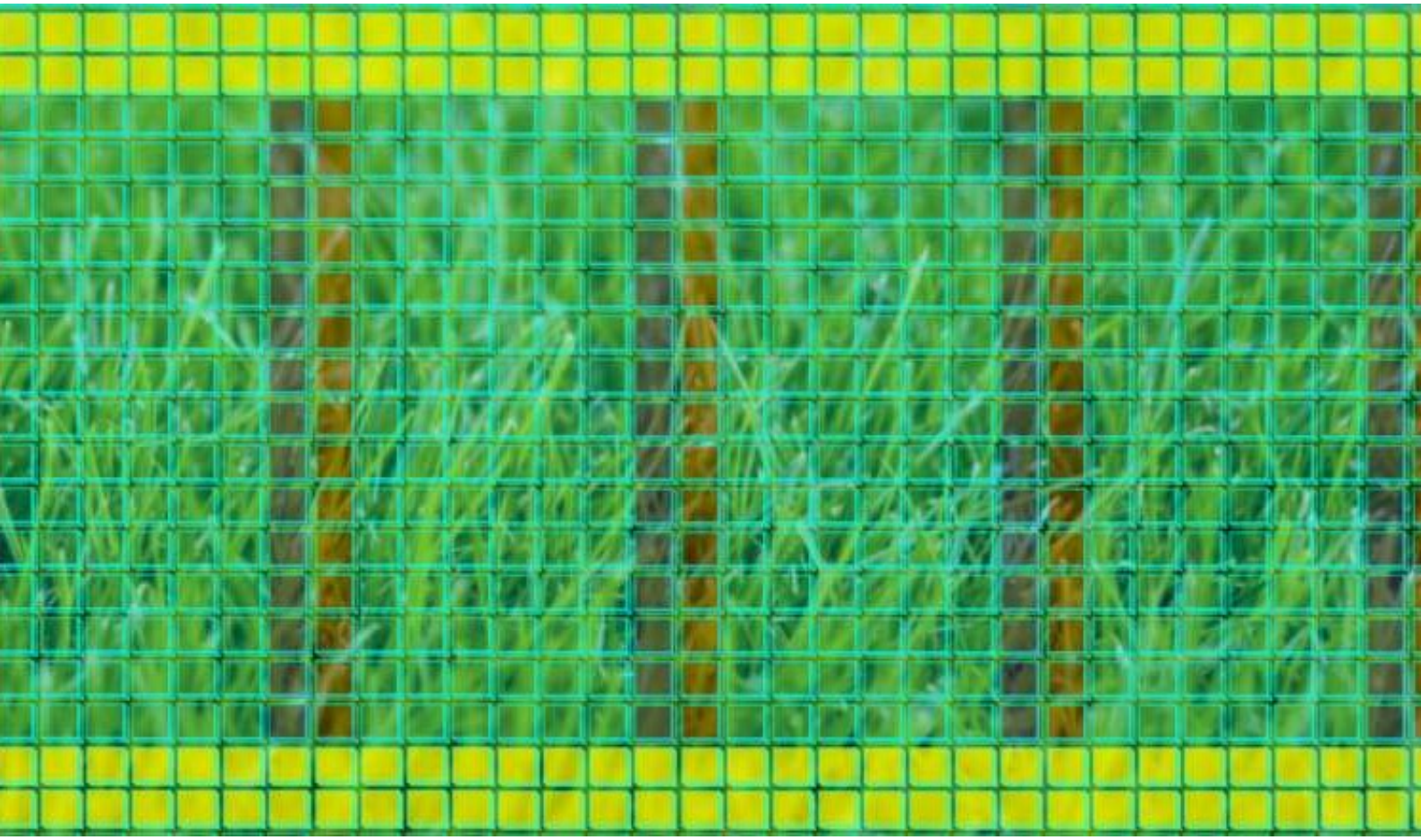
Universitat Autònoma de Barcelona

Scalable Parallel Architectures on Reconfigurable Platforms

David Castells Rufas



David Castells Rufas

Scalable Parallel Architectures on Reconfigurable Platforms

Ph.D. Thesis, David Castells Rufas
Department of Microelectronics and Electronic Systems
Universitat Autònoma de Barcelona
December 2015

UAB
Universitat Autònoma de Barcelona

Universitat Autònoma
de Barcelona

Scalable Parallel Architectures on Reconfigurable Platforms

Ph.D. Thesis Dissertation

Author:
David Castells i Rufas

Advisor:
Jordi Carrabina i Bordoll

Universitat Autònoma de Barcelona

December 2015

Bellaterra, Catalonia

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Dr. Jordi Carrabina i Bordoll

This work was carried out at the
Microelectronics and Electronic Systems Department of the
Universitat Autònoma de Barcelona

In fulfillment of the requirements for the degree of
Doctorat en Informàtica

Abstract

The latest advances on reconfigurable systems resulted in FPGA devices with enough resources to implement multiprocessing systems with more than 100 soft-cores on a single FPGA. In this thesis I present how to implement solutions based on such systems and two scenarios where those kind of systems would be valuable: hard real-time systems and energy efficient HPC systems. FPGAs are increasingly being used in HPC computing as they can bring higher energy efficiency than other alternative architectures such as CPUs or GPGPUs. However, the programmability barrier imposed by HDL languages is a serious drawback that limits their adoption. The recent introduction of OpenCL toolchains for FPGAs is a step towards lowering the barrier, but still produces application-specific designs that cannot be reused for other general-purpose computing unless the platform is reconfigured. Building many-soft-cores solutions using simple hardware accelerators can bring performance and energy efficiency and still provide a general purpose system to be used for different applications. To make it possible the current FPGA toolchains has been complemented to add support for fast many-soft-core platform description and verification (especially at higher levels of abstraction), to provide convenient parallel programming models and efficient methods to optimize the designs. On the other hand, I show how the same methods can be applied to implement systems using a task isolation approach to meet the requirements of a hard real-time safety critical system. Finally, the application of these technologies and optimization methods are used to efficiently implement both industrial and computational research problems.

Resum

Els últims avenços en els sistemes reconfigurables han fet possible que les darreres famílies de FPGAs tinguin prou recursos per permetre la implementació de sistemes multiprocessadors utilitzant més de 100 nuclis (soft-core) en una única FPGA. En aquesta tesis presento com implementar solucions basades en aquests sistemes i dos escenaris on aquests sistemes tenen interès: els sistemes de temps real dur i els sistemes d'altes prestacions eficients energèticament. Les FPGAs s'utilitzen cada vegada més en la computació d'altes prestacions ja que poden arribar a oferir més eficiència energètica que altres arquitectures com ara CPUs o GPGPUs. Tanmateix, la baixa programabilitat d'aquests sistemes, fruit de l'ús de llenguatges HDL és un inconvenient que limita la seva adopció. La introducció recent de fluxos de disseny per FPGA basats en OpenCL és un avenç per reduir la dificultat de programació, però malgrat tot, produeix arquitectures de propòsit específic que no poden reutilitzar-se per altres aplicacions a no ser que es reconfiguri la plataforma. Construint many-soft-cores que utilitzin acceleradors simples, es poden aconseguir prestacions i eficiència energètica i tot i així, encara disposar d'un sistema útil per executar altres aplicacions. Per fer-ho possible les eines de disseny actual per FPGA s'han complementat per afegir suport per construir arquitectures many-soft-core, proporcionar models de programació paral·lels i mètodes eficients per analitzar i optimitzar els dissenys. Per altra banda, es mostra com els mateixos mètodes es poden utilitzar per desenvolupar un sistema de temps real i alts nivells de seguretat mitjançant l'aïllament de tasques en els diferents nuclis del sistema.

Enthusiasm is followed by disappointment and even depression, and then by renewed enthusiasm.

Murray Gell-Mann

Acknowledgments

When I started my particular journey to Ithaca I prayed that the road would be long. Unfortunately, it looks as if gods were listening me…

I want express my deepest gratitude to all the people that encouraged me, that helped me, or contributed (no matter how) to make this long journey enjoyable and fun.

To my daughters Anna and Carla. For being lovely…most of the time. Anyway, I love you… all the time. I am happy to finish before you get to College! I will play more with you from now on!

To my ex-wife Mar. I probably devoted too much time to this, I'm sorry…and thanks for the many times you've expected a "thanks" coming out of my mouth that has never arrived.

To my parents, my brother and his family. Thank you for being there whenever I need help. Quite often lately.

To Txell. Thank you for the good times we have shared. I swear I am going to reach my toes one day.

To Disposats: Àngels, Carles, Marga, Miquel, Paquito, Montse, and Rita. I miss those dinners and adventures.

To my advisor Jordi Carrabina. Thanks for encouraging me to start, and not falling into despair during this last sprint. I hope you never have another Ph.D. student like me! …and this is not a self-compliment!

To Eloi. Thank you for the coffees, discussions, and everything that helped me think that I was not travelling a lonely road. I hope our roads can cross again in the future.

To Carme. I wish you much luck in your [shorter] journey!

To Albert. For not quitting smoking for some time, and for being always ready to roll up your sleeves if needed. Lab is empty without you, man! I wish we can climb together again soon!

To Jaume Joven. I had much fun working together.

To Eduard. For liking disguising! All the best with you growing family!

To Joan Plana. …for our discussions about the meaning of life and for, too often, paying my lunch. Man, I will be the one inviting next time!

To Josep Mesado and Martí Bonamusa. For continually asking me about some Drivers… I do not know what you are talking about! I sincerely wish that you have the success you deserve. All the best!

To colleagues from UAB, especially Lluis Terés, Toni Espinosa, Juan Carlos Moure, and Emilio Luque. Thanks for sharing wisdom, thoughts and experiences.

To Toni Portero and David Novo. It's long time since we worked together now. But I'm still convinced that you are great people to have around.

To David Marin. I admire your resolution and British style.

To Josep Besora. Be prepared! I am going to beat you in the Tennis court someday.

To my "Flama's fathers" friends, especially Pep Claret and Xavier Montanyà. Sharing drinks and laughs with you has been one of the most effective medicines to combat disappointment and depression.

To Chit Yeoh Cheng. Thanks for your help and for sharing your Chinese wisdom.

To Jose Duato, Greg Yeric, Rob Aitken, and Francky Catthoor, for your comments and valuable insight.

To Takehiko Iida and David Womble, for answering my mails asking for estimations of the power consumption of former #1 top500 machines.

To the Catalan, Spanish, and European taxpayers. Thanks for supporting my research.

To the anonymous reviewers of my papers. You were always right.

To the people that smile to strangers in the street. The world is better with you in it.

Contents

List of Acronyms

ALU	Arithmetic-Logic Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
CI	Custom Instruction
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DSP	Digital Signal Processor
DRAM	Dynamic Random-Access Memory
FF	Flip Flop
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GPC	General Purpose Computing
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HPC	High Performance Computing
ILP	Instruction Level Parallelism
IPC	Instructions per Cycle
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
ITRS	International Technology Roadmap for Semiconductors
LASER	Light Amplification by Stimulated Emission of Radiation
LE	Logic Elements
LUT	LookUp Table
MCAPI	Multicore Communications API
MIMD	Multiple Instruction Multiple Data

MISD	Multiple Instruction Single Data
MPI	Message Passing Interface
MPSoC	Multiprocessor System-on-Chip
NA	Network Adapter
NOC	Network On-Chip
NORMA	No Remote Memory Access
NRE	Non-Recurring Engineering
NUMA	Non-Uniform Memory Access
OPC	Operations per Cycle
OS	Operative System
PC	Program Counter
QoS	Quality of Service
RISC	Reduced Instruction Set Computer
RTOS	Real Time Operative System
SDRAM	Synchronous Dynamic Random-Access Memory
SIMD	Single Instruction Multiple Data
SMT	Symmetric Multi-Threading
SoC	System on Chip
SPMD	Single Program Multiple Data
TET	Total Execution Time
TLP	Thread Level Parallelism
UMA	Uniform Memory Access
VLIW	Very Long Instruction Word
WCET	Worst Case Execution Time
XML	Extensible Markup Language

List of Figures

List of Tables

1. Introduction

Since their inception in the eighties, Field Programmable Gate Arrays (FPGAs) have been benefiting from the continued effects of Moore's law. The reconfigurable devices, which were first used to reduce the glue logic to connect other more essential chips in a design, soon showed their ability to perform computing tasks.

As the integration capacity increased over the years, FPGAs were able to assimilate many functions that were implemented by the chips they were previously connecting. The main advantages of that replacement were: cost reduction, size reduction, and flexibility. In short, ASICs have fixed functions, FPGAs are flexible.

These have been good reasons for the market acceptance of this kind of devices. In the semiconductor industry, the economy of scale rules. That means that, the more units you fabricate, the lower the cost of production per unit you get. Since this industry is highly influenced by the advances on the capacity to integrate more transistors per silicon area, the level of integration can be generally predictable and are named after some physical feature of the build transistors. The term used to refer to this feature is "technology node". The physical feature that it measures has been changing over time, as the technology of transistors has been evolving as well [Arnold09]. Anyhow, as the capacity has increased, the Non-Recurring Engineering (NRE) costs of technology nodes has hiked as well, becoming harder to create new digital logic ASICs for low volume markets. These effects have maintained and fostered the demand for FPGA devices (see Figure 1). FPGAs are regular, replicable in large quantities, and flexible to be used in several application domains were they offer great performance and power efficiency [Coupé12].

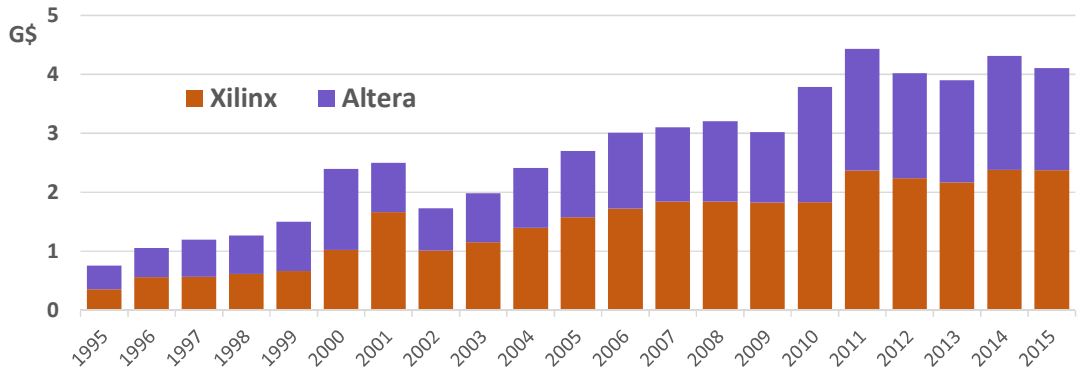


Figure 1 Evolution of the revenues of the two main FPGA providers: Xilinx and Altera

Their large unit sales and their simple replicable structure, allowed FPGA manufacturers to adopt new technology nodes as they were available. Currently, FPGA manufacturers are the second adopters of new technology nodes after major processor companies (like Intel, AMD, Samsung, Apple, NVIDIA, etc.). Low volume ASIC vendors are lagging behind (as shown in Figure 2).

