Università degli Studi di Cagliari

DOTTORATO DI RICERCA

INGEGNERIA ELETTRONICA ED INFORMATICA

Ciclo XXV

Combining FPGA prototyping and high-level simulation approaches for Design Space Exploration of MPSoCs

Settore/i scientifico disciplinari di afferenza

ING-INF/01 ELETTRONICA

Presentata da:	SEBASTIANO POMATA
Coordinatore Dottorato	PROF. ALESSANDRO GIUA
Tutor/Relatore	PROF. LUIGI RAFFO

Esame finale anno accademico 2011 – 2012

Ph.D. in Electronic and Computer Engineering
Dept. of Electrical and Electronic Engineering
University of Cagliari

Combining FPGA prototyping and high-level simulation approaches for Design Space Exploration of MPSoCs

Sebastiano POMATA

Advisor: Prof. Luigi RAFFO
Curriculum: ING-INF/01 Elettronica

XXV Cycle
May 2013

Ph.D. in Electronic and Computer Engineering
Dept. of Electrical and Electronic Engineering
University of Cagliari

Combining FPGA prototyping and high-level simulation approaches for Design Space Exploration of MPSoCs

Sebastiano POMATA

Advisor: Prof. Luigi RAFFO
Curriculum: ING-INF/01 Elettronica

XXV Cycle
May 2013

Dedicated to family, friends, colleagues.

Abstract

Modern embedded systems are parallel, component-based, heterogeneous and finely tuned on the basis of the workload that must be executed on them. To improve design reuse, Application Specific Instruction-set Processors (ASIPs) are often employed as building blocks in such systems, as a solution capable of satisfying the required functional and physical constraints (e.g. throughput, latency, power or energy consumption etc.), while providing, at the same time, high flexibility and adaptability. Composing a multi-processor architecture including ASIPs and mapping parallel applications onto it is a design activity that require an extensive Design Space Exploration process (DSE), to result in cost-effective systems. The work described here aims at defining novel methodologies for the application-driven customizations of such highly heterogeneous embedded systems. The issue is tackled at different levels, integrating different tools.

High-level event-based simulation is a widely used technique that offers speed and flexibility as main points of strength, but needs, as a preliminary input and periodically during the iteration process, calibration data that must be acquired by means of more accurate evaluation methods. Typically, this calibration is performed using instruction-level cycle-accurate simulators that, however, turn out to be very slow, especially when complete multi-processor systems must be evaluated or when the grain of the calibration is too fine, while FPGA approaches have shown to perform better for this particular applications.

FPGA-based emulation techniques have been proposed in the recent past as an alternative solution to the software-based simulation approach, but some further steps are needed before they can be effectively exploited within architectural design space exploration. Firstly, some kind of technology-awareness must be introduced, to enable the translation of the emulation results into a pre-estimation of a prospective ASIC implementation of the design. Moreover, when performing architectural DSE, a significant number of different candidate design points has to be evaluated and compared. In this case, if no countermeasures are taken, the advantages achievable with FPGAs, in terms of emulation speed, are counterbalanced by the overhead introduced by the time needed to go through the physical synthesis and implementation flow.

Developed FPGA-based prototyping platform overcomes such limitations, enabling the use of FPGA-based prototyping for micro-architectural design space exploration of ASIP processors. In this approach, to increase the emulation speed-up, two different methods are proposed: the first is based on automatic instantiation of additional hardware modules, able to reconfigure at runtime the prototype, while the second leverages manipulation of application binary code, compiled for a custom VLIW ASIP architecture, that is transformed into code executable on a different configuration. This allows to prototype a whole set of ASIP

solutions after one single FPGA implementation flow, mitigating the afore-mentioned overhead.

A short overview on the tools used throughout the work will also be offered, covering basic aspects of Intel-Silicon Hive ASIP development toolchain, SESAME framework general description, along with a review of state-of-art simulation and prototyping techniques for complex multi-processor systems.

Each proposed approach will be validated through a real-world use case, confirming the validity of this solution.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Main objectives and thesis organization

A common feature of modern embedded systems is the need for highly optimized application-specific processing elements. Application Specific Instruction-set Processors (ASIPs) are often the only solution to the required functional and physical constraints able to provide, at the same time, high flexibility and programmability.

These processors are typically performance- and power-optimized for a specific application domain. The optimizations in terms of extension of the processor instruction set often include vector processing and SIMD support, complex domain-specific arithmetic operations (e.g. MAC for digital signal processing). In terms of architecture organization, it is not infrequent to find register files with particular configurations (depth, data width or number of ports), separate local memories for different kinds of application data, customized data channels that implement real-time data flow into and out of the processing units, or synchronization ports shared with other SoC blocks. As opposed to classic general-purpose CPUs, because of the narrower and more regular application domain, ASIPs often rely less on cache hierarchies, to save on the power budget. Instruction caches can sometimes be avoided at all, in which case external agents pre-load the program code into dedicated program memories.

As a consequence of such extreme configuration possibilities, to efficiently explore the hardware-software customization of such systems appropriate emulation techniques are required to provide fast but accurate performance estimates. Along with the classic characterization of hardware modules and applications with standard functional metrics (i.e. execution time, cache performance, resource congestion), there is increasing interest in obtaining early estimations of physical metrics, such as area occupation and power/energy consumption. For all these requirements, hardware-based emulation techniques have been proposed as an alternative, more scalable, solution to cycle-accurate software-based simulation approach.

FPGA devices, with their high flexibility, have shown to serve well for hardware-based emulation but, at the same time, the achievable advantages are mitigated by the overhead introduced by the physical synthesis/implementation flow. This overhead impacts on the emulation time and thus on the number of explorable design space points.

In this work FPGA-based on-hardware emulation for design space exploration of com-

plex, heterogeneous systems composed by multiple ASIPs is explained, combined with high-level simulation techniques allowing a faster, accurate evaluation of design possibilities.

Two different and alternative methods that uses hardware modules and software-driven runtime reconfiguration of the emulating platform to enable the evaluation of different architectural configurations after a single synthesis/implementation process are shown. This helps in maximizing the speed-up of the overall evaluation process. The mechanism employed to reconfigure at runtime the emulating platform does not rely on standard partial reconfiguration capabilities that are offered by current FPGA devices and toolchains: instead, it's done through a custom-developed algorithm to identify the logic to be placed on the FPGA and the hardware modules that support the actual logic reconfiguration.

The majority of the research presented here has been conducted for the MADNESS FP7 project, where University of Cagliari was project leader. MADNESS aims at the development of an automated framework of heterogeneous MPSoCs, with adaptivity and fault-tolerance support. The project successfully concluded on March, 2013 (www.madnessproject.org).

The organization of this work can be summarized as follows: Chapter 2 will briefly introduce state-of-art techniques for FPGA emulation and Design Space Exploration of complex systems, illustrating also author's previous contributions on the topic; Chapter 3, on the other hand, describes the complex industrial toolchain by Intel (formerly SiliconHive) for the design of custom VLIW ASIPs, presenting the tools and languages used to create a processor starting from an high level input specification.

In 4, the combined approach of mixing high-level simulation and on-hardware prototyping through FPGA is explained: along with this, reader can find a description of how software simulation and FPGA execution cooperates to achieve desired result. Following Chapters 5 and 6 will describe two different approaches to build an FPGA platform that serves as a fast prototype for different architectures: the first will describe a solution based on instrumenting processor hardware with an additional module, automatically created starting from a number of input specifications, while the second one proposes a pure software method that leverages the ability of manipulating instruction words from the binary file, thanks to *a priori* knowledge of ASIPs instruction format.

Last Chapter (7) presents the SESAME framework to the reader, including a short overview on Kahn Process Network paradigm, and the contribution of this work, represented by the extension of the simulation framework to support Network-On-Chip interconnects, thus allowing the evaluation of hybrid systems with multiple cores communicating through such network.

Chapter 2

State of the art

In this part of the thesis, academic research on the topics of ASIPs, VLIW processors, FPGA simulation methods and Design Space Exploration algorithms will be presented. Each section describes a different part of a general outlook on the topics developed during the doctoral course.

2.1 Application-Specific Instruction Set Processors (ASIPs): an overview

2.1.1 Different approaches in speeding up execution

Very Long Instruction Word (VLIW) architectures allow for parallel execution of multiple processing operations with a single instruction word. After a brief walkthrough the evolution of computing approaches during history, key points of custom ASIPs will be described.

Sequential execution

During the seventies, the majority of the processors were purely sequential in two different aspects: instructions were executed on a sequence (Figure 2.1), and each instruction was decomposed in a set of actions (called *stages*) executed in a strict order. Typical stages that compose an instruction are:

- IF - instruction fetch;
- DE - instruction decode;
- OF - operand fetch;
- EX - execution;
- WB - result write-back;

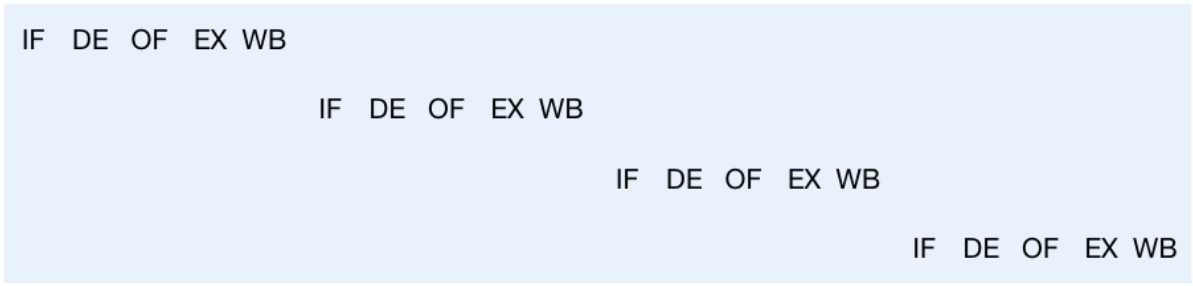


Figure 2.1: Sequential execution of stages/instructions

Pipelining

Later on, engineers moved to pipelining instruction stages, thanks to the relatively low complexity of RISC (Reduced Instruction Set Computing) processors. After the first stage was completed, it was easy to continue the execution of other stages in parallel, keeping the pipeline always full (Figure 2.2). However, this is not true for some cases, e.g. when processor stalls on a memory load or write event.

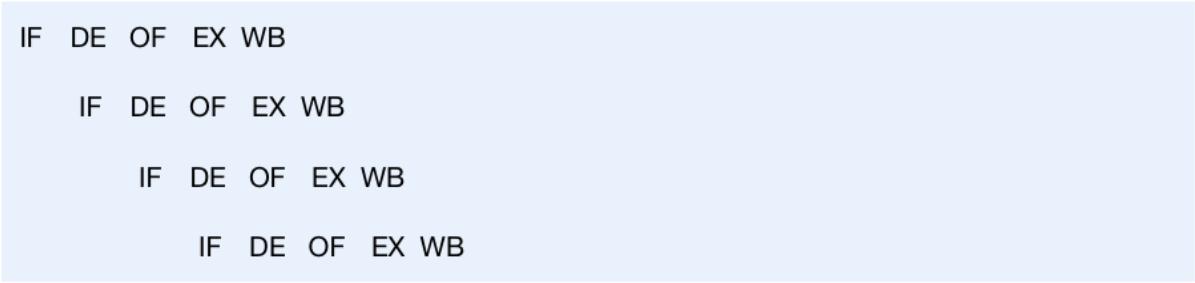


Figure 2.2: Pipelined execution of stages/instructions

Instruction scheduling: Superscalar and VLIW

Once engineers reached the target of virtually one instruction per clock cycle, the next challenge was to further improve the execution speed, performing more than one instruction per clock cycle. This challenge is still open, though, since it's not trivial to execute multiple instructions at once, due to data dependency between instructions, mutual exclusion due to hardware resources contention, etc. Basically, both the Superscalar and VLIW approaches leverage the instruction scheduling as a powerful tool to increase parallelism, with a significant difference: superscalar architectures perform scheduling at run-time (Pentium IV, PowerPC G4), while VLIW processors include the scheduling algorithm in the compiler, thus allowing for a faster (and predictable) execution flow, but increasing the compilation time. So, the VLIW processor, once it fetches an instruction, is able to perform each operation in parallel, without the need of dependency check or hardware resource conflicts inspection between operations.

Compiler or Hardware complexity?

As previously said, VLIW architectures shift the complexity burden on the compiler development: even though also RISC superscalar implementations need sophisticated compilation techniques, VLIW compilers represent the highest level of complexity, thus setting an initial obstacle to a prospective implementation. However, it should be considered that such a complex task as writing the compiler must be performed only once, rather than at each change of the hardware implementation as with superscalar processors. Designers can thus obtain a smaller, faster chip (this means a cheaper device), and the design phase itself can be more rapid, requiring less debugging effort. Lastly, the compiler can be improved at a later stage, adding or enhancing its capabilities, without the need of changing the hardware design of the processor.

Real world implementation problems

One of the issues regarding VLIW instruction format is due to the fact that, even when a function unit has no operation to execute, its relative portion of instruction bits still must be loaded from memory. As one can imagine, even the most sophisticated compilation algorithms cannot schedule operations on each function unit at each cycle. This leads to waste of memory space in instruction memory, in instruction cache, and also bus bandwidth resources. Furthermore, if a VLIW processor is equipped both with integer and floating point function units, it's hard to keep the latter working while the software application is only performing integer calculations. However, these problems can still be addressed by mainly two different approaches: compressing the instruction with a coded representation, or using an instruction word format that covers only partially the available hardware resources. For the first method, Huffman coding can be used to allocate fewer bits for most frequent operations. In the second technique, the operations contained in the instruction word also carry the information about which function unit has to execute it: this will lead to significant saving of memory space thanks to instruction word reduction, but will also limit the maximum number of operations to be conducted in parallel. Again, the designer is responsible of analyzing the application flow in order to choose the best tradeoff between instruction length and execution speed.

2.1.2 Advantages in using custom ASIPs

Electronic systems developers nowadays face more complex challenges, requiring flexibility in design and ability to quickly reach the market with innovative, even more powerful products. This trend has surely contributed to the adoption of application-specific processors (ASIPs) under the form of IP cores, allowing to obtain high performances and low power consumption, paired with easy software programmability, as demonstrated by [10].

ASIPs propose to exploit an a-priori knowledge about the processor target application, in order to enhance area and energy efficiency of the architecture. As an example, a custom-designed special instruction can be used to perform a particular operation faster (i.e., in less cycles), thus allowing the designer to slow down the clock frequency and consequently, cut the power consumption. Furthermore, inefficient ALU instructions can be removed if there's no reason to support them, enabling a significant reduction of needed transistors and area occupation on chip.

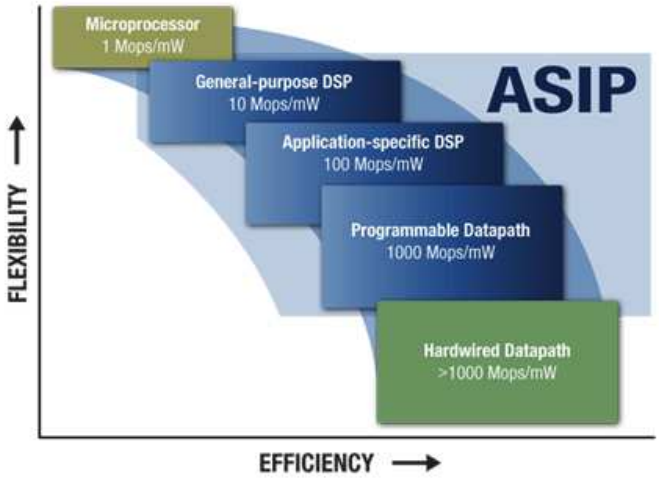


Figure 2.3: ASIPs vs other architectures: flexibility and efficiency (courtesy of www.retarget.com)

ASIPs represent a sort of convenient trade-off between flexibility and efficiency (see Figure 2.3), since they can be placed, inside the whole architectural spectrum, among general-purpose CPU and hardwired data paths. On the one hand, they are more efficient in terms of instruction-level and data-level parallelism than general-purpose microprocessors, offering an higher ratio of operations executed with respect to power consumption. On the other hand, though, they are way more flexible than hardwired data paths, that offer to the designer little-to-none reconfigurability nor possibility of being used for different software application than the one they were designed for.

2.2 FPGA-based evaluation platforms for heterogeneous systems

2.2.1 Cycle-accurate software simulation

Today, the vast majority of architectural simulation is performed, at maximum accuracy (i.e. cycle-level) in software. Among the most famous solutions, many are still sequential, like SimpleScalar [5], Simics [30], Gem5 [7] or MPArm [6]. To cope with the increasing system complexity, parallel software simulators have entered the scene [34]. Some of them give up on pure cycle-level accuracy and look at complex techniques, like event-based simulation, statistical sampling, dynamic accuracy switching and simulation state roll-back, to model complex, multi-core architectures that run full software stacks [3],[15]. Moreover, specific solutions have been developed for particular classes of processor architectures, and specifically optimized to enable rapid design space exploration of such architectures. For instance, in [4] and [14], a software-based simulation and exploration framework targeting optimization of a parametric VLIW microprocessor is presented.

Despite these innovations in the field, there is general consensus on the fact that classic software approaches are not anymore able to provide cycle-accuracy for complex multi-

core hardware-software designs in reasonable simulation times. A promising alternative approach aims at preserving cycle-accuracy, resorting to the extremely high speeds that can be achieved by running the target architecture on some kind of configurable hardware. FPGA devices naturally provide this configurable hardware substrate, potentially enabling hundreds of MHz of pure emulation speed already in the earliest stages of the design flow. In addition, their integration capacity, scalability, the relatively low cost and the decreasing power consumption figures suggest FPGAs are going to be the reference platform for hardware-based emulation for the next future[45].

2.2.2 The role of FPGAs as prototyping platforms

The most important contribution to the field of large hardware FPGA platforms for simulation of complex systems is brought by the RAMP project [46]. Several research activities have been condensated within the scope of this large project, including the first FPGA prototype of a transactional memory chip multiprocessor [47], a thousand-core high-performance system running scientific benchmarks on top of a message-passing communication library [25] and a SPARC based multi-threaded prototype which implemented FPGA-accelerated emulation by introducing the separation between functional and timing simulation [43]. Within the RAMP project, moreover, runtime configuration has been investigated to the extent of some cache-related parameters.

Other examples of hardware-based full-system emulators are [13], [16], [11] and [28], in which the FPGA-based layer is employed to accelerate the extraction of several metrics of the considered architecture, specified and automatically instantiated in a modular fashion. These papers also quantify the speedup achievable through FPGA prototyping in three/-four orders of magnitude in emulation speed, when compared to software-based simulators. Nevertheless, as mentioned, FPGA-based approaches are still affected by an overhead introduced by the synthesis and implementation flow [32]. With the standard flow, this amount of additional time has to be spent every time a hardware parameter is changed. Several approaches aim at the reduction of the number of necessary synthesis/implementations, by looking at FPGA reconfiguration and programmability capabilities.

In [26], the authors use the FPGA partial reconfiguration capabilities to build a framework for Network on Chip based (NoC) emulation. Also in [48], relying on partial reconfigurability techniques, FPGAs are used to optimize register file size in a soft-core VLIW processor. Both these solutions implement platform runtime reconfiguration employing specifically designed logic that is increasingly being included in the latest high-end FPGA devices. As opposed to these approaches, presented work devises a more generally applicable mechanism, which is more oriented towards software-based reconfiguration and employs minimal hardware modifications to the logic under emulation.

The main use scenario of described toolchain is along with a DSE engine, that provides as output the candidate architectures according to the design requirements, asks for emulation of a set of those and iteratively proceeds in the exploration. Some examples that can benefit from using this approach during the evaluation phase are [37], [17] and [12].

2.3 High-Level simulation techniques aimed to Design Space Exploration (DSE)

2.3.1 System-Level DSE

The process of system-level DSE is typically performed exploiting support from two different kind of tools [18]: an evaluation platform that examines the design points in the design space, using for example analytical models or (system-level) simulation, and an exploration engine that iteratively searches and decides which points have to be evaluated. There is a significant variety of approaches that aim at defining novel methods to perform either one or the other step in a time-effective manner, [40, 27, 21], especially targeting heterogeneous MPSoCs [44, 21, 29].

The majority of the approaches rely on system-level simulation to do the evaluation [18]. Basing on the Y-Chart principle we can find simulation tools that work at a high level of abstraction like [23, 24].

Modular system-level MP-SoC DSE framework are proposed in [36, 8, 29], for DSE of embedded systems. The MultiCube project [1] has similar objectives, but it mostly targets micro-architectural exploration of multiprocessors rather than system-level architectural exploration.

In [2], authors present a framework specifically targeting ASIPs, integrating a design toolchain with a virtual platform to explore a number of axes of the MP-SoC configuration space.

2.3.2 Introducing FPGAs to speed up DSE

However, none of the mentioned approaches, to the best of author's knowledge, experiments the integration of high-level simulation and FPGA prototyping. In literature, the use of emulation on reconfigurable hardware has been often limited to the analysis and exploration of high-performance computing systems, mainly enabling the prototyping of *static* architectural templates to speed-up the evaluation of architectural design techniques on complex applications. The most important contribution to the field of large hardware multi-FPGA platforms for simulation of complex systems is brought, again, by the RAMP project [46].
Examples of hardware-based full-system emulators are [13] and [28], in which the FPGA-based layer is employed to accelerate the extraction of several metrics of the considered architecture, specified and automatically instantiated in a modular fashion. Such papers report a speedup achievable with the use of FPGA prototyping of three/four orders of magnitude in emulation speed, when compared with software-based simulators. The Daedalus framework [35] can be considered a baseline for the work presented here. In Daedalus, on-hardware evaluation is used for DSE purposes. FPGA-based evaluation platforms are automatically created using the ESPAM tool. However, support for prototyping of highly heterogeneous (e.g. ASIP-based) architectures was not completely provided, since configuration at component-level was not allowed and no countermeasures are taken to balance the overhead related with the synthesis and implementation flow. Some works that aim at the reduction of the number of necessary synthesis/implementations, by looking at FPGA reconfiguration and programmability capabilities, can be found in literature. RAMP Gold, a framework developed within the RAMP project, also provides some capabilities of changing at runtime the cache-related parameters during the emulation. In [48], relying on partial reconfigurability techniques, FPGAs are used to optimize register file size in a soft-core VLIW

processor. In author's previous work [31] a hardware reconfigurable prototyping platform to allow fast ASIP design space exploration was implemented.

Chapter 3

Intel reconfigurable ASIPs development platform

This chapter will briefly introduce the architectural template used in most parts of this thesis, represented by Intel's family of configurable VLIW processors. First, an introduction on the architecture will be made, moving later to the description of a common architecture template, and finally describing the compiler and the tools shipped by Intel with the processor development kit.

3.1 HiveLogic

The cores described in these section are built using SiliconHive ([19]) (now Intel) development platform for custom VLIW processors, called HiveLogic. The platform offers to the designer a good amount of degrees of freedom at design time: on the one hand, this makes the generated processors highly customizable and tailored for a particular application/set of constraints, while on the other hand exposes greater complexity to the designer, having to explore such a considerable design space. This poses a problem that this work tries to tackle and optimally solve thanks to the support of a novel Design Space Exploration algorithm and an enhanced FPGA platform, as described in following chapters.

