilized for transfers.

The time required for data transfer can only be reduced by properly sizing the width of the network-on-chip. Figure 3.24 shows the implementation of the applications with different sizes of the NoC, while maintaining the number of processors fixed to 8 and the interleaving factor to 1. It is interesting to notice how, differently from the multi-processor software implementation showing a saturating trend, speed-ups of the multi-accelerated platform linearly raise together with the NoC data width. This means that when utilizing powerful

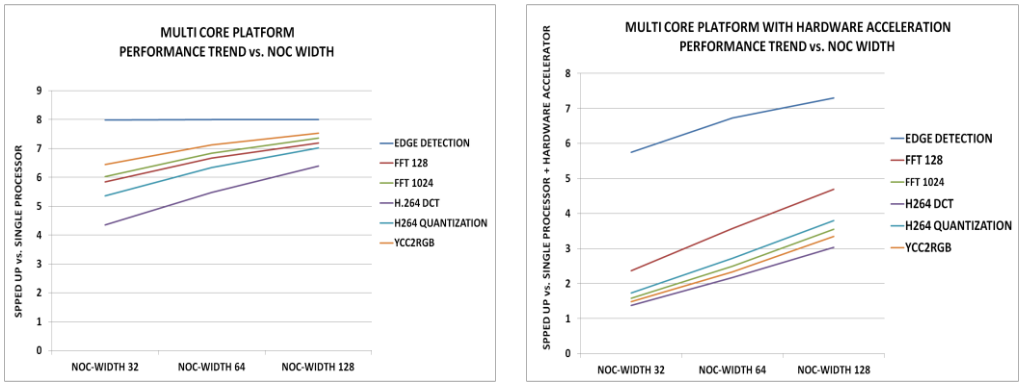


Figure 3.24: *Speed-ups of applications implemented on the Manyac platform without hardware accelerators (a) and with hardware accelerators (b). Speed-ups are normalized with respect to the single processor implementation without and with hardware acceleration, respectively.*

hardware accelerators, if we want to achieve a further performance improvement through parallelism we need to guarantee to the accelerators an adequate bandwidth being able to sustain their throughput.

From the above described analysis, we can state that the speed-ups of parallel applications can be achieved acting on many parameters of the platform, and the choice of each one affects the global performance. Thus the mapping of applications on a multi-accelerated platform cannot disregard system-level aspects. In particular, all the elements analyzed in the last sections, such as the choice of the right granularities of accelerators, the choice of the data- or task-parallel computational model and the selection of appropriate architectural parameters concur to the matching of the performance and power constraints of signal processing applications. The effectiveness of the utilization of hardware accelerators or thread level parallelism for improve performance must be carefully balanced with these parameters which form the main design choices for the described architecture.

3.8 Implementation results

This section describes the implementation of the Manyac platform in CMOS65 STMicroelectronics technology. As described previously in this chapter, the platform mainly consists of two components: the IO tile and the

CMOS65 Implementation of the Manyac platform
Number of Computational Tiles: ***4***
Network on chip and CT local bus data width: ***64bit***
CT local, data and program memory size: ***4K+4K+4K***
PGA Buffers: ***4x1024x32-bit + 16 registers***
Area: ***Computational Tile 1 mm^2 , IO Tile 0,5 mm^2***
Maximum Frequency: ***200MHz (WC-125°C-0,9V)***
Power consumption: ***86 mW@200MHz (TYP-25°C-1,0V)***

Table 3.5: *Manyac platform implementation results.*

computational tile, this last replicated over the silicon area of the platform. On the other hand, the customized macros implementing the metal-programmable, via-programmable, or run-time programmable accelerators are separate, components, pluggable at design time. For this reason we first analyze the implementation of a typical platform without customization, and then we analyze the impact of its customization with different technologies.
Results of the implementation of the Manyac in CMOS65 STMicroelectronics technology are shown in Table 3.5, which refers to the platform components without customization. The computational tile area with the reported configuration is equal to 1 mm^2, partitioned and shown in Figure 3.25. It is

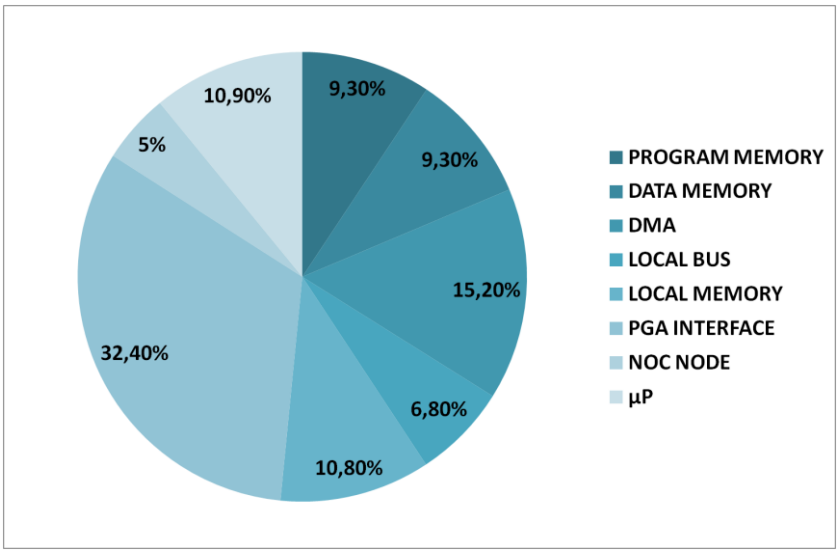


Figure 3.25: *Area breakdown of the computational tile component by logic entity.*

interesting to notice how a relevant portion of the computational tile area is utilized for communication (DMA+NoC+Bus) and local storage (Memories + PGA interface), emerged in the analysis of the last section as the most important factors in the exploitation of powerful hardware accelerators. The platform is capable to achieve a maximum working frequency of 200 MHz estimated in worst case commercial conditions, with an average power consumption of 86 mW. The power consumption is estimated utilizing the Synopsys PrimePower® tool, assuming a switching activity of 20%. Results reported in Table 3.5 are accomplished with the architectural parameters selected for the implementation. The layout view of a 4-tiles implementation of the Manyac platform is provided in Figure 3.26.

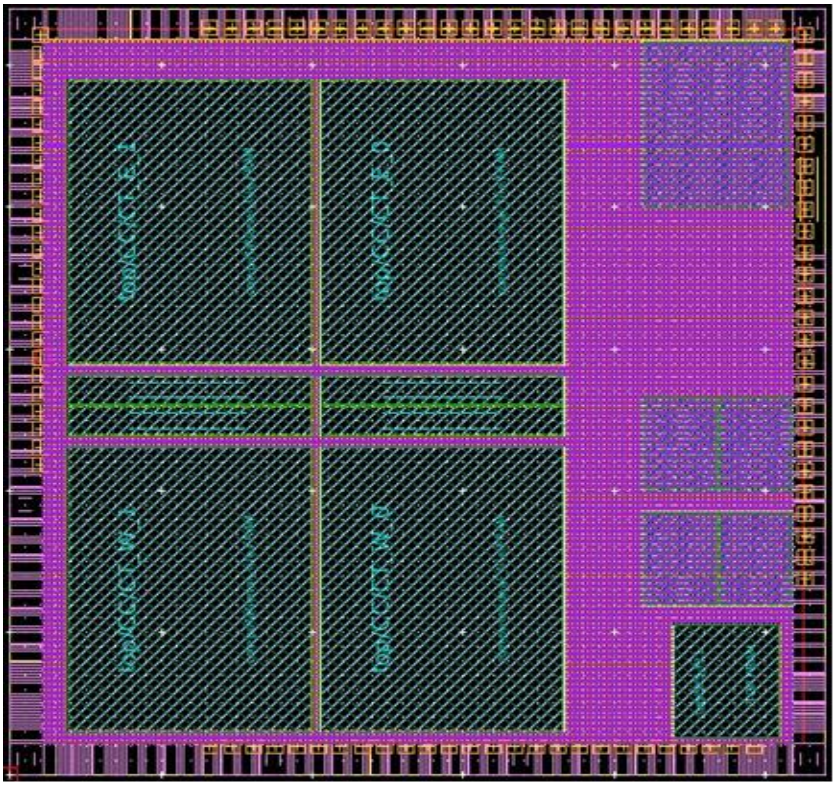


Figure 3.26: Layout *view of a 4-tiles implementation of the Manyac Platform*

	RUN TIME CONFIGURABLE GATE ARRAY	*VIA CONFIGURABLE GATE ARRAY*	*METAL CONFIGURABLE GATE ARRAY*	*ASIC*
FREQ. [MHz]	200	200	250	250
AREA [mm^2]	7,6	3	0,3	0,2
POWER [mW]	66,8	38,3	0,75	0,55
# CUSTOMIZATION LAYERS	0	1	9	33

Table 3.6: *Implementation Results of Customizable Hardware Accelerators.*

Moving the focus on the customizable areas of the platform, in order to evaluate the performance of the different configuration technologies, the accelerators realized to accelerate the applications described in the previous section have been implemented on the different gate-arrays. As each customization technology is based on a different kind of structured-silicon solution, each one introduces some overhead in terms of maximum frequency, area and power with respect to the case of standard-cell based ASIC approach. The goal of the proposed analysis is to quantify the gap in terms of power and area with respect to the ASIC approach, utilized in this context as reference.