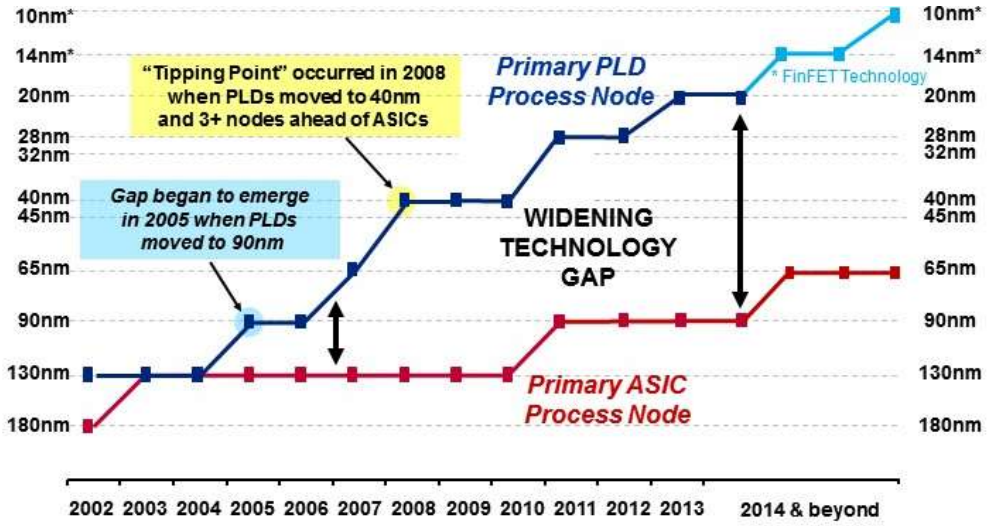


Figure 2 Evolution of the adoption of technology nodes by ASIC and FPGA vendors [Altera15]

The access to new technology nodes consequently increases the logic density of new devices. This creates a virtuous circle in which higher logic density allows wider adoption, rises demand, reduces manufacturing costs, and allows the access to new nodes. Figure 3 shows the number of Logic Elements (LEs) that were available in subsequent biggest Altera devices from 1995 to 2015.

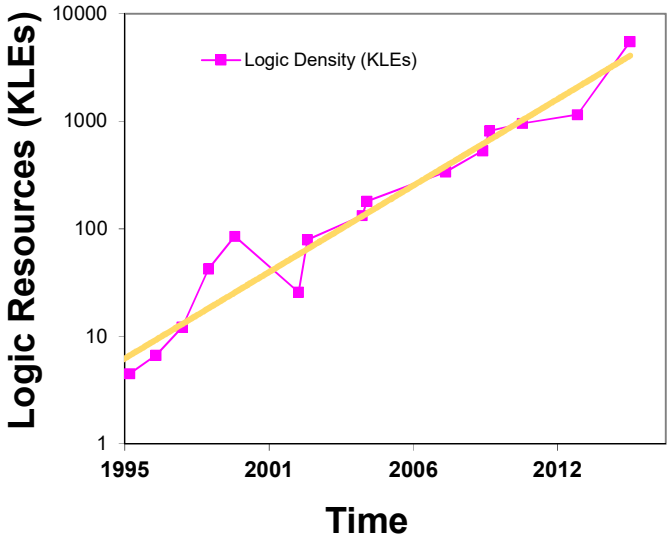


Figure 3 Evolution of the logic resources of biggest Altera devices from 1995 to 2015

A LE is a block that contains Look Up Tables (LUTs) and Registers. In the late nineties FPGAs already had thousands of them. A simple 32 bits Reduced Instruction Set Computing (RISC) processor needs few thousands of LEs to be implemented. With the new millennium it become possible to configure a subset of the LEs to work as a processor, and use the rest of the available logic to implement other system interfaces that could communicate with it by processor buses or other simple mechanisms. The term Soft-Core Processor was coined to denote a processor implemented using the reconfigurable cells of a reconfigurable device. Now, the logic density is high enough to be able to implement several processors using the reconfigurable cells of the devices or to save some portions of the silicon die to instantiate a real silicon processor design, memory blocks, or other complex Intellectual Property block.

If it is possible to create several soft-core processors in a FPGA, the questions that immediately arise are: How many? …and… What for?

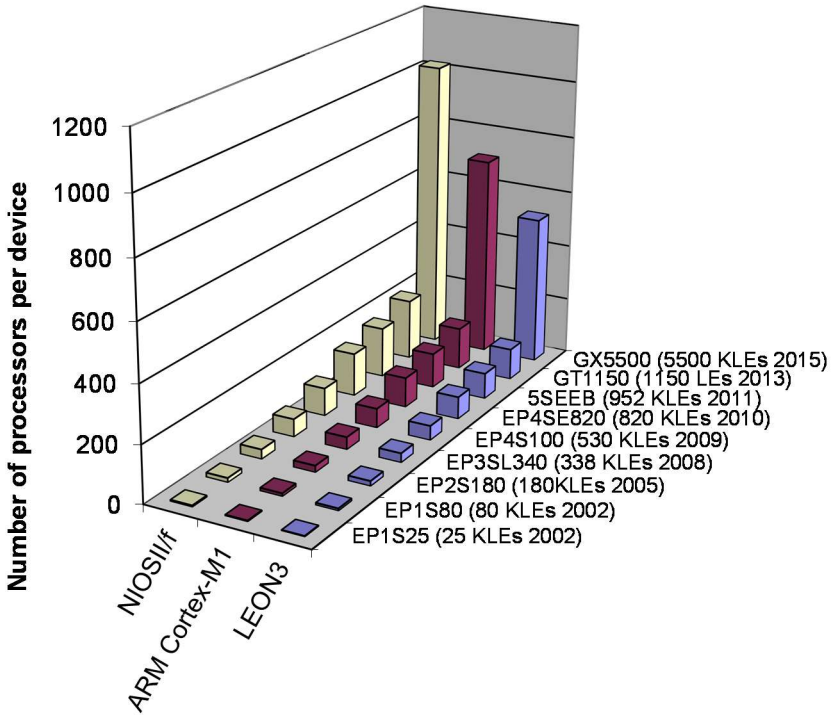


Figure 4 Potential number of processors per device

The first question is easier to answer. A simple estimation can be made, assuming that the number of LEs used by known soft-cores like NIOSII/f, ARM Cortex-M1, and LEON3 are 1800, 2600, and 3500 respectively, as used in [Pantelopoulos2012]. Note that this numbers are not considering other resources usually needed to create a multiprocessor system, such as the elements of the memory hierarchy or interconnect. To estimate the

usual additional logic needed be a system I multiplying this value by a constant factor 3, which is obtained from the empirical observation of typical system designs. Then I divide the number of device's reported resources by these values. The results are shown in Figure 4.

In summary, in early 2000s biggest FPGAs could already host few soft-core processors, by 2005 biggest FPGAs could host few tens of them, by 2010 that number was already a hundred, and now, in 2015, we should be able to host few hundreds of them.

Once we have an estimation of the number of soft-core processors that we could host, we face the tougher question: why would we do it?

My answer is two-fold. First, I propose that many-soft-core processors could be used to improve energy efficiency of some high performance workloads avoiding to invest too much effort in hardware design with classic HDL languages. Second, I maintain that they could be used to simplify the development of some hard real-time systems.

I will try to answer the question in more detail in the rest of this thesis, but I will also describe how it can be done.

1.1. Energy efficiency

Energy consumption of digital systems is driven by different factors determined by the logic technology. In the early days of computing, bipolar transistors were used. Their elementary logic gate circuitry requires to constantly drive current. After the initial boom in transistor count, the energy required for computers soon became very large. Power consumption was the major drawback of bipolar technology that forced the adoption of a much more energy efficient technology: CMOS.

We are still living in the CMOS era, and the power consumption of a CMOS circuit (1.1) is driven by two components: dynamic and static power. Dynamic power (P_{dyn}) is caused by the activity of the transistors when a change of state occurs. If no state change occurs, there is no dynamic power consumption. On the other hand, static power (P_{static}) is always present as soon as transistors are powered, and is mainly caused by various leakage currents happening in transistors.

$$P_{total} = P_{dyn} + P_{static} \tag{1.1}$$

In the past, static power (1.2) was usually considered as being negligible, but nowadays it is becoming more and more significant with the scaling down of technology nodes since the addition of the leakage produced by billions of powered transistors in a chip becomes a significant value. This is especially important in the FPGA domain since

FPGA designs usually suffer from a worse static power consumption, compared with their ASIC equivalents due to the fact that they need many more transistors to implement simple logic gates due to its inherent reconfigurability. The overhead can be as much as 80x factor [Kuon07].

If we make the simple assumptions that a circuit has a single voltage source leakage currents are equal for all the transistors of the circuit, the total static power would be defined by the number of transistors (N) used by the design (1.4). In FPGAs this value must be derived from the resources used [Leow13].

$$P_{static} = P_{leakage} \tag{1.2}$$

$$P_{static} = \sum V \cdot I_{leakage} \tag{1.3}$$

$$P_{static} = N \cdot V \cdot I_{leakage} \tag{1.4}$$

On the other hand, the dynamic power consumption (1.5) has two components. The power consumed to charge and discharge the load capacitance in every transition between two logic states and the power consumed during the interval where both N and P transistor networks connecting power to ground are conducting. This is known as the power caused by short circuit current. Since there are a discrete number of transitions per second, the formula can be rewritten as (1.6).

$$P_{dyn} = P_{trans} + P_{sc} \tag{1.5}$$

$$P_{dyn} = \sum (E_{trans} + E_{sc}) \tag{1.6}$$

The energy needed to charge a load depends on the capacitance of the load in a linear way, and on the voltage, in a quadratic way (1.7). The short current energy (E_{sc}) has not a simple equation as it depends on the time needed to charge the load and other factors such as the slope of the input transition, the ratios among transistors, etc. However, the short-cut current is usually either negligible with respect to the charge load current, or it is modelled as an additional load (1.8).

$$E_{trans} = \frac{1}{2} \cdot C \cdot V^2 \tag{1.7}$$

$$E_{sc} = \beta \cdot E_{trans} \tag{1.8}$$

To compute the total power consumption of a circuit we need to sum up all the contributions from all the nodes that experience transitions as expressed in (1.6). The number of transitions (T) that occur in a circuit during a second is usually a factor of the

number transistors (N), the clock frequency (f_{clk}), and a factor α, which denotes the probability of a transition of each transistor (also known as switching activity).

$$T = N \cdot \alpha \cdot 2 \cdot f_{clk} \tag{1.9}$$

We can rewrite (1.6) by using (1.7), (1.8), and (1.9) as (1.10)

$$P_{dyn} = N \cdot \alpha \cdot (1 + \beta) \cdot f_{clk} \cdot (C \cdot V^2) \tag{1.10}$$

… and taking (1.1) and using (1.10), and (1.4), we can rewrite it as (1.11)

$$P_{total} = N \cdot \left(\alpha \cdot (1 + \beta) \cdot f_{clk} \cdot (C \cdot V^2) + V \cdot I_{leakage}\right) \tag{1.11}$$

Power consumption if often dominated by switching activity. We are interested in minimizing energy consumption, not power. Energy consumption is the result of integrating the power consumption over a period of time (1.12), usually the time of a known task that can be used as a reference to compare the energy consumption of different systems.

$$E_{task} = P_{total} \cdot T_{task} \tag{1.12}$$

For our convenience, we can divide a task into primitive operations. Given a task consisting in Op_{task} number of operations, the time to complete it depends on the number of concurrent operations that we can execute per clock cycle, and the frequency of operation. If we use OPC (Operations Per Cycle) to denote the first factor. The expression would be (1.13)

$$T_{task} = \frac{Op_{task}}{OPC \cdot f_{clk}} \tag{1.13}$$

To avoid working with different tasks when reporting the energy efficiency of a system, a more task agnostic metric is usually adopted which takes into account simple operations of typical microprocessors. We can use the letter G (from *greenness*) to describe this energy efficiency metric as (1.14). The first factor of the equation is the number of operations per second that the system can perform, and the second factor is one over the power consumption of the system. The units of the energy efficiency factor G are Operations per second per Watt. As the bottleneck of scientific applications is usually floating point computing, a common variation of the metric is FLOPS/Watt (Floating point operations per second per Watt).

Note that Power times Time equals Energy, so (1.14) can be rewritten as (1.15), and simply read as the number of operations that we get per energy budged. The units of (1.15) should be Operations per Joule.

$$G = \frac{Op}{T}\frac{1}{P} \tag{1.14}$$

$$G = \frac{Op}{E} \tag{1.15}$$

If we substitute (1.13) into (1.14), we get an expression that depends on the power, the OPC and the frequency of operation (1.16)

$$G = \frac{Op_{task}}{P_{total} \cdot T_{task}} = \frac{Op_{task}}{P_{total} \cdot \frac{Op_{task}}{OPC \cdot f_{clk}}} = \frac{OPC \cdot f_{clk}}{P_{total}} \tag{1.16}$$

…and if we substitute (1.10) into (1.16) we get a final expression that takes into account most of the fundamental factors that drive the energy efficiency of a computing system.

$$G = \frac{OPC}{N \cdot \left(\alpha \cdot (1+\beta) \cdot \ (C \cdot V^2) + \frac{V \cdot I_{leakage}}{f_{clk}}\right)} \tag{1.17}$$

In order to have a better comprehension of how every parameter affects the energy efficiency factor, one can plot the contribution of every parameter while keeping other parameters constant. The only parameters with a positive correlation with the energy efficiency factor are OPC and f_{clk}. It is clear that executing more operations per cycle, if consumed power or frequency do not change, will increase the efficiency of the system in a linear way as shown in Figure 9. All other parameters appear in the denominator of (1.17), and all except f_{clk} have a negative correlation with the energy efficiency factor.

The parameter α indicates the probability of transitions occurring on transistors. The higher its value, the less energy efficient a system will be as it contributes to increase power consumption (see Figure 5). Nevertheless, it is unlike that the value of α goes to extremes, as it is typically between 10% and 20%.

Similarly, the number of transistors of a design (N) also contributes negatively to energy efficiency (see Figure 6). Using more transistors increases the total charge that causes dynamic power consumption, and also increase the silicon areas affected by leakage currents. FPGAs have a fixed number of resources that would normally be all

powered, contributing to static power. However, some advances [Altera12] allow controlling the number of active resources to eliminate their contribution to dynamic and static power consumption.

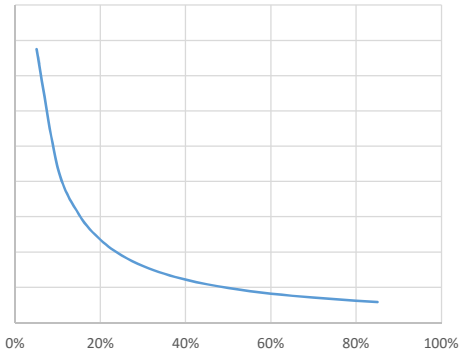


Figure 5 Energy efficiency as function of α

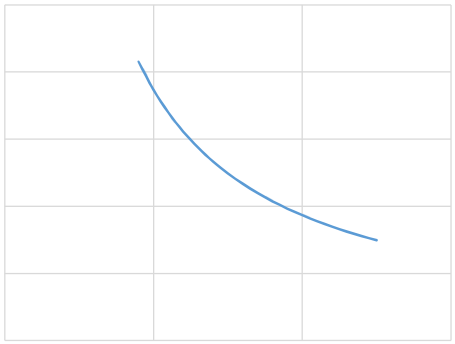


Figure 6 Energy efficiency as function of silicon area / resources

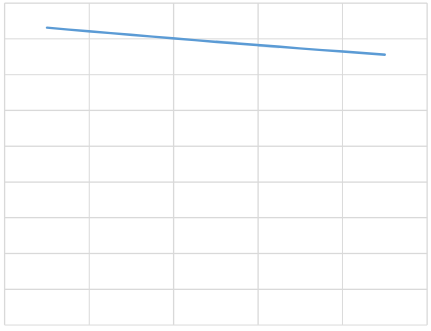


Figure 7 Energy Efficiency as function of $I_{leakage}$

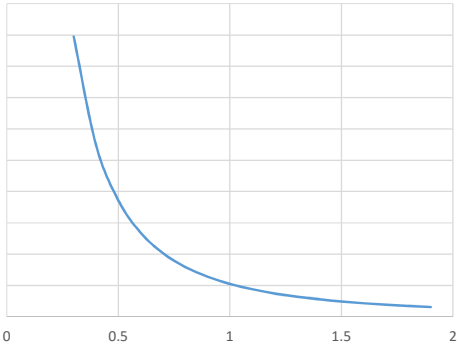


Figure 8 Energy Efficiency as function of supply Voltage

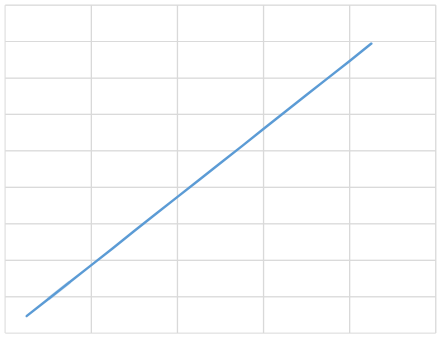


Figure 9 Energy Efficiency as function of OPC

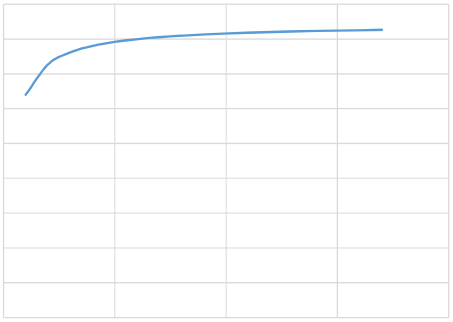


Figure 10 Energy Efficiency as function of F_{clk}

Leakage current also affects negatively to energy efficiency (as shown in Figure 7) as it increase the static power consumption, again some FPGA manufacturers [Altera12] allow reducing it by modifying the back bias voltage of transistors.