The main input for the processor generation is represented by a TIM file (custom SiliconHive description language), that contains every detail about the configuration, in terms of issue slots, description of each function unit, number and size of register files, interconnect between issue slots, local data and program memories, custom operations and FIFO adapters to interface with an host processor.

3.1.1 Architectural template

All Silicon Hive processors are derived from a common Processor Architecture Template (PAT). The PAT defines a processor that consists of one or more interconnected so-called Cells. A single cell defines a Very Long Instruction Word (VLIW) machine that is capable of executing parallel software with a single thread of control. According to the PAT a cell consists of a Core that performs computations under software program control, and a so-called

CoreIO that provides a memory and I/O subsystem allowing the core to be easily integrated in any system, while providing local storage to increase local memory bandwidth and largely reduce strain on shared system resources. The core consists of a VLIW data path and a sequencer controlling the data path under software control. The sequencer is a simple state machine containing a program counter register as well as a status register. The status register in the sequencer is used to enable special processor modes under software control. The program counter value is used to locate instructions stored in a local program memory or instruction cache. The presence of a local program memory guarantees an instruction fetch throughput of one complete VLIW instruction per cycle enabling maximum performance. This works especially well for smaller program sizes that warrant the cost of such a local program memory. For larger programs, additionally or alternatively an instruction cache is present coupled through a dedicate master interface. The instruction cache typically improves the performance in fetching instructions stored in external memory. Obviously, a certain performance penalty due to cache misses cannot be avoided. The data path contains a number of functional units organized in a number of parallel issue slots. The issue slots are connected to registers organized in a number of register files via programmable interconnect. The functional units perform compute operations on intermediate data stored in the register files. On each issue slot an operation can be started in every clock cycle. Some functional units, termed load/store units, have access to so-called logical memories in CoreIO. Each logical memory contains one or more storage or I/O devices. These allow the load/store units memory mapped access to local physical memory in CoreIO or to perform memory mapped I/O with the system. Supported storage and I/O devices include SRAM memories, flipflop-based memories, blocking streaming interfaces, and master interfaces. The system in which a cell is integrated has access to the storage devices in CoreIO via one or more slave interfaces. System access to program memory and status and control registers of the cell is provided through slave interfaces as well. The stream, master, and slave interfaces support commonly used standard protocols to provide clean and easy integration of a processor in a wide variety of system architectures.

According to the *Processor Architecture Template* (see Fig. 3.1), every processor consists of a composition of sub-structures called *processor slices*, that are complete vertical datapaths, composed of elementary functional elements called *Template Building Blocks*, such as:

- register files (RF): holding intermediary data in between processing operations, configurable in terms of width, depth, number of read and write ports;
- issue slots (IS): basic unit of operation within processor; every issue slot includes a set of function units (FU), that implement the operations actually executable. Every issue slot receives an operation code from the instruction decoding unit and, accordingly, accesses the register files and activates the function units;
- logical memories: container for hardware implementing memory functionality;
- interconnect: automatically instantiated and configured, implementing the required connectivity within the processor.

In order to simplify the description of our work, an assumption on the architecture is made: Hive processors will always contain a control slice, in charge of handling the program counter and updating the status register, plus a variable number of different processing slices, with no restrictions on their internal composition.

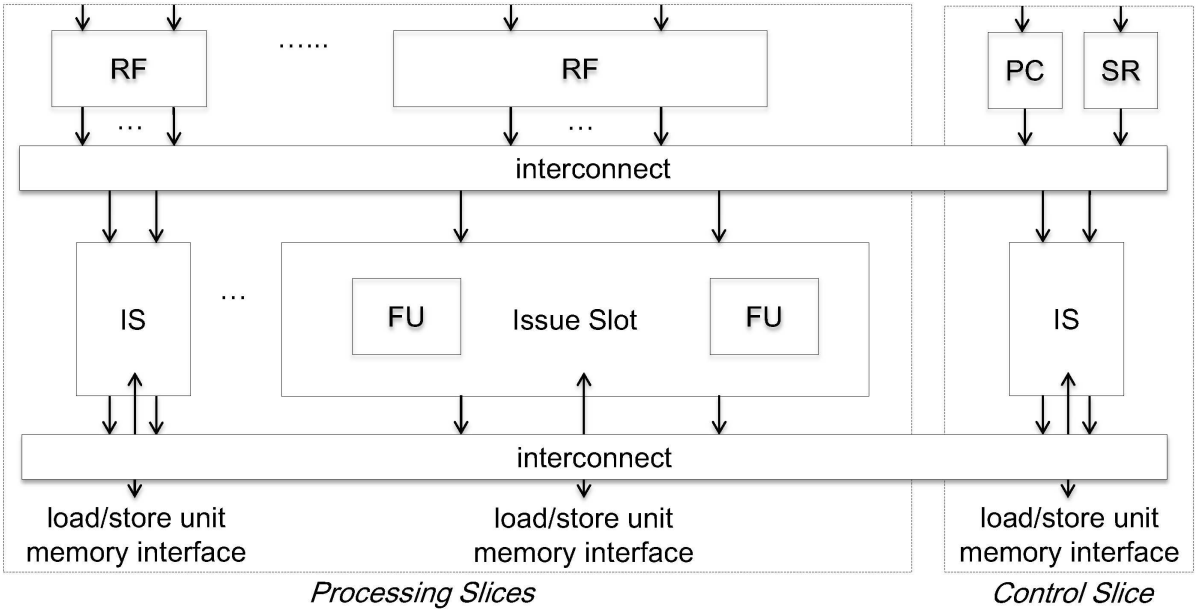


Figure 3.1: Reference VLIW ASIP template

3.1.2 Processor description and specification: TIM language

All of Silicon Hive's programming and processor generation tools are configured to target a specific processor instance described in the TIM language. TIM is therefore crucial for Silicon Hive processor design. To convert a TIM description from human readable form into a form that can be understood by Silicon Hive's tools, the TIM language is supported by a dedicated TIM compiler, invoked by a straightforward make flow (thus hiding the complexity to the final user). Here an example of what a TIM processor description looks like:

```
// cluster includes
#include <pbb/pse/std_base/pearl/bpse_pearl.tim>


/* ============================ */
/*  Cell declaration.           */
/* ============================ */
Cell pearl <signed intWidth, signed llWidth, signed bpFifoWidth>
    ( mmioW_DTL<intWidth> sl_ip, fifoW_DTL<bpFifoWidth> st_ip0, st_ip1 )
 -> ( mmioW_DTL<intWidth> sl_op, fifoW_DTL<bpFifoWidth> st_op0, st_op1 )
{

    /* type size properties */
    CharBits      := 8;
    ShortSize     := 2;
    IntSize       := intWidth/8;
    LongSize      := 4;
    LongLongSize  := llWidth/8;
```

```
    /* Default memory */
    DefaultMem    := bp.BP_DEFAULTMEM;

    /* Stack pointer */
    SPs         := bp.BP_LSUBASE_RF[0];
    Stacks := bp.BP_DEFAULTMEM;

    /* Return pointer */
    RP_rf         := bp.BP_LSUDATA_RF[1];

    signed pmemWidth                  := 67;
    signed pcBitsPmemCap              := 12;
    signed pcBitsViewCount            := 0;
    signed pcBitsSmallestViewFactor   := 0;

    signed statusControlRFcap         := 8;
    signed branchLatency              := 2;

    signed bpRF1cap       := 32;
    signed bpRF2cap       := 32;
    signed bpS1ImmBits    := 13;
    signed bpS2ImmBits    := 5;

    signed bpMemCap       := 16384;
    signed bpFfgCount     := 2;
    signed bpFfgWidth     := bpFifoWidth;
    signed bpFfgCap       := 2;

    bpse_pearl  bp < pmemWidth, pcBitsPmemCap,
pcBitsViewCount, pcBitsSmallestViewFactor,
                     statusControlRFcap, branchLatency,
                     intWidth,
                     bpRF1cap, bpRF2cap,
                     bpS1ImmBits, bpS2ImmBits,
                     bpMemCap,
                     bpFfgCount, bpFfgWidth, bpFfgCap >

                    ( bp.op, sl_ip, sl_ip, sl_ip, st_ip0, st_ip1 );

    sl_op       = { bp.sl_op_config, bp.sl_op_pmem, bp.sl_op_dmem };
    st_op0       = bp.st_op0;
    st_op1       = bp.st_op1;
};


/* ========================== */
/*  Processor declaration.    */
```

```
/* ========================== */
Processor pearl_processor
    ( mmioW_DTL<intWidth> sl_ip, fifoW_DTL<bpFifoWidth> st_ip0, st_ip1 )
 -> ( mmioW_DTL<intWidth> sl_op, fifoW_DTL<bpFifoWidth> st_op0, st_op1 )
{
    /* basic parameters that hold throughout the whole cell */
    signed intWidth := 32;
    signed llWidth :=  40;   when intSize==16
    signed bpFifoWidth := 32; // in bits

    pearl pearl <intWidth, llWidth, bpFifoWidth>
(sl_ip, st_ip0, st_ip1);
    sl_op = pearl.sl_op;
    st_op0 = pearl.st_op0;
    st_op1 = pearl.st_op1;
};
```

3.2 Retargetable software toolchain

HiveLogic comes equipped with a retargetable scheduler and compiler, capable of optimally exploiting the hardware resources of chosen processor configuration in a transparent way for the user. Once the designer builds the processor configuration(s) he plans to use, the toolchain generates needed symbols and libraries for each architecture. Thanks to this, user can compile the application and have it automatically scheduled for the hardware under consideration. Together with the compiler suite, also an instruction-level simulator is provided: through this tool, the designer is able to perform an evaluation of processor performances when running a given application or computation kernel.

3.2.1 ANSI C Compiler

Hive C Compiler is an ANSI C compiler, provided with the Intel SDK installation. Through a complex chain of makefiles inclusions, the designer is able to invoke different simulations of a target application by issuing always the same commands, no matter how complex the application is or which system it is mapped on. User must provide the code running on the Hive processors (called *cells*), tipically a processing kernel, and the code in charge of initializing the cells from the host processor. Here follows an example of Makefile for an application mapped on a single-core system.

```
SYSTEM = pearl_system
PROGRAMS = img_conv_3x3
METHODS = crun sched target

img_conv_3x3_CELL = pearl_16_3is
img_conv_3x3_FILES = img_conv_3x3.hive.c
img_conv_3x3_CFLAGS = -fno-stack -Whivesched,"-trace 2" -html
```

```
img_conv_3x3_CFLAGS +=  -Werror
img_conv_3x3_LDFLAGS += -embed

HOST_FILES = img_conv_3x3.c img_conv_3x3_c.c
HOST_CFLAGS = -W -Wall #-DHIVE_MULTIPLE_PROGRAMS -Werror
HOST_OUT = host.elf

include $(HIVEBIN)/../share/apps/hive_make.mk
```

The SDK allows the user to perform different kinds of simulations:

- C-Run: host code and cell code are both compiled with the workstation C compiler (gcc in this case), and a simple consistency and syntax check is performed;
- Unscheduled: cell code is compiled by Hive C compiler but it's not scheduled; generated simulator will now use the cell datapath and it's aware of cell memory dimensions, but execution statistics are not provided.
- Scheduled: cell code is compiled and scheduled by Hive toolchain, allowing the developer to obtain accurate execution statistics (for the processor datapath, but still not for external memory accesses). This simulation is the most useful when it comes to optimizing the code to exploit instruction-level parallelism.

3.2.2 Instruction scheduler

HiveSDK offers to the developer two different schedulers, called hivesched and manifold. The main difference between the two is that the first performs a non-exhaustive scheduling inspection, while the latter, being obviously slower, can lead to better results. Anyway, it's not possible to leave the complete burden of code optimization to scheduling algorithm, so the developer must take care of avoiding common pitfalls when writing the application code and apply good programming practices to allow the best exploitation of a parallel processor architecture.

3.2.3 C Debugger

Hive SDK comes with a debugger, similar in its usage to GNU Debugger, that takes scheduled code, executes it on the hardware model and it's able to read local memories and registers, plus providing watchpoints for variables and conditional execution. However, it should still be considered that on a VLIW architecture, a single instruction can (and often does) contain operations for multiple issue slots, thus stimulating more than an operation inside the cell processor. Developer should take this behavior into account when dealing with debugging Hive applications.

3.2.4 Host-Cell Runtime functions

To simplify the work of software developers, SiliconHive SDK provides runtime functions that can be used to access cell memories and FIFOs from the host processors. The retargetable toolchain takes care of transparently converting cells and variables names to the

correspondant memory mapped addresses, so that user can conveniently refer the variables by their names inside the cell's software. Most important functions are listed below:

- hrt_fifo_rcv: allows the host to fetch a token from a cell's FIFO, and blocks the host if FIFO is empty;
- hrt_fifo_snd: allows the host to send a token inside a cell's FIFO, and blocks the host if FIFO is full;
- hrt_scalar_load: loads a variable value from cell's data memory, transparently masking memory-mapped address of the variable;
- hrt_scalar_store: stores a variable value to cell's data memory, transparently masking memory-mapped address of the variable.

3.3 Multi-Processor systems instantiation

HiveLogic toolchain allows the designer to develop systems with more than one processor communicating through FIFO channels, containing custom memories, interconnects and I/O peripherals instantiated as black-box units (in this work, a custom Network-On-Chip interconnect has been used to connect different ASIPs in a system). As for a single processor, the platform can produce VHDL code of the whole described system, enabling an FPGA execution and evaluation process of the configuration or a prospective ASIC implementation.

System description is provided by the user through an HSD file (a format similar to the already described TIM), where the designer can instantiate the desired elements, interconnect the elements to the bus, add interface I/O signals and tune additional elements parameters (e.g. bus latency, clock gating etc).

Once user has completed the instantiation of system elements, it's possible to compile the description and the toolchain will take care of producing necessary header files and libraries. Among other things, the system header file contains needed information to allow host-cells interconnect, including memory-mapped addresses of Hive cell memories, FIFO adapters address and sizes and I/O ports interconnect. The header file must be included by the application intended to be executed on that system. The designer also has the possibility of generating VHDL code for system interconnection elements and cells, obtaining a set of files that can be integrated in an FPGA development flow to perform synthesis/implementation on a prototyping device.

Below an example of a real multi-processor system (used for the porting of an H.264 decoder), composed by three slightly different Hive processors, interconnected with FIFO adapters, exchanging data with an host processor through Silicon Hive's proprietary CIO bus protocol, and equipped with a master interface to provide the ability of performing load/stores on other Hive cells data memories.

```
System h264_hive_system
{
  signed bus_data_width := 32;
  signed bus_addr_width := 32;
  signed bus_burst_size := 4;
```

```
  pearl_freshcoco_economy_32 pearl_1 ( bus.sl_ip_1,
 pearl_bp_fifo_fifo_fifo0.st_out,
 pearl_bp_fifo_fifo_fifo1.st_out,
 );
  pearl_freshcoco_economy_32_3fifo pearl_2 ( bus.sl_ip_2,
pearl_1.st_op4, pearl_3.st_op3,
pearl_bp_fifo_fifo_fifo7.st_out );
  pearl_freshcoco_economy_32_4fifo pearl_3 ( bus.sl_ip_3,
 pearl_bp_fifo_fifo_fifo6.st_out,
 pearl_2.st_op1 );

  FifoAdapter_new_2s pearl_bp_fifo_fifo_fifo0
<32, bus_data_width, bus_addr_width, 4> (pearl_1.st_op0,
bus.bp_fifo_fifo_fifo0);
  FifoAdapter_new_2s pearl_bp_fifo_fifo_fifo1
<32, bus_data_width, bus_addr_width, 4> (pearl_1.st_op1,
 bus.bp_fifo_fifo_fifo1);

  SystemBus bus (host.op0, pearl_1.mt_op, pearl_2.mt_op);

  HostProcessor host<bus_data_width, bus_addr_width, 4>;

  User_Properties := { NoDefaultResetGate, 1 };
  User_Properties := { Processor, true };
};
```

Chapter 4

Combining on-hardware prototyping and high-level simulation for DSE of MPSoCs

The problem of efficient Design Space Exploration for complex MPSoCs is the spark behind the novel approach presented in this thesis. On one hand, exploration and evaluation of a particular architecture can be done with cycle-accurate software simulators, if available: the major drawback of this method is the time needed to simulate complex, real-world applications (think of multimedia as an example), that can take up to hours to complete and produce detailed statistics. On the other hand, reconfigurable devices such as FPGA seem to provide a solution for this problem, thanks to the flexibility and speed they can offer: however, if designer should undergo the complete synthesis and implementation flow each time he needs to evaluate a different architectures, advantages offered would be easily overtaken by these time-consuming tasks.

Proposed approach, starting from this practical problem, combines the benefits of an high-level simulation framework and a genetic algorithm for DSE, that receives as input the characterization number obtained by the calibration of a given architecture on FPGA devices. FPGA standard flow is improved, thanks to novel approaches described in the following chapters.

The rest of this chapter will describe how a combined FPGA-software simulation approach can be successfully exploited, thanks to the cooperation between the SESAME framework and an FPGA prototyping platform. In the next Chapters (5 and 6), two different and alternative approaches for characterization of multiple ASIPs configurations on FPGA will be presented: the first exploiting a custom-developed hardware module, automatically instantiated in the processor hardware, and the last one based on a pure software solution, leveraging the possibility of manipulating the instruction words for a given application binary. Both these methods can be used to provide execution metrics to the DSE algorithm described hereafter. At the end of each chapter, a use case will be described, supporting and confirming the validity of proposed approaches.

4.1 General toolset description

The toolset here presented integrates three main components: a search engine, a simulation tool and a FPGA-based prototyping platform. The interaction between the tools is depicted in Figure 4.1.

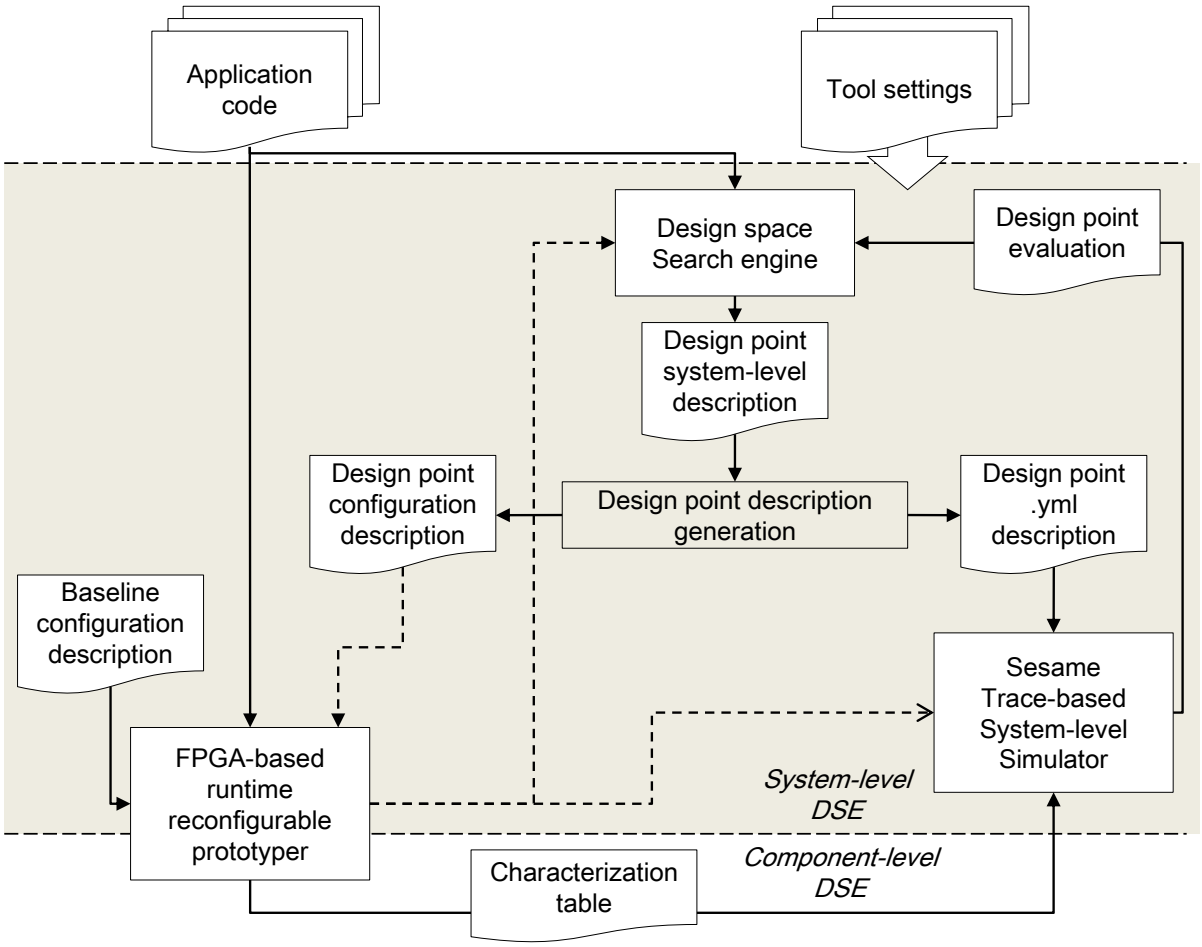


Figure 4.1: General toolset overview

The main input to the toolset is the application code, along with the specifications of the constraints that must be considered during the optimization process. Moreover all the tools can receive some input directives related to the settings of their operation mode. The search of the optimal design point is an iterative process that is driven by the *Design space search engine* (search engine hereafter). This tool, that will be described more in detail in section 4.2, embeds novel techniques for effectively pruning the design space by means of heuristic search algorithms and techniques for avoiding the use of relatively time-consuming simulations during DSE. When the search engine requires the evaluation of a (set of) design point(s), it produces a system-level description of those design points (*Design point system-level description* in the figure). At the output of the search engine, such description is expressed using a very abstract format to specify the design point to be evaluated and, adequately translated by a utility (*Design point description generation* in the figure), can be elaborated by the lower level of the toolchain. At this level, the FPGA-based prototyping platform and the simulator co-operate in different ways during the evaluation process.

More in detail, two use cases are typically possible:

- preliminary calibration;
  At the starting point, the FPGA-based prototyper is exploited for calibrating the simulation model. The execution of the application tasks is emulated on a baseline single-ASIP hardware prototype, to perform a detailed component-level (processor-level) DSE. According to such emulation, the tasks are conveniently characterized in terms of their computation latency over different processor configurations. Once this detailed characterization has taken place, the resulting numbers are passed as input to the simulation model, as a *characterization table*, so that it can start serving as an evaluation platform for the search engine.

- periodic tuning and detailed analysis;
  When needed, during the iterative process, the search engine is able to directly ask the prototyping for a (set of) customized multi-ASIP design point(s) under evaluation (dashed line in Figure 4.1). The prototyper is exploited, in this case, for system-level DSE, obtaining a detailed characterization that may be provided as a feedback to the search engine and to refine the tuning of the simulator.