Table 3.6 summarizes the results of the implementation of the YCC2RGB, H264 DCT, FFT and edge detection the proposed configurable gate arrays in CMOS065 technology, resuming the number of lithography masks required for the customization of each technology. Considering the working frequency, estimated referring to the worst case commercial conditions (wc, 125°C, 1V), it is possible to notice that it does not depend on the application mapped on each instance of the gate array, but only on the chosen configuration technology. This is achieved through the pipelined structure of the accelerators implemented utilizing the Griffy flow, which avoids the kernel mapping on hardware to be a bottleneck for the system, thus leading to a high performance predictability. The run-time programmable and the via-programmable gate arrays can reach a frequency of 200 MHz. Although an overhead in performance would be expected by the run-time programmable gate array, the full custom design approach utilized for its design fills the gap between the run-time programmable and the via-programmable solution. On the other hand, the metal programmable gate array can reach the target frequency of 200 MHz, as well as the standard cell-based ASIC

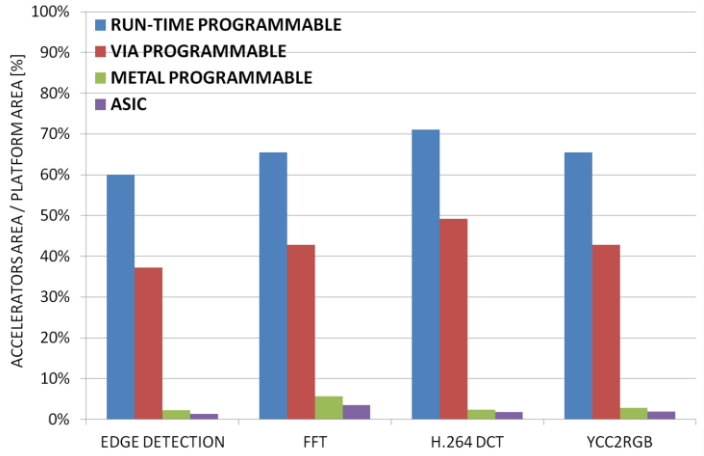


Figure 3.27: *% of the Manyac platform area utilized for configurable accelerators.*

implementation. It should be noticed that both metal programmable and ASIC solutions would reach even higher frequency, but their implementations were constrained at the maximum frequency achieved by the rest of the system.

The utilization of configurable technologies as application specific acceleration within MPSoCs introduces an increment of the die area which depends on the level of flexibility (i.e., number of masks for customization) allowed by the different customization technologies. For this reason, amount of area utilized for the implementation of configurable hardware accelerators when utilizing the different customization technologies has been analyzed. Results of Figure 3.27 show the percentage of the Manyac platform area utilized for the implementation of accelerators. Data are reported per application. For comparison, results of the run-time and via-programmable gate array are scaled to the actual number of rows utilized by each application. Results show how the heavily structured architecture of those two solutions, which form the basis for their higher flexibility, causes relevant overheads in area with respect to the fully programmable area of the platform. On the other hand, the metal-programmable solution relies on a synthesis-based design flow, which eliminates all the structural overheads of the other two solutions (at the cost of flexibility). This solution shows an overhead with respect to the related ASIC implementation, which is less than 1.5x.

The power consumption of the different gate array implementations is estimated with Synopsys PowerCompiler® assuming a switching activity of 20% at the nominal commercial condition (nom, 25°C, 1,2V). The run-time programmable and via-programmable gate array implementations are based

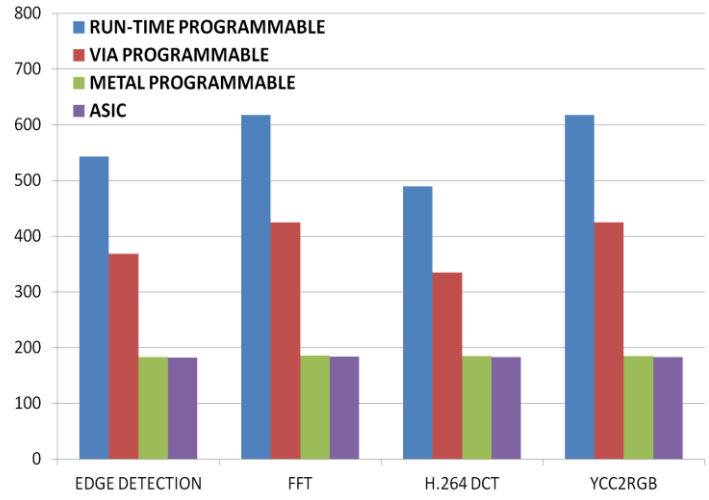


Figure 3.28: *Power consumption of applications running on the Manyac platform. Different configuration technologies are assumed as implementation platform for the hardware accelerators.*

on a fixed array structure. As the considered applications utilize only a subset of the array, while the rest is clock-gated, the power results are calculated scaling the dynamic power according to the actual number of rows utilized by each implementation. Figure 3.28 shows the average power consumption of the platform equipped with the proposed gate array implementations when running the chosen applications. The results highlight that, despite the full-custom implementation utilized for the run-time programmable and the optimized implementation of the via-programmable gate array, the gap in power with respect to the standard-cell based approach falls from about 70x of via-programmable solutions to 120x of the run time programmable solutions.

Results of the proposed implementations show how the performance of the accelerators implemented on the proposed configurable datapaths are only minimally affected by the chosen customization technology. Utilizing the proposed implementation flow, considering the standard-cell based approach as reference, the overhead of via- and run time-programmable datapaths is 20%, while the overhead of the metal-programmable solution is nearly null. The most overhead paid by the configurable solutions (especially the run time configurable and via-configurable datapath) resides in the power and area, mainly caused by the structured architecture of the two solutions which allow their customization without masks post-fabrication or with the fabrication of only one lithography mask.

Chapter 4

4 Evaluation of multi-core platforms with configurable accelerators

This chapter describes the evaluation of the proposed platforms with respect to the other platforms representing the state of the art of embedded computing for signal processing.

The evaluation first goes through the analysis of the application development time for different computational platforms. In this context, the models and languages utilized for the programming and customization of the described platforms will be analyzed. Then, the programming productivity of the different platforms will be estimated on the basis of a commonly used cost model.

The second evaluation considers the most commonly utilized metrics for evaluating computing platforms for embedded computing, the performance, energy efficiency, and area efficiency.

Finally, as the proposed platform spaces the different trade-offs between flexibility (i.e., number of masks for customization), and efficiency (frequency, area, power), the cost of manufacturing of the different solutions will be analyzed, highlighting benefits and drawbacks of each solution according to the products market volumes, and giving a perspective based on the scaling of the CMOS technologies.

4.1 Applications development cost

The aim of this section is to evaluate the aspect related to the cost of the applications development on multi-processor system on chip, and the related implementation of application specific or reconfigurable acceleration.
When dealing with multi-core systems the first step in the application development consists of the partitioning of the target application among the computational cores of the platform that can be either homogeneous or heterogeneous. When programing a heterogeneous MPSoC, and the number of cores composing the system is relatively low (i.e., up to 4) this partitioning can be performed manually, and handled with commonly used programming languages, such as C or C++.
This is the case of most ASSPs described in Section 1, that utilize a controlling core plus a set of application specific hardware accelerators or powerful DSPs. The choice of such kind of programming languages has the advantage of offering a very high programming legacy due to their large utilization in many kinds of domains. On the other hand, these languages do not provide natural statements to provide synchronization or, more in general, to handle parallelism. The lacks of theses languages are often compensated in such kind of devices by the utilization of pre-packaged libraries provided by the devices vendors for standard kernels, that drastically reduces the development time of the final users which only perform the wrapping between the application kernels. This is the programming style of Morpheus, where the ARM processor programmed with the C language provide synchronization and control of the reconfigurable engines, while the programs running on each reconfigurable engine are developed independently, packaged in a configuration bitstream and loaded at run-time by the ARM processor.
Even if this kind of programming can be acceptable when we deal with a relatively small number of processors, the manual partitioning and synchronization of applications cannot be handled manually when dealing with a large number of processors working concurrently. Programming languages for parallel systems (MPI, OpenMP, CUDA, OpenCL) are usually based on the C or C++ language, extended with application programming interfaces (APIs), or pre-processor directives that provide support for synchronization, explicit description of parallelism, handling of memory space allocation and vectorized data transfers. In particular, the OpenCL programming model utilized for the Manyac platform has the advantage of supporting a heterogeneous set of devices, either characterized by high data-

Language	Average Source Statements per FP	Productivity Average per Staff Month
C	128	9 FP
ASM	213	5 FP
VHDL	19	18 FP

Table 4.1: *Function point analysis parameters.*

level parallelism (for which a data parallel programming model is provided) or task level parallelism.

The second step in the application development consisting of application development on multi-core (re)configurable processors is performed partitioning the computational workload between software and configurable hardware. Although many high level synthesis tools have been proposed, from the practical point of view, the most common methodologies for the design of hardware on both silicon and FPGAs are still based on the register transfer level (RTL) description, created by hand utilizing, for example the VHSIC Hardware Description Language (VHDL). Such task leads, in the proposed platforms at the programming of the configurable engines of Morpheus utilizing the reconfigurable engines proprietary tools, and the designing of hardware accelerators utilizing the Griffy flow of the Manyac platform.

The proposed evaluation should then take into account an estimation of both management of thread/task-level parallelism, synchronization and wrapping of an application and the implementation of the application specific accelerators on run-time configurable fabrics, or application specific circuits. The evaluation of the programming productivity is a very important problem that has been the object of studies over the last 30 years. Although software productivity estimations and evaluation methodologies are currently under investigation, some of those have already been ported to the field of the hardware description languages [76]. As the implementations of applications described in the context of this work are handled with a heterogeneous set of software programmable processors and run/design-time configurable components, a heterogeneous set of languages was analyzed. In order to

TABLE

Application	ARM [C]	DREAM [C]	XPP [C]	DREAM [Griffy-C]	XPP [NML]	eFPGA [VHDL]
RGB2YUV	1,3	0	0,5	0	5,6	0
Edge Detection	1,6	2,5	0	13,2	0	0
Binarization	1,6	0	0	0	0	14,9
AES	1,9	8,8	0	9,8	0	0
CRC	2,3	1,9	0	26,9	0	0
ME	5,0	0	2,6	0	60,0*	0
MC	7,0	0	8,6	0	25,0	0
Ethernet	1,2	0	0	0	0	39,5

*Manual optimization and placement required

Table 4.2: *Estimation of design effort of applications implemented on the Morpheus platform.*

evaluate the development productivity of applications on the different platforms a well-known technique, which provides ready to use data for many programming languages has been utilized: the Function Point Analysis (FPA) [77].