Reducing supply voltage should be a good contributor to energy efficiency as it contributes quadratically in the denominator (see Figure 8), however reducing it is not

straightforward because it cannot lower than the transistor threshold voltage and, often, the frequency must also be reduced to permit a correct operation of the system. In addition, FPGAs commercial boards usually work with a fixed supply voltage.

Finally clock frequency is positively correlated with energy efficiency, as shown in Figure 10. This could seem counter intuitive, since reducing frequency is often perceived as a power saving strategy (as in [Anghel11]). This is often the case when no computing is needed, and there is no option to switch off the device or finish the task. For instance, in a MPEG decoder that has already rendered a frame to display, but it must wait for the appropriate time to display it to satisfy the specified frames per second (as in [Lu02]). On the contrary, increasing clock frequency allows the task to finish faster, thus making the contribution of static power proportionally smaller. A more detailed analysis of the impact of frequency in energy efficiency will be done in section 2.7.3.

1.2. Energy Efficiency evolution

The energy efficiency of computing systems has evolved a lot during the last 20 years. The Top500 list [top500] collects information of the world's top *public* supercomputers. If we look at the evolution of the performance and power consumption of the best computer on the list over time (see Figure 11) we can easily realize that performance has increased more than 5 orders of magnitude while power consumption just increasing 2 orders of magnitude.

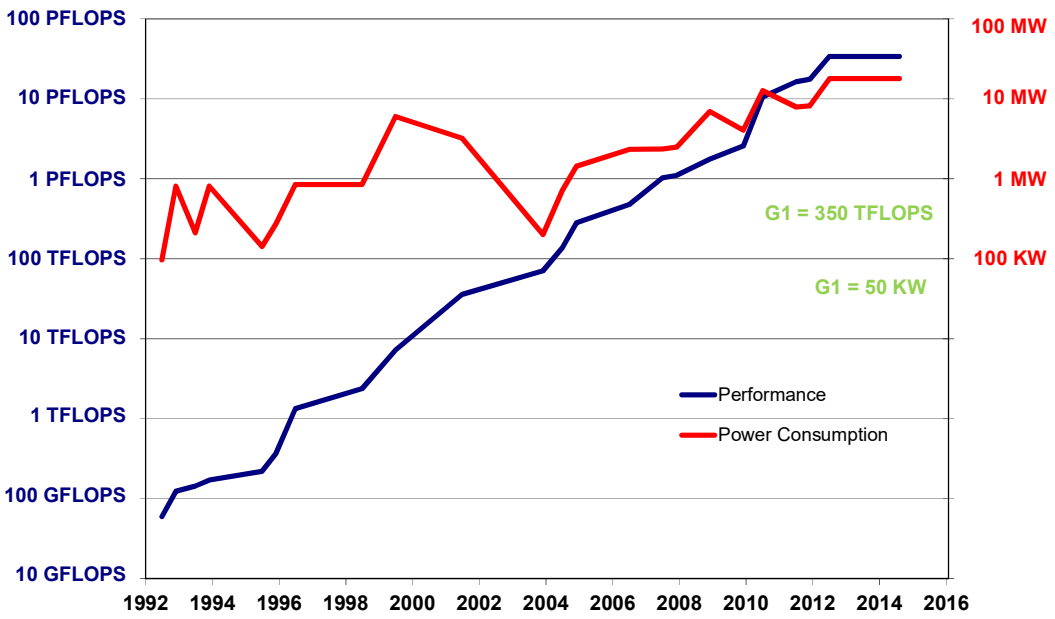


Figure 11 Evolution Performance and Power Consumption of top supercomputing systems [top500]

This means that the High Performance Computing (HPC) community has increased the energy efficiency of computing platforms by more than three orders of magnitude (see Figure 12). That is a remarkable achievement! The biggest advance was done between 2000 and 2005, when the whole HPC community understood that energy consumption, which had never been an important design constraint before, suddenly became a mandatory issue to tackle [Feng03]. This power-awareness forced the adoption of dynamic and voltage frequency scaling (DVFS), as well as other techniques (see [Valentini13]) as major drivers of energy efficiency. While a frequency reduction causes a decrease in energy efficiency, it also opens the possibility to reduce voltage. Voltage contributes quadratically to the denominator of (1.17), thus, the resulting combination of reducing both f_{clk} and V allows for an energy efficiency gain.

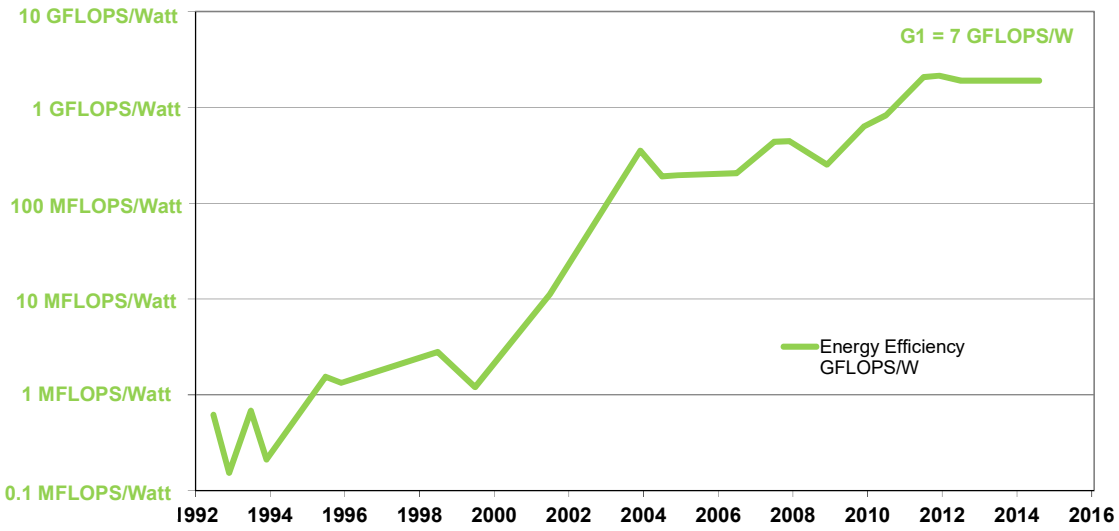


Figure 12 Evolution of energy efficiency of supercomputing systems

Another big improvement was done from 2008 to 2012, with the introduction of the IBM Blue Gene/Q chip, which used multiple techniques to reduce the power consumption while working at higher frequencies than its predecessors. Again, this example illustrates the fact that an increase in frequency can positively contribute to energy efficiency. The other used techniques are described in [Sugavanam13] and include clock gating, new floating point units, and multiple clock domains. Clock gating can be interpreted as a method to reduce the switching activity (α). When no clock is present in a circuit, no transition happens, so transition probability for that circuit becomes zero, as well as its dynamic power consumption. When averaged with the rest of the system, the effect is perceived as a reduction of global α. On the other hand, a new floating point unit

with multiple ways is a mean to increase the number of operations executed per cycle (OPC), which gives a linear increase to energy efficiency.

The recent move towards heterogeneous computing and the widespread use of accelerators like GPGPUs is also focusing on increasing OPC, but still has not produced a similar impact. The green500 list [green500], collects information from the systems listed in top500 but sorts the list by a descending value of the energy efficiency measured in GFLOPS/Watt. Green500 list is recently dominated by systems based on GPGPUs and many-cores, but the highest reported value is just 7.

1.3. Real-time systems

Real-time systems are described as the computing systems that have real-time constraints. Such constraints could be: to complete a certain task before a dead-line (as in Figure 13), or to react to a certain event in a given time frame (as in Figure 14). In other words, the correctness of a real-time system depends not only on the correctness of the program output, but also on the time at which the output is produced.

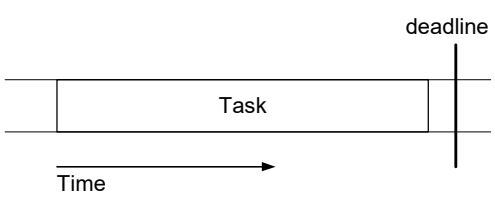


Figure 13 Real-time task with a certain deadline

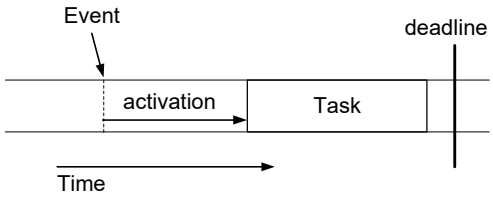


Figure 14 Real-time task triggered by an event

Generalizing, a real-time system works correctly if the time constraints (1.18) are met for all the tasks of the system.

$$T_{activation} + T_{task} < T_{deadline} \tag{1.18}$$

The complexity of real-time systems comes from different angles. First, task time can often not be well specific, as in (1.13), because of the variability on the inputs of the task. For example, the number of operations to encode a MPEG frame depend on the image to encoder. Images with high frequency components need more operations, while images with low frequency components need less. In most cases, these variations can be computed and expressed as a mean number of operations multiplied by a factor that considers the statistical variability at inputs γ_{in} as expressed in (1.19).

$$Op_{task} = Op_{mean}(1 \pm \gamma_{in}) \tag{1.19}$$

Real-time system designers usually try to determine the worst case execution time (WCET) with various means (like in [Lv09] and [Bernat12]). In an ideal single task environment, it could be just computed as (1.20).

$$WCET_{task} = \frac{Op_{mean}(1 + \gamma_{in})}{OPC \cdot f_{clk}} \tag{1.20}$$

On the other hand, in the same ideal environment activation time could be determined by the time needed to respond to an external event, for instance a hardware interrupt (1.21).

$$T_{activation} = T_{ISR} \tag{1.21}$$

But in reality real-time systems deal with many concurrent different tasks and their associated time constraints. Tasks are usually scheduled by a real-time operating system, and tasks potentially share resources interfering with each other. Thus, activation time depends heavily on the OS scheduler and the presence of other tasks. To cope with that, we need to rewrite (1.21) as (1.22) to introduce T_{sch} as the time to decide which is the next task to execute, and T_{enq} as the time devoted to finish other tasks that the scheduler considers more urgent than current one.

$$T_{activation} = T_{ISR} + T_{sch} + T_{enq} \tag{1.22}$$

Real-time systems community has been addressing the problems of scheduling and resource sharing for a long time. Several methods have been proposed in the literature to address them, like associating priorities to tasks and taking them into account for the scheduling decisions. In this context, the WCET has also to be rewritten to consider the interferences caused by sharing resources γ_{rsh}, and the time elapsed when the task was preempted by the scheduler T_{pre} as shown in (23).

$$WCET_{task} = \frac{Op_{mean} \cdot (1 + \gamma_{in}) \cdot (1 + \gamma_{rsh})}{OPC \cdot f_{clk}} + T_{pre} \tag{1.23}$$

If we rewrite (1.18) by using (1.22) and (1.23) we get a complex expression (1.24).

$$T_{ISR} + T_{sch} + T_{enq} + \frac{Op_{mean} \cdot (1 + \gamma_{in}) \cdot (1 + \gamma_{rsh})}{ILP \cdot f_{clk}} + T_{pre} < T_{deadline} \quad (1.24)$$

If we consider that applications can be executing on a multicore system and tasks can be parallel, with variable degree of scalability, it is even more complex (if even possible) to get an expression that allows analyzing system feasibility. Obviously, some academic attempts are made (like [David11]), but problems in industry are often solved by over-engineering. Note that the $OPC \cdot f_{clk}$ factor appears in the denominator of the quotient as well as the quotients that you would get after expanding $T_{ISR}, T_{sch}, T_{enq}$, and T_{pre} (since all them result from executing a number of instructions on a processor). Thus, increasing OPC would reduce the task activation and execution times so that the deadline could be easily met.

My main proposal is to use many-soft-cores architectures to provide a particularly appropriate platform to implement real-time applications used in cyber-physical systems, and to obtain a better estimation whether the deadline goals can be met.

Activation time ($T_{activation}$) can be reduced if tasks are isolated in processors, meaning that a whole processor is devoted to every single task. This will be possible if the number of tasks is lower than the number of processors. But this is not unlikely in a many-soft-core as the potential number of cores that they can embed is very large. Additionally, by implementing a system following this approach T_{sch}, T_{enq} disappear and activation is just determined by T_{ISR}. (1.21)

As T_{enq} also disappears, and the worst case execution time equation ($WCET_{task}$) is again determined by the single task equation function (1.20) simplifying the development of complex hard real-time systems.

1.4. Objective of this thesis

The main objective of this thesis is to propose a framework and set of methodologies to build and program effectively multiprocessor systems based of soft-core processors, and to analyze and provide estimates of the benefits of building such systems, in terms of their real-time capabilities, energy efficiency, and design productivity.

This goal can be unfolded on several specific bullet points, which are detailed as follows:

1. **Demonstrate how applications can benefit from many-soft-cores in the real time domain**. Making it easier to estimate its worst case execution time, and reducing their activation times.

2. **Provide the framework to materialize many-soft-architectures in a systematic way**. Current FPGA tools allow the creation of almost all required building blocks, but they still lack a complete effective toolchain.

3. **Adapt standard parallel programming models to many-soft-core architectures**, so that applications codes can be developed and deployed fast enough to cope with market demands.

4. **Enhance the development toolchain to provide a complete development process** that can iteratively improve and optimize applications to meet their requirements.

5. **Determine whether many-soft-cores can provide energy efficiency platforms**, which can help to solve the challenges of the HPC community in their path to Exascale.

1.5. Thesis organization

This thesis is organized in the following way. Chapter 2 will describe the benefits that FPGAs could theoretically provide, and the existing experiences to implement MPSoC systems in them. Chapter 3 will detail the proposed methods and tools that will allow creating many-soft-core processor architectures in a systematic way. Chapter 4 will focus on the adaptation of parallel programming models to many-soft-core architectures and the need of additional tools to get the complete development flow starting from the embedded application. Chapter 5 will present the implementation of several solutions based on multi-soft-core and many-soft-core platforms. They will be used to demonstrate their convenience and efficiency for hard real-time systems and to provide high performance energy efficient platforms. Finally Chapter 6 will present conclusions and outline potential future research in this domain.

2. Reconfigurable Logic and Soft-Core Processors

In the last 20 years, the microelectronics market has witnessed a fierce competition between ASICs and FPGAs to become the vehicle for new custom-computing systems. This battle has been fought in a changing context according to the evolution provided by the Moore's law. FPGAs have been increasing their market share during this period, displacing ASICs implementations to those requiring very high volumes. Flexibility, price and much shorter time-to-market are the main advantages of FPGAs vs. ASICs [Trimberger15]. At the same time, there is a strong interest in the HPC community on the energy efficiency that FPGAs could provide since it is no clear that alternative architectures (like processors and GPGPUs) can provide a significant increase on efficiency that would allow to reach the Exascale Challenge [Donofrio09] [Trefethen10]. Although the flexibility of FPGAs comes at a price on area and speed overheads compared with ASICs, market dynamics and economies of scale must be considered to have a better picture of the benefits that FPGAs can offer in terms of energy efficiency. A hardened design is obviously better in terms of area, speed, and power. However, it has to be sold in large quantities and must not need any structural change in order to be mass produced in ASIC technology. As shown in previous chapter, many designs cannot afford to use the latest technology nodes and must use more mature ones, e.g. 65nm instead of 14nm. When comparing FPGAs vs. ASICs it would be necessary to consider this difference in the access to technology nodes. Moreover, system-level designs have other alternative implementing platforms, such as general-purpose, or application-specific or domain-specific standard processors. In the following sections I intend to quantify the expected energy efficiency differences offered by those main alternative implementation platforms.