At the end, a Pareto front is provided to the user, to be considered when choosing the optimal application-specific architecture.

4.2 Design Space Exploration: search engine

To optimally explore the design space for optimum design points, a search engine that utilizes heuristic search techniques, such as multi-objective Genetic Algorithms (GAs), has been developed. Such GAs prune the design space by only performing a finite number of design-point evaluations during the search, evaluating a *population* of design points (solutions) over several iterations, called *generations*. With the help of genetic operators, a GA progresses iteratively generating new populations towards the best possible solutions. In the search engine, the design space is explored in an iterative fashion using the NSGA II evolutionary algorithm. This module constructs a chromosome, a string of values representing the architectural- and mapping-related, which, for the sake of the exploration that is considered, may be defined as follows:

$$[p_1, p_2, \cdots p_j \cdots p_N, k_1, k_2, \cdots k_j \cdots k_N]$$

where the position j refers to a specific application process, the value p_j indicates respectively the ID of the processing unit in the system onto which the application process is mapped, and the k_j indicates the architectural configuration chosen for the processor (obviously if $p_i = p_j$ (task i and j are mapped on the same processor), then $k_i = k_j$). As mentioned, such string is analyzed by the *Design point description generation* utility to produce two different design description formats: the input for the prototyping platform (expressed using an industrial proprietary format) and the input for the simulation tool. Both formats will be described more in detail in the following sections.

To further optimize the DSE process, the search engine also allows for hybrid DSE in which fast but slightly less accurate analytical performance estimations are interleaved with more accurate but slower Sesame system-level simulations to evaluate design points during

DSE. Evidently, the aim is to interleave the analytical evaluations with the simulative evaluations in a way such that most evaluations are performed analytically. As a consequence, such an approach could significantly improve the efficiency of the DSE process, allowing for searching a much larger design space. The hybrid DSE part of the Search module is, however, beyond the scope of this work. The interested reader is referred to [39] for more details.

4.3 System-level simulation

For simulative evaluation of design points during the DSE, the Sesame MPSoC simulation framework [38] is deployed. Sesame is a modeling and simulation environment for the efficient design space exploration of heterogeneous embedded systems. According to the Y-chart design approach, it recognizes separate application and architecture models within a system simulation. An application model describes the functional behavior of a (set of) concurrent application(s). An architecture model defines architecture resources and captures their performance characteristics. Subsequently, using a mapping model, an application model is explicitly mapped onto an architecture model (i.e., the mapping specifies which application tasks and communications are performed by which architectural resources in an MPSoC), after which the application and architecture models are co-simulated to qualitatively study the performance consequences of the chosen mapping. For application modeling, Sesame uses the Kahn Process Network (KPN) model of computation in which parallel processes implemented in a high-level language communicate with each other via unbounded FIFO channels. Hence, the KPN model unveils the inherent task-level parallelism available in the application and makes the communication explicit. Furthermore, the code of each Kahn process is instrumented with annotations describing the application's computational actions, which allows to capture the computational behavior of an application. The reading from and writing to FIFO channels represent the communication behavior of a process within the application model. When the Kahn model is executed, each process records its computational and communication actions, generating a trace of application events, an abstract representation of the application behavior, necessary for driving the architecture model. Application events are generally coarse grained. Typical examples are:

- *read(channel id, pixel block)* that represents a communication event, in this case a data read from a FIFO channel
- *execute(DCT)* that represents an atomic computation event, in this case the execution of a DCT kernel.

The architecture model simulates the performance consequences of such computation and communication events generated by the application model. It is parameterized with an event table (the previously mentioned *calibration table*), that contains latency values that are associated to a given event. A table entry could include, for example, the number of cycles needed by a given processor architecture to complete a DCT function, that is the computation latency associated with the event *execute(DCT)*. Other kind of events, such as the previously mentioned communication actions or remote memory access can be modeled, associating a latency to a different architectural component, but are not strictly related with the scope of this method that mainly discusses about processor characterization. To realize

trace-driven co-simulation of application and architecture models, Sesame has an intermediate mapping layer that controls the mapping of Kahn processes (i.e., their event traces) onto architecture model components by dispatching application events to the correct architecture model component.

This description (*Design point .yml description* in Figure 4.1) is specified in YML (Y-chart Modeling Language), an XML-based format that consists of three parts: a high-level architectural description of the design point, an application graph description and a description of the mapping of application processes and communication channels onto the architecture resources. This information is automatically generated (by the *Design point description generation*). Moreover, it receives the *calibration table* by the FPGA-based prototyping environment.

4.3.1 FPGA prototype

In the next two chapters, the two different approaches for FPGA evaluation of design points will be presented. Both approaches were developed during the doctoral course, and present their own advantages and drawbacks. The method described in Chapter 5 was developed at an earlier stage, through the creation of a custom tool that parses the complete set of design points and instantiates an additional hardware block, while the latter, described in Chapter 6, leverages the predictability of each instruction format for every design point under consideration and performs binary manipulation at runtime.

The hardware approach would prove to serve better in an application where software changes more often than architectural configurations: the additional hardware module is created just once and software doesn't need any manipulation. The second method, on the other hand, introduces no overhead in terms of hardware resources and is suitable for every application that involves benchmarking a particular target application.

Chapter 5

An hardware-based FPGA flow to evaluate performances of ASIPs

5.1 Fast ASIP DSE: an FPGA-based runtime reconfigurable prototyper

In the section, a complete emulation toolchain is described. This toolchain, given a set of candidate ASIP configurations, identifies and builds an overdimensioned architecture capable of being reconfigured via software at runtime, to emulate all the design space points under evaluation. The approach has been validated against two different design space exploration case studies, with a filtering kernel and an MJPEG encoding kernel. Moreover, the presented emulation toolchain couples FPGA emulation with activity-based physical modeling to extract area and power/energy consumption figures. Furthermore, it's shown how the adoption of the presented toolchain reduces significantly the design space exploration time, while introducing an overhead lower than 10% for the FPGA platform resources and lower than 0.5% in terms of the operating frequency.

5.1.1 Approach overview

As already introduced, the objective of this work is to enable fast exploration of the processor configuration space, to identify the best customization for a given application. In order to do so, in the context of FPGA-based on-hardware emulation, authors aim at minimizing the overhead introduced by the FPGA platform synthesis and implementation process, when different processor configurations have to be prototyped. In fact, the elementary approach to this problem would imply a different FPGA synthesis run for each candidate configuration, impacting significantly on the speed-up over pure software simulation. Instead of doing that, investigation has been conducted on the possibility of identifying what it's named a *worst case* processor configuration (WCC). The WCC is a processor configuration that is oveprovided to include all the hardware resources necessary to emulate on FPGA every configuration included in the predefined set of candidates. The size and complexity of the WCC configuration will depend on the kind and number of architectural variations between the different configurations. After this process, synthesis and implementation on the FPGA plat-

form of *only* the WCC processor configuration is performed, thus limiting the mentioned time overhead to a single run. After its implementation on the FPGA, a runtime map of each specific configuration onto the implemented WCC is performed, by activating/deactivating hardware sub-blocks when needed, through dedicated software-based configuration mechanisms. The specific hardware/software mechanisms for runtime reconfiguration allow to select the functional blocks included in the WCC processor configuration and to adapt the connections between them. When emulating a configuration on top of a larger set of resources, the interconnection elements configuration are as relevant to the functional correctness as the block selection: this is due to the fact that WCC configuration should not generate incorrect communication latencies possibly caused by delays that are not part of the currently emulated configuration.

The conceived algorithm to build the WCC configuration, together with the hardware modules that implement the mechanisms for its runtime configuration, were designed to preserve full cycle-accuracy. In detail, the number of clock cycles that an arbitrary set of instructions will take to execute on the WCC configuration, when configured to emulate on the FPGA a candidate configuration, has to be exactly the same that it would take on the same candidate configuration, when it is synthesized and emulate alone on the FPGA. Similarly, all the functional metrics (congestion, latency, CPI) will be exactly the same, when measured in terms of clock cycles. The only difference between the WCC configuration and each single candidate configuration, when synthesized and placed on the FPGA, will be the number of utilized resources and the operating frequency of the FPGA emulator. Overall, this mechanism preserves the correct functional behavior of the ASIP processor and the binary compatibility of the WCC configuration to the executable code that would run on every candidate topology. The algorithm that we use to synthesize the WCC configuration is described in Section 5.1.6.

The design flow that implements the proposed prototyping technique refers, as baseline, to the industrial ASIP customization flow of SiliconHive. This flow has been extended to provide the needed support for runtime configuration. On the hardware side, some further HDL generation capabilities were added, that have been integrated with the baseline flow and will be explained in Section 5.1.7). On the software side, author implemented the generation of software functions allowing a user to manage the reconfiguration at application level. The reference baseline flow is described in Sect. 5.1.3 and shown in Fig. 5.2. The extensions are presented in Sect. 5.1.5 and in Fig. 5.3.

5.1.2 Reference architectural template and DSE strategy

This section will present the ASIP architecture template taken as reference for exploration purposes. Also, which variables identify the design space to explore are defined. The considered processor template belongs to the class of VLIW ASIPs, and is composed of instances of industrial IPs, based on a flexible *Processor Architecture Template* (PAT). It employs an automatically retargeting compiler.

Fig. 5.1 shows the main building blocks of the VLIW template. Every processor generated from this template consists of a composition of sub-structures called *processor slices*, which are complete vertical datapaths that propagate data through the Processor Template. The processor slices are composed of elementary functional elements called *Template Building Blocks*, such as:

- register files: they hold intermediary data between processing operations and are configurable in terms of width, depth, number of read and write ports. Typically, ASIPs generated from this architecture template include many register files.
- issue slots: they are the basic units of execution within the processor. Every issue slot includes a set of function units (FUs), that implement the operations actually executable and compose the execution stage. From an operating viewpoint, every issue slot receives an operation code from the instruction decoding phase and, accordingly, accesses the register files and activates the proper function units;
- logical memories: these are containers for hardware implementing memory functionality;
- interconnect modules: configurable connections automatically instantiated and configured. They implement the required connectivity within the different processor building blocks. In detail, connections can appear on the forward propagation data path along the processor slice, but also on the backward path to implement the writeback into the different register files. The interconnect module from the register files to issue slots is called Argument Select Network (ASN), while the interconnect from the issue slots to the register files is called Result Selection Network (RSN).

In this work, starting from SiliconHive's Pearl processor, author enables a design space exploration that covers most of the degrees of freedom exposed by the PAT, but not all of them. The control processor slice is made of two issue slots, two general-purpose register files, one local memory and one slave interface for external control. The two issue slots contain a minimal set of function units, which are mainly in charge of managing the program flow (handling the program counter and updating the status register) and the interaction with the program memory. This control logic includes a decoder that generates the opcodes to the function units from the VLIW instructions and a sequencer that handles instruction fetching.

Moreover, to limit the degrees of freedom, the processing slices must have only one issue slot and one register file. This is to say that, in order to have more than one issue slot inside a processor configuration, more than one processing slice is required. Finally, FUs inside every issue slot are taken from a pool of pre-defined FUs. Although the industrial methodology supports full extensibility of the instruction set through the definition of custom instructions, for the scope of this work this possibility is not taken into account.

Having defined these limitations on the range of possible template configurations, the design point including only the control slice is the simplest stand-alone processor configuration that can be evaluated. Other design points are processor configurations that instantiate an arbitrary number of additional processing slices and feature different parameterizations of the building blocks included in them. As a result of what was introduced so far, the design space under consideration is thus determined by the following degrees of freedom:

- $N_{IS}(c)$ is the number of issue slots inside the generic configuration c;
- $FU\_set(x,c)$ is the set of function units in the generic issue slot x, for the configuration c;

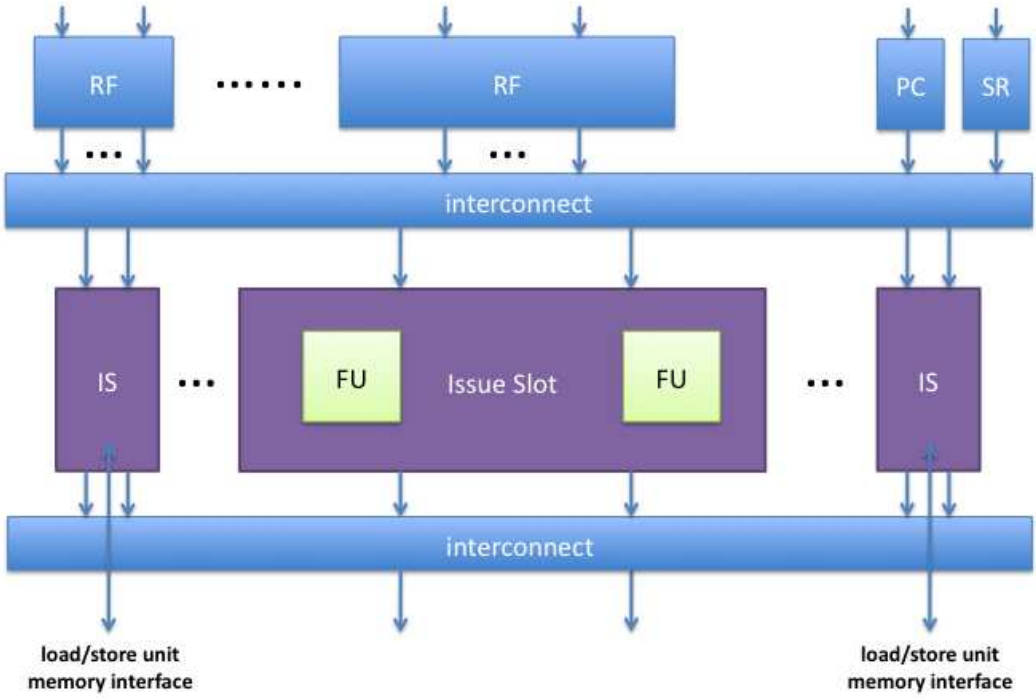


Figure 5.1: Reference VLIW ASIP template

- $RF\_size(x,c)$ is the size (depth) of the register file associated with issue slot x, in configuration c;
- $n\_mem(c)$ is the number of local memories in configuration c.

Although the way of specifying the parallel datapaths to be instantiated in the processors is limited, the configurability of the template keeps being very wide. With further engineering effort, that is beyond the scope of this research activity, the techniques and the tools here presented can be extended to overcome the mentioned limitations, keeping the correctness of the theoretical approach.

5.1.3 The reference design flow

Figure 5.2 plots the baseline flow for configuration of the ASIP template described in Sect. 5.1.2. The reference toolchain is SiliconHive's HiveLogic toolchain, composed by a core generator, a system generator and the HiveCC compiler. The figure shows the simplest mechanism to perform exploration of a given design space employing the baseline ASIP configuration flow. Every configuration to be evaluated during the DSE process is described using a proprietary description format. The description customizes the composition of the ASIP architecture under prototyping, in terms of number and kind of blocks, and their connectivity. In the baseline flow, every configuration description is passed to an RTL generator, that analyzes it and provides as output the VHDL hardware description of the whole architecture. This HDL code is then used as input for the FPGA implementation phase, that can be performed with commercial tools.

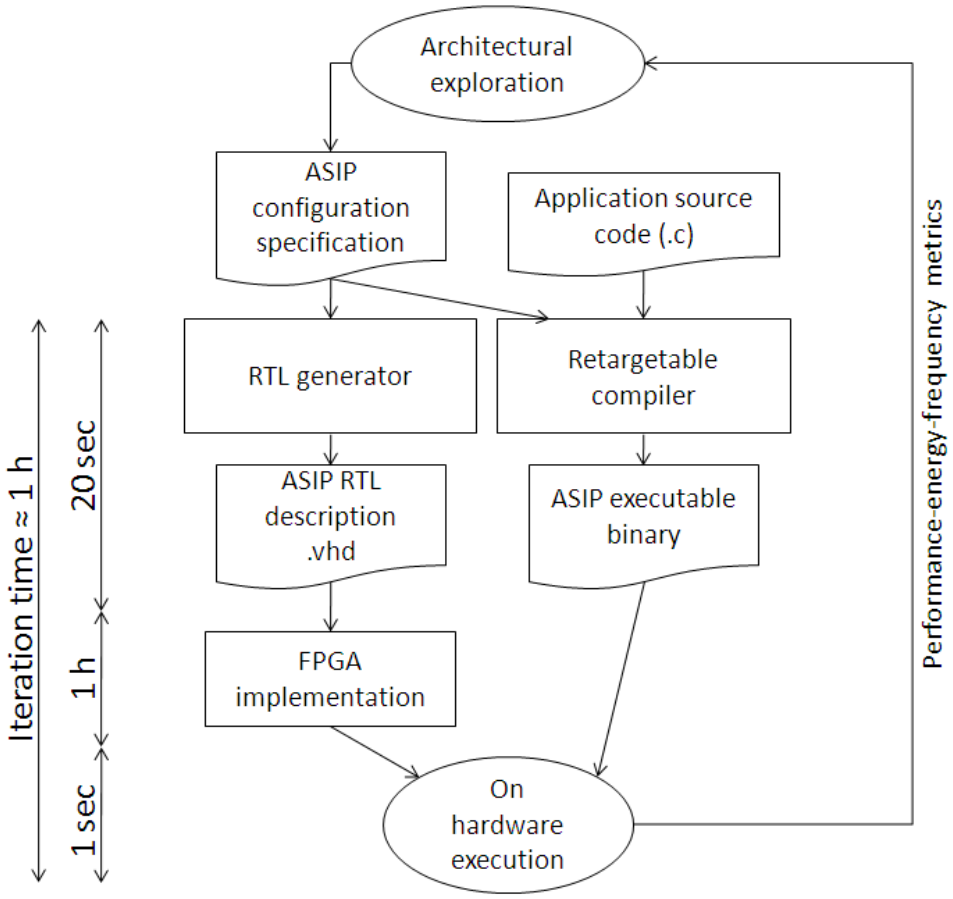


Figure 5.2: Baseline prototyping flow (Evaluation time for N candidate architectures can be measured to be approximately N hours for a typical design case like those presented in 5.1.10)

On the software side, in order to perform the evaluation of the architecture with the desired application, the target source code is compiled by means of an automatically retargetable compiler. The compiler is able to optimize the instruction flow with respect to the instruction set and to the architecture of the processor under prototyping. In order to do so, it extracts information on the underlying ASIP architecture from the same configuration specification provided in input by the DSE engine. The compiler then retargets itself according to the considered ASIP configuration specification. After compilation, the program can be executed on the ASIP actually implemented on FPGA.

The hardware structures, within the scope of this work, have been instrumented with dedicated activity probes and counters, capable of collecting performance (in terms of numbers of cycles) and the activity figures (i.e. number of accesses) of the blocks in the considered ASIP. In addition to estimation of functional metrics, these activity traces are used to estimate physical metrics, such as dynamic power consumption. This estimation is performed through a layer of analytical modeling that will be described in Sect. 5.1.4.

The left side of Figure 5.2 highlights the time necessary for traversing the entire baseline flow, for each processor configuration. On a workstation equipped with a Core2 Q6850 processor running at 2 GHz, 8 GB of DDR3 RAM memory and running Ubuntu 10.04 Linux OS with the Xilinx ISE FPGA synthesis/implementation toolchain for a Xilinx Virtex5 LX330

device, roughly 20 seconds for the platform VHDL generation and code compilation were needed. The largest part of the time was consumed by the FPGA synthesis/implementation toolchain which took about a hour to complete. This time has been measured with the FPGA device operating far from its resource capacity limit. Therefore, the basic scenario for design space exploration, employing an entire run for each of the N candidate configurations, would require approximately N hours to complete.

5.1.4 Area and Power/Energy models

After the execution, collected emulation data are translated to estimated physical metrics by means of dedicated area, frequency and energy models, and fed back to the DSE engine. The translation is performed by means of a set of analytical expressions that allow the evaluation of the energy, area and critical path contributions of the functional blocks inside the library. Such analytical expressions depend both on their static parameterization and switching activity. The power modeling layer is able to separately account for leakage and dynamic power consumption. Such models refer to a target technology library.

The modeling phase has to acquire information on the architecture configuration, therefore it takes as input a description of the current configuration with a dedicated file generated during the RTL generation phase. Examples of the relevant architectural information are the number, kind and depth of the different issue slots, number of ports, width and depth of the register files, size of the memories, and so on. The on-hardware emulation then provides the activity traces necessary to perform the analytic estimation of interest.

For space reasons, it's preferrable not to list here all the model formulae for estimation of area occupation, leakage and dynamic power consumption of every functional block. However, tables 5.1 and 5.2 provide information on how the area and power models account for the different processor blocks. The entries in the tables report the dependency between the related block and the architecture parameters. These parameters are then used in conjunction with technology-dependent normalized values (e.g. per-bit area occupation, per-bit leakage power numbers) to obtain the actual metric estimation of interest. For instance, the second line of Table 5.1 reports on the area contribution related to multiplexing logic around the FUs inside an issue slot. Such area occupation is proportional to the number of operations actually available inside the issue slot. Similarly, the third line of Table 5.2 indicates that the leakage power consumed by the same logic depends linearly on the occupied area, while the dynamic power consumption is proportional to the logic access rate. This access rate factor is extracted, within the design flow, by the activity traces obtained through on-FPGA emulation.

In reading the Tables, it's useful to point out that the $\#operations_{IS}$ parameter is different from the $\#operations_{P}$ in that the former refers to the operations executable by a single issue slot, while the latter refers to all the operations simultaneously executable by the entire processor, i.e. by all the issue slots. Also it's useful to highlight the difference between the operation count parameters $\#operations_{IS}$,$\#operations_{P}$ and the current clock tick operation count parameter OPC. The first two parameters are static quantities, depending only on the architecture configuration, and impact on area occupation and static power consumption, while the latter is a dynamic quantity that accounts for the number of currently executing operations, and obviously impacts on dynamic power consumption.

The models, that is, the formulae used to calculate power and area and whose main dependencies are reported in Tables 5.1 and 5.2, were obtained through limited experimenta-

	Area
FU	$\propto FU\_Port\_Size$
IS mux logic	$\propto \#operations_{IS}$
Register File	$\propto RFdepth, RFwidth$
Decoder	$\propto \#operations_{P}$
Result Select Network	$\propto \#operations_{P}(\#RFports + \#ISoutput\_ports)$
Sequencer	constant
FIFOs	$\propto Mem\_size$
Program Memory	$\propto Mem\_size$
Data Memory	$\propto Mem\_size$

Table 5.1: Area models dependency recap. Subscripts for operations separate operation count for the single issue slot (IS) from the overall processor count (P)

	Leakage Power	**Dynamic Power**
FU	$\propto FU\_Port\_Size$	$\propto FU\_Port\_Size, Access\_Rate$
IS mux logic	$\propto Area$	$\propto Access\_Rate$
Register File	$\propto Area$	$\propto RFwidth, Access\_Rate, \#ports$
Decoder	$\propto Area$	$\propto OPC$
Result Select Network	$\propto Area$	$\propto Access\_Rate, OPC$
Sequencer	constant	constant
FIFOs	$\propto Mem\_size$	$\propto Mem\_size, Access\_Rate$
Program Memory	$\propto Mem\_size$	$\propto Mem\_size$
Data Memory	$\propto Mem\_size$	$\propto Mem\_size, Access\_Rate$

Table 5.2: Power models dependency recap. OPC stands for Operation Per Cycle. Program Memory is assumed to have 100% access rate

tion. As part of the experiments, different processors were synthesized and analyzed in detail, using Synopsys front-end and back-end tools, including lay-out and wireload models. The results of these analyses were used to calculate the detailed normalized per-bit power and area numbers. However, the results are not yet suitable for very wide ranges of parameters. For example, the linear dependency on port width for function unit metrics is not always applicable.