Table 4.1 summarizes the parameters necessary for performing the analysis, referred to the languages utilized to program the evaluated platforms. In order to evaluate the programming productivity of the proposed platforms, we approximated the wrapping/synchronization stage of the application to be implemented utilizing the C language even if actually realized with extensions of the C language (i.e., OpenCL). Thus, according to the model, the advantage in the utilization of these languages consists of the fewer number of statements utilized to achieve the same.

On the other hand, the implementation of the accelerators, or the programming of the reconfigurable engines has been performed utilizing VHDL, Griffy-C or NML depending on the application. Griffy [65] and NML [64] show similarities with intermediate representations (IR) utilized by most compilers to produce assembly code. In fact, modern compilers utilize high-level intermediate representations, often based on Static Single Assignments

(SSA) to implement more efficient optimization steps [78][79][80]. For that, we consider ASM as a reference for NML and Griffy FPA. In order to perform the FPA on the selected test cases, we inspected the source codes of the applications implemented on the described platforms, the pure software implementation and the FPGA implementations. Then, we utilized the parameters reported in Table 4.1 to perform the design effort estimation, starting from the number of statements extrapolated from the application source codes. Table 4.2 reports the resulting data, expressed in person day. According to the estimations, implementation of kernels utilizing the reconfigurable engines proprietary languages requires much of the design effort, while the control and synchronization tasks implemented on the ARM processor only require a minor effort. In addition, the VHDL implementations on the eFPGA core require a development time considerably higher that the other applications. In Figure 4.1, design efforts of applications implemented utilizing the different languages have been compared with the C language ARM implementations of the same algorithms and with the VHDL implementations.

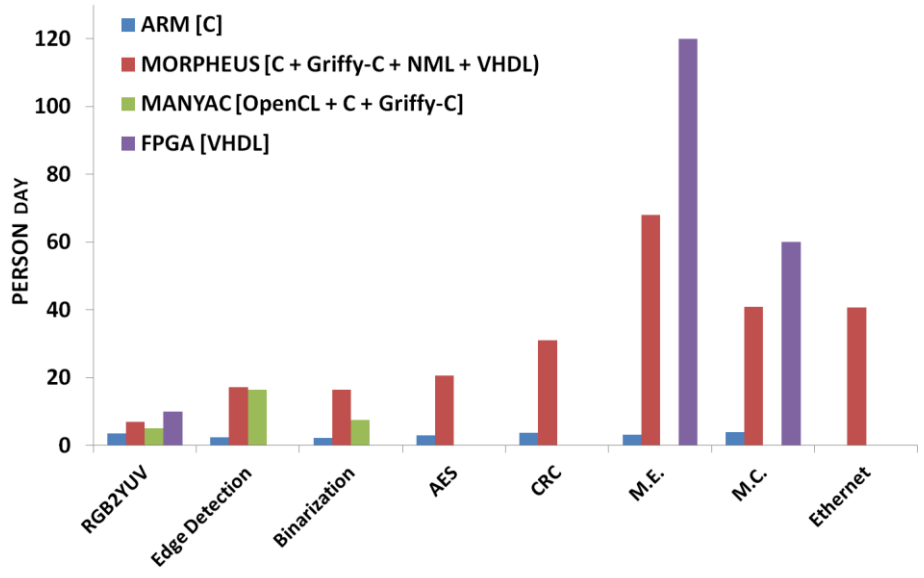


Figure 4.1: *Estimation of design effort required to implement selected applications on different computational platforms.*

The results shows that the FPGA implementations of the proposed applications require a design effort 42% to 76% larger compared to the Morpheus implementation. On the other hand as expected, the C implementations require smaller efforts. Nevertheless, it should be noticed that manual optimizations typical of signal processing algorithm implementations on embedded processors and DSPs (e.g., assembly coding of critical kernels) were not performed in this context. Although the absolute number of person days seems to be under-estimated, results of the analysis are in line with our practical experience from the qualitative point of view, giving a good view of development time ratios among different implementations.

4.2 Performance

This section provides a quantitative evaluation of the performance of the proposed platforms. The main metrics adopted for the evaluation of the performance considered in this context are those most commonly utilized for embedded applications. The first one consists of the computation capabilities, expressed in this context as Giga Operations Per Second (GOPS), where an operation is considered in this context as an equivalent RISC operation. The second metric consists of the energy efficiency, represented by the number of GOPS delivered by a device per each watt consumed. Please note that the GOPS/W metric is equivalent to Op/nJ (number of equivalent RISC operations per nano-Joule), which is an expression of energy. The third metrics considered in this context is the computational density of devices, expressed as GOPS/mm^2. As the digital signal processors proposed in this thesis were benchmarked with a slightly different set of applications, the performance of the two devices will be first evaluated separately, then results will be generalized.

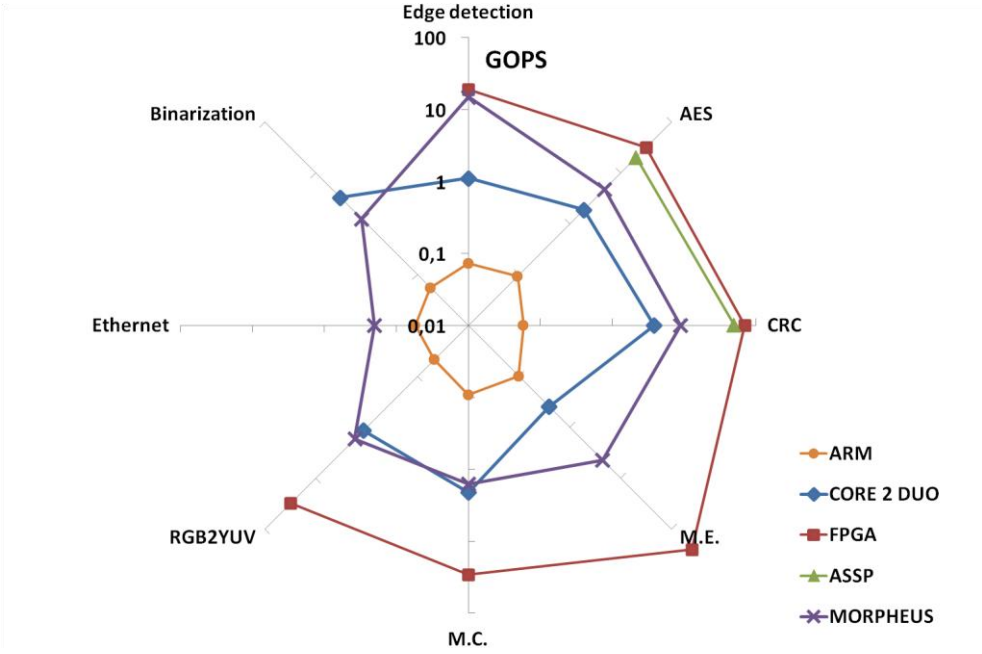


Figure 4.2: *Performance of Morpheus and other SoA devices.*

* For comparison data are scaled to 90 nm technology assuming a 1/λ reduction in delay.
**Reported data refer to Intel Core 2 DUO E6400 for the M.E. M.C and RGB2YUV applications, Intel Core 2 DUO C6600 for the others.

Figure 4.2 reports the performance of applications utilized for benchmarking the Morpheus platform. The performance of the Morpheus platform was compared with the devices that represent its design space boundary: GPPs, FPGAs, and ASSPs. As the performance reported by most of referenced works refers to application bandwidths, all the data were re-processed, and annotated in terms of equivalent Giga Operations Per Second (GOPS). Morpheus' performance is half way between GPPs and FPGAs/ASSPs spanning from 1.25 GOPS of Binarization to15 GOPS of Edge Detection. As expected, in terms of absolute performance, the Morpheus platform cannot challenge either ASSP or FPGA implementations. In the first case, this is due to specific optimization of the hardware implementations realized to match the application requirements; in the second case it is due to the huge amount of logic and I/O resources available on modern FPGA devices by which they widely surpass the capabilities of Morpheus, ASSPs considered in this context, and GPPs. Considering the analysis performed in Section 3 the main limitation of the Morpheus platform with respect to FPGA devices can be

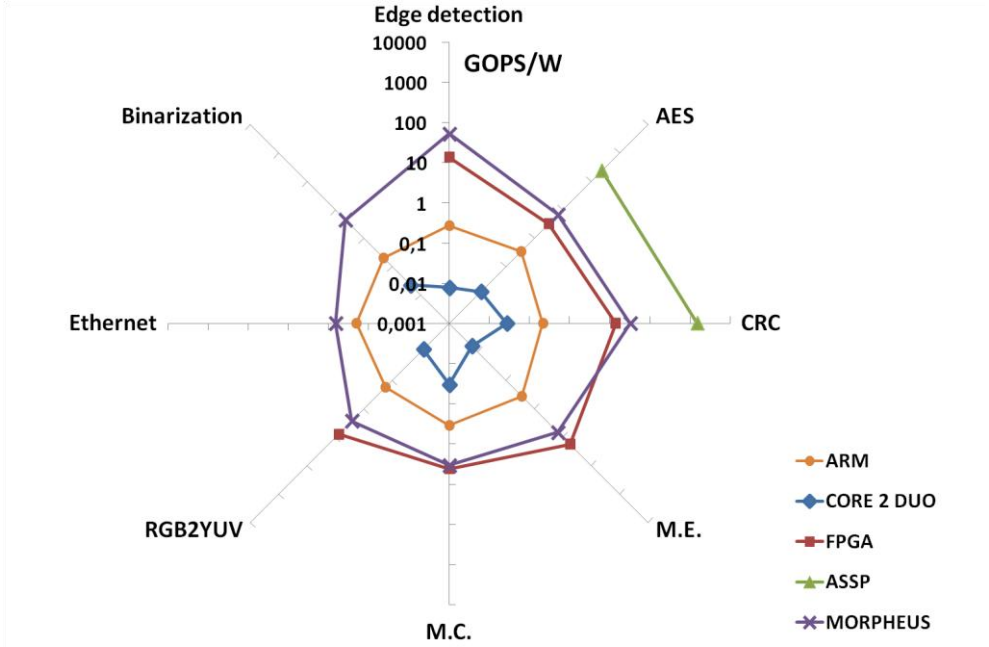


Figure 4.3: *Energy efficiency of Morpheus and other SoA devices.*

* For comparison data are scaled to 90 nm technology assuming a $1/\lambda^2$ reduction in power.

determined in the external bandwidth, that causes slight performance degradation that limits its computational throughput.