2.1. Energy Efficiency relations and ratios between technologies

As shown in (1.17) energy efficiency is determined by some technology parameters (like V, C, β and $I_{leakage}$) and some design constraints (like α, N, OPC, and f_{clk}). In this section I will intend to extract the relations between parameters on different

technology nodes and between FPGAs and ASICs. Those relations would allow to predict or speculate about the energy efficiency achievable on different platforms.

2.1.1. N

When comparing FPGAs against ASICs we cannot assume that N is the same. FPGAs are organized in logic blocks that usually include few LUTs and registers together with the configuration circuitry, while ASICs directly instantiate logic gates and registers. Due to this organization the same exact design uses more transistors (and silicon area) in FPGAs than in ASICs.

[Kuon07] studied the overheads of FPGAs on several small benchmark designs using a 90 nm process. Their conclusion is that the overhead in area, delay, and power is around 35, 3, and 14 respectively. [Lu07] did a similar analysis with a much complex design: a complete Pentium processor. Their results showed a factor 53, and 30, for area and delay respectively. More recently, [Wong11] did a similar analysis based on OpenSparc, Atom and Nehalem processors, again with similar results (see Table 1).

Table 1 Area, delay, and power overheads of FPGAs compared with ASICs

	Technology	Area	Delay	Power
Small Benchmarks [Kuon07]	90 nm	23-55	2-6	9-18
Pentium [Lu07]	65 nm	53	30	-
OpenSparc, Atom, Nehalem [Wong11]	65 nm	17-27	18-26	-

Given the inherent variability in the reported results, and in order to perform a simple analysis, I make the assumption that, for the same technology, the penalty factor for area is 40 when implementing in FPGAs (2.1).

$$K_a = \frac{N_{FPGA}}{N_{ASIC}} \approx 40 \tag{2.1}$$

2.1.2. f_{clk}

Operation frequency is a design parameter, but since we will try to increase it as much as possible to get higher energy efficiency factors we will be limited by a technology parameter: the maximum frequency of operation. [Kuon07] reports only a 3 or 4 factor for speed of ASICs vs. FPGAs, but I argue that this value could only be achieved for very small designs.

As circuit complexity increase the contribution of the flexible interconnection network increases, causing a degradation of the maximum attainable frequency. This is experienced in [Lu07], which reports a factor 30. [Wong11] estimates a factor between 18 and 26, which is much in line my experience. The operation frequency for most of the designs I implemented using different FPGAs is below 150 MHz, while standard processors have much stabilized its operation frequency around few GHz. Since this frequency of operation looks quite stable during recent years, I select a constant penalty factor of 20 for clock frequency between FPGAs and ASICs (2.2) for most technologies except 14nm.

$$K_f = \frac{f_{clk\ ASIC}}{f_{clk\ FPGA}} \approx 20 \tag{2.2}$$

With its line of devices implemented in the 14 nm process, Altera has introduced a new registered interconnect that allows to duplicate clock frequency. So in that case the frequency relation should be defined as (2.3), giving a penalty factor of 10. Nevertheless, this value is speculative and based on marketing announcements and not backed by any research.

$$K_{f,14} = \frac{f_{clk\ ASIC}}{f_{clk\ FPGA,14}} \approx 10 \tag{2.3}$$

2.1.3. *C*

One of the problems of the analytical model presented in (1.17) is that the total effective load capacitance per transistor is usually not reported. Dennard's scaling rules [Dennard74] stated some scaling factors for CMOS technologies that were valid until 2005. They use a scaling factor s, which was empirically found to be $\sqrt[4]{2}$ per year. [Taylor13] recently revisited them to adapt to the recent industrial regime (see Table 2).

The C reported in Dennard's and Taylor's scaling rules is related to the transistor capacitance, not the total effective load capacitance. The effective load capacitance depends on the transistor input gate capacitance, the drain-source capacitance, and other parameters linked with the circuit dimensions and wiring density. For the same circuit, a similar scaling factor of the effective load capacitance could be expected.

Table 2 Dennard's and Taylor's scaling rules

	Dennard	Taylor
Device Dimension tox, W, L	$1/s$	$1/s$
Devices	s^2	s^2
Voltage V	$1/s$	1
Current I	$1/s$	1
Capacitance C	$1/s$	$1/s$
Intrinsic delay CV/I	$1/s$	$1/s$
Power dissipation VI	$1/s^2$	1
Power density	1	s^2

Both predictions agree in that C scales as $1/s$ because it is basically determined by device dimensions. The International Technology Roadmap for Semiconductors (ITRS) do not reports the effective transistor load capacitance, but reports some figures that follow a similar trend. Namely, the total gate capacitance of transistors ($C_{g,total}$), and the intrinsic delay (CV/I). Figure 15 shows the predictions values for CV/I found in [ITRS02], [ITRS06], [ITRS07], [ITRS08], [ITRS12], and [ITRS13]. The important mismatches between the predicted values in 2008 and 2013 reports suggest that the reports have lost some of its predictive power for CV/I. When reporting the capacitance value of transistors, ITRS do not report "capacitance per transistor". Instead, the reported C_{total} value is a relative measure in fF/μm. To get the final gate capacitance value, we should multiply by the W of the transistors, but a design will have different Ws for different transistors. We cannot use a specific value for W, but we can assume that W scales with nodes ([Dennard74], [Taylor13]), so we obtain a gate capacitance value multiplying the ITRS relative gate capacitance (expressed in fF/um) by the ½ pitch length of the technology node (2.4). A quite similar approach is used in [Meenderinck09] to estimate the effective gate capacitance.

$$C_g \propto C_{total} \cdot \text{Pitch}/2 \tag{2.4}$$

Figure 16 depicts the values of C_g (expressed in aF) derived from the previous ITRS reports. In this case the predictive power for C_g seems slighly better than until year 2013, where an important mismatch occurs.

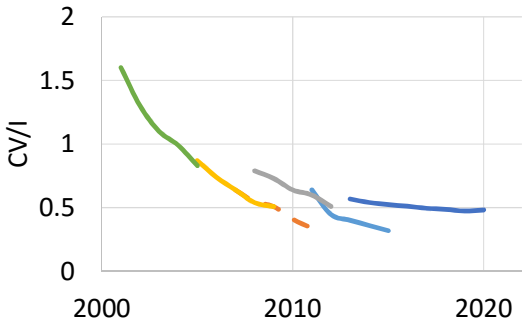


Figure 15 CV/I factor predictions reported by various ITRS reports ([ITRS02], [ITRS06], [ITRS07], [ITRS08], [ITRS12], and [ITRS13])

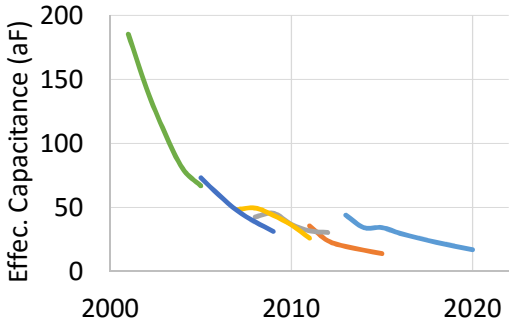


Figure 16 Cg computed from predictions reported by various ITRS reports ([ITRS02], [ITRS06], [ITRS07], [ITRS08], [ITRS12], and [ITRS13])

Although ITRS predictions report the expected evolution of different parameters along the coming years, the silicon market uses node names. Below the micron scale the node name is not directly related to any physical dimension but to the general view that the integration density still scales as square, as it used to be ([Arnold09] and [Iwai09]).

Table 3 Values for technology nodes of interest

Node Name	"65nm"	"45nm"	"32nm"	"22nm"	"16/14nm"
Year	2005	2007	2009	2011	2013
½ $pitch$ (nm)	90	68	52	38	40
	[Arnold09]	[Arnold09]	[Arnold09]	[ITRS12]	[ITRS2013]
L_{gate} (nm)	32	38	29	24	20
	[Arnold09]	[Arnold09]	[Arnold09]	[ITRS12]	[ITRS13]
CV/I (ps)	0.87	0.64	0.73	0.64	0.56
	[ITRS06]	[ITRS07]	[ITRS08]	[ITRS12]	[ITRS13]
$\frac{(CV/I)_{node}}{(CV/I)_{node-1}}$	-	0.73	1.14	0.87	0.88
$C_{g,total}$ $(fF/\mu m)$	0.81	0.71	0.87	0.93	1.1
	[ITRS06]	[ITRS07]	[ITRS08]	[ITRS12]	[ITRS13]
C_g (2.3)	72.9	48.28	45.24	35.34	44
$\frac{C_{g_{node}}}{C_{g_{node-1}}}$	-	0.66	0.93	0.78	1.24
Relative C_g	1	0.66	0.62	0.48	0.60

I am particularly interested in recent technology nodes 65nm, 45nm, 32nm, 22nm and 14nm. Table 3 collects some information from the ITRS reports with the node name equivalence of [Arnold09]. Note that the selected technologies follow a two year period, and the scaling factor s for a two year period is $\sqrt{2}$ [Moore75]. Nevertheless, the predictions show that CV/I and C_g do not follow the $1/s$ factor very well.

C_g does not decreases much in the 32nm node, and even increases in the 14nm node. We do not have to forget that ITRS reports predictions, and that those predictions should be backed by industry data. To validate the quality of the predictions, a comparison can be made between the data provided by Intel and that contained in the ITRS reports. This comparison can be made, for instance, taking the values for ½ pitch and gate length reported by both organizations.

Table 4 ½ pitch and L_{gate} predicted by ITRS and reported by Intel in different technology nodes

Node Name	"65nm"	"45nm"	"32nm"	"22nm"	"16/14nm"
ITRS	90	68	52	38	40
½ $pitch$ (nm)	[Arnold09]	[Arnold09]	[Arnold09]	[ITRS12]	[ITRS13]
Intel's	110	80	56	45	35
½ $pitch$ (nm)	[Bai04]	[Mistry07]	[Packan09]	[Auth12, Jan12]	[Natarajan14]
ITRS	32	38	29	24	20
L_{gate} (nm)	[Arnold09]	[Arnold09]	[Arnold09]	[ITRS12]	[ITRS13]
Intel's	35	35	30	26	20
L_{gate} (nm)	[Bai04]	[Mistry07]	[Packan09]	[Natarajan14]	[Natarajan14]

Table 4 shows such analysis. ITRS reports are in line with industry with a typical error of ±10% (as shown in Figure 17, and Figure 18). So we would expect to get an C_g estimate from the data reported by Intel in line with ITRS reports that allow us further analysis.

Intel follows a two-step approach (called tick-tock) to evolve the architecture of its processor lines. They change the technology node in the tick step, and they try to implement the new architectural changes in the tock step. This scheme seems perfect to make comparisons about the contributions of node scaling done in the tick step, however it looks like some tick steps are not as ideally isolated from architectural changes as we would like.

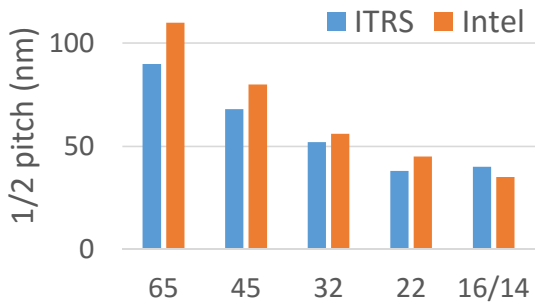


Figure 17 Difference between Intel's data for ½ pitch and ITRS predictions

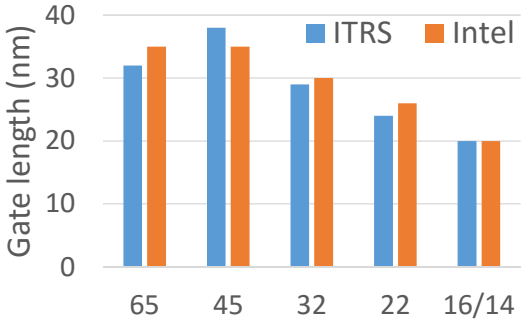


Figure 18 Difference between Intel's data for L_{gate} and ITRS predictions

[George07] reports the improvements on power consumption, comparing the *Penryn* architecture implemented in the 45nm node, with the *Memron* core 2 architecture implemented in the 65nm node. [Czechowski14] do not directly report C but analyzes the power consumption (both static and dynamic) of 45nm, 32nm, and 22nm. By comparing first the same *Nehalem* processor implemented in 45nm, and 32nm, and seconds the *Sandy Bridge* processor implemented in 32nm and 22nm. [Deval15] and [Nalamalpu15] report power savings for the *Broadwell* architecture (14nm) compared with previous *Haswell* architecture (22nm) running different applications. I take the worst value corresponding to a video display application. In top demanding applications there will be less opportunities to do clock gating or voltage scaling, methods that new Intel processors use extensively. I assume that remaining power savings should be caused by the technology node advance. In this case the power saving is 21%.

P_{dyn} is determined by the expression (1.10). Although it is not possible to direcly get C from P_{dyn} without knowing the rest of the parameters, I assume that most of them will be constant across technology nodes because they are replicating exactly the architecture. [Czechowski14] relates dynamic power is basically determined by C, but considering that all processors in this range include DVFS it is hard to make a strong statement from their data. Assuming that we are working at the full potential (f_{max}, V_{max}) I consider that maximum frequency is quite stable across the analyzed nodes. This makes sense due to the power density limit and the intrinsic delay evolution observed in ITRS predictions. On the other hand voltage has been decreasing moderately. Thus, I include a voltage correction factor from ITRS considering the quadratic contribution of voltage to power (2.5).

$$\frac{C_{g,node}}{C_{g,node-1}} \propto \frac{\frac{P_{dyn,node}}{P_{dyn,node-1}}}{\left(\frac{V_{node}}{V_{node-1}}\right)^2} \tag{2.5}$$

Table 5 Industry reported dynamic power factors between different technology nodes and my inferred capacitance factor value

Node Name	65nm	45nm	32nm	22nm	14nm
$\frac{P_{dyn,node}}{P_{dyn,node-1}}$	-	0.7 [George07]	0.57 [Czechowski14]	0.68 [Czechowski14]	0.79 [Nalamalpu15]
V_{node}	1.1 [ITRS06]	1.1 [ITRS07]	0.97 [ITRS08]	0.9 [ITRS12]	0.86 [ITRS13]
$\frac{V_{node}}{V_{node-1}}$	1	1	0.88	0.92	0.95
$\frac{C_{g,node}}{C_{g,node-1}}$	-	0.7	0.73	0.78	0.86
Relative C_g	1	0.7	0.51	0.40	0.35

When we compare the numbers obtained from industry data with the ITRS predictions there is a slight divergence, especially in the 14nm node (see Figure 19). The work described in [Martins15] synthesizes the same circuit for two different technologies using NanGate open cell libraries. The effective capacitance factor they found between a 45nm and a 15nm node is is 0.85. Taking the ITRS derived data our factor for the same technologies would be 0.91, and taking the data from industry is would be 0.50. Since the value reported by [Martins15] is between both estimates, I decided to take the mean of both estimates for our analytical model. The final values are reported in Table 6.