To date, the experiments leading to the normalized per-bit power and area numbers involved 16- and 32-bit datapaths. When scaling these datapaths between these numbers, and comparing additional detailed synthesis results with results from the models, obtained total overall accuracy of the formulae remains within 10%.

5.1.5 The proposed design flow

To allow fast prototyping of multiple candidate interconnect configurations inside the system, the baseline flow has been extended with a utility that analyzes the whole set of configurations under prototyping, synthesizes the WCC and creates the configurable hardware and the software functions needed to map each candidate configuration on top of the overdimensioned hardware.

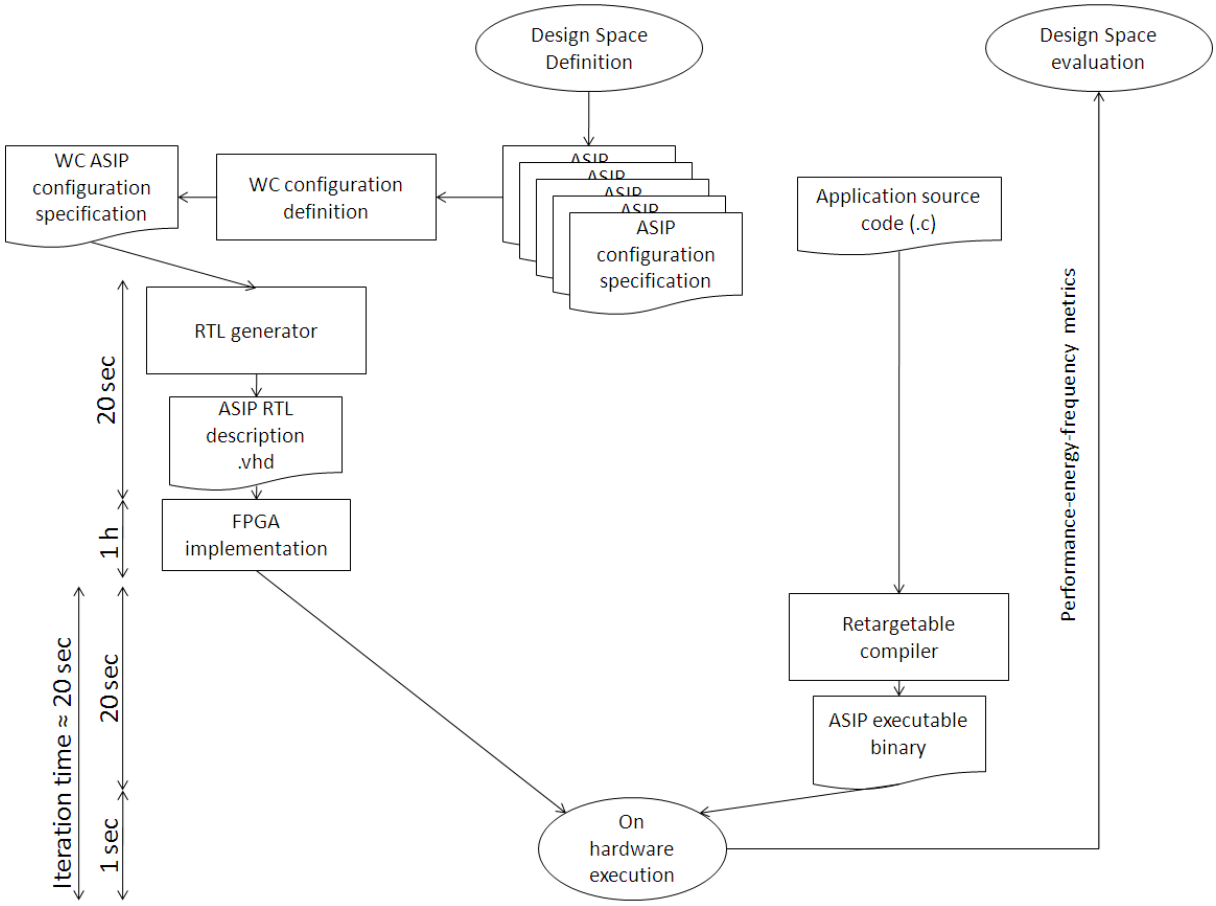


Figure 5.3: Prototyping flow extended with runtime reconfiguration capabilities (Evaluation time for N candidate architectures is approximately 1 hour and $N \times 20$ seconds)

Figure 5.3 shows the extended flow. By comparison with Figure 5.2, it can be noticed how many ASIP configuration descriptions can be passed as input to the flow. Based on such input sets, the extended flow identifies the WCC and creates its configuration description, in compliance with the same description format of the reference flow. As a consequence of this modification of the flow, Figure 5.3 shows how the time necessary to perform the evaluation of N different candidate ASIP configurations is now reduced to roughly $N \times 20sec + 1hour$. Reported times are meaningful examples of the duration of every evaluation step, as can be measured in real design cases. The precise numbers are obviously dependent on the application, on the hardware architectures and on the system used for the implementation flow.

The synthesis algorithm is described in Sect. 5.1.6. The hardware and software support implementation details are respectively provided in Sects. 5.1.7 and 5.1.8.

5.1.6 The WCC synthesis algorithm

Algorithm 1 is the algorithm used to identify the worst case configuration, for the considered input set of candidate ASIP configurations. In the extended flow, all the design points under

test must be provided to the flow at the beginning of the iterative process.

ALGORITHM 1: Worst Case Configuration Identification

Input: A set of K candidate ASIP configurations
Output: Worst Case Configuration (WCC)
$N_{IS}(WCC) = 1$;
$RF\_size(1, WCC) = 0$;
$FU\_set(1, WCC) = \{\}$;
for *each candidate configuration c with* c=1,...,K **do**
  $N_{IS}(WCC) = \max\{N_{IS}(i)\}$; with i=1,...,c
  for *each issue slot x of the configuration c* **do**
    $RF\_size(x, WCC) = \max\{RF\_size(x, i)\}$; with i=1,...,c
    $FU\_set(x, WCC) = FU\_set(x, c) \cup FU\_set(x, i)$; with i=1,...,c
  end
end

The WCC is defined iteratively while analyzing all the candidate configurations. At every iteration, it is updated according to the design point currently under analysis. At iteration N (i.e. parsing the $N-th$ candidate configuration under test c)

- The number of issue slots inside c is identified and compared with previous iterations. A maximum search is performed, then, if needed, the WCC is modified to instantiate $N_{IS}(WCC)$ issue slots. For every issue slot of every candidate configuration c, there must be one and only one corresponding issue slot in the WCC.
- For every issue slot x inside c, the size of the associated register file is identified and compared with previous iterations. A maximum search is performed, then, if needed, the register file related to the issue slot x inside the WCC is resized to have $RF\_size(x, WCC)$ locations. Since there is one and only one issue slot in the WCC that corresponds to the issue slot x of c, the related register file in WCC can be identified without any ambiguity.
- For every issue slot x inside c, the set of FUs is identified and compared with previous iterations. The issue slot x inside the WCC is modified, if needed, to instantiate a set of FUs being the minimum superset of FUs used in previous configurations.

To clarify the WCC construction algorithm, let's consider a possible design space with the following three different candidate configurations:

- A first candidate configuration with 3 issue slots, respectively instantiating the functional units $\{FU\_A, FU\_B\}$, $\{FU\_A, FU\_C, FU\_D\}$, $\{FU\_B, FU\_C\}$ and the register files of sizes 12, 8 and 16 (always expressed in terms of 32-bit registers).
- A second candidate configuration with 2 issue slots, respectively instantiating the functional units $\{FU\_A, FU\_B, FU\_D\}$, $\{FU\_A, FU\_B, FU\_C\}$ and the register files of sizes 24 and 16.
- A third candidate configuration with 4 issue slots, respectively instantiating the functional units $\{FU\_C\}$, $\{FU\_A, FU\_B, FU\_C, FU\_E\}$, $\{FU\_C, FU\_D, FU\_E\}$, $\{FU\_A, FU\_B, FU\_C\}$ and the register files of sizes 8, 32, 24 and 16.

Decomposing the outer loop of the algorithm, the WCC is iteratively constructed as follows:

1. The WCC instantiates 3 issue slots, with functional units $\{FU\_A, FU\_B\}$, $\{FU\_A, FU\_C, FU\_D\}$, $\{FU\_B, FU\_C\}$ and register file sizes 12, 8 and 16 respectively.

2. The WCC is then modified to instantiate 3 issue slots, with functional units $\{FU\_A, FU\_B, FU\_D\}$, $\{FU\_A, FU\_B, FU\_C, FU\_D\}$, $\{FU\_B, FU\_C\}$ and register file sizes 24, 16 and 16 respectively. It can be noticed how the functional unit sets are being populated according to the union of the candidate configurations sets.

3. Finally, the WCC is modified to instantiate 4 issue slots, with functional units $\{FU\_A, FU\_B, FU\_C, FU\_D\}$, $\{FU\_A, FU\_B, FU\_C, FU\_D, FU\_E\}$, $\{FU\_B, FU\_C, FU\_D, FU\_E\}$, $\{FU\_A, FU\_B, FU\_C\}$ and register file sizes 24, 32, 24 and 16 respectively.

5.1.7 Hardware support for runtime reconfiguration

The software runtime reconfiguration capability is supported by two hardware modules, automatically generated and instantiated in the overdimensioned WCC architecture basing on the set of different input configurations that are passed to the exploration engine.

The first module is the *instruction adapter*, a programmable decoder that interprets and delivers every single chunk of the VLIW instruction to the relevant hardware element. For each candidate architecture in input, knowing the complete set of architecture parameters, the instruction bits can be split in sub-ranges that identify specific control directives to the datapath. Examples of such bit ranges are operation codes (that activate specific function units and specific operations inside the issue slots), index values (used to address the locations to be accessed in the register files), and configuration patterns (used to control the connectivity matrices that regulate the propagation of the computing data through the datapath). The width and the position of the boundaries between the bit ranges are not fixed but instead depend on the architectural configuration that must execute the instruction.

The configurable instruction adapter is in charge of translating the instructions produced by the compiler, which re-targets itself for each candidate ASIP configuration, into an instruction executable on the WCC. All the sequences in the instruction related to a given slice of the configuration under evaluation are adapted in size and dispatched to the corresponding slice of the WCC. The value of each control directive is modified to ensure the instruction will provide the correct functionality on the overdimensioned prototype, despite the presence of additional hardware. All slices that do not exist in the configuration under test are disabled using dedicated opcodes.

Figure 5.4 shows an example of how the instruction adapter works. In the example, an instruction produced by the compiler for a configuration under test *c* requires the activation of the FU in charge of performing shift operations (*shu*) in $IS1$. Inside the candidate configuration *c* alone, the instruction decoder would statically split the VLIW instructions as it is stored in the program memory into different opcodes and pass each of them to the proper issue slot. Inside the issue slot $IS1$, only the *shu* function unit would then be activated, basing on the ocpode value.

In the extended flow, where the number of issue slots is potentially different (more issue slots are usually instantiated in the WCC) from the one of each candidate configuration, the instruction adaptation is necessary to execute the same instruction binary on the WCC. The adapter is adequately programmed via software through a memory-mapped register write, in order to obtain information on the configuration identifier. According to this value, it then decodes the different instruction fields, generates a new (longer) instruction word and dispatches the new opcodes according to the mapping strategy. In the example of Figure 5.4 $IS1$ is mapped onto ISm in the WCC. The opcode originally targeted for $IS1$ is thus dispatched to ISm. Its value is translated to activate the *shu*, taking into account the architectural composition of ISm in terms of its worst case function units. Since the opcode values may differ from each candidate configuration to the WCC, the opcode width is adapted to the WCC architecture. Similar dispatching/translation is applied by the adapter to the other instruction fields.

The second hardware module, the *memory router*, is introduced in order to support different connections between the pool of data memories instantiated in the WCC and the issue slots. The baseline flow supports, directly inside the application code, the explicit positioning of variable and data structures in each memory inside the architecture. To keep this capability in the extended flow, and to allow at the same time the possibility of arbitrarily dispatching the operations to the issue slots inside the WCC, the memory router provides connectivity between memories and issue slots to be programmable according to the configuration under prototyping. The programmability mechanism follows the same logic of the opcode dispatching process implemented inside the instruction adapter that we previously mentioned. An identifier is stored, for each candidate configuration, inside a memory-mapped register. Its value drives the connections to the different memory modules.

5.1.8 Software support for runtime reconfiguration

Software support for reconfiguration is realized simply writing a memory mapped register, which stores a unique configuration identifier and acts as an architecture selector, directly accessible by a function call at C application level. The automatic flow provides, in the form of a simple API, the function that accesses this register. The value stored in the register, as already described in Sect. 5.1.7 controls the instruction adapter and the memory router, to select one among the candidate configurations under emulation. The generated routines are suitable to be compiled and linked by the application executable file running on an host processor controlling the ASIP.

In case the user does not modify the application code to instrument it with the function calls that select the architecture configuration under test, the extended framework provides an alternative way, which employs Xilinx System Generator (SysGen). The utility was used to enable direct access from a host workstation to the configuration selector, and to allow easy access to the emulation results. SysGen consists of a set of Matlab/Simulink blocks and routines that enable, among other features, selecting a hardware system, implemented on an FPGA device, as if it was a Matlab/Simulink block, and to allow the cooperation of software-based simulation and on-hardware prototyping.

SysGen provides the capability of automatically creating the hardware and software support for data exchange between the FPGA board and a host processor. It also enables stimuli generated by a software-based simulation environment (such as a Matlab function itself or an HDL simulated stimuli generator like Modelsim) to be fed as input to the hardware. In

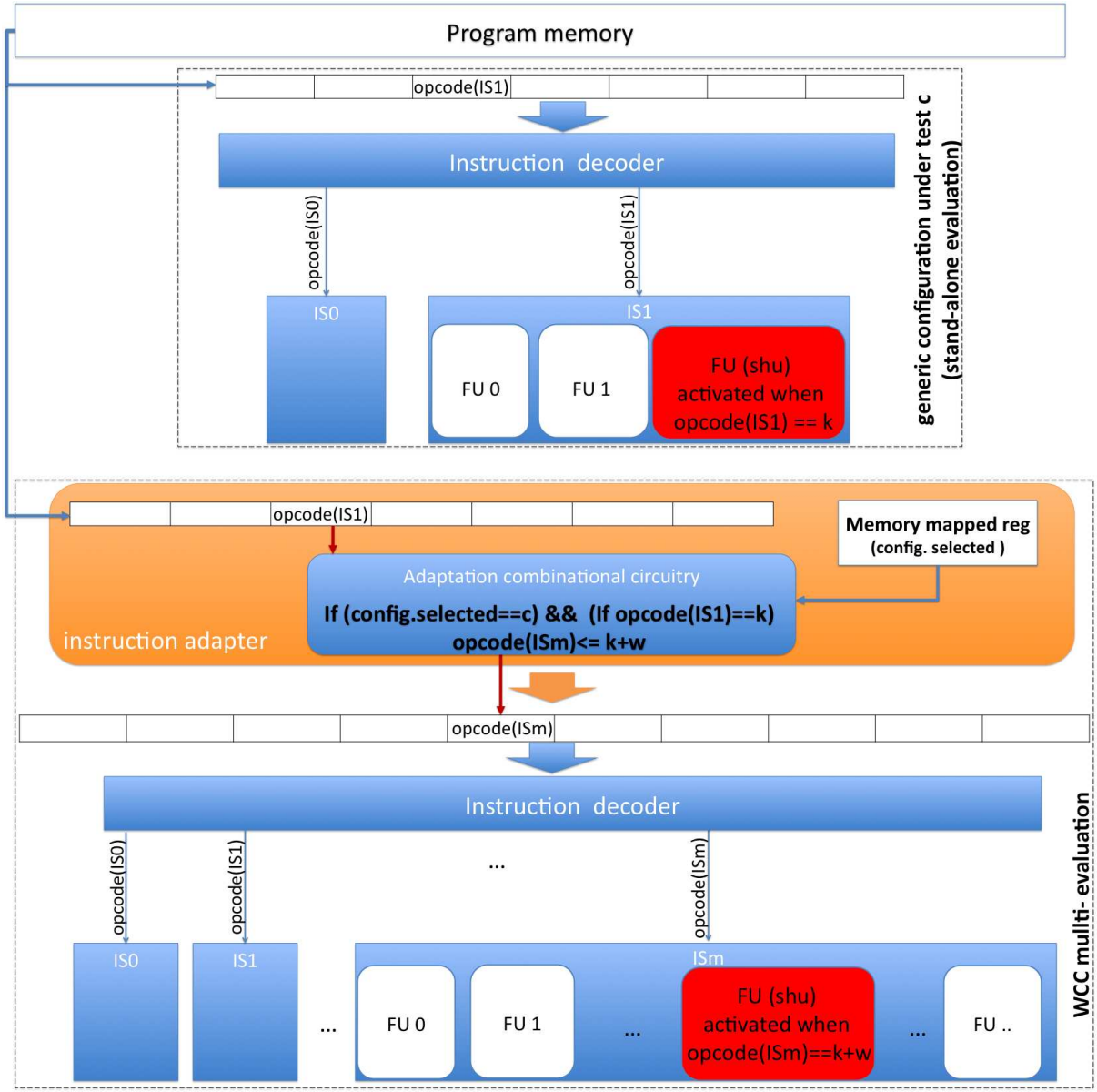


Figure 5.4: Example of instruction adapting

this way, after every execution, the user can choose the next configuration to evaluate directly from the Simulink graphic interface and automatically restart the application.

5.1.9 Implementation degradation and overhead reduction techniques

As specified in the definition of the instruction adapter and memory router in Section 5.1.7, the modules were designed to not introduce any spurious latency cycle in the WCC with respect to the single configurations. For this reason, cycle-accuracy is always guaranteed. However, as a consequence of the provision of runtime reconfiguration capabilities, a degradation of the quality of results may be expected with respect to the hardware implementa-

tion of a single sample configuration on the FPGA device. In particular, two aspects related to the implementation quality can be affected and potentially preclude the usability of the approach. Firstly, the FPGA area occupation of the WCC determines whether the prototyping platform fits on one given target programmable device or not. Secondly, in case the additional hardware (instruction adapter and memory router) affects the critical path of the candidate system, the WCC working frequency might have an impact on the emulation time.

These possible drawbacks have been evaluated in detail while implementing the flow extensions. In order to minimize the area occupation of the WCC netlist, the minimum needed subset of hardware resources is identified by the WCC synthesis algorithm. The mapping of the instruction fields inside the instruction adapter is optimized to reduce as much as possible the number of different adaptations that must be allowed by the circuitry, to minimize its area and, being the adapter a combinational module, its propagation delay to prevent impacting the overall working frequency of the prototype. The resulting area/frequency overhead is quantified in Sect. 5.1.10. Readers can find there the experimental results and discuss how the mentioned overhead can be effectively controlled and how the proposed approach is applicable to systems characterized by considerable complexity.

5.1.10 Use Cases

In this section two use case scenarios for the previously described runtime reconfiguration techniques are described. The first use case involves exploration of a possible configuration space for the architecture of a single ASIP, running an image filtering kernel application. The second use case involves exploration of a multi-ASIP system, composed of three processors, a packet-based on-chip interconnection switch, two different shared modules (a UART controller and a hardware Test&Set semaphore bank for lock-based synchronization) and a MicroBlaze platform control processor. The running application is an MJPEG codec, partitioned and mapped on the three ASIP processing elements. For both sets of experiments, the adopted hardware FPGA-based platform features a Xilinx Virtex5 XC5VLX330 device, counting over 2M equivalent gates.

Single-ASIP exploration

Author will present the results obtained while performing the single-ASIP architecture selection process over a set of 30 different ASIP configurations. The explored design points were identified considering different permutations of the following processor architectural parameter values:

- $N_{IS}(c)$: 2 or 3 or 4 or 5;
- $FU\_set(x,c)$: from 3 to 10 FUs per issue slot;
- $RF\_size(x,c)$: 8 or 16 or 32 entries, each 32-bits wide;
- $n\_mem(c)$: 2 or 3 or 4 or 5.

A filtering kernel was compiled for every candidate configuration and the resulting binaries were executed on the WCC prototype, adequately re-configured. Although the abovementioned design space could seem small, this design case is very realistic. In fact several

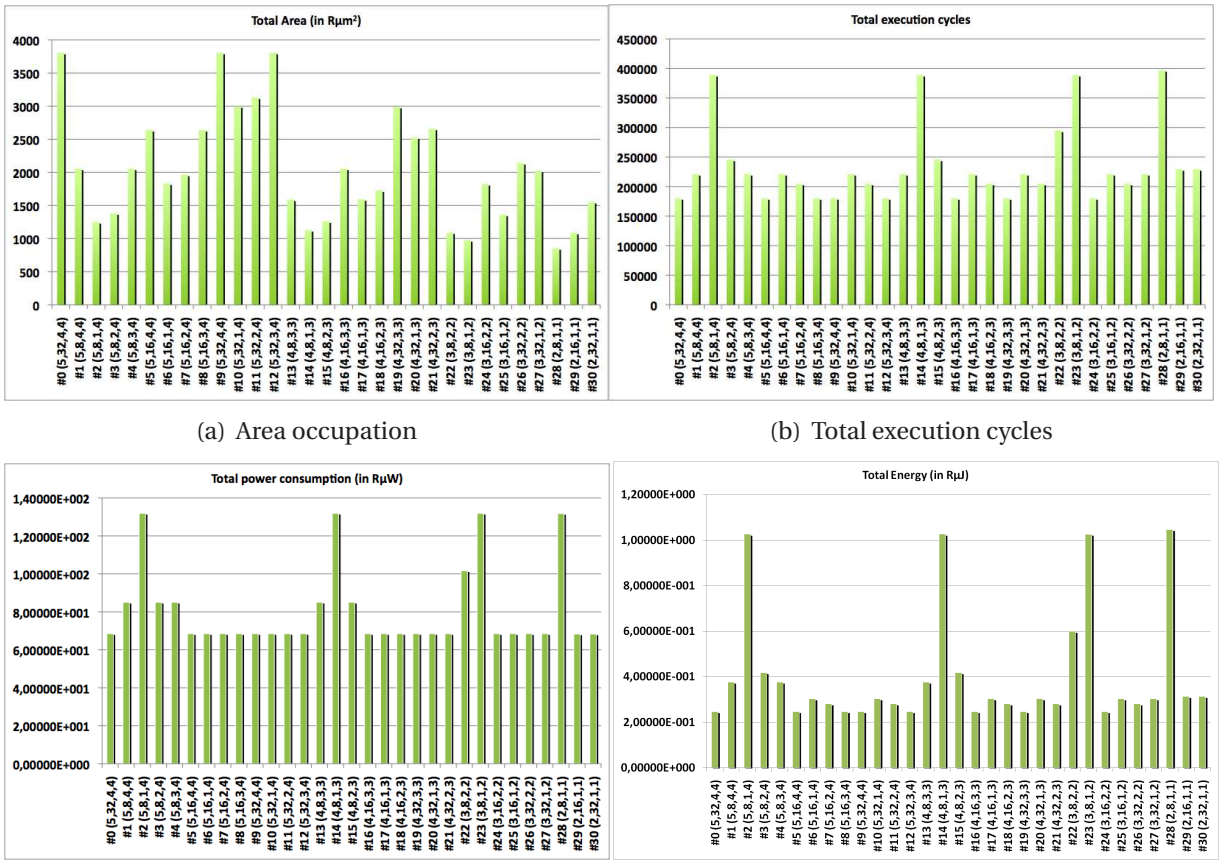
DSE engines, like [12], when selecting the best system-level configuration over millions of design points, often start by exploring the system-level composition (number and kind of cores in the system) or the interconnection topology and application task mapping. This first space design space pruning is usually performed using a tuned high-level software simulator, which is very fast but not capable of detecting the precise functional and physical behavior of the micro-architectural modules. To be effective, the simulator is usually fed with latency and power numbers related to the execution of an application on a given processor instance. Such numbers are typically obtained from a single-processor detailed prototyping. This use case not only allows to assess the feasibility of the approach, but also presents an example of such kind of analysis.