The situation reverses when energy is introduced as the criterion of comparison. As shown in Figure 4.3, the energy efficiency of applications implemented on Morpheus span between 2 GOPS/W of RGB2YUV and 50 GOPS/W of Edge Detection. In this scenario, ASSPs represent the upper limit, due to the high efficiency of their hardwired accelerators. By contrast, both embedded and mainstream GPPs are inefficient in terms of energy. The first, even if consuming a relatively small power are not able to deliver high performance due to their software sequential execution and small working frequency. On the contrary, the seconds are able to reach extremely high operating frequency. This feature allows mainstream processors to achieve higher performance, but on the other hand causes high power consumption resulting in poor energy efficiency. Considering the energy efficiency, the Morpheus platform is able to reach, and in some cases exceed FPGA performance. Even if mitigated by the frequency scaling, the degradation of performance caused by the external memory accesses has an impact on its

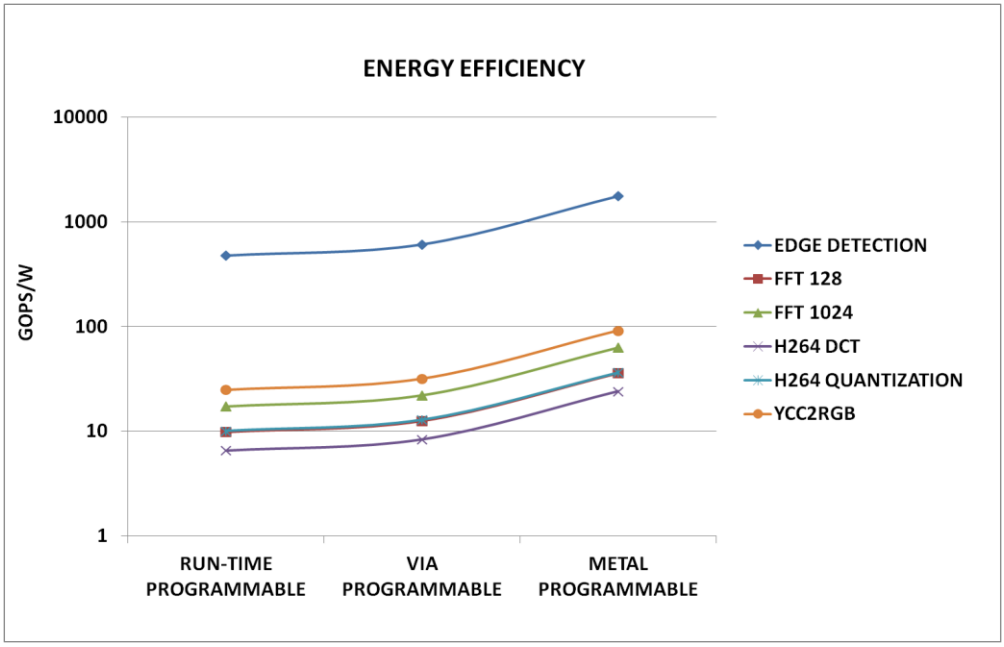


Figure 4.4: *Energy efficiency of applications implemented on the Manyac platform considering the different configuration technologies.*

energy efficiency due to the power offsets as described in Section 3. A higher external memory bandwidth would allow the Morpheus platform to reduce the throughput gap with respect to FPGAs, and widely overcome the FPGA performance in terms of energy efficiency.

As the Manyac platform does not rely on a prototype implementation, but only on estimations based on its physical implementation, the performance of the platform has been normalized by the power consumption of the platform and by the area of the platform for the different configuration approaches utilized. The architectural parameters of the Manyac platform are assumed to be fixed to those utilized for its physical implementation as described in Section 4. Figure 4.4 shows the energy efficiency of the Manyac platform when running the analyzed signal processing applications. As the accelerators implemented utilizing the run-time programmable, via-programmable, and the metal-programmable arrays feature the same computational model the curves related to all applications feature a common trend. Energy efficiency of the platform with run-time programmable gate array falls between 6,5 GOPS/W and 420 GOPS/W. The energy efficiency of the platform with via-

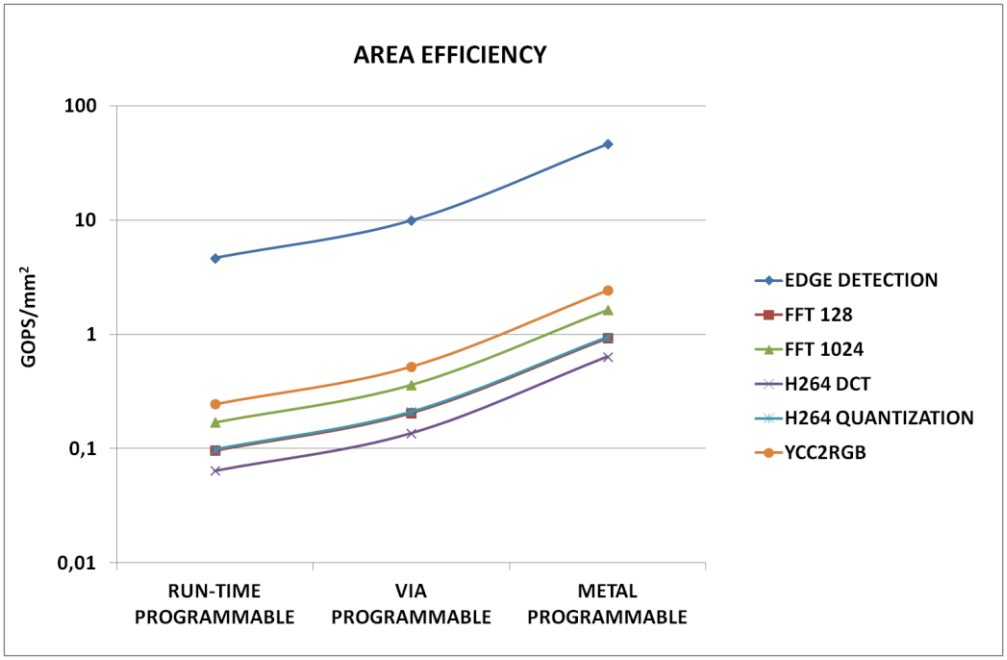


Figure 4.5: *Area efficiency of applications implemented on the Manyac platform considering the different configuration technologies.*

programmable gate array falls between 8,4 GOPS/W and 608 GOPS/W. Finally, the energy efficiency of the platform equipped with metal programmable gate array falls between 24 GOPS/W and 1765 GOPS/W. The energy efficiency delivered by the applications are very different. This mainly depends on the computational complexity of the applications as well as the granularity of the operators. The definition of "operation" as equivalent RISC operation, favor those applications featuring bit-level granularity.

The evaluation of the area efficiency of the platforms equipped with the three different kinds of configurable accelerators is shown in Figure 4.5. The area efficiency of the platform equipped with run-time configurable gate array falls between 0,06 and 4,6. The area efficiency of the platform equipped with via programmable gate array falls between 0,13 and 10, while the area efficiency of the platform equipped with metal-programmable gate array falls between 0,6 and 46. It is possible to notice that the curves follow the same trend as the area efficiency, but they raise more rapidly than the power efficiency curves,

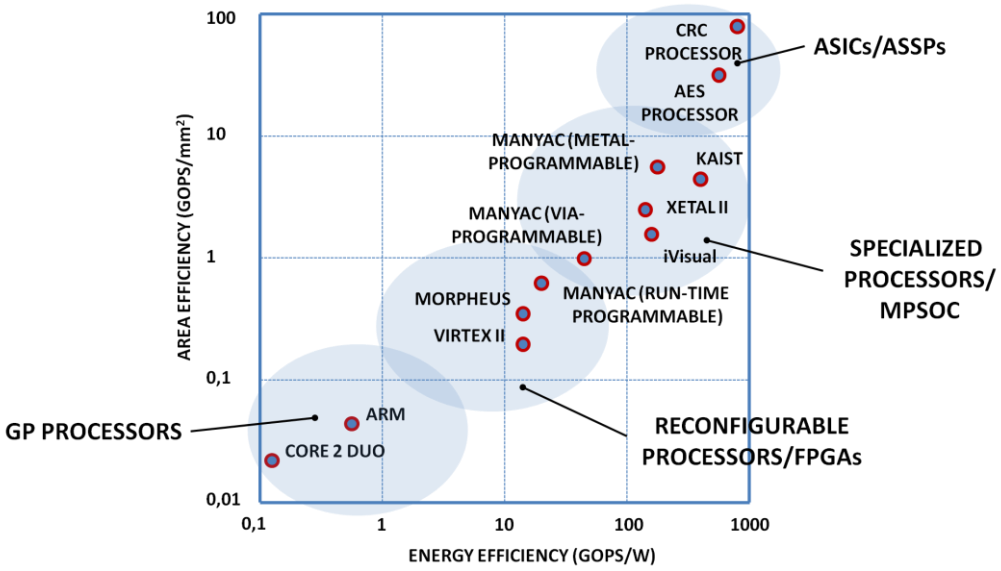


Figure 4.6: *Energy efficiency vs. Area Efficiency of computational devices for signal processing.*

meaning that the area form an overhead bigger than the power for the analyzed technologies.