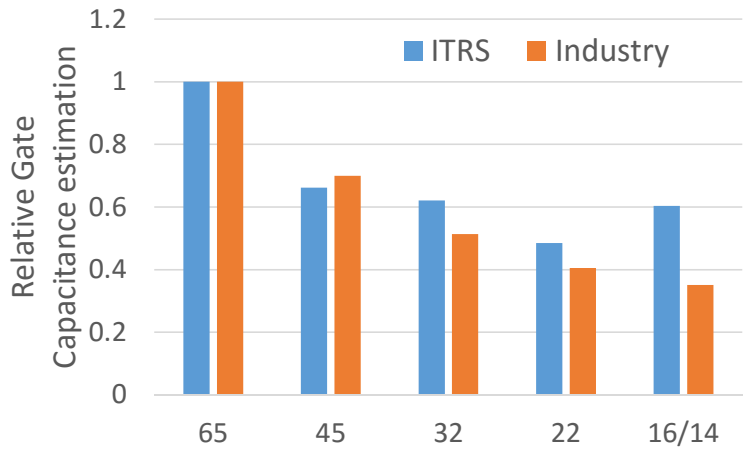


Figure 19 Relative gate capacitance inferred from ITRS data and Industry data

Table 6 Proposed gate capacitance relative factor for technologies, computed as the mean between the C_g values obtained from ITRS predictions, and the C_g values from industry

Node Name	65nm	45nm	32nm	22nm	14nm
C_g (aF)	72.9	49.65	41.32	32.44	34.78
Relative C_g	1	0.68	0.56	0.44	0.47

2.1.4. I_{leak}

The leakage current is also not easily found. ITRS reports leakage in relative dimension units. Again we use a correction factor to infer the leakage current per transistor as (2.6).

$$I_{leak} \propto I_{leak} \cdot \text{Pitch}/2 \tag{2.6}$$

Figure 20 depicts the obtained leakage current per transistor base on the information of previously mentioned ITRS reports. In this case the predictive power is almost inexistent until year 2012.

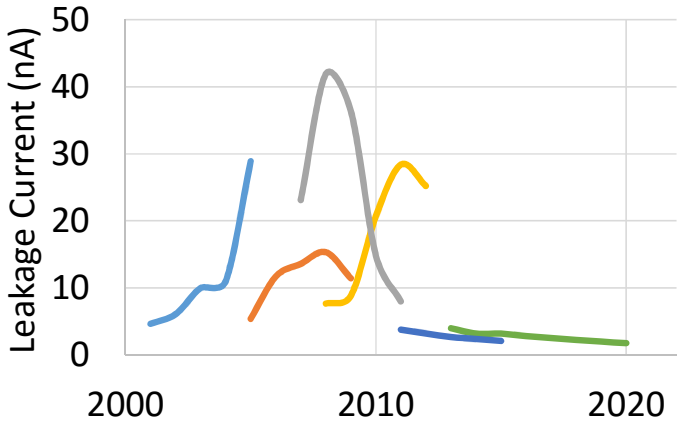


Figure 20 Leakage current using (2.6) and ITRS predictions

Table 7 $I_{leakage}$ derived from ITRS data

	"65nm"	"45nm"	"32nm"	"22nm"	"16/14nm"
ITRS	90	68	52	38	40
½ $pitch$ (nm)	[Arnold09]	[Arnold09]	[Arnold09]	[ITRS12]	[ITRS2013]
I_{off} $(nA/\mu m)$	60	340	170	100	100
	[ITRS06]	[ITRS07]	[ITRS08]	[ITRS12]	[ITRS13]
$I_{leakage}$ (nA)	5.4	23.12	8.84	3.8	4

Again I try to confront this information with data from industry. Collecting the I_{off} values reported in [Bai04], [Mistry07], [Packan09], [Auth12], and [Natarajan14] I compose Table 8.

Table 8 $I_{leakage}$ derived from industry data

	"65nm"	"45nm"	"32nm"	"22nm"	"16/14nm"
Intel's	110	80	56	45	35
½ $pitch$ (nm)	[Bai04]	[Mistry07]	[Packan09]	[Auth12, Jan12]	[Natarajan14]
I_{off} $(nA/\mu m)$	100	100	200	10	10
	[Bai04]	[Mistry07]	[Packan09]	[Auth12]	[Natarajan14]
$I_{leakage}$ (nA)	11	8	11.2	0.45	0.35

There is some divergence (see Figure 21). Again, to test we compare with the data from [Martins15]. They report 8.6 nA for 45nm and 5.3 nA for 15nm. Given the fact that ITRS predictive power seems limited for $I_{leakage}$ and that [Martins15] is also based on a predictive model, we use the values of Table 8 as a reference for our study.

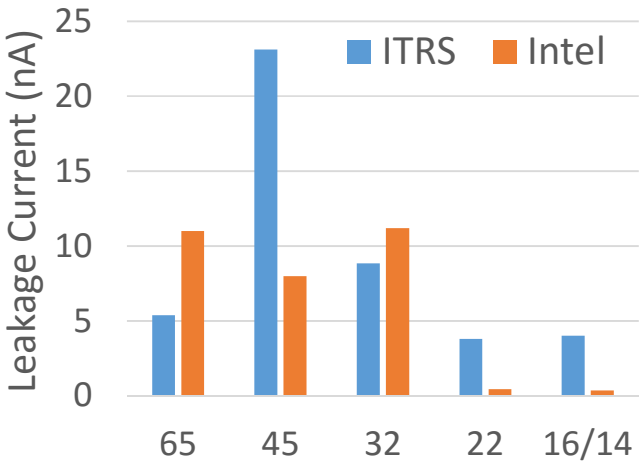


Figure 21 Leakage current estimations

2.2. Comparing the same technology nodes

In the case of having to decide between implementing a given circuit using FPGA or ASIC platforms on the same technology node, we should have to analyze the relation between the energy efficiency for both implementations by using a ratio based on the expression (1.17). Again, I consider that this scenario is highly improbable because the costs of ASIC manufacturing and market dynamics make new technology nodes only accessible to high volume manufacturing devices like processors and FPGAs.

Anyhow, if that would be the case, the efficiency factor would be defined by equation (2.7). Since, in this case, the technology node and the circuit are the same, α, β, C, $I_{leakage}$, and V would be the same for both designs.

$$\frac{G_{FPGA}}{G_{ASIC}} = \frac{N_{ASIC}}{N_{FPGA}} \cdot \frac{\left(\alpha \cdot (1+\beta) \cdot (C \cdot V^2) + \frac{V \cdot I_{leakage}}{f_{clk\ ASIC}}\right)}{\left(\alpha \cdot (1+\beta) \cdot (C \cdot V^2) + \frac{V \cdot I_{leakage}}{f_{clk\ FPGA}}\right)} \tag{2.7}$$

The differences would be due to the area overhead of FPGA designs and the lower maximum frequencies available to them. If we define some common factors from dynamic power as (2.8) and common factors from static power as (2.9) and use the area and frequency overheads that were defined by (2.1) and (2.2), we can rewrite (2.7) as (2.10).

$$\delta = \alpha \cdot (1+\beta) \cdot (C \cdot V^2) \tag{2.8}$$

$$\theta = \frac{V \cdot I_{leakage}}{f_{clk}} \tag{2.9}$$

$$\frac{G_{FPGA}}{G_{ASIC}} = \frac{1}{40} \cdot \frac{\delta + \theta}{\delta + K_f \cdot \theta} \tag{2.10}$$

The (2.10) expression should be interpreted as being determined by the activity factor α. If the activity factor is high, δ will probably be significantly higher than θ and the efficiency relation will tend to be 1/40. This means that the FPGA implementation will be 40 times more energy inefficient than ASIC one. In case that activity is low, static power will dominate, and things will get even worse, up to 800 or 400 when using the 14nm node. Figure 22 depicts the penalty factor G_{ASIC}/G_{FPGA}, which is the inverse of (2.9), as function of the quotient δ/θ.

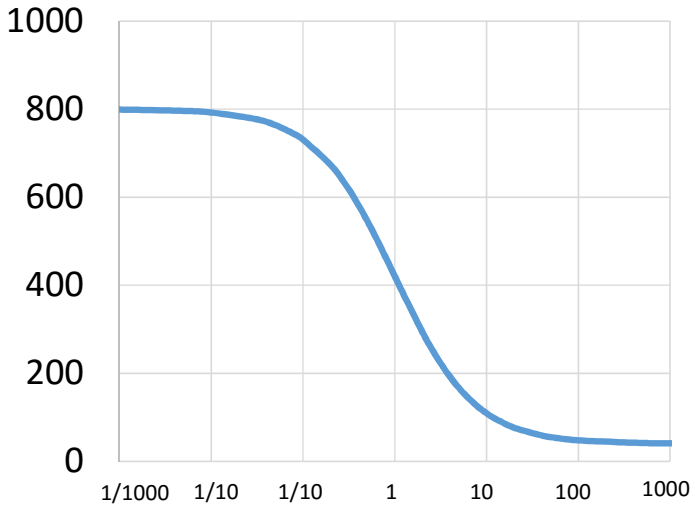


Figure 22 Energy Efficiency penalty of FPGA designs vs. ASIC implementation as a factor of the δ / θ relation

2.3. Comparing latest node for FPGA with ASIC entry node

Being mass market devices, FPGA manufacturers have been able to be second to access new technology nodes like 14nm, while the increasing NRE costs and foundry market dynamics have relegated ASIC designs to the older 65nm nodes (as shown in Figure 2). When comparing the same implementations in ASIC and FPGA for medium volume, where economics would not be the main constraint, it would make sense to compare 14nm node for FPGA (as Stratix 10) and 65nm for ASIC implementations (available in major foundries like TSMC, UMC, and Global Foundries).

To perform the comparison, for exactly the same design, we could assume the same α, β, but we should consider diferent values for C, and V. As proposed in Table 6 the relation factor between device capacities in 65nm and 14nm is given by (2.10) and the relation between voltages would be given by (2.11) as reported in Table 3. A summary of the used parameters are shown in Table 9. I speculate that the total effective capacitance could be a constant factor (like 10) from the gate capacitance. This is an over simplification, but could be useful to get an idea of the relative values.

Table 9 Parameters used by comparison between FPGAs in 14nm and ASICs in 65nm

	FPGA	ASIC
Node Name	"14/16"	"65"
C_g (F)	$34.78 \cdot 10^{-18}$	$72.9 \cdot 10^{-18}$
C (F)	$347.8 \cdot 10^{-18}$	$729 \cdot 10^{-18}$
V (V)	0.86	1.1
f_{clk} (Hz)	$300 \cdot 10^{6}$	$3 \cdot 10^{9}$
$I_{leakage}$	$0.35 \cdot 10^{-9}$	$11 \cdot 10^{-9}$

$$\frac{G_{FPGA_{14}}}{G_{ASIC_{65}}} = \frac{N_{ASIC_{65}}}{N_{FPGA_{14}}} \cdot \frac{\left(\alpha \cdot (1+\beta) \cdot \left(C_{ASIC_{65}} \cdot {V_{ASIC_{65}}}^2\right) + \frac{V_{ASIC_{65}} \cdot I_{leakage}}{f_{clk\ ASIC}}\right)}{\left(\alpha \cdot (1+\beta) \cdot \left(C_{FPGA_{14}} \cdot {V_{FPGA_{14}}}^2\right) + \frac{V_{FPGA_{14nm}} \cdot I_{leakage}}{f_{clk\ FPGA}}\right)} \tag{2.11}$$

$$\frac{G_{FPGA_{14}}}{G_{ASIC_{65}}} = \frac{1}{40} \cdot \frac{(\alpha \cdot\ 8.82 + 0.04)}{(\alpha \cdot\ 2.57 + 0.01)} \tag{2.12}$$

In short, the FPGA implementation would be around 11 times less energy efficient than its ASIC equivalent.

At that point I am asking myself whether we could do anything to reduce G even further.

The flexibility of the FPGA could allow us to re-spin the design to get higher OPC factor or reduce the N in a higher factor. Another option to increase G would be to reduce the static power by powering off inactive cells of the FPGA as proposed in [Gayasen04]. But power gating can be generally applied in ASIC as well.

2.4. Comparing 40nm node for FPGAs with 14nm node for ASICs

FPGAs could be just an order of magnitude less energy efficient than ASICs when comparing different nodes (such as 65nm vs 14nm). Considering the flexibility of the FPGAs and their faster time-to-market, there would be less reasons to choose ASIC instead of FPGA as the implementation platform for many designs. But, how do FPGAs compare against processors? Do processors achieve better energy efficiency?

Mainstream processors are already build. They are top selling products, and they have first access to technology nodes. They are fast and super-flexible. Is there any reason to use FPGAs instead of processors for energy efficiency?

The answer is yes. FPGAs can be more energy efficient than processors. More details to elaborate this answer will be given in section 2.7 The better energy efficiency of FPGAs vs. processors is often the result of creating a specific architecture that increases the OPC to a high number. Processors have a fixed maximum OPC determined by their architecture.

However, my thesis is that, for certain applications, better energy efficiency can be obtained using soft-core processors instead of processors.

In section 2.6 I will compare the energy efficiency level obtained in the Intel i7-5500u processor (a modern low power microprocessor) with the energy efficiency level achieved in a NIOSII on a Stratix IV 530K FPGA.

Before this comparison, it would be necessary to know the contributions of having a different technology node, and a different implementation platform. In Table 10 I list the technological parameters used to do the comparison. The core Voltage in Stratix IV is reported to be 0.9 V (from Altera). I compare i7-5500u, build in 14nm with Stratix IV 530K build in 40nm.

Table 10 Parameters used by comparison

	ASIC	FPGA
Node Name	"14/16"	"40"
C_g (F)	$34.78 \cdot 10^{-18}$	$49.65 \cdot 10^{-18}$
C (F)	$347.8 \cdot 10^{-18}$	$496.5 \cdot 10^{-18}$
V (V)	0.86	0.9
f_{clk} (Hz)	$3 \cdot 10^{9}$	$150 \cdot 10^{6}$
$I_{leakage}$	$0.35 \cdot 10^{-9}$	$8 \cdot 10^{-9}$

The expression (2.13) if obtained from expanding (1.17) in the energy efficiency factor of FPGA vs. ASIC.

$$\frac{G_{FPGA_{40}}}{G_{ASIC_{14}}} = \frac{N_{ASIC_{40}}}{N_{FPGA_{14}}} \cdot \frac{\left(\alpha \cdot (1+\beta) \cdot \left(C_{ASIC_{14}} \cdot {V_{ASIC_{14}}}^2\right) + \frac{V_{ASIC_{14}} \cdot I_{leakage}}{f_{clk\ ASIC}}\right)}{\left(\alpha \cdot (1+\beta) \cdot \left(C_{FPGA_{40}} \cdot {V_{FPGA_{40}}}^2\right) + \frac{V_{FPGA_{40nm}} \cdot I_{leakage}}{f_{clk\ FPGA}}\right)} \tag{2.13}$$

By substituting the parameters of Table 10 in (2.13), an expression depending of $\alpha \cdot (1+\beta)$ is obtained (2.14).

$$G_{FPGA_{40}} = \frac{G_{ASIC_{14}}}{40} \cdot \frac{(\alpha \cdot (1+\beta) \cdot 2.57 + 0.01)}{(\alpha \cdot (1+\beta) \cdot 4.01 + 0.48)} \tag{2.14}$$

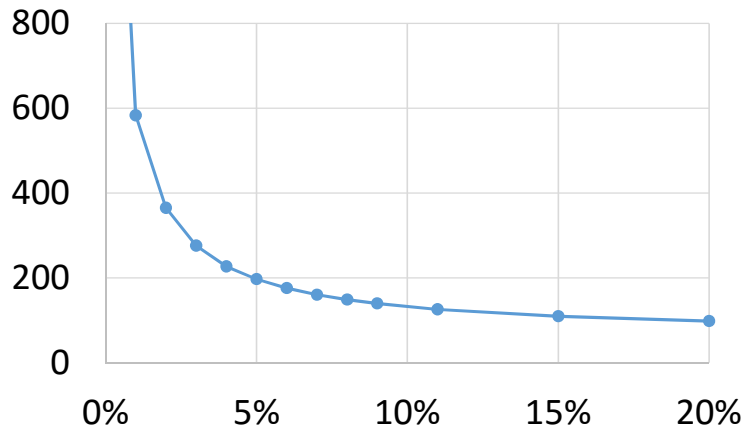


Figure 23 Energy Efficiency penalty for FPGAs as factor of α

Figure 23 depicts the energy efficiency relation for various values of switching activity. For the same design, working above a 10% activity the energy efficiency penalty of FPGAs is around 100 with respect to ASICs in the last technology node.