In Fig. 5.5, the results of the evaluation obtained with respect to total execution time, total latency, total energy and power dissipation are shown. The energy results have been calculated assuming an identical clock frequency for all the considered ASIP configurations. In theory, energy could be modeled only after an operating frequency estimation step is performed. The reason behind this is obviously the different critical paths that might appear in different configurations. However, since in this use case logic synthesis results identified the critical path in the same piece of logic for all the ASIP configurations, the operating frequency can be assumed to be the same without introducing any error.

To avoid the disclosure of sensitive industrial information, authors do not report absolute power, frequency and area numbers. Instead, a back-annotation of the emulation results referring to "comparative" numbers for energy and area contributions for the functional blocks in the ASIPs has been done. The area, power and energy figures are thus reported respectively in $R\mu m^2$ (relative square microns), $R\mu W$ (relative microWatts) and $R\mu J$ (relative microJoules), while, for the execution time, the number of cycles is chosen. The use of such relative units does not hide the usefulness of the performed analysis for a prospective designer approaching comparative architecture selection. Multi-constraint optimization can be effectively performed. For example, imposing a constraint on maximum execution time (e.g. 200K cycles), the user could identify a subset of candidates satisfying the constraint (configurations $\#\{0,5,8,9,12,16,19,24\}$). Then, among these, one could choose the best configuration with respect to power or area (#24).

From the performed analysis a designer could estimate $topology\_29$ to be the joint optimal configuration, for the considered target application, from the points of view of energy consumption, execution time, and area occupation. To identify possible computation bottlenecks and power hot-spots inside the architecture, performance and power profiling at the functional unit level can also be obtained, referring to each single functional unit included in the configurations under test. As an example, it's shown in Fig. 5.6 a plot reporting power consumption of each function unit in a particular configuration, during the execution of the already mentioned filtering kernel binaries.

The cycle-accurate correctness of the emulation of a candidate configuration with the WCC is guaranteed by construction of the WCC architecture. In fact, every instruction that traverses the ASIP datapath, both in the candidate configuration and in the WCC, undergoes the same exact logic path. The WCC architecture does not insert any new pipeline stage in the instruction path with respect to the ASIP. What can change is only the operating frequency of the WCC and thus the resulting execution real time, due to the more complicated combinatorial logic (e.g.: instruction adapter), but the emulated CPI will be exactly the same (as a count of clock cycles). To confirm this behavior, it's useful to compare the experiment results obtained by the prototyping all the candidate configurations on the WCC architecture



(a) Area occupation

(b) Total execution cycles

(c) Power consumption of FUs

(d) Energy consumption of FUs

Figure 5.5: Use case results. Every configuration is labeled with a 4-tuple, whose elements represent the total number of issue slots, the register file capacity (in 32-bit words), the number of fully-featured issue slots, the number of data memories. Execution cycles are reported for the different configurations under emulation. Moreover, the modeled power consumption (expressed in $R\mu W$), area occupation (expressed in $R\mu m^2$) and total energy consumption (expressed in $R\mu J$) figures are reported.

with the results of their stand-alone evaluation. As expected, exactly the same "functional related" (execution time, latency, switching activity) performances are estimated. Cycle/signal level accuracy can thus be assessed for the presented approach.

In order to evaluate the speed-up that was allowed by the proposed approach, it should firstly be considered the time needed for the same design exploration performed using the classic approach. This requires the designer to go through the implementation flow for all the candidate design points. The related time depends on the complexity of the considered design point. For the set of candidates in this example, implementation time ranged from roughly 20 minutes to 45 minutes, for a total of 15 hours. When using our approach, on the other hand, only one synthesis is required. Such synthesis of the WCC required roughly 45 minutes, allowing for a 20x speed-up. In this analysis, author is not accounting for the time needed for the execution of the kernels on the prototypes, since it is negligible (few seconds) with respect to the implementation time.

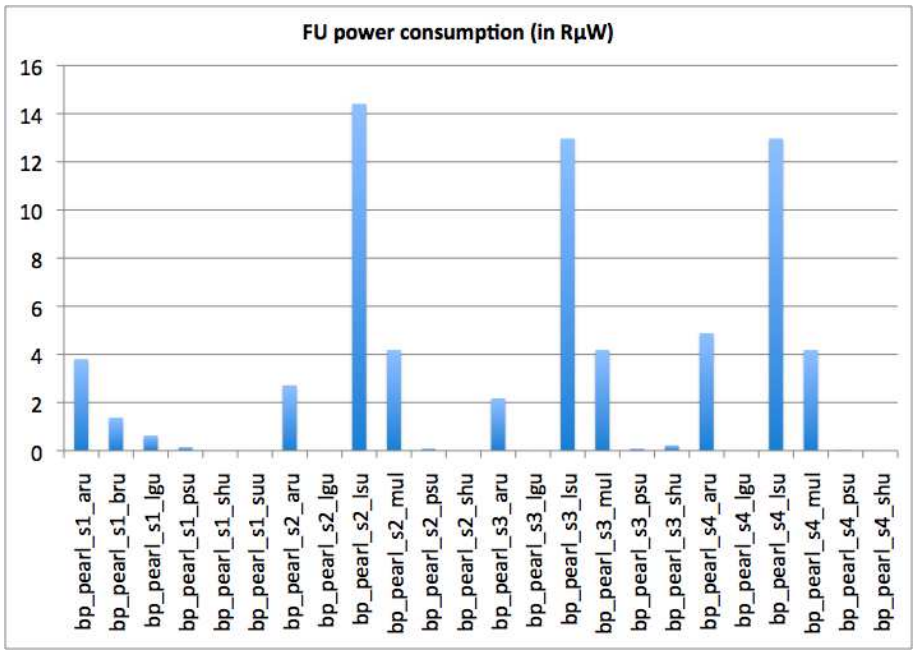


Figure 5.6: Power consumption for each FU in a particular configuration, composed of four issue slots, with register files of 16 entries, reported in $R\mu W$.

Multi-ASIP configuration

A second use case is now presented, which validates the proposed emulation approach inside the exploration of a multi-ASIP system. Several are the reasons for which such kind of use case is important. Firstly, since cycle-accurate simulation becomes slower as the size of the simulated system increases, obtaining execution traces of multi-core systems by means of software-based methods quickly becomes unpractical. As opposed to software-based simulators, the emulation speed achievable with FPGA devices is not affected, in general, by the system size. The only requirement is that the system under prototyping fits in the target configurable device. Therefore, a multi-ASIP system exploration should, in terms of overall speed-up, favor FPGA-based emulation approaches as opposed to pure software simulation.

Moreover, the use case shows that it is possible to cross-optimize the micro-architectures of the ASIPs, exploiting results obtained from a complete system prototyping. In fact, it's demonstrated how the designer is able to observe the mutual influence among the processors and between the processors and the surrounding environment (interconnect infrastructure, peripherals, shared memories) without relying on further software-based simulation steps.

Author now presents the prototyping results obtained by the execution of an MJPEG encoder on a parallel MPSoC composed of a host processor and three ASIPs, interconnected by means of a Network on Chip subsystem. The system is represented in Figure 5.7.

The application is partitioned into four parallel computing tasks, communicating through FIFO channels, according to a programming model based on Kahn Process Networks ([9]). In detail:

- the host processor is in charge of initializing the program and data memories inside the ASIPs and executing a parallel task (named *Video_in*). The *Video_in* task dispatches the

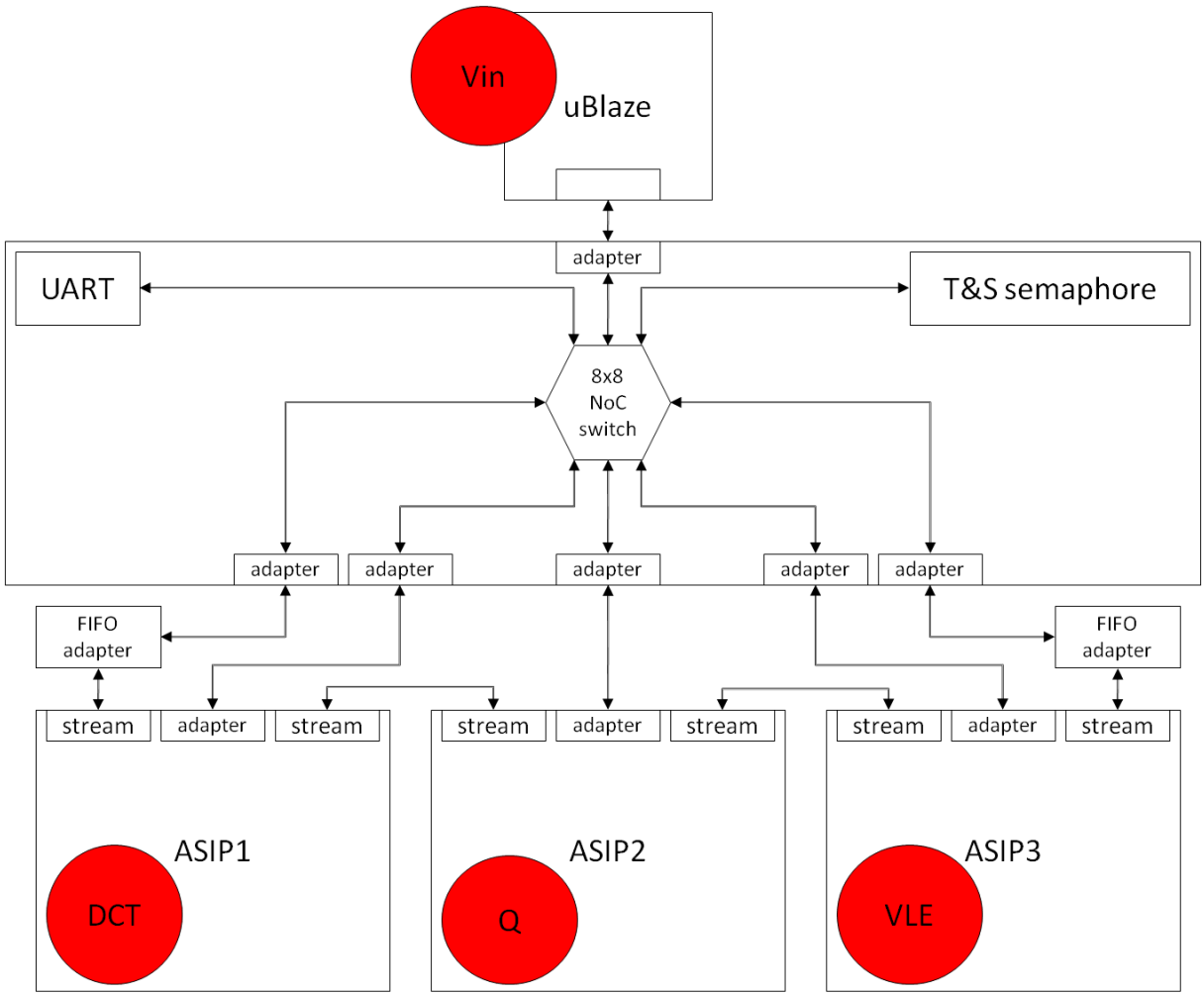


Figure 5.7: Multi-ASIP platform under exploration

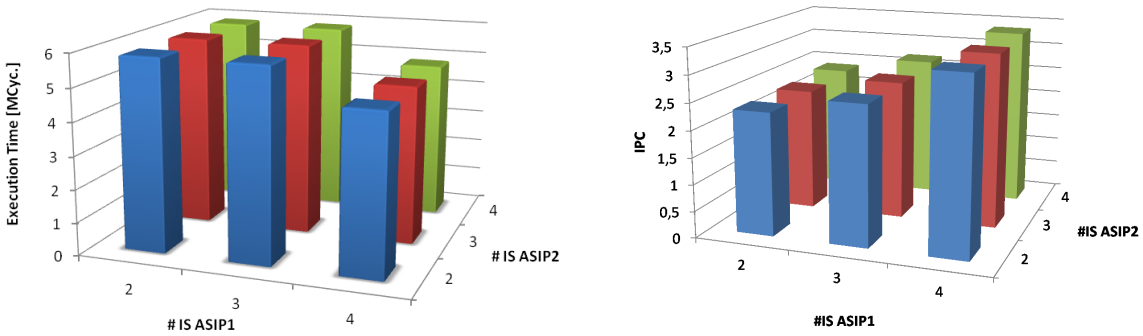
input stream pixels, the header and footer information to the other tasks

- the second task then involves the DCT encoding calculations and is mapped on the first ASIP (ASIP1)
- the third task takes care of the quantization process (*Q* in short) and is mapped on the second ASIP (ASIP2)
- the fourth and last task performs the variable length encoding part (*VLE*) and is executed by the third ASIP (ASIP3).

In order to explore the micro-architectures of the single ASIPs, ASIP1 and ASIP2 were enriched with the support for fast prototyping. ASIP3 on the contrary has been implemented as a single static configuration. By doing so, ability to investigate on the impact that the customization of ASIP1 and ASIP2 has on the metrics related with the execution of the *VLE* task, which runs on ASIP3, has been preserved.

In detail, in the presented results the number of issue slots and the kind of included function units inside ASIP1 and ASIP2 were the variables that defined the exploration space. We

let ASIP1 and ASIP2 processors issue slot counts assume all the possible combinations of values from 2 to 4.



(a) Execution Times

(b) IPC per ASIP

Figure 5.8: Second use case results. Every horizontal axis captures the number of issue slots inside processors ASIP1 and ASIP2. Execution cycles are reported for the different configurations under emulation. The IPC for every ASIP is also reported.

Figure 5.8 plots the results of the exploration in terms of execution times (measured in clock ticks) and IPC per ASIP. The execution times are probed on the ASIP3 processor. Considering the typical communication pattern of an MJPEG encoder, and considering the task mapping explained in Figure 5.7, it is clear how the ASIP3 will have to wait for the ASIP1 and ASIP2 to complete their assigned tasks (*DCT* and *Q*) to complete its own task. In fact, this is a classic example of dataflow communication pattern, therefore the execution time of ASIP3 fully characterizes the overall application execution time.

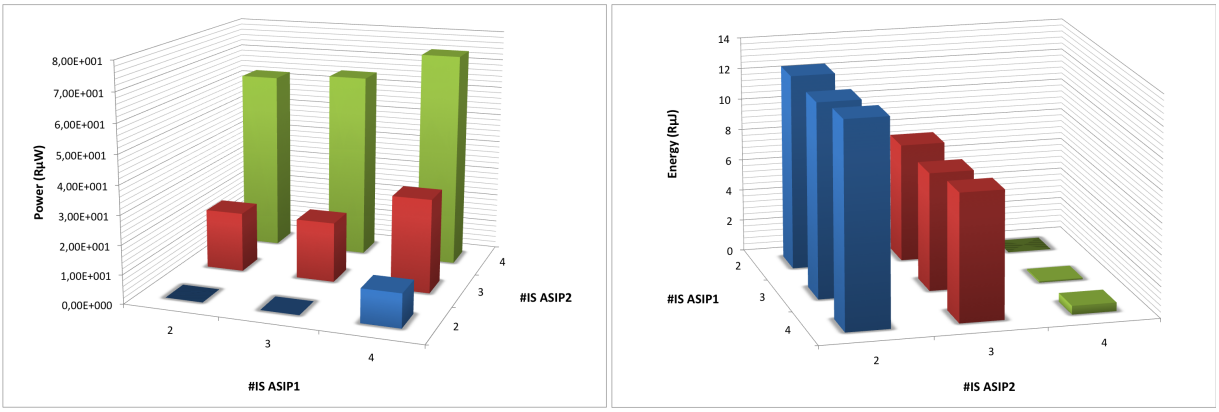
The power results were instead acquired as a sum of the power dissipation occurring in the ASIP1 and ASIP2 processors, for their different configurations. ASIP3 was not considered in the power estimation process, since its architectural configuration has been kept constant.

By looking only at the execution time results, it seems that the architectural modifications performed inside the ASIP2 processor (which is assigned the quantization task) were unable to produce an impact on the ASIP3 execution time. This means that, as for what regards the execution time, the DCT task is the most hungry, and optimizing its execution is key to obtaining an overall performance improvement.

Interestingly, changing the number of issue slots from 2 to 3 does not affect the execution time. We could argue that this is related to how much the VLIW compiler proficiently uses the available issue slots to exploit the DCT task available parallelism.

It's possible to provide an estimation of the system power and energy consumption. Figure 5.9 shows the energy (measured in $R\mu J$) and power consumption (measured in $R\mu W$) figures. It is important to mention that, as already explained in Sect. 5.1.10, the energy estimation has been carried out without accounting for the real operating frequency of the processors. This is still a reasonable assumption, since the most relevant piece of logic is included in all the processor configurations and limits the critical path, regardless of the architectural modifications that we made. We are also assuming that the entire processing section of the system runs within a single frequency domain.

By looking at the power numbers, on the other hand, we see that the architectural changes to both the ASIP1 and ASIP2 processor configurations have an impact on the overall results.



(a) Power consumption

(b) Energy consumption

Figure 5.9: Second use case results. Every horizontal axis captures the number of issue slots inside processors ASIP1 and ASIP2. Execution cycles are reported for the different configurations under emulation. The modeled power consumption (expressed in $R\mu W$) and energy consumption (expressed in $R\mu J$) figures are reported. Values are expressed as offset with respect to a zero-point (lowest value of power and energy consumption).

The numbers suggest that the most convenient architectural configuration, in terms of energy, features 2 issue slots in the ASIP1 processor and 4 in the ASIP2 processor.

In the use case exploration, we kept the NoC switch configuration constant and changed the configurations of ASIP1 and ASIP2. This means that the candidate configurations with the highest number of issue slots could have experienced more network congestion than the ones with less issue slots, simply because of the increased memory references issued per cycle. Since this difference exists among the various candidate configurations, the WCC architecture exactly reproduces it when configured to emulate the different candidate configurations. This is another consequence of the cycle-accuracy that the WCC architecture has with respect to the direct emulation of each single candidate configuration.

Both the functional results and the cycle counts obtained with this FPGA approach and the baseline software simulation were completely equivalent. But, while cycle-accurate software simulation required few minutes (roughly five on average per configuration), onboard execution on the FPGA prototype required only few seconds (roughly two) to emulate each candidate architecture. A synthesis/implementation flow, performed on an Intel Quad-Core machine with commercial tools, required less than half an hour to complete. Such time obviously depends on the size of the system, but can be estimated in the order of one hour for moderately complex systems. According to the mentioned numbers, the presented approach allows a time saving that increases with the number of candidate topologies under prototyping, outperforming soon (for approximately ten candidate design points involved in the design process) software-based simulation. Moreover, the software-based simulation is not always effective. For example if design cases imply the evaluation of runtime middleware policies or network routing protocols, whose effectiveness must be measured on execution times much longer than a single processing kernel, simulation times become unaffordable, making FPGA emulation the only available evaluation method and our approach fundamental for its application to DSE.

Hardware overhead due to runtime configurability

In this section, a quantitative analysis on the overhead introduced by the addition of reconfiguration support is presented, in terms of occupied hardware resources and critical path degradation. For this scope, referring to the previously presented use case, a comparison between the results obtained by a synthesis of the WCC and the most hardware-hungry configuration under test (when implemented for a stand-alone evaluation, without support for runtime reconfiguration) is performed. Again, the WCC is generated to support 30 different configurations. This degree of exploration is reasonably interesting for this kind of applications, though larger pools of configurations can always be used.

As can be noticed from Table 5.3, the introduced overhead in FPGA resource utilization is limited, also considering that WCC is built iteratively from a set of 30 different candidates. Considering that author is comparing the WCC to the most power-hungry of the 30 input ASIP configurations, the result would be expected more related to the overhead introduced by the logic necessary for the reconfiguration mechanism (i.e. instruction adapter and memory router modules) than to the overprovision of actual architectural functional blocks (i.e. issue slots and function units). Table 5.3 confirms that, in terms of FPGA resources, this overhead is limited to 10% of the largest candidate ASIP configuration. Moreover, since the memory router and instruction adapter are mostly combinational modules, the overhead is almost entirely consumed by the look-up table (LUT) FPGA slice logic.

Also, a comparison related to the logic synthesis maximum operating frequency is presented, being this a limiting factor for the overall emulation speed. Results presented in Table 5.4 show that critical path is almost not impacted (less than 0.1%) by the presence of the hardware structures implementing reconfiguration support. This result is particularly relevant, since the instruction adapter and memory router are made out of almost completely combinational logic, therefore their potential impact on the critical path was high. However, after the insertion of such modules, the critical path still resides inside more complex function units (e.g. multiply-and-accumulate) and is not affected by the inserted logic. The minimal increase is due to unpredictable behaviors of the synthesis algorithm.

The results show how the overhead reduction mechanisms explained in Sect. 5.1.9 effectively allow to perform prototyping of a reasonable number of different configurations without significant emulation time degradation.

	Occupied Slices	**Slice Registers**	**Slice LUTs**
Largest configuration	19859	6923	16387
WCC	21278 (+7.1%)	6931 (+0.001%)	17951 (+9.5%)

Table 5.3: FPGA hardware overhead figures

	Critical path
Largest configuration	$9.809ns$
WCC	$9.817ns$

Table 5.4: FPGA critical path overhead figures

Chapter 6

Extending FPGA fast prototyping through binary manipulation

This chapter will present an alternative approach to the hardware-based fast FPGA prototyping, thanks to a custom-developed algorithm designed to operate on application binaries, obtained from SiliconHive C compiler. In this approach, to increase the emulation speed-up, author exploits translation of application binary code, compiled for a custom VLIW ASIP architecture, into code executable on a different configuration. This allows to prototype a whole set of ASIP solutions after one single FPGA implementation flow, mitigating the aforementioned overhead. In the toolset, the resulting evaluation platform serves as an underlying layer for a Design Space search algorithm. This method is able to provide latency numbers, or any other interesting performance metrics such as area, energy and power figures, that can fed SESAME simulator and start a design space exploration process with the aforementioned tools.