Figure 4.6 shows a graphic viewof the computational platforms discussed in this thesis in terms of area efficiency and energy efficiency. Although the positioning of each device in the graph should be considered as purely qualitative, as both energy efficiency and area efficiency are affected by "noise" caused by many factors, it gives a good view of the flexibility/efficiency trade-off in the field of computing devices. In this scenario, the most efficient devices are positioned on the top-right of the graph, while the efficiency decreases towards the bottom-left area. Considering the Morpheus platform and the Manyac platform equipped with run-time configurable hardware accelerators, it is possible to notice that the Manyac platform provides slightly better efficiency both considering energy and area. The main cause of this difference consists of the full-custom technique with manual optimization utilized for the implementation of run-time configurable gate array of the Manyac platform, and on the external memory access of the Morpheus platform that limits the computational power of its reconfigurable engines. On the other hand, the Manyac platform with the via-programmable and especially metal-programmable gate arrays provide

better performance in terms of both energy and area efficiency. More in general, the most flexible devices such as processors are positioned on the bottom-left area of the graph due to their general-purposity and intrinsic overheads in the execution of signal processing applications. On the other hand, the most specialized devices such as ASICs/ASSPs are those providing better performance, due to the high computational power of their accelerators, but also due to their high efficiency achieved by application-specific optimization performed at all levels of design.

4.3 Cost of Manufacturing

This section analyzes the cost of manufacturing of the different implementations of the proposed platform that utilize the three analyzed configuration technologies: run-time programmable, via-programmable and metal-programmable. In order to perform the analysis we consider three instances of the Manyac platform with equivalent computation capabilities, whose accelerators are implemented with the three different configuration technologies.

The cost of manufacturing of integrated circuit technologies are often categorized into *fixed costs* and *variable costs*. *Fixed costs* are those costs that are independent on the number of pieces realized for a given IC, while *variable costs* are those which depend on the number of pieces realized for a given IC implementation, in other words on the market volume of a product. Fixed costs of IC manufacturing are includes the cost of the masks realized to print each layer of the IC on the silicon wafer. On the other hand, variable costs are mainly dominated by the costs of lithography, by means of costs of "printing" the realized masks on the silicon wafers. In order to perform an analysis of costs of the proposed technologies we utilize the cost of ownership model proposed by Paramanik et. al. [81].

Reticle Strategy

In order to describe the utilized mask cost model, we first provide a brief description of the reticle strategy adopted in the IC manufacturing. A reticle is printed on each silicon wafer, which is utilized to alienate the masks over the wafer surface. Each field of the reticle contains one or more dies, and all dies in a reticle are printed at the same time. In this context, our first assumption is to utilize the reticle strategy believed to achieve the highest printing throughput: the *single-layer reticle on a large field* (SRL-L). In this scenario, the overall number of fields within a reticle (i.e., number of exposures per wafer) is calculated as:

$$Num_{fields} = \frac{\pi d^2}{4 \cdot S_{field}}$$

Assuming shape of the die as square, the overall number of dies within a wafer can be estimated as:

$$Num_{dies} = \frac{\pi d^2}{4 \cdot S_{die}} - \frac{1}{\sqrt{2 \cdot S_{die}}}$$

Where d is the wafer diameter, S_{field} is the area of the field, and S_{die} is the area of the die.

Cost of Masks

In order to analyze the cost of the overall mask set of a given technology, we consider a subdivision into three main categories of masks, which depends on the technology utilized for printing each masks sub-set:

- very critical layers (e.g., 193 nm)
- critical layers (e.g., 248 nm)
- non critical layers (e.g., I-line)

Technology node	*65nm*	*45nm*	*32nm*	*22nm*
Mask cost per layer (very critical)[$]	37,580	75,161	150,323	300,647
Mask cost per layer (critical)[$]	9,395	18,790	37,580	75,161
Mask cost per layer (non critical)[$]	3,355	6,710	13,421	26,843
Number of layers (very critical)	11	11	12	13
Number of layers (critical)	11	12	12	13
Number of layers (non critical)	11	12	13	13
Cost of exposure (very critical)[$]	2,8	3,44	4,22	4,48
Cost of exposure (critical)[$]	1,57	2,06	2,53	2,68
Cost of exposure (non critical)[$]	0,56	0,69	0,84	0,89
Yield	90%	80%	70%	60%

Table 4.3: *Parameters of the Manufacturing Cost Model.*

According with the utilized model, the overall mask cost is calculated as:

$$Cost_{maskset} = c_{m,vc} \cdot n_{m,vc} + c_{m,c} \cdot n_{m,c} + c_{m,nc} \cdot n_{m,nc}$$

Where $c_{m,vc}$, $c_{m,c}$, $c_{m,nc}$, is the cost of very critical, critical and non-critical masks, and $n_{m,vc}$, $n_{m,c}$, $n_{m,nc}$, is the number of very critical, critical and non-critical masks, respectively.

Pramanik et.al. estimated the cost of 90nm mask at the introduction year. According with our SRL-L assumption, which leads to a field size of 25x25mm^2 the mask costs per layer are 112.000$, 28.000$, 10.000$ for very critical, critical, non-critical masks, respectively. In order to scale the cost of masks for the more recent technology nodes, we utilized the following assumptions: (a) mask cost doubles at the introduction year of every technology node, (b) mask cost decreases by 20% every year, (c) the introduction year of 90*nm* technology node is 2003. The assumptions give mask cost in 2011 as:

$$Cost_{mask,i} = Cost_{mask,90} \times 2 \times i \times (0.8)^{2011-2003}$$

Where *i* is 2, 3, 4, 5 for 65*nm*, 45*nm*, 32*nm*, 22*nm*, respectively and $Cost_{mask,90}$ is the 90*nm* technology node initial mask cost. The overall number of mask layers are predicted from ITRS 2007, while we assume the portion of very

critical, critical and non-critical layers is equal. Table 4.3 show the calculated mask set costs and the related number of masks.

Cost of Lithography

The overall cost of lithography is proportional to the number of wafers developed n_w and to the cost of lithography of a single wafer. Considering a single wafer, the lithography cost depends on the cost of a single exposure (C_e), the number of exposures per wafer (n_e) (i.e., number of fields within a wafer), and the number of mask layers (n_m). The lithography cost for producing a wafer can be calculated as:

$$Cost_{litho,w} = n_e \cdot (c_{e,vc} n_{m,vc} + c_{e,c} \cdot n_{m,c} + c_{e,nc} \cdot n_{m,nc})$$

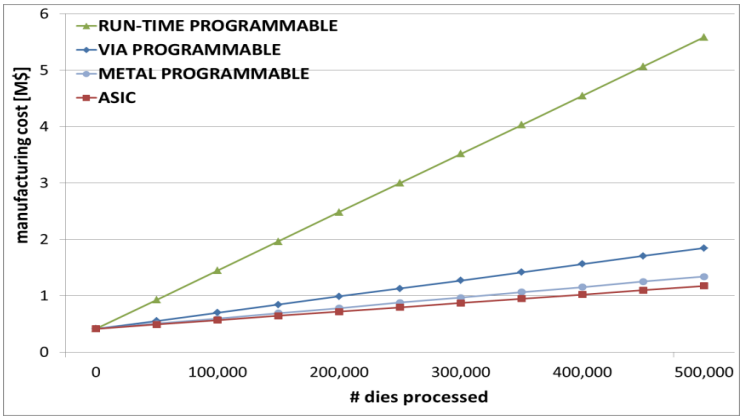
The cost of a single exposure for the 90nm technology node is assumed to as 2.5$, 1.5$, 0.5$ for very critical, critical, and noncritical layers, respectively, based on the Parmanik estimation [81]. In order to estimate the cost of lithography for the technology generations, we scaled the 90nm cost of exposure, according with lithography tool cost. The cost of lithography tool is assumed as 40M$, 49M$, 52M$ for 45nm, 32nm, 22nm, respectively. The cost of 65nm lithography tool is estimated from curve-fitting.

Analysis of overall manufacturing costs

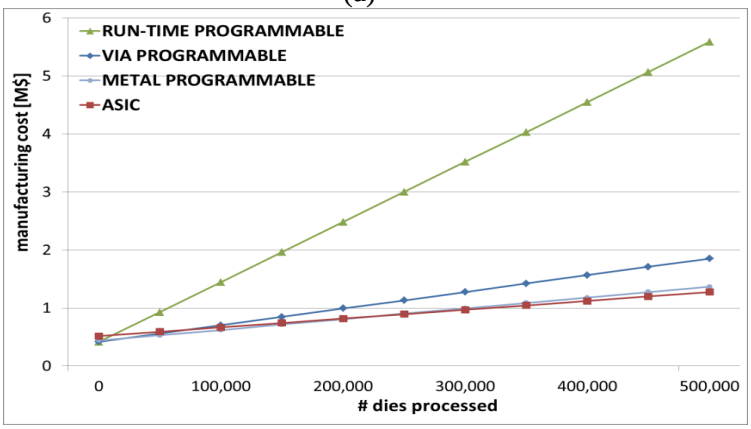
Finally, the overall cost of manufacturing consists of the overall cost of the maskset required and the overall lithography cost, which depends on the number of processed wafers. The cost of manufacturing for n dies can be calculated as:

$$Cost_{n,dies} = Cost_{maskset} + \frac{Cost_{litho}}{Y}$$

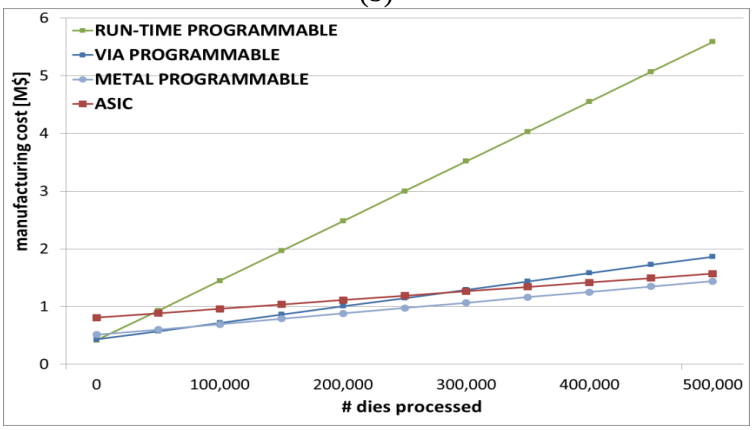
Where Y is the lithography yield.
In order to analyze manufacturing cost of the platform we assume the application specific accelerators of different implementations of the platform



(a)



(b)



(c)

Figure 4.7: *Manufacturing cost of platform implementation utilizing the different configurable gate arrays as hardware accelerators assuming 1 product (a), 5 product (b), and 10 product (c) realized utilizing the same architectural template.*

realized with run-time programmable, via-programmable, metal-programmable, and ASIC gate arrays, respectively. Considering fixed costs, each of the proposed technology requires a different number of masks to modify the application specific acceleration of the platform. For each implementation of the platform, the realization of a first product require the overall mask set, while the realization of a second product require a number of masks that depends on the configuration approach utilized, so that more flexible is the solution the lower is the mask cost. Considering the lithography, the overall cost depends on throughput, thus on how many dies fit a wafer, so that the smaller solution provides lower volume costs.