2.5. Processor Technologies

The evolution of computing has been basically a quest for higher OPC. To increase OPC the application must have some degree of parallelism, i.e. parts of the code must be able to execute concurrently. Depending on whether the concurrent parts are fine grain or coarse grain, we find Instruction Level Parallelism (ILP) or Thread Level Parallelism (TLP). ILP is found when, in the stream of instructions executed by a processor, several of them can be executed concurrently.

Before comparing soft-core processors and mainstream processors, we should briefly review some of the optimizations that have been introduced during processor architecture evolution to understand the differences that we will find in terms of ILP, often measured as Instructions executed Per Cycle (IPC). A much more detailed analysis is found in [Hennessy11].

In the most simplistic way, a processor is a system containing an ALU, and all the necessary complements to keep it busy.

Usually, a memory is indispensable to keep it busy. From the early days of computing, most computers works following the concept of stored program [VonNewmann45]. Meaning that the memory contains the program which is executed as well as the data used by the program to operate. Data can also be exchanged with the external world through I/O operations, but for our analysis we would ignore that. In order to analyze the problem we will present the stages needed to execute an instruction. Whatever the implementation is, either hardwired in an FSM, microcoded, etc. First, in the IF stage the instruction has to be fetched from memory. In the ID stage the instruction has to be decoded in order to know what to do. Usually instructions need operands, so there is a OF stage to fetch the operands. When operands are ready the instruction can be executed during the EX stage. Finally the result has to be stored during the WR stage.

First computers, like the EDSAC [Wilkes49], where taking the Von Neumann architecture ideas. EDSAC clock frequency was very low (500 KHz), but worse, the IPC as low as 1/769. The problem with early designs is that they needed several cycles to complete the execution of an instruction, since the control execute one stage every clock cycle. In early designs only a single accumulator register was used and hardwired as an input to the ALU (see Figure 24). So, typically 3 memory accesses would be needed complete an instruction, one to read the instruction, another one to read the operand (as the other was implicitly the accumulator register), and an additional one to store the result.

The move to transistorized electronics first, and microelectronics later, gave a boost to performance. So, it took a while before implementing architectural changes to increase the IPC. First, mass production microprocessors, like Intel 4004, Motorola 6800, Zilog Z80, MOS 6502 (running at 1Mhz), started to use more registers, as well as higher systems like VAX11, creating register files that later were a central part of the RISC concept [Patterson85]. Having more registers a program can avoid going that often to external memory, thus reducing the cycles waiting for memory. Register files are often considered as a kind of cache of level zero, as depicted in Figure 25. In these processors the number of operations executed per cycle was still around 1/10.

Mainframe IBM System/360/85 introduced the use of cache memory in 1969, and during the 80s, various processors like Motorola 68000, and Intel 80386 (running at 16 MHz) used cache memories. Cache memories, as depicted in Figure 26, were introduced after the observation that memory positions accessed by programs are often repeated, so some cycles could be saved if some accessed positions could be stored in faster memories [Goodman83]. This recall of same accessed positions happen when fetching instructions for a number of reasons, mainly because of loops and function calls. But it also happens with memory positions, due to variable use. Because the external memory was cheap, but very slow compared with on-chip registers, cache were implemented using fast SRAM memory while external memory was high latency DRAM. SRAMs could cope with chip processor speed, while DRAMs not. The ILP was still around 1/10.

The processor has a bottleneck in memory access; [Cragon80] describes how the Harvard architecture can improve performance by having separate code and data memory. In this case instruction fetch and operand fetch can happen simultaneously. Although originally thought as separate memories like many DSP processors (as [Yasui91]), implementing Harvard architectures with different caches connected to the memory, had a good performance while allowing the Von-Neumann idea of stored program.

The RISC concept proposed by [Patterson85] and executed by [Hennessy81] exploited the concept of pipelining. Although instructions still need several clock cycles to complete all needed stages, the different stages can be executed concurrently so that different instructions can be executed simultaneously on different stages. Pipelining is implemented by introducing registers between stages (as shown in Figure 29). In a pipelined processor an instruction could be fetched every cycle. Since RISC processors have a common instruction size, the PC register can be automatically increased to fetch the next consecutive instruction without needing extra cycles. The benefit in performance

is obvious and, since the architectural changes are not significant, the benefits in energy efficiency as well. Not only because the higher ratio of ILP per power, but also because power consumption can be reduced as a consequence of glitch reduction (as explained in [Wilton04]). However there are situations in which the processor pipeline cannot be fully utilized. For instance when there are cache misses, when there are dependencies from registers, or when there are branches. In these situations the pipeline must be either stalled or filled with bubbles. Because of shorter combinational paths, pipelining had a size effect of allowing to increase f_{max}. Processors like MIPS R2000, Intel i860, HP PA-RISC were implemented following the RISC principle, achieving not only higher frequencies, but much higher values of IPC (close to 1).

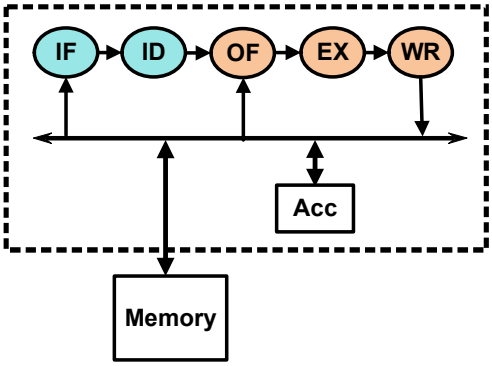


Figure 24 Architecture of simple Von-Neumann processor. The instruction is fetched from memory int the IF phase. Operands can be fetched from Memory and the Accumulator register. Results can be writen to Accumulator or memory. The different stages cannot be simultaneously executed.

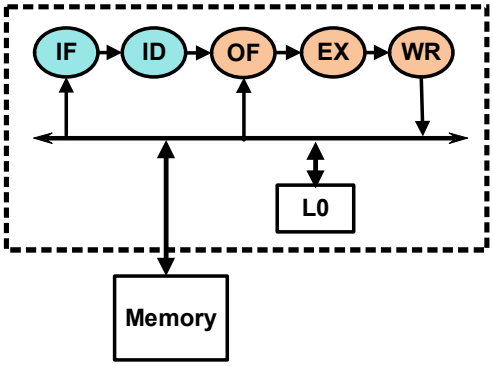


Figure 25 Simple architecture with a register File. Instruction still needs several cycles to execute but the operand fetching time can be reduced if they are in the register file.

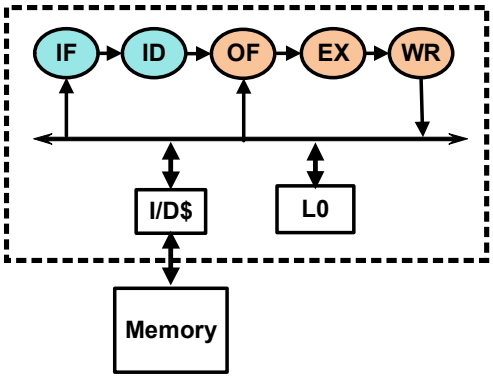


Figure 26 Introduction of a cache to reduce the time to access the external memory.

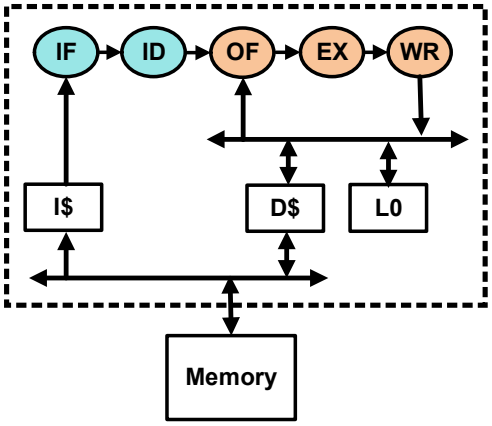


Figure 27 Harvard architecture. Simultaneous access to instruction and memory could double performance.

One of the important penalties paid by pipelined designs is the amount of in-flight operations that must be canceled when a branch occurs. All the pipeline is filled with sequential instructions after the branch operation instead of having the instructions from the addresses where the program branch to, so all those operations must be discarded. Early studies showed that up to 30% of the instructions could be branches [Wiecek82]. The effect on the relative IPC of the application is given by the expression (44), where P_{flu} is the probability of pipeline flush due to executing a branch instruction and $L_{pipeline}$ the number of pipeline stages that have to be invalidated. Figure 28 shows how IPC can be affected by the probability of branches and the pipeline length. A processor with a pipeline of 20 stages (in orange) is much penalized than a processor with a pipeline of 5 stages (in blue) if branches occur.

$$K = \frac{1}{(1 - P_{flush}) + P_{flush} \cdot L_{pipeline}} \tag{2.15}$$

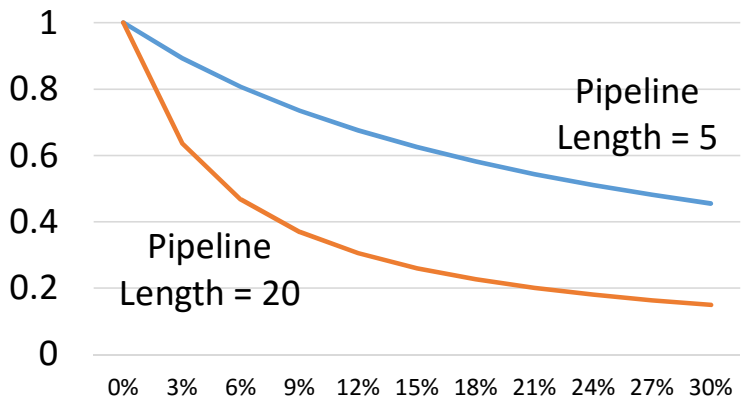


Figure 28 Relative IPC as function of the probability of pipeline flush for different lengths of the pipeline. In orange $L_{pipeline} = 20$. *In blue* $L_{pipeline} = 5$.

Actually, there is a way to mitigate this penalty: branch prediction. Branch prediction can be implemented in many different ways. The idea is that a new stage is introduced in the pipeline to predict whether the next instruction is a branch and what should be the branch address. By knowing the predicted target branch address the instruction fetch unit move the PC to it and fetches it, instead of fetching PC + instruction size. Branch predictors usually store the same history of the branches taken, to be able to optimize them (see Figure 30). After much research during the nineties, accuracy of

branch prediction stabilized around 97% for many benchmarks ([Yeh14]), so the probability of misprediction ($P_{misprediction}$) is around 3%.

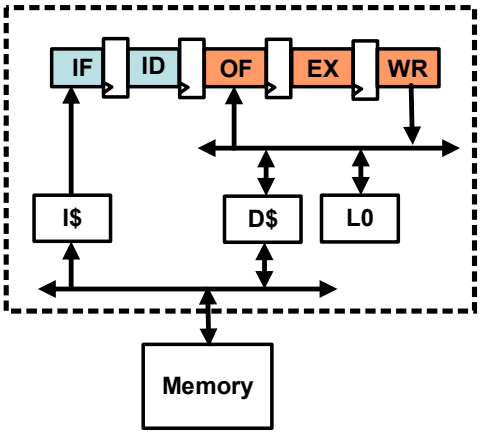


Figure 29 Pipelined architecture. Pipelining allows to increase the throughput while maintaining the latency of instruction execution.

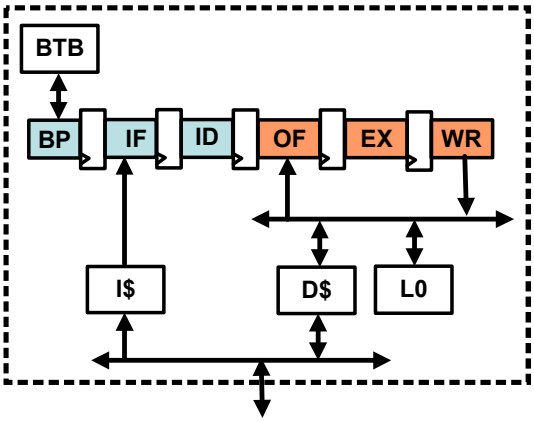


Figure 30 Branch prediction

Currently it is assumed modern programs have a probability of branch P_{brach} around 20% [Yeh14]. As the probability of a pipeline flush (P_{flush}) is given by (2.16), the current probability of flushing the pipeline is 0.6%.

$$P_{flush} = P_{brach} \cdot P_{misprediction} \tag{2.16}$$

By using (44) we can plot the case penalty factor K as function of pipeline length.

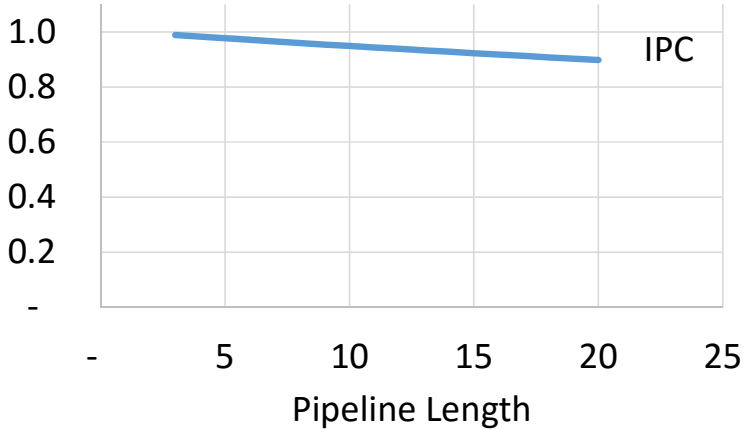


Figure 31 IPC factor as function of pipeline length assuming a branch prediction accuracy of 97% and branch probability instruction of 20%.

During the nineties, as the processor frequency was increasing, but the external memory performance, based on Dynamic Random-Access Memory (DRAM), was not increasing at the same pace [Matas97]. This divergence, illustrated in Figure 32,

motivated for the insertion of customized cache memories implemented in DRAM technologies, but with lower latencies than external memory. First, L2 was introduced and then L3, as shown in Figure 33, and Figure 34. More levels and bigger caches contributed to reduce cache misses and reduce the probability to be negatively impacted by them in the IPC, however they did not allow to increase the IPC over the factors already achieved.