6.1 FPGA-based prototyping platform

The FPGA-based prototyping platform, as mentioned, is used to provide detailed characterization numbers when needed by the optimization process. The evaluation can be done at component-level on a single ASIP (during preliminary characterization) or on a complete system-level configuration. The inputs to the prototyping phase are:

- the partitioned application code (coded in plain C)
- a set of ASIP architectural specifications, describing the processor configurations to be evaluated, expressing number and kind of template building blocks and their connectivity, according to the reference architectural template previously described;
- a system-level specification, describing the system architecture in terms of number and kind of processing elements and defining their connectivity. When performing the preliminary calibration step, the system-level specification is a default single-ASIP system that includes a host processor in charge of uploading the binaries in the ASIP memories. When performing a system-level evaluation, the multi-ASIP specification is

automatically created by the *Design point description generation* tool shown in Figure 4.1.

Both the ASIP- and the system-level specifications (both indicated as *Design point configuration description* in Figure 4.1) are expressed in a proprietary industrial format. This enables to exploit the tools in the industrial flow ([41]) that is taken as reference within the project, aimed at the design and the programming of ASIP architectures compliant with the previously described general template. The tool suite includes HDL generators and a retargeting compiler, and envisions a typical ASIP design flow. A configuration description is passed to the RTL constructor, that analyzes it and provides as output the VHDL description of the whole architecture. This HDL code is used as input for the FPGA implementation phase, that can be performed with standard commercial tools. The target application code is compiled by means of an adequate compiler, retargeting itself according to the instruction set and the architectural features of the processor under prototyping. After compilation, the program can be executed on the ASIP implemented on FPGA. In order to enable fast on-FPGA evaluation of multiple design points, such flow has been extended within the project as shown in Figure 6.1.

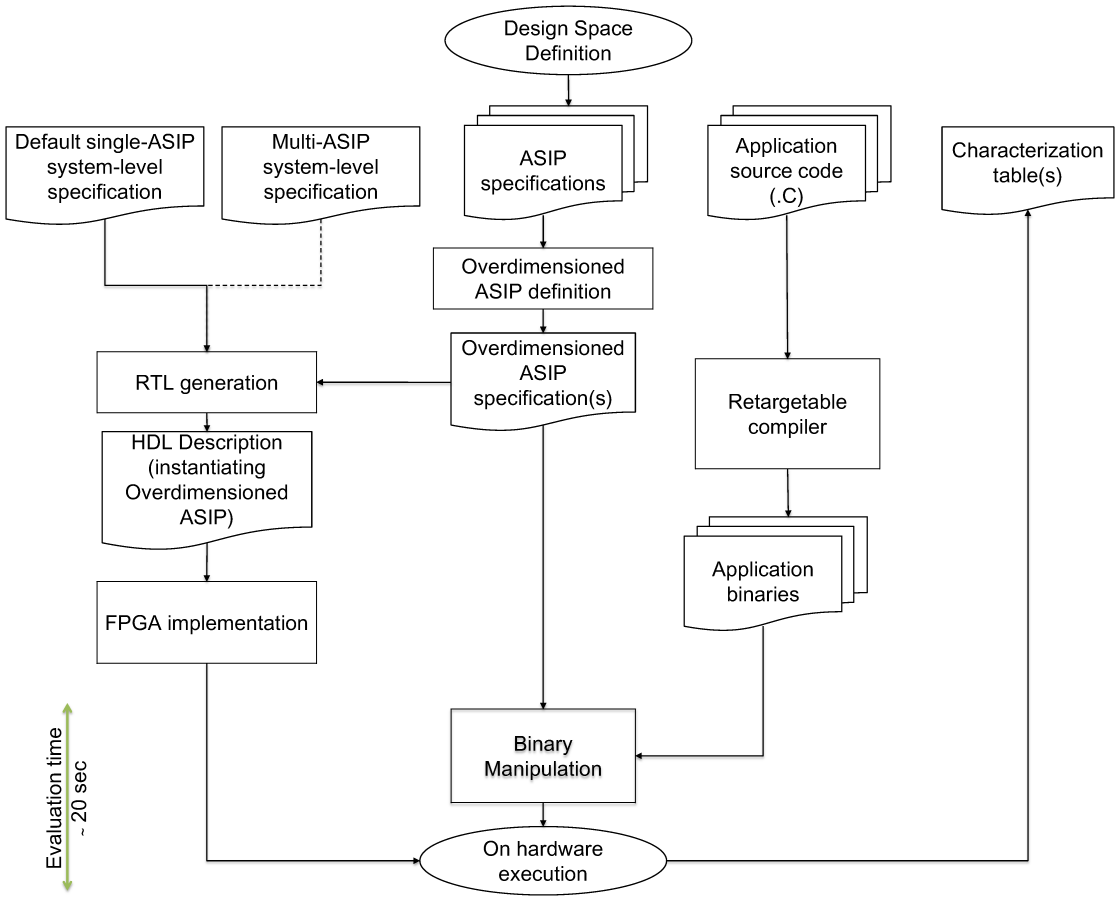


Figure 6.1: Prototyper block diagram

The prototyping speed-up technique developed within MADNESS focuses on the identification of what it's named a *worst case* configuration (WCC), i.e. a processor configuration

that is over-dimensioned with the hardware resources necessary to emulate all the configurations included in the predefined set of candidates. Once the WCC has been implemented on FPGA, to evaluate different design points, on top of it the binaries obtained compiling the target application for each candidate configuration are *adapted* by means of a custom-defined manipulation algorithm and then executed. The manipulation is aimed at activating only the needed subset of the WCC circuitry, to mime the prototyping of the considered design point after a stand-alone implementation and programming flow. During the execution, it's possible to obtain, by means of dedicated counters automatically instantiated inside the HDL code before synthesis, performance and switching activity metrics. To evaluate every candidate architecture only the meaningful counters inside the WCC are considered, assuring the obtained results to be perfectly equivalent to those obtainable from its "single-configuration" prototyping.

6.1.1 The WCC synthesis algorithm

In the extended flow, all the design points under test are provided to the flow at the beginning of the iterative process. The WCC is defined by updating it at each iteration according to the design point under analysis. At iteration N (i.e. parsing the $N-th$ candidate configuration under test c)

- The number of issue slots inside c is identified and compared with previous iterations. A maximum search is performed, then, if needed, the WCC is modified to instantiate $N_{IS}(WCC)$ issue slots, where

$$N_{IS}(WCC) = max\{N_{IS}(i)\} \text{ for } i = 1,...,N;$$

- For every issue slot x inside c, the size of the associated register file is identified and compared with previous iterations. A maximum search is performed, then, if needed, the register file related to the issue slot x inside the WCC is resized to have $RF\_s(x, WCC)$ locations, where

$$RF\_s(x, WCC) = max\{RF\_s(x, i)\} \text{ for } i = 1,...,N;$$

- For every issue slot x inside c, the set of FUs is identified and compared with previous iterations. The issue slot x inside the WCC is modified, if needed, to instantiate a set of FUs being the minimum superset of FUs used in previous configurations:

$$FU\_set(x, WCC) =$$

$$FU\_set(x, c) \cup FU\_set(x, i) \text{ for } i = 1,...,N;$$

6.1.2 The binary manipulation algorithm

For each candidate architecture, knowing the architectural parameters, the instruction bits can be partitioned in sub-ranges that identify specific control directives to the datapath, such as operation codes (selecting specific function units and specific operations inside issue slots), index values specifying the locations to be accessed in the register files, and

configuration patterns for the connectivity matrices controlling the propagation of the computing data through the datapath. The width and the position of each range are statically dependent on the architectural configuration that must execute the instruction. For each field, a disabling configuration is defined, able to determine a no-operation for the related datapath part. The algorithm presented hereafter assumes, according to the structure of the instruction word for the considered ASIP flow and for most VLIW architectures, the size and position of each instruction field to be univocally determined *a priori* once processor architecture is specified, thus allowing to correctly predict instruction structure. The general idea is then to manipulate each single instruction field of a candidate configuration, in order to fit it (modified in position, size and value) in the WCC instruction format. The entire set of these structures represent a full description of the instruction format for the candidate architecture: these values, automatically extracted in the analysis phase and paired with WCC description, are sufficient to implement the binary translation feature. Furthermore, if differences in hardware require modifications of the value associated to a particular architecture instruction field (e.g. an issue slot has a different set of function units in a candidate architecture than the corresponding one instantiated in WCC), offset values are computed by the parser and stored in an extension of cited data structure associated with that field, ready to be added to field values by the algorithm when needed. We now present the main steps involved in translating a binary application for a target configuration. First, each parsed candidate architectural description is analyzed by the tool, and compared to the WCC description, to identify: the position and the size of the field inside the candidate instruction word, the position and the size of the field in the WCC, and an "offset" indication to be considered during adaptation.

```
struct field_s field_A_n = {6, 4}; //{position, size}
struct field_s field_A_wcc = {8, 6};
struct offset_s offset_A_wcc = {0};
struct field_s field_B_n = {14, 3};
struct field_s field_B_wcc = {25, 6};
struct offset_s offset_B_wcc = {0};
struct field_s field_C_n = {31, 3};
struct field_s field_C_wcc = {44, 5};
struct offset_s offset_C_wcc = {1};
```

A strict one-to-one relationship is thus established between each processor slice (and the related instruction fields) in the candidate architecture and a corresponding processor slice miming it in the WCC. The information contained inside the "offset" structure indicates how the value in the related candidate instruction must be modified, taking into account hardware structures instantiated in the WCC but not involved in the prototyping of the considered design point.

For every instruction in the candidate binary, a WCC instruction word is then *populated* with corresponding values read from the candidate instruction word.

If value needs to be modified due to differences in hardware between WCC and candidate, the corresponding offset value is added to the value written in destination field. It may also happen that a disabling configuration (i.e. an opcode that disables a function unit, or a register file output port) is found: this value is eventually extended on its most significant bits with ones, to match differences in length, and written in the appropriate field of WCC

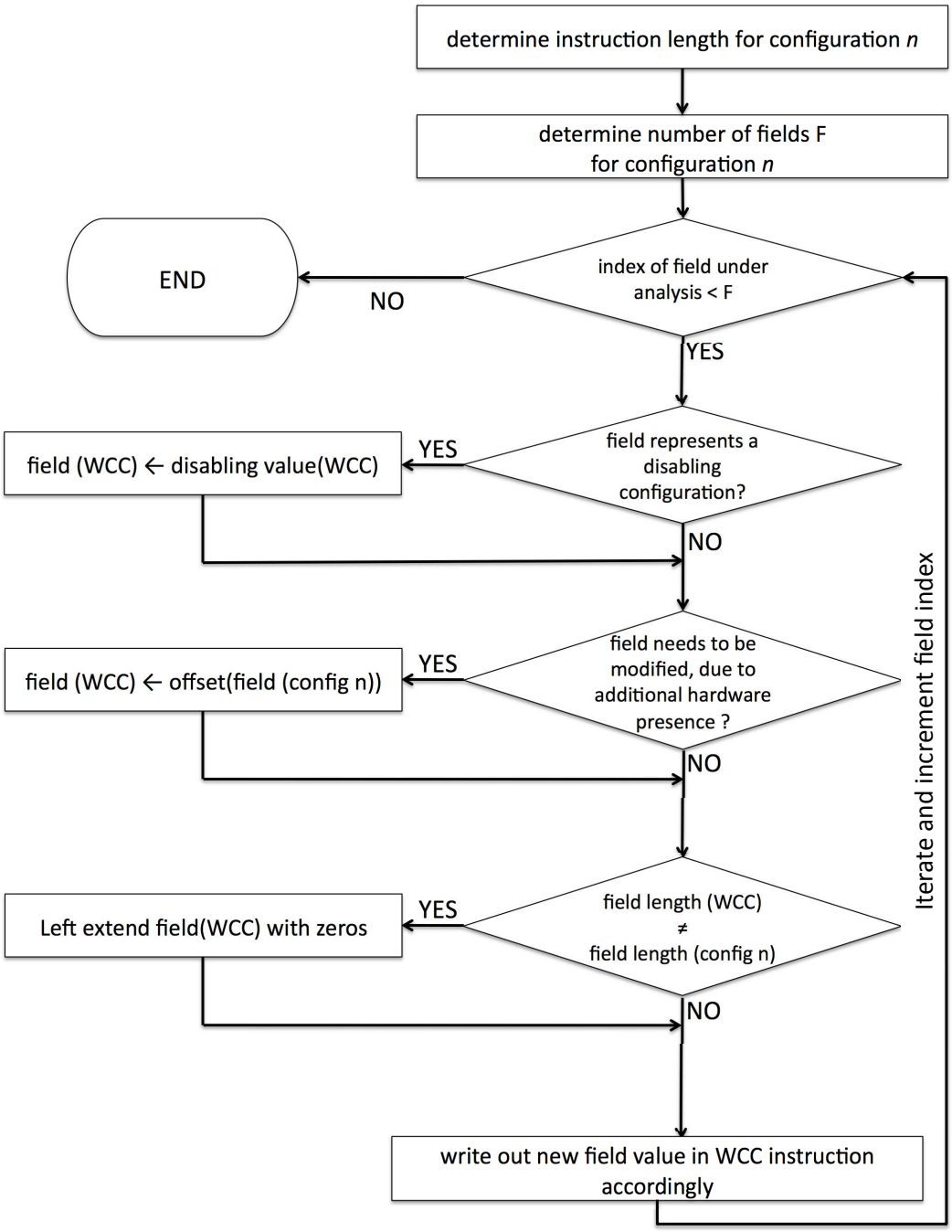


Figure 6.2: Flowchart for instruction manipulation algorithm.

instruction.
An example is depicted in Figure 6.3.

6.1.3 Software support for binary manipulation

Software support for binary translation is provided within the proposed framework, and is included in already presented tools. All the binary manipulations are done on a host machine before programming the FPGA and uploading application code on it, such that is convenient to perform multiple translations of the application binary and adapt it for different configurations. Obtained translated binaries can be loaded on FPGA onboard memory, and subsequently executed on the ASIP prototyping platform only by invoking a custom C function in the application flow, in charge of selecting the correct binary code for the desired emulated configuration and uploading it to ASIP program memory. At the end of each on-ASIP execution, metrics are automatically extracted from the platform, accessing memory-mapped counters, obviously excluding those related to hardware elements that are instantiated within the WCC but are not involved in the prototyping of a specific configuration under test.

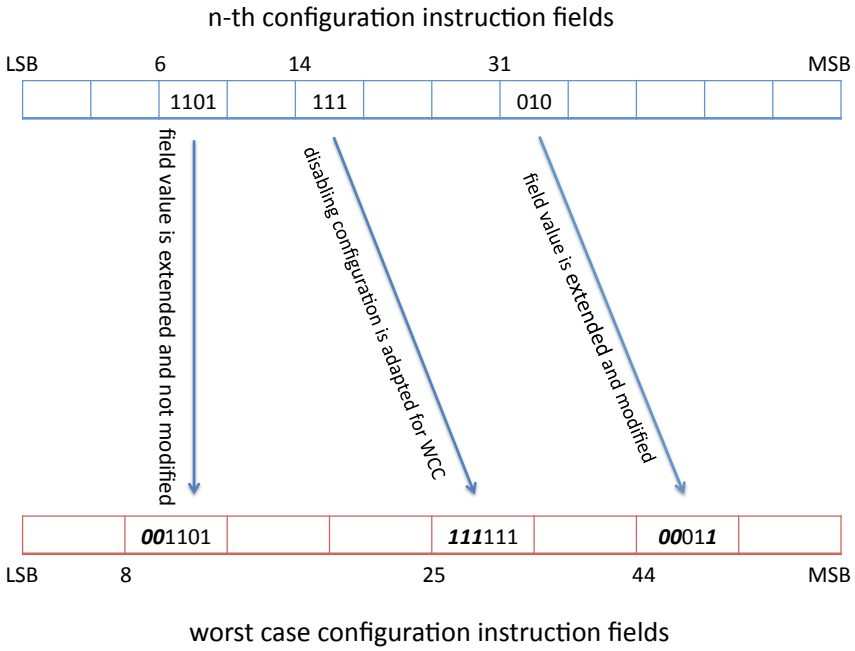


Figure 6.3: Example of manipulation of an instruction word. First field is left-extended to obtain the same length of corresponding field on the WCC; second field represents a disabling configuration adapted for WCC; third field is left-extended and modified, due to the presence of an offset value.

6.2 Interfacing the tools through co-simulation

In order to enable the calibration data to be comfortably accessed by the DSE environment, a dedicated support for extracting the emulation results from the FPGA was implemented, exploiting Xilinx SysGen toolbox for Matlab.

The toolbox enables to define shared memories that can be accessed either by the hardware modules implemented on the FPGA, or by a Simulink instance running on a host workstation. In this way, it is sufficient to connect the performance counters inside the processors and the other modules in the system to such memories, to have a user-friendly interface to the evaluation platform.

The HDL generator was enhanced in order to automatically set-up the needed connections and wirings to support the counter values fetching. More in detail, the HDL generator was slightly modified in order to set-up the needed connections, defining the needed wiring resources and instantiating an "address generator" to dispatch the different counter values to different shared-memory locations.

Simulink objects and Matlab funtions can, at the other end, read from the shared memory activity values and counts to make them available for plotting or in the workspace. Being the DSE environment also implemented using Matlab, this results in an efficient method implementing the exchange of data between the tools (i.e. to implement the transfer of the previously described *calibration table*). In Figure 6.4 a screenshot of the framework user interface is shown.

6.3 Use Case

In this section, a typical use case of the previously described integrated toolset, where the FPGA-based prototyping platform is used for preliminary calibration of the simulation model, is presented. The target application is a motion-JPEG (MJPEG in the following) video compression kernel. The use case is a DSE process that optimizes the mapping of the application parallel tasks on a selected set of ASIP configurations. During the calibration, an exploration of the component-level design space exposed by the ASIP template was done, evaluating 18 different ASIP configurations under the workload related with the execution of the parallel tasks inside the MJPEG task-graph. The explored design points were identified considering different permutations of the following parameter values:

- $N_{IS}(c)$: 2 or 3 or 4 or 5;
- $FU\_set(x,c)$: two different sets of FUs were considered, basically differing only for the inclusion of a *multiply-and-accumulate* FU (MAC), that is the most power/area hungry inside the library;
- $RF\_size(x,c)$: 8 or 32 entries, each 32-bits wide;

A filtering kernel was compiled for every candidate configuration and the resulting binaries were executed on the WCC prototype, after being adequately manipulated. During the preliminary calibration phase, as previously explained, a system with a host processor and one over-dimensioned ASIP core was designed. The host processor is in charge of reading the adapted binary from its local memory and upload it to ASIP program memory. After the binary file has been uploaded, host processor triggers the ASIP core for its start and wait until

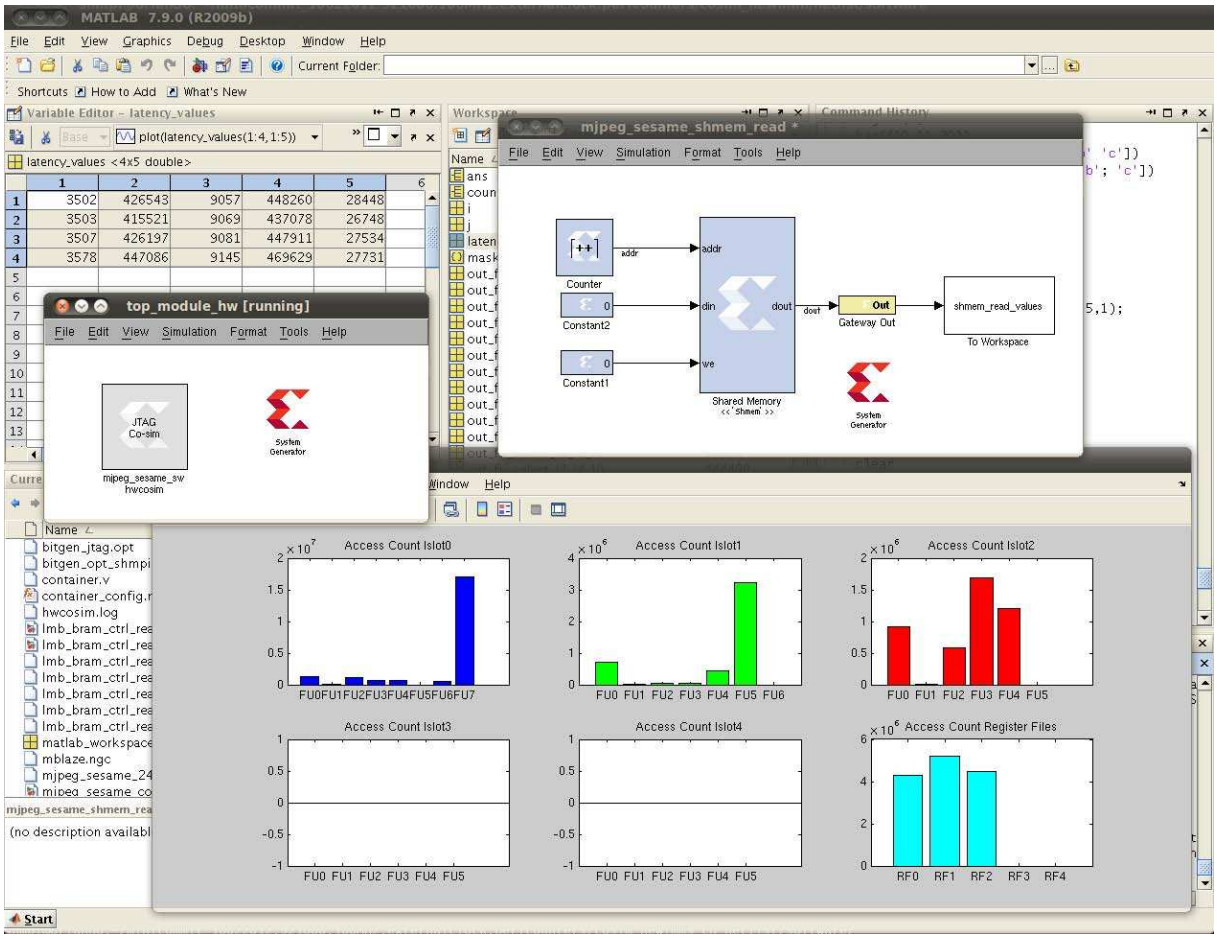


Figure 6.4: Screenshot of the Matlab GUI after a prototyping of a 4-tasks application kernel executed on 4 ASIP configurations. On the top-left (in the workspace) the *calibration table* of latency numbers to be passed in input to the simulation tool. On the right the Simulink model used to access the shared memories implementing the interface with the FPGA prototype. At the bottom detailed emulation data plotted as histograms. Below the latency values, the FPGA seen as a black box by the Matlab/Simulink environment.

the end of its execution, to fetch from ASIP local memory the results of the execution and eventually to check them for the presence of any errors.
The adopted hardware FPGA-based platform features a Xilinx Virtex5 XC5VLX330 device, counting over 2M equivalent gates.