Figure 4.7 shows the results of the analysis, where the number of different products realized through customization for each implementation is 1 (a), 5 (b), 10 (c). The results shows that the proposed customization technologies demonstrate cost-effective even for relatively small number of customization. Figure 4.7a shows the cost of production of one product, utilized as reference. In this case it is evident that the manufacturing fixed cost of all the proposed solutions are equal as a complete set of masks has to be realized in all cases. On the other hand the CMOS implementations takes benefits on volumes due to the smaller area of the related implementation. Moving the focus on Figure 4.7b it is possible to notice that when moving to 5 products realized utilizing the proposed approach, the metal-programmable customization appears as the most appealing for the low market, while is surpassed by the ASIC customization for volumes over 250.000 pieces per product. Finally, analyzing Figure 4.7c, which refer to the realization of 10 products, the situation changes again. In this scenario the via-programmable solution appears the most suitable solution for low market volumes, while is being surpassed by the metal- programmable customization for volumes higher than 20.000 pieces per product. In this scenario the ASIC implementation appears convenient only for extremely high volumes. On the other hand, considering the 65nm technology node, the run- time programmable solution appears appealing only for very low market volumes, or for a very high number of customization.

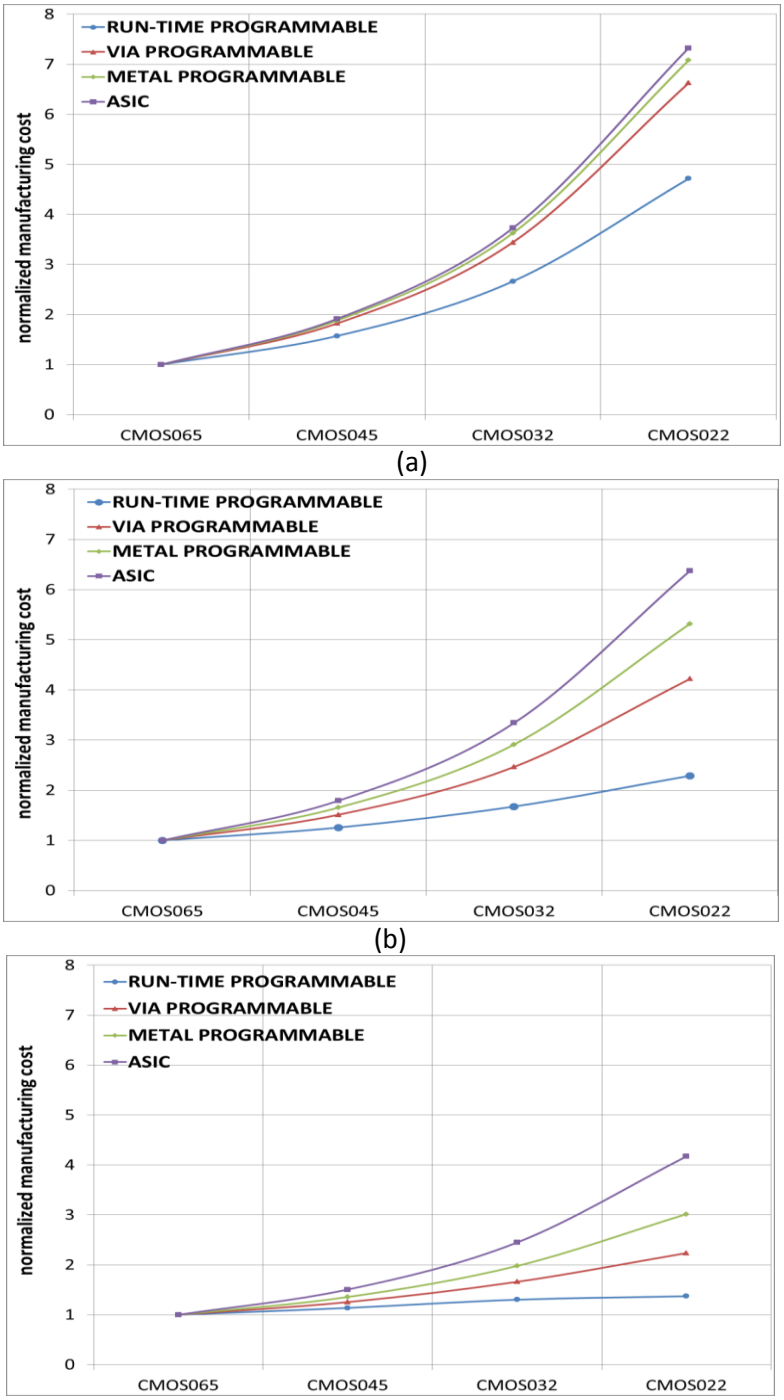


Figure 4.8: *Manufacturing cost of platform implementation utilizing the different configurable gate arrays in different technology nodes. (a) A market volume of 5.000 pieces is assumed for 1 product. (b) A market volume of 50.000 pieces is assumed for 5 products with the same architecture template. (c), and 10 product (c) (b) A market volume of 250.000 pieces is assumed for 5 products with the same architecture template.*

Perspectives

In order to analyze perspectives of configurable and reconfigurable solutions, we consider now the cost of products manufacturing for more recent technology nodes, according to the model described previously in this section. In this context we assume the area of the dies to scale according to the half-pitch of each technology node, and the parametric yield affecting the different technologies to scale according with the 65nm learning curve [82]. We considered three different scenarios. Figure 4.8a shows the manufacturing cost for realizing 5.000 pieces of one product. Figure 4.8b shows the manufacturing cost for realizing 50.000 pieces of 5 products. Figure 5.8c shows the manufacturing cost for realizing 250.000 of 10 products. Data are normalized to the manufacturing cost of the 90nm technology node. In general, results show that the spread between the different customization technologies caused by their area overhead with respect to the ASIC implementation is expected to drop. This effect is mainly caused by the continuous rise of the mask costs that, especially during the first years of production for each technology node remain prohibitively high. On the other hand, the variable cost associated with lithography, even if affected by parametric yield are expected to be mitigated by the larger lithography throughput achieved by realizing smaller die sizes. Considering the proposed customization strategies, it appears evident how more flexible solutions are expected to be appealing for even larger market volumes. In particular, considering the run-time programmable approach, results show that even if at the moment they do not appear convenient for the mid and large scale production, it can be expected that they will become an appealing and cost-effective solution for the future of signal processing systems.

Chapter 5

5 Conclusion

In this thesis, a computational paradigm based on the cooperation between multi-core computing and configurable hardware acceleration has been presented, utilizing different run-time programmable and silicon-structured configuration technologies for the specialization of the platform. This computational paradigm first implies a partitioning of the applications among the available computational cores, being either homogeneous or heterogeneous, and the successive migration of kernels from the software programmable processors of the platform to the customizable hardware accelerators. The performance achieved by such kind of computational platforms depends on a careful balance between the portions of applications being executed on hardware and software, but also on an accurate selection of the architectural parameters of the platform, as such kind of solution is much more sensitive to such parameters than the software-based solutions. For this reason, such kind of computational platform needs to be accompanied with design environments that allow the user to evaluate the trade-off spectrum among the parameters mentioned above.

In the first part of the thesis a reconfigurable digital signal processor with a heterogeneous set of computing units, featuring different computational paradigms and granularities has been presented. An accurate analysis of the platform performed through the implementation of signal processing applications has been performed. The analysis evidenced that the Morpheus platform is able to match computational requirements of most of applications. The input/output bandwidth of the platform and reconfiguration latencies emerged as main bottlenecks. If not properly managed they can cause relevant performance degradation with respect to the ideal case of reconfigurable engines that exploits all their computational power. The mapping of applications on the most suitable reconfigurable engine plays a crucial role in the performance achieved by the platform. On the other hand, the intrinsic

heterogeneity of the devices require to the final user the knowledge of different kind of programming and hardware description language.
A second approach aimed at the exploitation of parallelism at both thread-level and data-instruction level has been further presented. The Manyac platform joins the benefits and flexibility typical of software-programmable multi-processor systems with the performance and energy efficiency typical of hardware-based platforms. A peculiarity of such a platform is that of being customizable with three kinds of configuration technologies: run-time programmable, via-programmable and metal-programmable. An analysis of the Manyac platform proved that the choice of the configuration technology only minimally influences performance, which is rather much more sensitive to the trade-offs between implementation strategies of the hardware accelerators and to the architectural parameters of the platform. On the other hand, this choice has a large impact on the power consumption and area of the platform.
The proposed solutions have been finally evaluated from the quantitative point of view and compared against the state of the art in terms of programmability, area/energy efficiency and cost of manufacturing. Although utilization of run time-configurable logic provides ASIC-like performance, it pays most of its overheads in terms of power and area. Considering the power, this means that utilization of reconfigurable logic is likely unsuitable for applications with high portability requirements. On the other hand, the area overheads of reconfigurable technology have a direct impact on the manufacturing cost. The analysis shows that even if at the moment it does not appear convenient for mid and large scale product volumes it is possible to expect that they may be appealing and cost-effective solutions in the future technology nodes. In the mean time, the utilization of structured solutions (i.e. via- and metal-programmable) as silicon platforms for hardware accelerators in multi-processor systems provide highly efficient figures (almost ASIC-like), and a huge reduction of manufacturing costs even when spread over a small number of products based on the customization of the same platform.