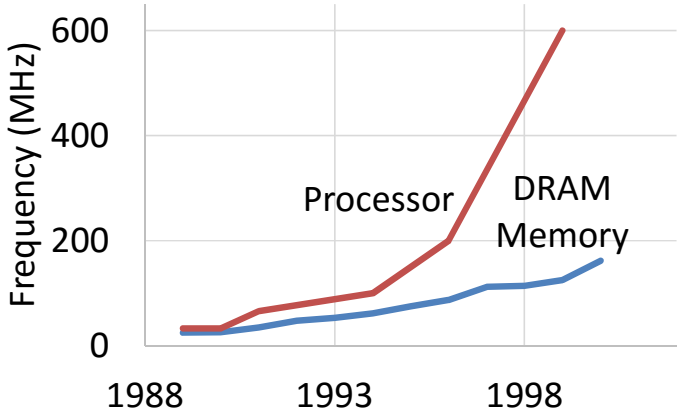


Figure 32 Frequency of operation of mainstream processors and their external DRAM memories during the nineties

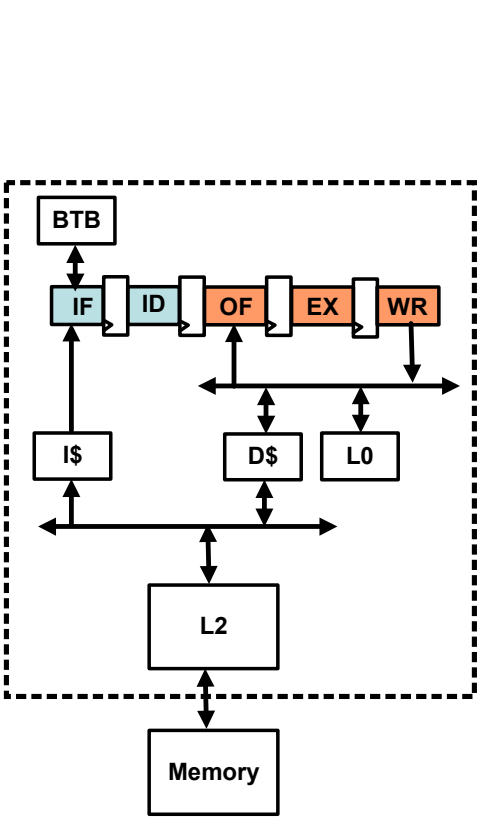


Figure 33 Level 2 Cache

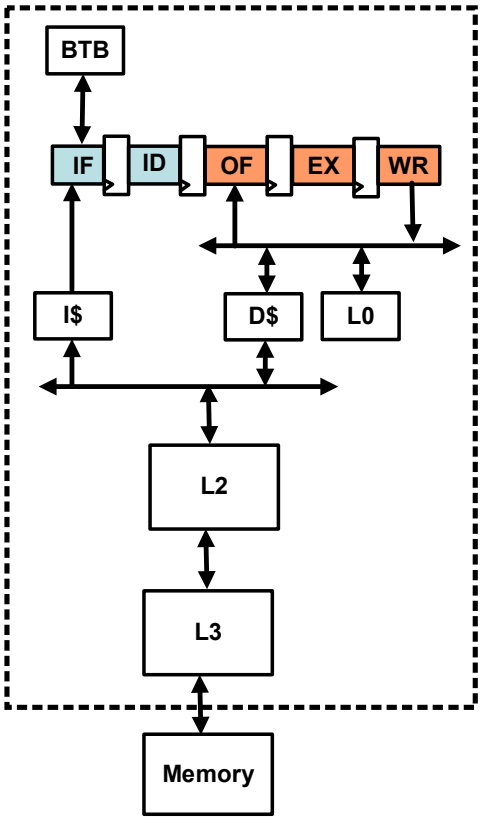


Figure 34 Level 3 Cache

Previous reviewed optimizations were good to reach and maintain the IPC factor 1. However they were not allowing to go over 1.

To get higher values, functional units must be replicated, so the idea changes from being an ALU and all the necessary surrounding components to keep it busy, to the concept of having multiple ALUs and all what it's needed to keep them busy.

One of the early implemented options was to replicate the ALUs and make them execute the same instruction simultaneously on different data (see Figure 35). This kind of architecture was called vector architecture because the different data fed to the different ALUs were though as elements of a vector of registers. See Figure 35. Vector architectures were made popular in HPC by Cray during the 70s, and later by NEC during the early 2000s. NEC SX-9 (as described in [Zeiser09]) was able to reach ILP to 32 by having 8 sets of pipelines containing 2 multiply and 2 add functional units. During late nineties AMD and Intel incorporated vector instructions into the x86 architecture. There were some of them, 3Dnow, MMX, SSE, SSE2, SSE3, SSE4, XOP, FMA, CVT16, The outcomes of these instructions can be 4, 8, or 16 depending on the processor. It is not clear whether it is economically viable to sustain the manufacture of vector processor, although vector units in consumer processors are widely used as their benefits in multimedia workloads [Chi15].

The drawback of vector processing is that functional units (vector units on its jargon) have to execute the same instructions. The alternative is to let functional units execute different instructions. The Very Long Instruction Word (VLIW) architectures implement this approach by combining different instructions on a single wider instruction. In this case data dependency must be controlled by the compiler. The compiler has the challenge to try to put as many instructions in parallel as possible, (instead of filling with *nops*) to fully utilize the available performance. There are some drawbacks to this approach, the code is usually bigger (because of the padding) and more bandwidth is needed by the instruction fetch unit (see Figure 36). VLIW has been used extensively in embedded DSP processors by Texas Instruments, Analog, and other firms, They usually include 4 or 8 functional units, so max IPC is usually 8 [Kozyrakis02].

Vector and VLIW architectures put the pressure to the compiler to be able to fully exploit the functional units. The other alternative is to try to put the pressure on the architecture itself. So that the compiler produces a standard stream of instructions, and the architecture tries to execute them in parallel. That's what superscalar architectures do.

They fetch multiple instructions per cycle, but instead different than VLIW the instructions are from sequential positions on the stream (see Figure 37). As VLIW this has the drawback of increasing the memory bandwidth. In order to execute instructions the processor has to determine dependencies between operations and registers. To do so all operations are usually inserted in a queue and scheduled and dispatched to functional units. Pipelines of superscalar processors are usually much larger than simple processors. Pentium was the first Intel Superscalar processor. Pentium 4 reached up to 31 pipeline stages.

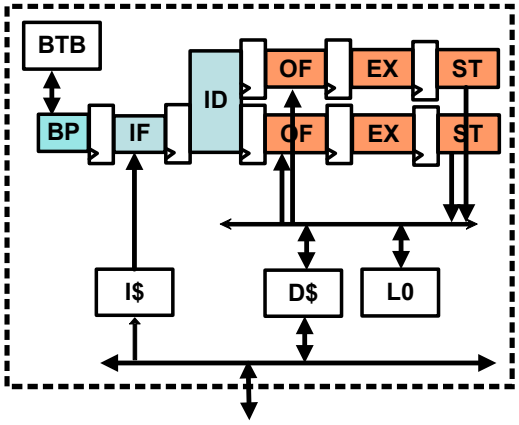


Figure 35 Vector architecture

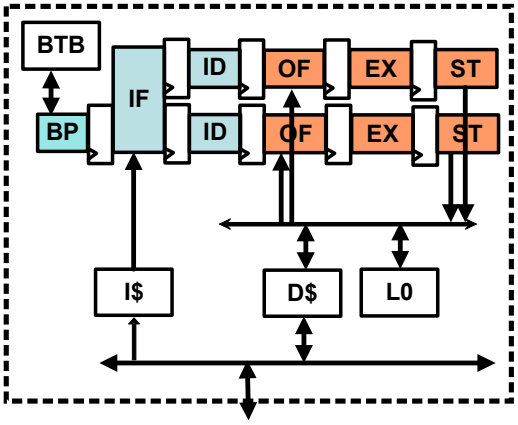


Figure 36 VLIW architecture

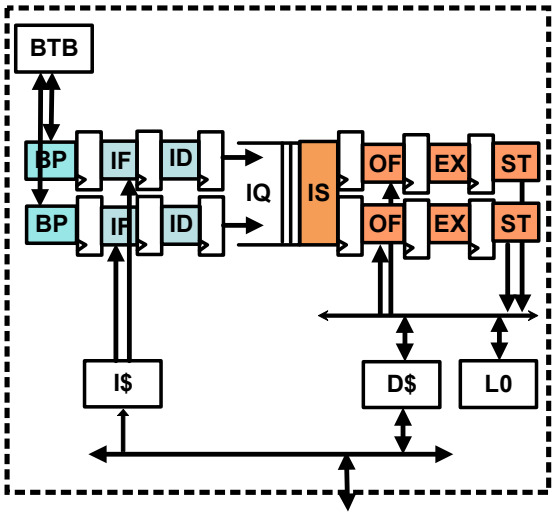


Figure 37 Superscalar architecture

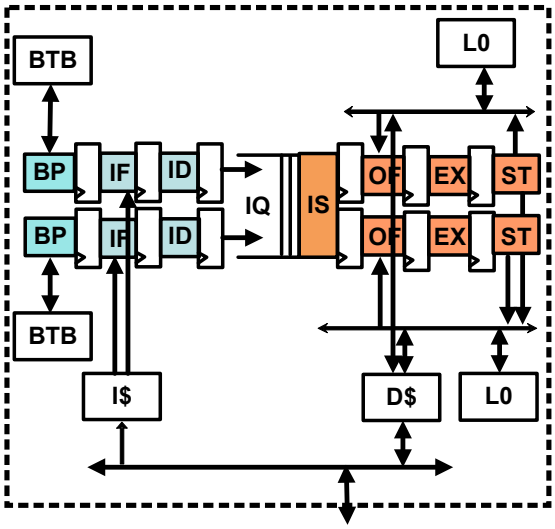


Figure 38 Symmetric Multi-Threading (SMT) architecture

With the superscalar design there is some probability that data dependencies do not allow to fully utilize the pipeline. Since processors were executing multitasking OS for long time, wouldn't it be wise to fetch instructions from different threads instead of the same one. In this way no data dependency would occur and pipelines would be fully occupied. This is why Symmetric Multi-Threading (SMT) architectures were proposed

(see Figure 38). SMT architectures are more efficient because they share functional units, instead of duplicating them.

In the context of the famous Flynn's taxonomy of computer organization (see Table 11), the increment of the instructions per cycle has been achieved by moving from SIMD architectures, to SIMD and MIMD. Vector, Superscalar, and even pipelined architectures can be considered SIMD architectures. VLIW, and SMT can be considered MIMD architectures.

Table 11Flynn's taxonomy of computer organization [Flynn72]

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SIMD	SIMD
	Multiple	MISD	MIMD

To continue increasing the number of instructions executed per cycle previous architectures can be replicated to create parallel architectures. Those, would fall in the MIMD classification. A deeper analysis of such architectures will be done in the next chapter.

2.6. Comparing Soft-core Processors with Processors

A soft-core is a simple processor available as HDL source code that can be synthesized on a FPGA. They were first introduced in 2001 by FPGA manufacturers like Xilinx, Altera and Lattice to compete with microcontrollers and DSPs simplifying the SoC concept. Some of their first uses were targeting the telecommunications market and embedded real-time equipment [Dalay03]. They were useful to cooperate in the control of DSP datapaths (like in [Guan01]), and provide control logic that was becoming too complex to implement using FSMs. But soon, they were able to displace microcontrollers doing all kind of real-time tasks.

Today, most popular soft-core processors are 32bits RISC processors with a not very long pipeline. Some processors like (Microblaze, NIOS, and RISC-V) allow to choose between some shorter and longer pipelines. Shorter pipelines allow for a much compact design by sacrificing performance. On the other hand, longer pipelines allow higher frequencies and higher performance.

Table 12 Soft-Core Processors sorted by academic popularity

Processor	Arch.	ISA	Open Source	Bits	Pipeline	Popularity
MicroBlaze [Xilinx06]	RISC	Microblaze	No	32	3/5	6970
NIOSII [Altera10]	RISC	NIOS	No	32	1/5/6	6640
LEON3 [Gaisler07]	RISC	SPARC V8	Yes	32	7	1400
PicoBlaze [Chapman03]	RISC	PicoBlaze	Yes	16	3	1320
LEON2 [Gaisler03]	RISC	SPARC V8	Yes	32	5	1020
OpenRISC [Tandon11]	RISC	OpenRISC	Yes	32	5	644
JOP [Schoeberl03]	CISC	Java	Yes	32	4	343
Cortex-M1 [ArmM1]	RISC	Thumb-2	No	32	3	269
OpenSPARC T1 [Parulkar08]	RISC	SPARC V9	Yes	64	6	204
LatticeMico32 [LM32]	RISC	LM32	Yes	32	6	103
SecretBlaze [Barthe11b]	RISC	Microblaze	Yes	32	5	50
TSK3000A [AltiumTsk]	RISC	MIPS	No	32	5	48
RISC-V [Waterman11]	RISC	RISC-V	Yes	64	2/3/5	48
xr16 [Gray00]	RISC	xr16	No	16	3	47
Zet [Zet]	CISC	8086	Yes	16	8	13

Table 12 shows some of the most popular soft-cores. The majority of soft-cores have a RISC architecture with the exception of JOP and Zet. CISC is more complex then RISC to implement, it usually performs worse, and it is less energy efficient.

In order to glimpse the energy efficiency differences between processors and soft-core processors I propose to compare NIOS II implemented in a Stratix IV device (on a 40nm die) against a low power computer processor, the Intel Core i7-5500-U (Broadwell architecture implemented in a 14nm node). Laptop processors are especially designed with energy efficiency in mind.

There are some measures of the energy efficiency of Intel Core i7 in the literature. [Vasudevan10] reports 0.01 GFLOPS/Watt for a Nehalem architecture, which is implemented in 32nm. [Amsler12] reports 1.43 GFLOPS/Watt for a Sandy Bridge architecture, implemented in 32nm. This is a big difference that would need further analysis.

In [Renbi09] the energy efficiencies of different implementations of matrix multiplication are analyzed. One of the designs is software based running on a NIOSII\e implemented on a Cyclone II (TSMC 90nm) running at 50 MHz. The work reports a power consumption of 160mW and an energy consumption of 18015nJ. By applying

(1.12) we can deduce that the application execution time was 0.1125 s. As the IPC for NIOSII\e is 0.15, as reported in [Altera15b], this results on 843.75 KFLOPS. By applying (1.16), this results in 0.00527 GFLOPS/Watt. The IPC value of NIOSII\f is 6.6 times higher than NIOSII\e, so if we take the previous value and multiply it by 6.6, the energy efficiency would be around 0.035 GFLOPS/Watt. It seems that literature do not shed enough light for a consistent comparison on the energy efficiency of both platforms, but if we take the best value for the Intel platform a first impression is that energy efficiency factor could be about 40x in favor of i7 processor.

Another approach to obtain comparable values could be to apply the analytical model presented in chapter 1. The issue width (number of instructions fetched per cycle) of *Broadwell* is 8, and the total thermal power reported for Intel Core i7-5500-U is 15 W. Considering a maximum frequency of 2.9 GHz, if we apply (1.16) we get an energy efficiency factor of 1.54 GFLOPS/W. A similar reasoning can be followed for the NIOS. In this case there is no manufacturer information of the processor total thermal power, but it can be obtained by the PowerPlay Altera tool. PowerPlay reports 2.1 W for a simple NIOS processor with FPU running at 160 MHz, from which only 0.6 W are from dynamic power. By applying (1.16) we get 0.07 GFLOPS/Watt.

So, this second look, which gives similar results for Intel processor, indicates that the energy factor could be just 20x in favor of i7.

In order to try to have a better approximation to the real energy efficiency ratio I execute some benchmarks on an FPGA system and a last generation laptop computer. The FPGA system is a Terasic DE4 board containing a Stratix IV 530K with the NIOS II design running at 160MHz. The laptop is a Lenovo Yoga 3 containing a Intel Core i7-5500-U. The power measurement is done with a power meter smart plug that reports power consumption in Watts with an error of ±10mW. Since both systems contains more elements than just the FPGA device and microprocessor the power consumption in idle state must be subtracted. The power measurement for idle state in the FPGA device is done by applying a global reset signal. This reduces the dynamic power to zero, but not the static one. The power measure for the idle state in the laptop is a little trickier. Intel processors have several power modes that basically modify the voltage and frequency operating points. I forced the highest performance setting on the operative system, which forces the highest voltage and frequency point, and run the minimal applications on the system. Once the idle state power is measured for both systems I execute the benchmarks in an infinite loop, so that their power consumption can be obtained. Having obtained the

benchmark execution time in both platforms and taking into account the baseline idle power, an energy efficiency factor considering only dynamic power G_{dyn} is obtained (2.4).

$$G_{dyn} = \frac{Op}{T}\frac{1}{P_{dyn}} \tag{2.17}$$

The first benchmark I executed is the Dhrystone benchmark. This is a small synthetic benchmark that measures the time to execute a simple integer arithmetic small loop. Results are given in MIPS_{VAX}. The benchmark is so simple that the code and data fits in the processor caches so that cache misses (ignored in my energy efficiency expression) do not occur. This ensures that architectures are executing in similar conditions because both are accessing internal memory. Test results are shown in Table 13.

These results show that the core i7 is 130 times faster than the NIOS but just 5 times better in dynamic power consumption. If we add an estimation of the consumed static power the energy efficiency relation between the i7 and the NIOS is around 15, which is slightly less than previously computed.