The synthesis/implementation flow, performed on an Intel Quad-Core machine with commercial tools, required less than half an hour to complete. Binary translation was also performed on the same machine, but the related overhead in terms of emulation time is negligible (less than a second). According to this numbers, the presented approach allows a time saving that increases with the number of candidate topologies under prototyping, easily outperforming software-based simulation. The results of the preliminary component-level DSE are plotted in Figure 6.5. All the presented data are obtained after traversing only one synthesis/implementation flow. Area numbers are evaluated according to the ASIP configuration features and to area models provided by the industrial partners in the MADNESS project. Similar models are also available for energy, but the possibility of evaluating power

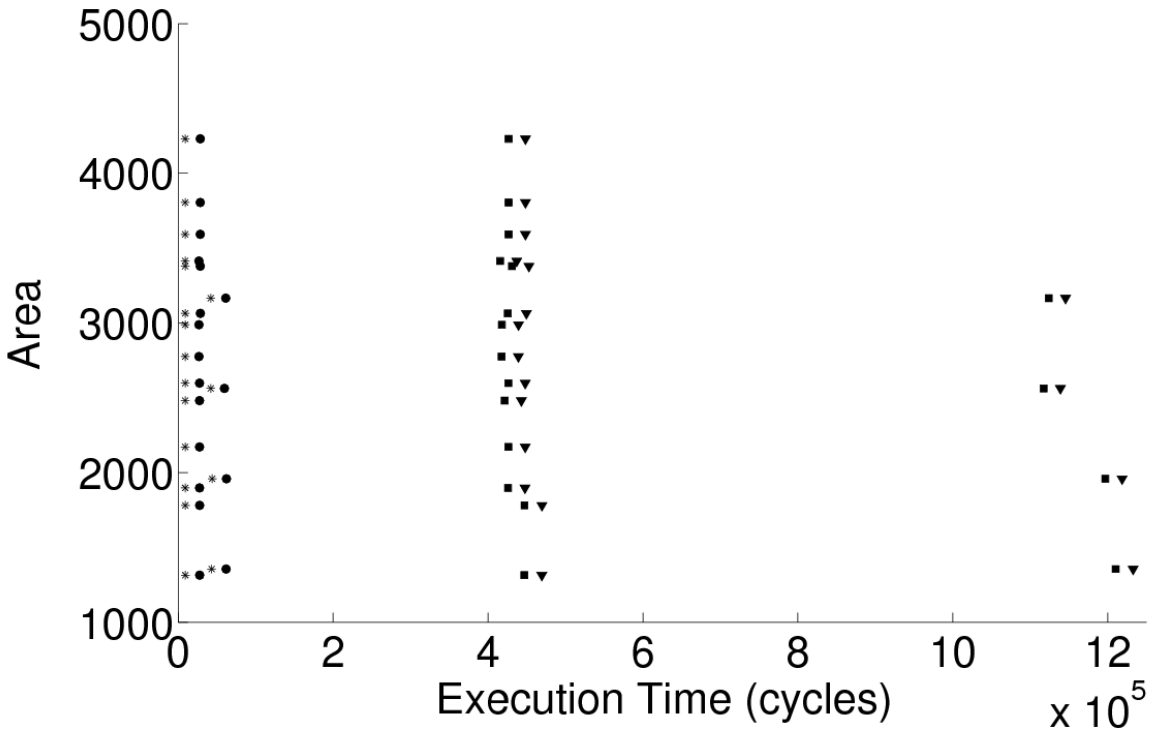


Figure 6.5: Pareto plot representing the latency values included in the *calibration table* annotated with the area value corresponding to the related ASIP configurations. Different symbols are used to represent values associated to different tasks in the MJPEG kernel

consumption, even if enabled at both simulation- and prototyping-level, is not discussed in this work. As may be noticed, all the tasks show to have similar behavior with respect to the fitting to the different candidate ASIP architectures. Design points that, for all the tasks, experience an execution time much longer than the others (right end of the graph) are those that do not feature any MAC, that, evidently, is intensively exploited for the kind of workload in the MJPEG kernel. Besides estimating computation latency for the tasks in the target application, the prototyping phase can be used to identify computation bottlenecks and congestion hot-spots inside the architecture. As an example, we show in Figure 6.6 a graph reporting number of accesses to every function unit and register file in a candidate ASIP configuration, during the execution of the MJPEG kernel.
As may be noticed, in the presented example, the WCC is used to evaluate a design point featuring only 3 issue slots, thus the activity counters related to issue slots 3 and 4 are never stimulated and must not been considered when evaluating the design point.

After the calibration step, the system-level DSE process can be initiated. The DSE engine can start evaluating different design points using the simulation model. The simulation model is able to tune itself by reading, directly from the Matlab workspace, the data inside the previously mentioned *characterization table*. As an example of the achievable results, we show in Figure 6.7 the Pareto graph obtained after an exploration process that involved an iterative evaluation of 100 generations with each population composed of 50 solutions each. This implies 500 evaluations performed by the toolset. The whole DSE experiment, after calibration, required 35 minutes on the previously mentioned Intel Quad-Core workstation.

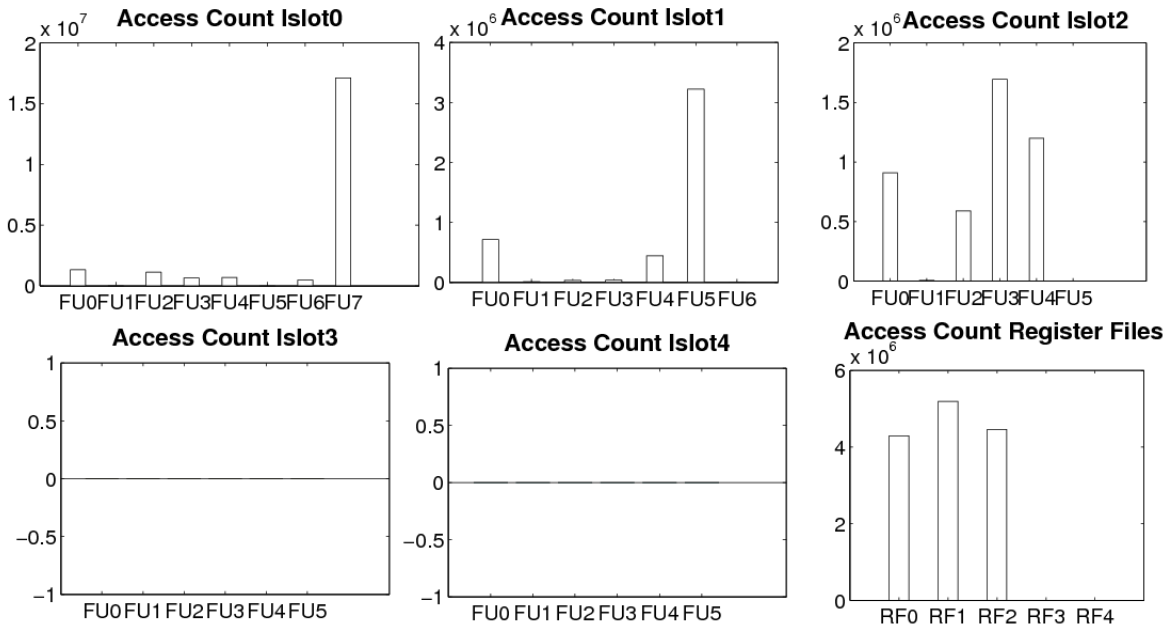


Figure 6.6: Detailed calibration results at functional block level

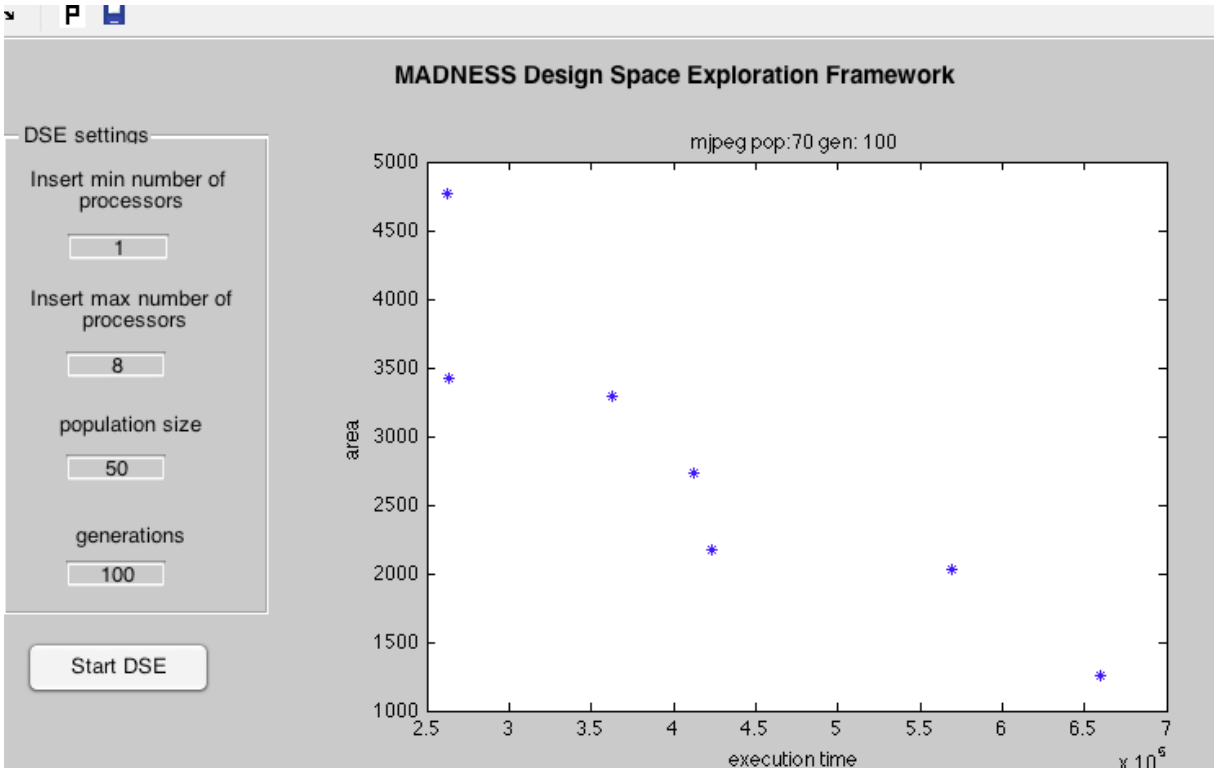


Figure 6.7: GUI of the DSE framework, plotting the results obtained for the MJPEG use case

As may be noticed in the Pareto graph, after the DSE process, a set of design points has been identified, showing different performance (execution time) vs cost (area) trade off. The fastest and more area-hungry Pareto point (top-left of the graph) features one host processor (executing the *VideoIn* task) and three ASIP processors. The three selected configurations are different, featuring respectively 4 ISs (3 equipped with MAC, executing the *DCT* task), 4 ISs (2 equipped with MAC, executing both *Vle* and *Q*) and 3 ISs (1 with MAC executing *Vout*). However a solution providing the same execution time but requiring less hardware is also identified, mapping *DCT* and *Vout* on the same processors, but using two different smaller processors for *Vle* and *Q* (respectively featuring 1 IS without MAC and 2 ISs with one MAC). The slowest and cheapest solution (the Pareto point at the bottom-right corner of the plot) is a single-ASIP featuring only one instance of the cheapest processor (1 *processing* issue slot without MAC), that is in charge of executing all the tasks in the target application kernel. Besides providing the plot in the figure, the process collects the simulation results for all the evaluated design points and the related HDL system-level description, in order to enable the protoyping of a multi-ASIP design point on the FPGA platform.

6.4 Extending binary manipulation techniques for fault-tolerance support

6.4.1 Overview on fault-tolerance techniques

With the growing number of transistor on a single chip, technology scaling and high wiring density, fault probabilities have grown alongside with performances, limiting the overall ability of the processors to be available in an extended life period. Indeed, BER (bit error rate) value in a modern processor is ten times bigger than the one of a memory chip, mainly due to the higher complexity of the first. Considering all of this, fault-tolerance techniques must be adopted in order to guarantee availability for mission critical applications: as an example, Intel Itanium architecture provides important mechanisms such as *Machine Check Abort* along with standard *Error Correcting Code* to optimize overall dependability of the chip.

A classic approach to fault-tolerance is represented by triple modular redundancy (TMR), consisting in three different phases of intervention: detection, where the presence of a fault is determined and, if possible, its nature is evaluated (temporary vs. fixed, minor vs. critical); location, during which fault is located in a precise hardware module; recovery, tentative approach to restore normal execution. Recovery is generally obtained thanks to redundant hardware resources, that can be *spare units* (not used during normal execution flow), or they can also be parts of the core that serves both during normal flow and recovery state.

To demonstrate the feasibility of supporting fault-tolerance in this platform, the work presented here is based on an off-line instruction remapping method, requiring easier control structures on the hardware and avoiding to interfere and slow down application execution with on-line checking/remapping routines. However, should this approach prove as being interesting for a prospective implementation, interesting research can be conducted on the topic of extending support to on-line remapping techniques, developing appropriate hardware modules to detect faults and leveraging the instruction manipulation techniques described here for task remapping.

6.4.2 Pearl and Pearl_FT processors

For the sake of this work, two simple VLIW architectures are considered: the first is a Pearl processor, a two-ways processor developed by SiliconHive, with the first issue slot equipped with basic control/logic function units, and a second slot sporting arithmetic and DSP-oriented function units. Starting from this template, Pearl_FT was developed, by means of replicating the second issue slot and instantiate it as a spare unit, where instructions could have been remapped by developed algorithm. A general assumption has been made, requiring that each detected fault can occur in the second issue slot, thus allowing to easily remap the operation on the spare issue slot. This also doesn't require recompilation of application code for both the processors, since applications compiled for Pearl processor will be executable transparently on the Pearl_FT variant.

6.4.3 Remapping algorithm

In the workflow described here, author assumes that, at some point of the application execution, a fault has been detected and program execution has to be aborted. A general assumption on fault localization was made, consisting in the possibility of finding faults only on the second issue slot of Pearl processor.

Exploiting the binary manipulation already described in 6.1.2, the toolchain is instructed to remap each operation from the second issue slot, where it was scheduled originally, onto the spare issue slot added in the Pearl_FT.

Performing this operation is more or less the same process as described in the already referenced section, but another important aspect on the interconnect wirings has to be kept into account. Since instruction word not only contains opcodes and operands for the function units, but also the signals that drive multiplexors and other interconnection hardware elements, relevant bits of the instruction have to be modified, so to move the input/output paths to/from the spare issue slot onto the correct register file connected to it.

Once these operations have been performed, it's possible to seamlessy execute an application compiled for the Pearl processor onto the Pearl_FT cell, without any access to the second issue slot hardware resources that were marked as faulty. Again, it's useful to point out that this can be obtained without any recompilation of the application, but simply leveraging the remapping and instruction manipulation algorithms.

Chapter 7

SESAME: high-level simulation for heterogeneous MPSoCs

7.1 General description

As described in Section 4.3, SESAME is a framework for high-level simulation. Inside the MADNESS FP7 Project, usage of SESAME has been envisioned as a center point for DSE of heterogeneous multi-processor systems. Its three-layered structure can be described as composed by:

- the application model, that describes the functional behavior for an application;
- the architecture model, that defines architecture resources and their performance characteristics;
- the mapping model, specifying which tasks of the application are to be mapped on which part of the architecture.

SESAME framework can be useful both at system-level and component-level DSE: for the first part, it tries to help designers in choosing how many processors to instantiate for a given application, how to map tasks onto processors, how should processor communicate among each other. Furthermore, for what concerns component-level DSE, the problem lies in how to configure the processing elements, or, in the case of multiple available ASIPs configurations, which ones to choose among them.

In this work, particular focus is put on the architectural model, since it was extended to support other forms of interconnect (see following sections).

7.1.1 Kahn Process Network paradigm

Application considered for this simulation framework all follow the KPN paradigm ([22]): in such an approach, parallel processes communicate among them through FIFO channels at their ends. In Kahn paradigm, reading from a FIFO is a blocking operation, while write operation is not: KPN model is very suitable for dataflow applications (like multimedia processing), thus is chosen by SESAME as the ideal paradigm to describe applications. Furthermore,

no matter what the process scheduling can be, they provide determinism in terms that each input application will always lead to the same output.

Workload of each application, transformed into a KPN-compliant model by means of manual intervention or through automated frameworks ([42]), is annotated by marking occurrencies describing computational and communication events of the application: this leads to the generation of traces driving the architectural model of SESAME simulator.

7.1.2 Design point description generation

In order to obtain the functionality already described in Section 4.3, a custom C tool application was developed by the author. Once the DSE algorithm produces as output a set of design points to be evaluated, both the FPGA prototyper and SESAME simulator needs adequate inputs to simulate those architectures and obtain relevant figures.

The developed tool is in charge of performing exactly this operation: starting from an abstract specification, obtained from DSE engine output, the tool produces adequate input files for FPGA prototype builder and for SESAME simulator.

For what concerns FPGA, a topology input file is created, representing an equivalent mapping of the design point, with automatic link instantiation between desired cores and route generation; on the other hand, a YML description of the architecture is produced for SESAME framework, along with routing functions that are in charge of simulating the flits exchange that occur on a NoC interconnect.

These operations are performed automatically and seamlessly among the different applications involved: also, files are placed in the correct directories by the tool itself, enabling an automated workflow based on scripted operations.

7.2 Extending SESAME to support Network-On-Chip interconnects

The aim of the work here described is the provisioning of an alternative approach for simulations of heterogeneous systems containing a NoC interconnect. Before this extension, an RTL simulation was needed to evaluate performance of a given architecture, coupled with a particular network topology and task mapping infrastructure. RTL simulations are time-hungry and overdetailed considering that one could just need latency numbers for their system being benchmarked. So, the solution was seen in an enrichment of SESAME framework, already used for complex MPSoCs high-level simulations, in order to support our Network-On-Chip interconnect. As already discussed before, SESAME is mainly composed by three different layers: application, mapping and architecure. With respect to the extension of the framework to support NoC, the architectural layer was enhanced with custom-developed building blocks, in order to model the missing Network On Chip components. Furthermore, the critical part of the work was represented by the calibration of NoC blocks latencies for communication/computational patterns, so that they could reflect real hardware behavior and provide consistent performance metrics (see Section 7.2.4).

7.2.1 NoC interconnect architecture

The interconnect structure is based on SHMPI framework ([33]), based on successive customizations of xpipescompiler ([20]) NoC architecture, that can be roughly described as composed by these basic elements:

- network interface;
- switch;
- link.

Network Interface

The NI connects cores (processing units) to the Network. It's in charge of building the header of the packet thanks to the information stored inside an internal look-up table. The packet is then divided in multiple chunks, called *flits*, and injected into the network.

Switch architecture

Switches are designed to interconnect and exchange packets between different elements of the network. They are designed with forward flow control, meaning that a flit is only sent to the next switch whether it has available space to store it. Switch is equipped with a set of registers that act as buffer to store incoming flits, given the fact that each flit has to be acknowledge upon reception by the receiver.

Link element

Links can have different lengths inside the interconnect, thus different paths may be instantiated inside a network: this suggested to design links as a collection of basic segments, each one requiring a single clock cycle to be traversed.

7.2.2 Topology file example

A completely custom interconnect can be described, in terms of number of cores, number of switches, links configuration and network topology. Description is done through a text file specification (example following).

```
topology(nocxplorer_custom_top, other);
core(core_0, switch_0, 1, 6, userdefined, initiator);
core(core_1, switch_0, 1, 6, userdefined, initiator);
core(core_2, switch_1, 1, 6, userdefined, initiator);
core(core_3, switch_1, 1, 6, userdefined, initiator);
core(pm_4, switch_0, 1, 6, double, target:0x10, high:0x1000ffff);
core(pm_5, switch_0, 1, 6, double, target:0x12, high:0x1200ffff);
core(pm_6, switch_1, 1, 6, double, target:0x14, high:0x1400ffff);
core(pm_7, switch_1, 1, 6, double, target:0x16, high:0x1600ffff);
switch(switch_0, 5, 5, 6, 0, 0);
switch(switch_1, 5, 5, 6, 1, 0);
```

```
link(link0, switch_0, switch_1);
link(link1, switch_1, switch_0);
```

The description file is parsed by a compiling utility that produces, as output, complete HDL description of each NoC element, including parametric switches and NIs, along with a convenient Xilinx ISE project file that allows user to synthesize and implement on FPGA the created design in a matter of minutes.

7.2.3 SESAME NoC blocks

Custom blocks can be easily added to SESAME by describing them in a particular format, called Pearl (a C-like descriptive language). To enable the usage of SESAME for this purpose, NIs, switches, links and virtual memory channels were created, aiming at a realistic description of network hardware dynamics. An extract from a file describing a NoC element (switch input port) is reported below:

```
#include "debug.h"

class switch_port_in

id        : integer
nswitches  : integer
switch_t = [nswitches] switch_port_out
switches   : switch_t

out_port   :integer
mask         : integer = -1
shift        : integer = -1

read_size     : integer = 0
read_time     : integer = 0
write_size    : integer = 0
write_time    : integer = 0
send_time     : integer = 0
send_count    : integer = 0
reply_time    : integer = 0
reply_count   : integer = 0

read:(vid:integer, size:integer,route:integer) {
    t : integer = timer()

    //  blockt(size * latency);
    blockt(SWITCH_LAT);
    out_port=port_select(route, nswitches);

    VP(VNOC,("%s: sending read over noc, route: %d\n", whoami(), route));
    VP(VNOC,("%s: using port: %d\n", whoami(), out_port));
```

```
    route = update_route(route, nswitches);
    VP(VNOC,("%s: sending read over noc, route: %d\n", whoami(), route));
    switches[out_port] !! read(vid,size,route);

    read_time += timer() - t;
    read_size += size;
}

write:(vid:integer, size:integer,route:integer) {
    t : integer = timer()

    //  blockt(size * latency);
    blockt(SWITCH_LAT);

    out_port=port_select(route, nswitches);

    route = update_route(route, nswitches);

    switches[out_port] !! write(vid,size,route);

    write_time += timer() - t;
    write_size += size;
}

// parameters not right. need to pass source
send_request:(vchan:vchannel, src:integer, route:integer) {
    t : integer = timer()

    //  blockt(size * latency);
    blockt(SWITCH_LAT);

    out_port = port_select(route, nswitches);

    VP(VSEND,("%s: sending send_request over noc, route: %d\n", whoami(), route));
    VP(VSEND,("%s: using port: %d\n", whoami(), out_port));

    route = update_route(route, nswitches);

    VP(VSEND,("%s: send_request updated route: %d\n", whoami(), route));

    switches[out_port] !! send_request(vchan, src, route);

blockt(REQ_PACKET_LAT);

    send_time += timer() - t;
    send_count += 1;
}
```

The files containing the description are compiled by SESAME internal tools, producing a simulator model. To proceed, user must provide an application to be mapped onto the architectural model (a simple producer-consumer benchmark and an MJPEG implementation were used), and a consistent mapping to specify which task should by executed by each of the processors instantiated in the system.