6 Publications

The following papers have been co-authored and are accepted for publication:

D. Rossi, F. Campi, A. Deledda, S. Spolzino, S. Pucillo, "A Heterogeneous Digital Signal Processor Implementation for Dynamically Reconfigurable Computing", *Custom Integrated Circuit Conference (CICC)*, 2009.

D. Rossi, F. Campi, A. Deledda, C. Mucci, S. Pucillo, S. Whitty, R. Ernst, S. Chevobbe, S. Guyetant, M. Kühnle, M. Hübner, J. Becker and W. Putzke-Roeming, "A Multi-Core Signal Processor for Heterogeneous Reconfigurable Computing", *International Symposium on System-on-Chip (SOC)*, 2009.

F. Campi, R. König, M. Dreschmann, M. Neukirchner, D. Picard, M. Jüttner, E. Schüler, A. Deledda, D. Rossi, A. Pasini, M. Hübner, J. Becker, R. Guerrieri, "RTL-to-Layout Implementation of an Embedded Coarse Grained Architecture for Dynamically Reconfigurable Computing in Systems-on-Chip", *International Symposium on System-on-Chip (SoC)*, 2009.

D. Rossi, F. Campi, S. Spolzino, S. Pucillo, R. Guerrieri, "A Heterogeneous Digital Signal Processor for Dynamically Reconfigurable Computing", *IEEE Journal of Solid-State Circuits (JSSC)*, 2010.

A. Grasset, P. Millet, P. Bonnot, S. Yehia, W. Putzke-Roeming, F. Campi, A. Rosti, M. Huebner, N. S. Voros, D. Rossi, "The MORPHEUS Heterogeneous Dynamically Reconfigurable Platform", *International Journal of Parallel Programming (IJPP)*, 2011.

D. Rossi, C. Mucci, F. Campi, S. Spolzino, L. Vanzolini, H. Sahlbach, S. Whitty, R. Ernst, W. Putzke-Röming, and R. Guerrieri, "Application Space Exploration of a Heterogeneous Run Time Configurable Digital Signal Processor", *IEEE Transactions on Very Large Scale Integration (TVLSI) Systems*, 2012.

7 References

[1] B. Ackland, A. Anesko, D. Brinthaupt, S. J. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C. J. Nicol, J. H. O'Neill, J. Othmer, E. Sackinger, K. J. Singh, J. Sweet, C. J. Terman, and J. Williams, "A single-chip, 1.6-billion, 16-b MAC/s multiprocessor DSP", IEEE J. Solid-State Circuits, vol. 35, no. 3, pp. 412–424, Mar. 2000.

[2] C-5 Network Processor Architecture Guide, C-Port Corp., North Andover, MA, May 31, 2001. [Online]. Available: http://www.freescale.com.

[3] S. Dutta, R. Jensen, and A. Rieckmann, "Viper: A multiprocessor SOC for advanced set-top box and digital TV systems," IEEE Des. Test. Comput., vol. 18, no. 5, pp. 21–31, Sep./Oct. 2001.

[4] OMAP5912 Multimedia Processor Device Overview and Architecture Reference Guide, Texas Instruments Inc., Dallas, TX, Mar. 2004. [Online]. Available: http://www.ti.com.

[5] A. Artieri, V. D'Alto, R. Chesson, M. Hopkins, and M. C. Rossi, Nomadik—Open Multimedia Platform for Next Generation Mobile Devices, 2003. technical article TA305. [Online]. Available: www.st.com.

[6] R.W. Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective", Proceedings of the conference on Design, automation and test in Europe, 2001, pp. 642-649.

[7] www.altera.com

[8] www.xilinx.com

[9] W. S. Carter, K. Duong, R. H. Freeman, H. C. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. go and S. L. Sze, "A user programmable reconfigurable logic array", IEEE 1986 Custom Integrated Circuits Conference, pp. 233 , 1986.

[10] R. Razdan; M.D. Smith A High-Performance Microarchitecture with Hardware-Programmable Functional Units, Proceedings of IEEE MICRO, Nov. 1994.

[11] T.J. Callahan, J.R. Hauser, J. Wawrzynek, "The Garp architecture and C compiler", IEEE Computer, April 2000.

[12] S. Vassiliadis, S. Wong, S. Gaydadjiev, K. Bertels, G. Kuzmanov, E.M. Panainte, "The MOLEN Polymorphic Processor", IEEE Transactions on Computers, Nov. 2004.

[13] R.W.Hartenstein, "Reconfigurable Computing Architectures and Methodologies for System-on-Chip", International Symposium on Systems-On-Chip, 2001.

[14] S. Vassiliadis, G. Gaydadjiev, K. Bertels, and E. Moscu Panainte, "The Molen Programming Paradigm", Proceedings of the Third Int'l Workshop Systems, Architectures, Modeling, and Simulation, pp. 1-7, July 2003.

[15] www.aboundlogic.com

[16] S . C. Goldstein et al.: PipeRench: A Coprocessor for Streaming Multimedia Acceleration; Proc. ISCA.99, Atlanta, May 2-4, 1999.

[17] H. Singh, et al.: *Morphosys: An Integrated Re-configurable Architecture*, Proceedings of the dAT0 RTO Symp. on S stem Concepts and Inteiration, Monterey, CA, USA, April 20-22, 1968.

[18] A. Marshall et al.:. "A Reconfigurable Arithmetic Array for Multimedia Applications", Proc. ACM/SIGbA FPGA'99, Monterey, Feb. 21-23, 1999.

[19] F. Campi, A. Deledda, M. Pizzotti, L. Ciccarelli, C. Mucci, A. Lodi, L. Vanzolini, A. Vitkovski, "A dynamically adaptive DSP for heterogeneous reconfigurable platforms", *IEEE International Conference on Design Automation and Test in Europe (DATE'07)*, Apr. 2007, pp. 1-6.

[20] A. Lodi, A. Cappelli, M. Bocchi, C. Mucci, M. Innocenti, C. D. Bartolomeis, L. Ciccarelli, R. Giansante, A. Deledda, F. Campi, M. Toma, and R. Guerrieri, "XiSystem: a XiRisc-based SoC with reconfigurable IO module", *IEEE Journal of Solid-State Circuits*, vol. 41, no. 1, pp. 85–96, Jan. 2006.

[21] M. Vorbach, J. Becker, "Reconfigurable Processor Architectures for Mobile Phones*", Proceedings of the IEEE Parallel and Distributed Processing Symposium*, April 2003, pp. 6.

[22] C.Brunelli, F.Garzia, D.Rossi, J.Nurmi, "A coarse-grain reconfigurable architecture for multimedia applications supporting subword and floating-point calculations", Elsevier Journal of System Architecture, Vol. 56 , Issue 1, Jan 2010, pp. 38-47.

[23] F. Garzia, W. Hussain, J. Nurmi: CREMA: A coarse-grain reconfigurable array with mapping adaptiveness. FPL 2009, 708-712.

[24] G, William, Lusk. Ewing, S. Anthony, "Using MPI: portable parallel programming with the message-passing interface", Cambridge, MA, USA: MIT Press Scientific And Engineering Computation Series, 1994.

[25] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald, "Parallel Programming in OpenMP", Morgan Kaufmann, 2000.

[26] T. R. Halfhill, "Parallel Processing with CUDA," *Microprocessor Report*, Jan. 2008.

[27] *OpenCL Specification v1.0r48*, Khronos Group, Oct. 2009 [Online], Available: http://www.khronos.org/registry/cl/

[28] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, J. Zook, "TILE64 processor: A 64-core SoC with mesh interconnect", *IEEE International Solid-State Circuits Conference (ISSCC'08)*, Feb. 2008, pp. 88-89.

[29] A. Duller, G. Panesar, and D. Towner, “Parallel Processing — the picoChip way!”, *Communicating Processing Architectures, 2003*, pp. 125–138.

[30] D.N. Truong, W.H. Cheng, T. Mohsenin, Yu Zhiyi, A.T. Jacobson, G. Landge, M.J. Meeuwsen, C. Watnik, A.T. Tran, X. Zhibin, E.W. Work, J.W. Webb, P.V. Mejia, B.M. Baas, “A 167-Processor Computational Platform in 65 nm CMOS”, *IEEE Journal of Solid-State Circuits*, vol. 44, no. 4, April 2009, pp. 1130-1144.

[31] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, K. Yazawa, "The Design and Implementation of a First-Generation CELL Processor", *IEEE Journal of Solid-State Circuits*, vol. 41, no. 1, January 2006, pp. 179-196.

[32] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture", *IEEE Micro*, vol. 28, no. 2, Mar./Apr. 2008, pp. 39-55.

[33] H. Nikolov, T. Stefanov, E. Deprettere, “Systematic and automated multi-processor system design, programming, and implementation”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, *27*(3), pp 542 - 555, March 2008.

[34] T. Stefanov *et al.*, “System design using Kahn process networks: The Compaan/Laura approach”, in *Proc. DATE*, Feb. 2004, pp. 340–345.