7.2.4 Calibrating the NoC model

Described NoC model for SESAME simulator it's not enough to prove it as an accurate way of simulating complex applications with a high-level approach, rather than benchmarking the platform through long and complex RTL simulations. As described earlier, initial calibration was needed, so to obtain, for internal network operation latencies, a consistent value to be inserted in the high-level model. Calibration involved two different types of operations:

- read-write operations, representing communication among NoC elements (e.g. flits exchange);
- execute operations, representing computations performed (e.g. FIFO channels instantiation, full/empty check).

To perform calibration, a simple producer-consumer application was chosen (see Figure 7.1): two different processors communicate through FIFOs, the first producing tokens while the latter consumes them. The protocol is engineered so that consumer sends a request each time its FIFO channel is empty, and the producer stops creating tokens once its FIFO is full.

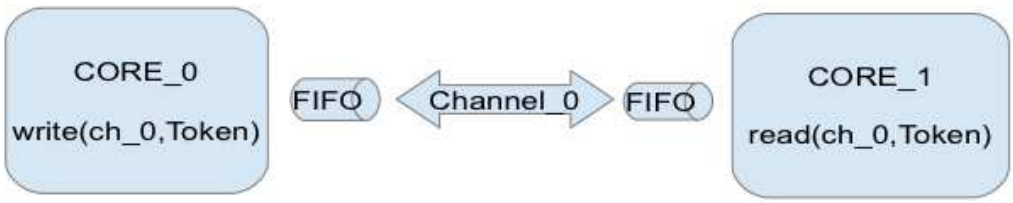


Figure 7.1: Producer-Consumer application used for the calibration of the model

More in detail, a list of the latency elements identified during NoC blocks design is hereby reported:

- delay in enabling/disabling interrupt - cycle of instructions needed to perform interrupt toggling;
- delay in token receiving - calculated from last token arrival to first token copied in FIFO;
- delay in transfer request - computation time needed to produce a request from CPU;
- delay in checking if FIFO is empty;

- delay in checking if FIFO is full;
- delay in copying a byte from memory to FIFO - only for producer;
- delay in copying a byte from FIFO to memory - only for consumer;
- delay in memory access for producer - independent of transfer size;
- delay in memory access for consumer - independent of transfer size.

To perform this tuning, the approach can be described in the following sequence of steps (algorithm 2):

ALGORITHM 2: Calibration of NoC building blocks

Input: a set of calibration applications; a set of network topologies
for *topology, application* **do**
    port the application on SESAME framework using the described topology;
    port the application on the RTL hardware model onto an equivalent topology;
    execute RTL simulation of the application;
    for *latency element* **do**
        extract the latency value from RTL simulation through counting of clock ticks;
        set the corresponding SESAME latency element to this value;
    end
end

RTL simulation were performed with ModelSim on a Intel Quad-Core workstation. Calibration was performed by means of accurate clock ticks evaluation for each operation, obtained from waveform simulation coupled with the help of disassembled code, in order to know at each instant the operation performed by the CPU. Once calibration phase has been completed, focus can be moved on validating the correctness of proposed method, through a typical use case of this setup, as described in the next section.

7.2.5 Use case

To demonstrate the validity of this approach, the calibrated model has been used to obtain performances of an MJPEG implementation, mapped onto different processors and interconnect network topologies, and comparing the results of the high-level simulation with those coming from RTL simulation of the very same application.

More in detail, the MJPEG application has been partitioned into different tasks, partitioned onto four MicroBlaze processors instantiated through the SHMPI system builder:

- video_in task, managing input sample acquisition - mapped on CORE0;
- DCT, discrete cosine transform - mapped on CORE1;
- Q, quantization - mapped on CORE2;
- VLE, variable-length encoding - mapped on CORE3;
- video_out task, transferring output image to external memory - mapped on CORE3.

To stimulate different conditions of work, three interconnect topologies were developed, each one with a different number of switches (one, two, three): thanks to this, different network congestion situations were introduced, allowing to extensively evaluate the behavior of the proposed high-level simulation model. As an example, Figure 7.2 describes the three-switches topology used to execute MJPEG application.

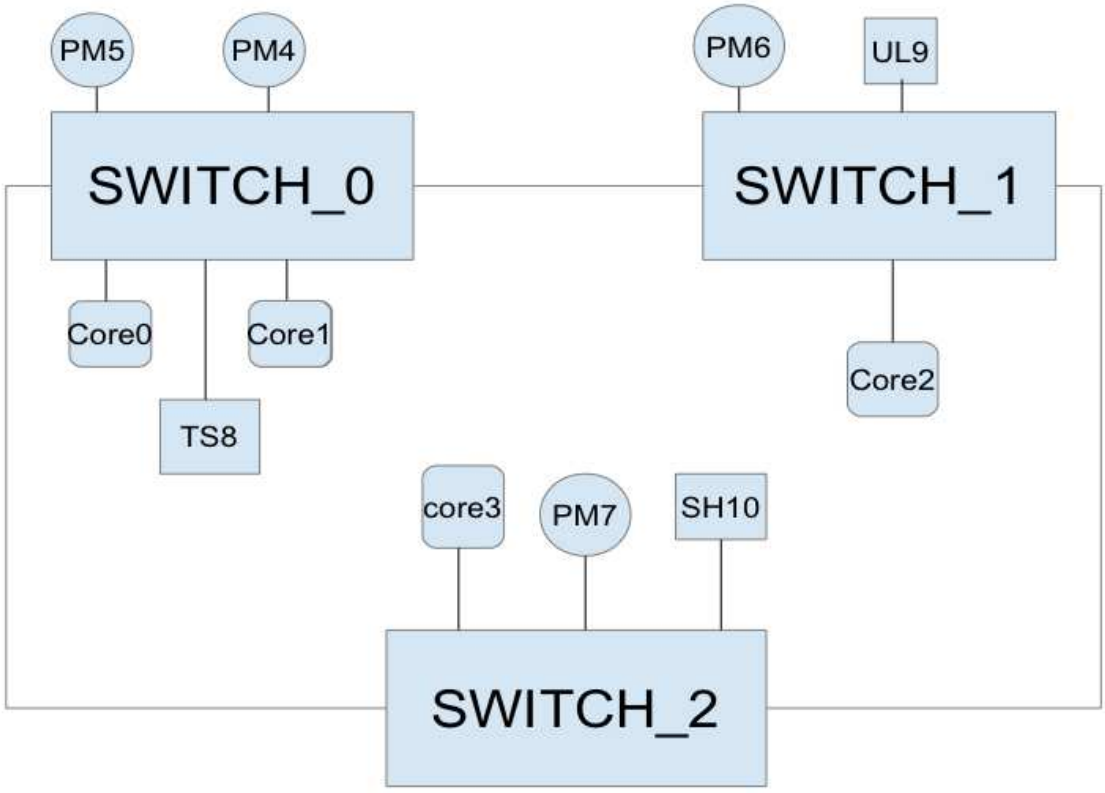


Figure 7.2: MJPEG application mapped on 4 cores, 3 switches network topology

In Figure 7.3 is shown the graph containing minimum, maximum and average unaccuracy of the high-level simulation model, for the MJPEG application (single frame run), mapped onto the different architectures. Results can be considered very promising, since the error rate is not heavily affected by the change of topology interconnect, and its average is under 2%, confirming that obtained latency numbers are quite dependable even compared to the exact, clock-precise latency values obtained from RTL simulation.

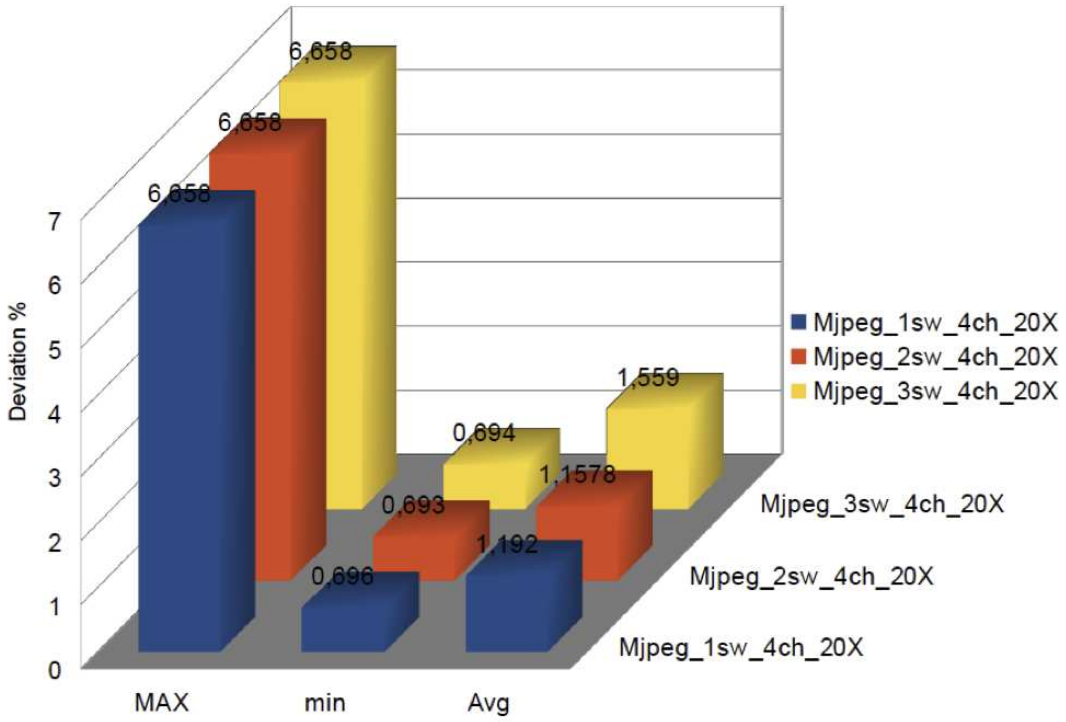


Figure 7.3: Min, max and avg error for latency values

Chapter 8

Conclusions

In this work, a novel approach to the application-driven design, configuration and programming of multi-ASIP systems, based on a combination of trace-based high-level simulation performed by SESAME framework and FPGA-based emulation, is presented and evaluated. The main point of strength of the proposed approach relies on the complementarity of the high-level simulation and the FPGA evaluation methods. While simulation, once duly calibrated, is capable of exploring vast design spaces in reasonable times, FPGAs, if the overhead related with on-hardware implementation is adequately reduced, are a convenient platform for rapidly evaluating sets of design points with component-level detail and complete accuracy. To reduce the amount of time needed for standard FPGA exploration flows, it has been shown how to exploit both an automatic hardware module instantiation or manipulation of the application binaries to obtain different VLIW ASIP architectures being emulated on-hardware by *mapping* them via software on a larger *worst case* configuration.
Furthermore, the SESAME simulation framework has been enriched with support for Network-on-Chip interconnects, broadening the range of systems that can be evaluated through this methodology. In addition to the classic functional metrics (e.g. execution time, access rate, IPC, resource congestion), the presented framework has been proven to be able to produce physical metrics (e.g. area obstruction, leakage static power, dynamic power and energy consumption) for a prospective implementation of the ASIP system. Each approach has been evaluated with use cases that helps to validate the usefulness of the entire framework as an effective support to quantitative design space exploration or simply as an environment for rapid prototyping of complex ASIP-based platforms. Future developments of this work will go towards the provision, by extending and improving the fundamental mechanisms presented in this article, of support for adaptiveness and on-line fault tolerance techniques in ASIP single- and multi-core platforms. Also, significant improvements can be obtained by developing more accurate models for the collection of interesting functional metrics, such as those presented before, allowing designers to be presented with a comprehensive and accurate benchmark over each configuration they intend to evaluate for a prospective ASIC implementation.

Bibliography

[1] Multicube. www.multicube.eu. [cited at p. 8]

[2] F. Angiolini, Jianjiang Ceng, R. Leupers, F. Ferrari, C. Ferri, and L. Benini. An integrated open framework for heterogeneous mpsoc design space exploration. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1 –6, march 2006. [cited at p. 8]

[3] Eduardo Argollo and et al. COTSon: Infrastructure for Full System Simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, 2009. [cited at p. 6]

[4] G. Ascia, V. Catania, M. Palesi, and D. Patti. A system-level framework for evaluating area/performance/power trade-offs of vliw-based embedded systems. In *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, volume 2, pages 940 – 943 Vol. 2, jan. 2005. [cited at p. 6]

[5] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35:59–67, February 2002. [cited at p. 6]

[6] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. Mparm: Exploring the multi-processor soc design space with systemc. *J. VLSI Signal Process. Syst.*, 41:169–182, September 2005. [cited at p. 6]

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39:1–7, August 2011. [cited at p. 6]

[8] Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler. Pisa: A platform and programming language independent interface for search algorithms. In *EMO*, pages 494–508, 2003. [cited at p. 8]

[9] Emanuele Cannella, Onur Derin, and Todor Stefanov. Middleware approaches for adaptivity of kahn process networks on networks-on-chip. In *DASIP'11: Proceedings of the Conference on Design and Architectures for Signal and Image Processing*, pages 1–8, Tampere, Finland, November 2-4 2011. [cited at p. 40]

[10] Joseph R. Cavallaro and Predrag Radosavljevic. Asip architecture for future wireless systems: Flexibility and customization, 2004. [cited at p. 5]

[11] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William H. Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators. In *MICRO*, pages 249–261, 2007. [cited at p. 7]

[12] Joseph E. Coffland and Andy D. Pimentel. A software framework for efficient system-level performance evaluation of embedded systems. In *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, pages 666–671, New York, NY, USA, 2003. ACM. [cited at p. 7, 38]

[13] P.G. Del Valle, D. Atienza, I. Magan, J.G. Flores, E.A. Perez, J.M. Mendias, L. Benini, and G. De Micheli. Architectural exploration of mpsoc designs based on an fpga emulation framework. 2006. [cited at p. 7, 8]

[14] Alessandro G. Di Nuovo, Maurizio Palesi, Davide Patti, Giuseppe Ascia, and Vincenzo Catania. Fuzzy decision making in embedded system design. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '06, pages 223–228, New York, NY, USA, 2006. ACM. [cited at p. 6]

[15] A. Falcon, P. Faraboschi, and D. Ortega. Combining simulation and virtualization through dynamic sampling. *Performance Analysis of Systems and Software, IEEE International Symposium on*, 2007. [cited at p. 6]

[16] N. Genko, D. Atienza, G. De Micheli, L. Benini, J.M. Mendias, R. Hermida, and F. Catthoor. A novel approach for network on chip emulation. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 2365–2368 Vol. 3, May 2005. [cited at p. 7]

[17] Tony Givargis, Frank Vahid, and Jörg Henkel. System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ICCAD '01, pages 25–30, Piscataway, NJ, USA, 2001. IEEE Press. [cited at p. 7]

[18] Matthias Gries. Methods for evaluating and covering the design space during early design development. *Integr. VLSI J.*, 38:131–183, December 2004. [cited at p. 8]

[19] Silicon Hive. Silicon hive sdk reference manual and tim developer's guide, 2010. [cited at p. 11]

[20] Antoine Jalabert, Srinivasan Murali, Luca Benini, and Giovanni De Micheli. xpipesCompiler: A Tool for Instantiating Application Specific Networks on Chip. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, volume 2, 2004. [cited at p. 59]

[21] Z.J. Jia, T. Bautista, A. Nunez, C. Guerra, and M. Hernandez. Design space exploration and performance analysis for the modular design of cvs in a heterogeneous mpsoc. In *Reconfigurable Computing and FPGAs, 2008. ReConFig '08. International Conference on*, pages 193 –198, dec. 2008. [cited at p. 8]

[22] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974. [cited at p. 57]

[23] K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(12):1523 –1543, dec 2000. [cited at p. 8]

[24] B. Kienhuis, E. Deprettere, K. Vissers, and P. Van Der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*, pages 338 –349, jul 1997. [cited at p. 8]

[25] Alex Krasnov, Andrew Schultz, John Wawrzynek, Greg Gibeling, and Pierre yves Droz. Ramp blue: a message-passing manycore system in fpgas. In *In 2007 International Conference on Field Programmable Logic and Applications, FPL 2007*, pages 27–29, 2007. [cited at p. 7]

[26] Yana E. Krasteva, Francisco Criado, Eduardo de la Torre, and Teresa Riesgo. A fast emulation-based NoC prototyping framework. In *RECONFIG '08: Proceedings of the 2008 International Conference on Reconfigurable Computing and FPGAs*, pages 211–216, Washington, DC, USA, 2008. IEEE Computer Society. [cited at p. 7]

[27] Choonseung Lee, Sungchan Kim, and Soonhoi Ha. A systematic design space exploration of mp-soc based on synchronous data flow specification. *J. Signal Process. Syst.*, 58:193–213, February 2010. [cited at p. 8]

[28] Slobodan Lukovic and Leandro Fiorin. An automated design flow for NoC-based MPSoCs on FPGA. In *RSP '08: Proceedings of the 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 58–64, Washington, DC, USA, 2008. IEEE Computer Society. [cited at p. 7, 8]

[29] Jan Madsen, Thomas K. Stidsen, Peter Kjaerulf, and Shankar Mahadevan. Multi-objective design space exploration of embedded system platforms. In *DIPES*, pages 185–194, 2006. [cited at p. 8]

[30] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, Feb 2002. [cited at p. 6]

[31] P. Meloni, S. Pomata, G. Tuveri, S. Secchi, L. Raffo, and M. Lindwer. Enabling fast asip design space exploration: an fpga-based runtime reconfigurable prototyper. *VLSI Design*, 2012(Article ID 580584), February 2012. [cited at p. 9]

[32] P. Meloni, S. Secchi, and L. Raffo. An FPGA-based framework for technology-aware prototyping of multicore embedded architectures. *Embedded Systems Letters, IEEE*, 2(1):5 –9, march 2010. [cited at p. 7]

[33] Paolo Meloni, Simone Secchi, and Luigi Raffo. An fpga-based framework for technology-aware prototyping of multicore embedded architectures. *Embedded Systems Letters, IEEE*, 2(1):5–9, 2010. [cited at p. 59]

[34] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. pages 1 –12, jan. 2010. [cited at p. 6]

[35] H. Nikolov and et al. Daedalus: Toward Composable Multimedia MP-SoC Design. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 574–579, New York, NY, USA, 2008. ACM. [cited at p. 8]

[36] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. A flexible framework for fast multi-objective design space exploration of embedded systems. In *PATMOS*, pages 249–258, 2003. [cited at p. 8]

[37] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. Multi-objective design space exploration of embedded systems. *J. Embedded Comput.*, 1:305–316, August 2005. [cited at p. 7]

[38] Andy D. Pimentel, Cagkan Erbas, and Simon Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Comput.*, 55(2), 2006. [cited at p. 22]

[39] Andy D. Pimentel and Roberta Piscitelli. Design space pruning through hybrid analysis in system-level design space exploration. In *Design, Automation and Test in Europe, 2012. DATE '12. Proceedings.* [cited at p. 22]

[40] V. Reyes, T. Bautista, G. Marrero, P.P. Carballo, and W. Kruijtzer. Casse: a system-level modeling and design-space exploration tool for multiprocessor systems-on-chip. In *Digital System Design, 2004. DSD 2004. Euromicro Symposium on*, pages 476 – 483, aug.-3 sept. 2004. [cited at p. 8]

[41] SiliconHive. Hivelogic configurable parallel processing platform, 2010. [cited at p. 46]

[42] T. Stefanov, E. Deprettere, and H. Nikolov. Multi-processor system design with espam. In *Hardware/Software Codesign and System Synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th International Conference*, pages 211–216, 2006. [cited at p. 58]

[43] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. RAMP gold: an FPGA-based architecture simulator for multiprocessors. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 463–468, New York, NY, USA, 2010. ACM. [cited at p. 7]

[44] L. Thiele, I. Bacivarov, W. Haid, and Kai Huang. Mapping applications to tiled multiprocessor embedded systems. In *Application of Concurrency to System Design, 2007. ACSD 2007. Seventh International Conference on*, pages 29 –40, july 2007. [cited at p. 8]

[45] K.D. Underwood and K.S. Hemmert. Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance. In *Field-Programmable Custom Computing Machines. FCCM 2004. 12th Annual IEEE Symp. on*, pages 219–228, 2004. [cited at p. 7]

[46] John Wawrzynek, Mark Oskin, Christoforos Kozyrakis, Derek Chiou, David A. Patterson, Shih lien Lu, James C. Hoe, and Krste Asanovic. Ramp: Research accelerator for multiple processors. 2006. [cited at p. 7, 8]

[47] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis, and K. Olukotun. A Practical FPGA-based Framework for Novel CMP Research. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 116–125, New York, NY, USA, 2007. ACM. [cited at p. 7]

[48] S. Wong, F. Anjam, and F. Nadeem. Dynamically reconfigurable register file for a softcore vliw processor. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 969 –972, march 2010. [cited at p. 7, 8]

List of Publications Related to the Thesis

Published papers

Journal papers

- MELONI, P., S. POMATA, G. TUVERI, S. SECCHI, L. RAFFO, and M. LINDWER, Enabling fast ASIP design space exploration: an FPGA-based runtime reconfigurable prototyper, VLSI Design, vol. 2012, no. Article ID 580584: Hindawi, February, 2012.

Conference papers

- JOZWIAK, L., M. LINDWER, R. CORVINO, P. MELONI, L. MICCONI, J. MADSEN, E. DIKEN, D. GANGADHARAN, R. JORDANS, S. POMATA, et al., ASAM: Automatic Architecture Synthesis and Application Mapping, Digital System Design (DSD), 2012 15th Euromicro Conference on, September, 2012.
- MELONI, P., S. POMATA, L. RAFFO, R. PISCITELLI, and A. PIMENTEL, Combining on-hardware prototyping and high-level simulation for DSE of multi-ASIP systems, Proc. 12th International Conference on Embedded Computer Systems (SAMOS-XII), 2012.
- POMATA, S., P. MELONI, G. TUVERI, L. RAFFO, and M. LINDWER, Exploiting binary translation for fast ASIP design space exploration on FPGAs, Design, Automation Test in Europe Conference Exhibition (DATE), 2012, pp. 566 -569, March, 2012.

Posters with published proceedings

- POMATA, S., TUVERI, G., MELONI, P., LINDWER, M., Fast ASIP Design Space Exploration on FPGAs through Binary Translation, ACACES 2011 HiPEAC Summer School, 2011.
- POMATA, S., TUVERI, G., SECCHI, S., MELONI, P., An FPGA-based runtime-reconfigurable prototyper for ASIP-based structured multi-core architectures, DEPCP - Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications (DATE 2011), 2011.