[35] A. Turjan *et al.*, “Translating affine nested-loop programs to process networks,” in *Proc. CASES*, Sep. 2004, pp. 220–229.

[36] B. Kienhuis *et al.*, “Compaan: Deriving process networks from Matlab for embedded signal processing architectures,” in *Proc. CODES*, May 2000, pp. 13–17.

[37] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere, “LAURA: Leiden architecture research and exploration tool,” in *Proc. FPL*, Sep. 2003, pp. 911–920.

[38] M. J. Rutten *et al.*, “A heterogeneous multiprocessor architecture for flexible media processing,” *IEEE Des. Test Comput.*, vol. 19, no. 4, pp. 39–50, Jul./Aug. 2002.

[39] D. Lyonnard *et al.*, “Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip”, in *Proc. DAC*, Jun. 2001, pp. 518–523.

[40] L. Gauthier, S. Yoo, and A. Jerraya, “Automatic generation and targeting of application specific operating systems and embedded systems software”, *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 11, pp. 1293–1301, Nov. 2001.

[41] Xilinx, Inc., *Xilinx Platform Studio and the Embedded Development Kit*. EDK version 8.1i edition. [Online]. Available: www.xilinx.com/ise/embedded_design_prod/platform_studio.htm

[42] Altera, Inc., (2005, Dec.). *Quartus II Handbook Volume 4: SOPC Builder*. [Online]. Available: www.altera.com/literature/quartus2/lit-qts-sopc.jsp

[43] A. Papakonstantinou, K. Gururaj, J.A. Stratton, D. Chen, J. Cong, W.-M.W. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs", Symposium on Application Specific Processors, 2009, pp. 35 - 42.

[44] http://www.altera.com/literature/wp/wp-01173-opencl.pdf

[45] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, A. Sangiovanni-Vincentelli, "System-Level Design: Orthogonalization of Concerns and Platform-Based Design", IEEE Transactions on Computer-Aided Design of integrated Circuits, Vol.19, n. 12, Dec. 2000, pp. 1523-1543.

[46] E.A. Lee, A. Sangiovanni-Vincentelli, "Component-based design for the future, Design", Automation & Test in Europe Conference & Exhibition (DATE), 2011, pp 1 - 5.

[47] P. Ienne and R. Leupers, Eds., *Customizable Embedded Processors*. San Francisco, CA: Morgan Kaufmann, 2006.

[48] P. Marwedel, "The MIMOLA design system: Tools for the design of digital processors," in *Proc. 21st Des. Autom. Conf.*, 1984, pp. 587–593.

[49] S. Kobayashi, K. Mita, Y. Takeuchi, and M. Imai, "Rapid prototyping of JPEG encoder using the ASIP development system: PEAS-III," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, Apr. 2003, vol. 2, pp. 485–488.

[50] A. Hoffman, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, "A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language," *IEEE Trans. Comput.-Aided Desgn Integr. Circuits Syst.*, vol. 20, no. 11, pp. 1338–1354, Nov. 2001.

[51] C. Rowen, *Engineering the Complex SoC: Fast, Flexible Design With Configurable Processors*. Upper Saddle River, NJ: Prentice-Hall, 2004.

[52] R. Taylor and P. Morgan, "Using coprocessor synthesis to accelerate embedded software," in *Proc. Embedded Syst. Conf.*, 2005.

[53] http://www.mentor.com/esl/catapult/overview/

[54] http://www.chipvision.com/products/index.php

[55] D. Goodwin and D. Petkov, "Automatic generation of application specific processors," in *Proc. CASES*, 2003, pp. 137–147.

[56] D. Burger, T. Austin *The SimpleScalar Tool Set, Version 2.0*, www.simplescalar.com

[57] M. Weinhardt and W. Luk *Pipeline Vectorization*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Feb. 2001, pp. 234-248.

[58] B. Mei, S. Vernalde, D. Verkest, H. De Man, R. Lauwereins *DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architecture*, International Conference on Field Programmable Technology, Dec. 2002.

[59] D. Rossi, F. Campi, A. Deledda, C. Mucci, S. Pucillo, S. Whitty, R. Ernst, S. Chevobbe, S. Guyetant, M. Kühnle, M. Hübner, J. Becker, W. Putzke-Roeming "A Multi-Core Signal Processor for Heterogeneous Reconfigurable Computing", *IEEE International Symposium on System-on-Chip (SoC'09)*, Oct. 2009, pp. 106-109.

[60] M. Kuehnle, "An Interconnect Strategy for a Heterogeneus, reconfigurable SoC", IEEE Design & Test of Computers, 2008.

[61] M.Coppola et al, "Spidergron: a novel on-chip communication network", IEEE SOC 2004.

[62] A. Deledda, C. Mucci , A. Vitkovski , P. Bonnot, A. Grasset, P. Millet, M. Kuehnle, F. Ries, M. Huebner, J. Becker, M. Coppola, L. Pieralisi, R. Locatelli, G. Maruccia., F. Campi, T. DeMarco, "Design of a HW/SW Communication Infrastructure for a heterogeneous reconfigurable processor", *IEEE International Conference on Design, Automation, and Test in Europe (DATE'08)*, 2008, pp. 1352-1357.

[63] F. Campi, R. König, M. Dreschmann, M. Neukirchner, D. Picard, M. Jüttner, E. Schüler, A. Deledda, D. Rossi, A.Pasini, M. Hübner J. Becker, R. Guerrieri, "RTL-to-Layout Implementation of an Embedded Coarse Grained Architecture for Dynamically Reconfigurable Computing in Systems-on-Chip", *IEEE International Symposium on System-on-Chip (SoC'09)*, Oct. 2009.

[64] J. M. P. Cardoso, M. Weinhardt, "Fast and Guaranteed C Compilation onto the PACT-XPP Reconfigurable Computing Platform", Proceedings of the 10 th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, Jan. 2003, pp. 291 – 292.

[65] C. Mucci, C. Chiesa, A. Lodi, M. Toma, F. Campi, "A C-based Algorithm Development Flow for a Reconfigurable Processor Architecture", Proceedings on the IEEE Symposium on System on Chip (SoC2003), Tampere (Finland), Nov. 2003.

[66] A. Grasset, P. Millet, P. Bonnot, S. Yehia, W. Putzke-Roeming, F. Campi, A. Rosti, M. Huebner, N. Voros, D. Rossi, H. Sahlbach, R. Ernst, "The MORPHEUS Heterogeneous Dynamically Reconfigurable Platform", *International Journal of Parallel Programming*, Feb. 2011, Vol. 39, pp 328-356.

[67] U. Pross,S. Goller, E. Markert, M. Juttner, J. Langer, U. Heinkel, J. Knablein, A. Schneider, "Demonstration of an in-band reconfiguration data distribution and network node reconfiguration", *IEEE International Conference on Design Automation and Test in Europe (DATE'08)*, Mar. 2008, pp. 1444-1449.

[68] http://www.itu.int

[69] NIST Specification for the ADVANCED ENCRYPTION STANDARD (AES), FIPS PUBS 197, November 26, 2001.

[70] C. Mucci, L. Vanzolini, A. Lodi, A. Deledda, R. Guerrieri, F. Campi, M. Toma, "Implementation of AES/Rijndael on a dynamically reconfigurable

architecture", *IEEE International Conference on Design, Automation and Test in Europe (DATE'07)*, Apr. 2007, pp. 1–6.

[71] C. Mucci, L. Vanzolini, F. Campi, G. Gaillat, A. Deledda, "Intelligent cameras and embedded reconfigurable computing: a case-study on motion detection", *IEEE International Symposium on System on Chip*, Oct. 2007, pp.1-4.

[72] C. Mucci, L. Vanzolini, I. Mirimin, D. Gazzola, A. Deledda, S. Goller, J. Knaeblein, A. Schneider, L. Ciccarelli, F. Campi, "Implementation of Parallel LFSR-based Applications on an Adaptive DSP featuring a Pipelined Configurable Gate Array", *IEEE International Conference on Design Automation and Test in Europe,* Mar. 2008, pp. 1444-1449.

[73] J.H. Derby, "*High-speed CRC computation using state-space transformations*", Global Telecommunications Conference, vol.1, Nov. 2001, pp. 166 – 170.

[74] S. Whitty, H. Sahlbach, R. Ernst ,W. Putzke-Roming , "Mapping of a film grain removal algorithm to a heterogeneous reconfigurable architecture", *IEEE International Conference on Design, Automation and Test in Europe (DATE'09)*, Apr. 2009, pp. 27-32.

[75] E. Markert, E. Billich, C. Tischendorf, U. Pross, T. Leibelt, U. Heinkel, J. Knäblein, A. Schneider, "An in-band reconfigurable network node based on a heterogeneous platform", Conference on Design and Architectures for Signal and Image Processing (DASIP), 2010, pp. 15 - 20.

[76] W. Fornaciari, F. Salice, U. Bondi, and E. Magini, "Development cost and size estimation starting from high-level specifications", *Proceedings of the ninth international symposium on Hardware/software codesign*, 2001, pp. 86–91.

[77] http://www.spr.com/programming-languages-table.html

[78] J. Merrill, "GENERIC and GIMPLE: A New Tree Representation for Entire Functions", *Proceedings of the GCC Developers Summit*, pp. 171-180, May 25-27, 2003.

[79] R. Leupers, "Compiler Design Issues for Embedded Processors", *IEEE Design & Test of Computers*, July–August 2002, pp. 51-58.

[80] http://impact.crhc.illinois.edu/index.php

[81] D. Pramanik, H. H. Kamberian, C. J. Progler, M. Sanie and D. Pinto, "Cost Effective Strategies for ASIC Masks", Proc. SPIE Cost and Performance in Integrated Circuit Creation, Vol. 5043, 2003, pp. 142–152.

[82] R. Scott Mackay, H. Kamberian, Y. Zhang, "Methods to reduce lithography costs with reticle engineering", Microelectronic Engineering, Volume 83, Issues 4–9, April–September 2006, pp. 914-918.