*Table 13 Dhrystone bechmark results in core i7-5500-U and NIOS II running at 160 MHz on a Stratix IV 530K. * means estimated*

	i75500U	NIOSII	NIOSII+FPU
f_{clk}	2.9 GHz	160 MHz	160 Mhz
MIPS_{VAX}	20028	153.82	153.82
$\frac{\text{MIPS}_{VAX}}{f_{clk}}$	6.91	0.96	0.96
P_{dyn}	11.812 W	0.451 W	0.490 W
$G_{dyn} = \frac{\text{MIPS}_{VAX}}{P_{dyn}}$	1696	341	314
relative MIPS_{VAX}	1	1/130	1/130
relative P_{dyn}	1	1/25.5	1/23.4
relative G_{dyn}	1	1/4.97	1/5.40
$P*$	12.812 W	1.451 W	1.49 W
$G*$	1563	103	106
relative $G*$	1	1/14.7	1/15.1

I compare our results with [Roberts09], which analyzes the energy efficiency of a Cortex A8 in the OMAP3530 implemented in 65nm, and the Intel Atom 330 implemented in 45nm. We see in Table 14 that the energy efficiency of a NIOS is 16 times smaller than the Cortex A8 and 8 times smaller than the Atom 330 implemented in a 45nm node. The Cortex A8 has a 2-issue superscalar architecture while the Atom is a simple scalar. Being implemented in a comparable node (45nm vs. 40nm for the Stratix IV), it is interesting to realize that the energy efficiency factor is just 8 and not 40 as predicted.

Table 14 Dhrystone benchmark results from [Roberts09]

	Cortex A8	Atom 330
f_{clk}	600 MHz	1.6 GHz
MIPS_{VAX}	883	1822
$\frac{\text{MIPS}_{VAX}}{f_{clk}}$	1.47	1.14
P	0.5 W	2 W
$G = \frac{\text{MIPS}_{VAX}}{P}$	1766	911

The Dhrystone benchmark is based on integer arithmetics, but many modern codes are based on floating point arithmetics. To test the performance and energy efficiency of floating point code I execute the Linpack benchmark on NIOSII and core i7. Results are shown in Table 15. The dynamic power efficiency of the i7 is 49 times higher than the NIOSII, and if we take into account the static power it raises to 91 times. This significant different between integer and floating point efficiency has a very simple explanation.

Notice that the number of floating point instructions per cycle IPC have been reduced drastically from 0.96 to 0.06 in NIOS. The reason for this drop is that floating point unit in NIOS is not part of the processor pipeline but it is implemented as a custom instruction. This means that the pipeline cannot be fully used when executing floating point instructions and the performance drops according to the latency of the floating point instructions.

*Table 15 Linpack bechmark results in core i7-5500-U and NIOS II running at 160 MHz on a Stratix IV 530K. * means estimated*

	i75500U	NIOS+FPU
f_{clk}	2.54 GHz	160 Mhz
$KFLOPS_{Linpack}$	5364873	10243
OPC	1.85	0.06
P_{dyn}	10.12 W	0.95 W
$G_{dyn} = \frac{KFLOPS_{Linpack}}{P_{dyn}}$	530126	10782
relative $KFLOPS_{Linpack}$	1	1/524
relative P_{dyn}	1	1/10.6
relative G_{dyn}	1	1/49
$P*$	11.12 W	1.95 W
$G*$	482453	5253
relative $G*$	1	1/91.84

If we compare with [Roberts09] we realize that in this occasion the Cortex-A8 is just 9 times more efficient than the NIOS, while the Atom is 89 times more efficient (see Table 16). The reason for the performance drop in the Cortex performance is very similar to the drop experienced by the NIOS in floating point operations. Floating point support for the Cortex A8 is implemented either by the VFP-lite coprocessor or through the NEON vector unit. Some compilers by default use the VFP-lite unit which has a latency of 18-21 cycles. As a consequence performance is penalized and energy efficiency drops.

Table 16 Linpack benchmark results from [Roberts09]

	Cortex A8	Atom 330
f_{clk}	600 MHz	1.6 GHz
$KFLOPS_{Linpack}$	23376	933638
OPC	0.04	0.58
P	0.5 W	2 W
$G = \frac{KFLOPS_{Linpack}}{P}$	46752	466819

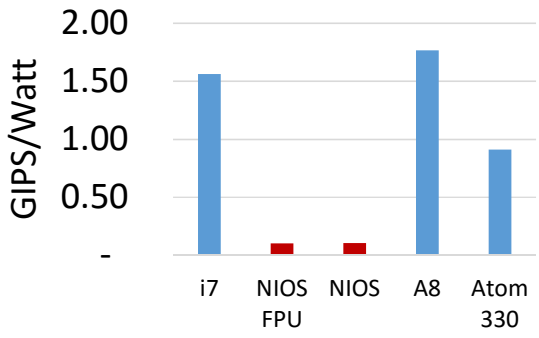


Figure 39 Integer Energy Efficiency

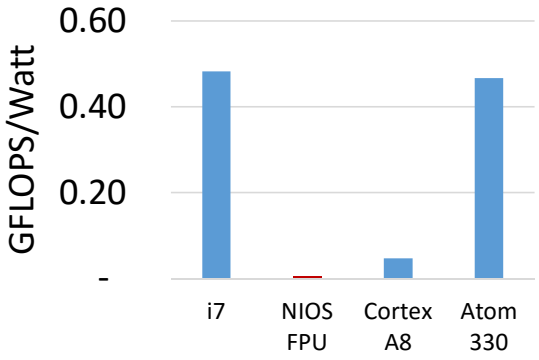


Figure 40 Floating Point Energy Efficiency

Seeing the previous results in integer (Figure 39) and floating point arithmetic (Figure 40) I realized that, with the right architectural choices, the energy efficiency of a soft-core implemented on a FPGA can be just about order of magnitude lower than that of a processor, even when not accessing the last technology nodes. It is clear that the increasing the IPC is crucial to increase the energy efficiency [Czechowski14]. Modern processors have limited architectural choices to be able to increase IPC because they have to maintain the general purpose, and moreover they have to deal with the increasingly important dark silicon problem [Esmaeilzadeh11], which states that, as transistor capacity increase, higher and higher portions of the chip must remain inactive to reduce thermal density.

On the other hand FPGAs can be tailored to application specifics and can implement specialized units to significantly increase the IPC for particular functions. By doing this, both performance and energy efficiency will be increased. But, moreover, coprocessors can be combined with parallel processors to increase even more the performance. Parallelism does not allow increasing energy efficiency per se. But it can indirectly do it applying the following logic: If an application is parallelized by a factor *M* and exhibits a linear scalability, the clock frequency could be reduced by a factor 1/M. Thus, would allow to reduce the voltage supply, which is a major driver to energy efficiency, or relax the synthesis requirements that would reduce the total effective capacitance of the circuit (as will be explained in more detail in 2.7.3).

Nevertheless, this strategy is often out of reach of FPGA designers if using commercial platforms. First, because they would have a fixed voltage supply. Second, because FPGA logic blocks are pre-build, and designers cannot modify the width of the transistors to adapt to a more relaxed requirements.

2.7. Soft-core Multiprocessors

The higher performance and better energy efficiency of FPGAs compared with CPUs and GPGPUs has been already demonstrated by several works (see Table 17). Although FPGAs are not always the best option in terms of performance, they usually become the best energy efficient alternative. A usual argument against FPGA implementations is the complexity in developing complete solutions using HDL languages. High performance applications in CPUs are easily programmed with high level programming languages like C/C++. GPGPUs use languages like CUDA, OpenCL, etc. that are based on C/C++, but introduce some constraints in the code executed by the coprocessor. It is not straightforward for a programmer to exploit the peak performance of GPGPUs because programming style must be very aware of the particularities of the GPGPU architecture, like its SIMD execution style and memory hierarchy. The skills needed for an optimal implementation in FPGA are even higher since designers have the challenge to create a complete new architecture from scratch and map the algorithm effectively on it.

Table 17 Relative performance and energy efficiency of FPGAs/CPUs/GPGPUs

	Relative Performance			Relative Energy Efficiency		
	FPGA	CPU	GPGPU	FPGA	CPU	GPGPU
[Tian09]	1	1/25	1/9	1	1/37	1/35
[Hamada09]	1/2.12	1/9.75	1	1	1/24.5	1/8.75
[Tian10]	1	1/544	1/10.8	1	1/336	1/16
[Kestur10]	1/3.15	9.81	1/3.24	1	1/3.3	1/13.3

To break this programmability barrier, FPGA manufacturers have promoted the adoption of OpenCL as a design entry language to build application specific accelerators that work in cooperation with an external host processor in a similar way than is done with GPGPUs. This approach is an important step forward in programmability, but limits the execution of the created designs to some very specific applications, and relegates the FPGA as a coprocessor of a host system and cannot execute other generic applications.

The superior energy efficiency of FPGA designs, shown in Table 17, is reached by combining deep pipelining, data parallelism, and data locality. They all contribute to a much higher level of effective instructions executed per cycle, which is the main driver of energy efficiency. As demonstrated in previous section, soft-core processors do not

have a superior energy efficiency level than hardened standard processors, but they have the flexibility to be extended. It is possible to easily attach soft-core coprocessors having the properties that make FPGAs so efficient: deep pipelines, and local memories.

The evolution on logic density of the last decade mimics the experienced by processors, which is the self-fulfilling prophecy described in Moore's law. But the availability of that huge number of reconfigurable logic elements puts pressure to design teams, who have challenge to take profit from all of them. There is a general feeling that the rate at which new resources are available is usually higher than the rate at what designers make use of them. This illusion was widespread in the microelectronic design community that called it the "design productivity gap". Recent studies [Foster13] suggest it did not happen thanks to the increase of IP reuse and the emerging of bus standards. In any case, replicating larger IP blocks, such as processors, is an easy way to close the gap.

Soft-core processors could be easily replicated to build multiprocessing architectures to exploit data parallelism. This could provide the energy efficiency that FPGAs have already proven to have, but with a simpler programming model based on parallel programming standards complemented with coprocessor invocation. With this approach they would still provide the flexibility of standard processors to allow their reuse for a wider range of applications.

The idea of replicating soft-core processors to collaborate in order to increase system performance appeared almost simultaneously as the introduction of the soft-cores themselves. At the beginning, FPGA devices had limited resources, allowing only the instantiation of just few soft-core processors. As mentioned in chapter 1, soft-core processors can be implemented with few thousands of look-up-tables and flip-flops.

2.7.1. Implementing devices

As FPGAs were implemented in new technology nodes, they increased its logic density and allowed the instantiation of larger numbers of soft-cores. Table 18 lists the FPGAs devices used to implement reconfigurable multiprocessing systems. The first collected device is a Xilinx XCV1000 FPGA, which was implemented in a 0.22µm node, and offered 22 K LUT/FF pairs. This early device was already able to host an 8-core multiprocessor system, as detailed in [Martina02]. Recently I have proven how an Altera EP4SGX530 device, which is implemented in a 40nm node, and is not the biggest device from Altera, can embed a 128-core multiprocessor system [Castells15].

Table 18 Multiprocessing systems based on reconfigurable logic.

Device	Node	Voltage	K LUTs	K FFs	Mem (Mb)	References
XCV1000	220	2.5	24	24	0.52	[Martina02]
XCV1000E	180	1.8	24	24	0.78	[Li03]
XC2V3000	150	1.5	28	28	2.16	[Hubner05]
XC2V6000	150	1.5	67	67	3.64	[Huerta05]
EP1S40	130	1.5	41	41	3.42	[Hung05] [Lehtoranta05] [Salminen05] [Kulmala06]
XC2VP20	130	1.5	20	20	1.8	[Freitas07]
XC2VP30	130	1.5	30	30	2.8	[Dykes07] [Tumeo07] [Tumeo10]
XC2VP50	130	1.5	53	53	4.91	[Jin05] [Ravindran09]
XC2VP70	130	1.5	74	74	6.93	[Krasnov07]
XC4VFX12	90	1.2	10	10	0.73	[Huerta07]
XC4VFX20	90	1.2	17	17	1.35	[Kornaros10]
XC4VFX140	90	1.2	142	142	10.92	[Wang08]
EP2C35	90	1.2	33	33	0.4	[Pitter08] [Bao09]
EP2C70	90	1.2	68	68	1.15	[Lee09]
EP2S60	90	1.2	60	60	2.54	[Khan09]
EP2S180	90	1.2	143	143	9.38	[Yan09] [Fernandez10] [Castells11]
XC5VLX110T	65	1.0	69	69	6.45	[Giefers10] [Kiefer15]
XC5VLX155T	65	1.0	97	97	9.27	[Lebedev10]
XC5VFX130T	65	1.0	81	81	12.31	[Chen11] [Jing13]
XC5VLX330	65	1.0	207	207	10.36	[Wang10]
XC5VLX330T	65	1.0	207	207	15.08	[Mplemenos08]
EP3C40	65	1.2	39	39	1.1	[Han12]
EP4CE22	60	1.0	22	22	0.60	[Castells12]
EP4CGX150	60	1.0	149	149	6.48	[Rashtchi14]
XC6VLX240T	40	1.0	150	301	18.60	[Stevens12] [Kondo13] [Plumbridge13] [Plumbridge14]
EP4SGX530	40	0.9	531	531	20.74	[Choi13] [Castells15]
XC7K325T	28	0.9	203	402	20.02	[Raza14]
XCZ7020	28	0.9	53	106	4.48	[Vestias14] [Jose15]
5SGXEA7	28	0.85	622	622	50.00	[Baklouti14] [Podobas14]

2.7.2. Type of replicated Soft-Cores

Some works have already surveyed how soft-core processors can be replicated to build reconfigurable multiprocessors [Dorta10],[Göhringer14]. The multi-soft-core processors and many-soft-core processors in the literature that are analyzed in this thesis listed in Table 19. A large number of research and industrial works use the soft-cores provided by the main FPGA vendors: Microblaze from Xilinx, and NIOS from Altera. Researchers usually embrace them because they provide a mature infrastructure that allows building complex systems using a standard toolchain to program it. Although I personally do not consider this toolchain mature enough for multiprocessing purposes, it clearly offers an advantage over alternative soft-cores.

Microblaze and NIOS have a classic 32 bit RISC architecture. They have a short pipeline, branch prediction, and Harvard architecture. As we could expect the maximum IPC they can reach is close to 1, but they offer a mechanism to surpass this value with custom instructions and coprocessors. An effective way to increase the IPC is including multiply-accumulate units (MAC), as done in [Martina02]. MACs are regularly used in digital signal processors (DSPs) and the typical signal processing algorithms (like FIR filters) can easily take profit of them. Since MACs combine a multiply and add operation they should tend to double the IPC.

Table 19 Properties of the soft-core processors used in the analyzed literature. The last entry does not show any property as it groups the works that use application specific architectures, in which cores are synthesized to address the needs of each different application.

Soft-Core	Instruction Width Data Width	Pipeline Branch Prediction	IPC	FPU	Instruction Stream Data Stream	LUTs FFs
			References			
Microblaze	32 32	5 Opt.	1	Opt.	I-Cache D-Cache	1290 1290
	[Hubner05] [Huerta05] [Jin05] [Ravindran09] [Dykes07] [Freitas07] [Huerta07] [Krasnov07] [Tumeo07] [Mplemenos08] [Wang08] [Giefers10] [Kornaros10] [Tumeo10] [Chen11] [Jing13] [Plumbridge13] [Plumbridge14]					
NIOS	32 32	6 Opt	1	Opt.	I-Cache D-Cache	1700 1700
	[Hung05] [Lehtoranta05] [Salminen05] [Kulmala06] [Khan09] [Yan09] [Lee09] [Bao09] [Fernandez10] [Castells11] [Castells12] [Han12] [Rashtchi14] [Baklouti14] [Castells15]					
Prop. DSP [Martina02]	24 16 [Martina02]	5 No	1.9	No	ScratchPad ScratchPad	3024 1618
SIMD PIC [Li03]	16 8 [Li03]	3 No	31	No	ScratchPad ScratchPad	18420 18420
JOP [Schoeberl03]	32 32 [Pitter08]	4 No	<1	No	I-Cache D-Cache	2100 2100
LE1 [Stevens10]	32·n 32 [Stevens12]	8 Yes	n	Opt.	I-Cache Local	5648 2032
Geyser [Seki08]	32 32 [Kondo13]	5 ?	1?	No?	I-Cache D-Cache	10942 7083
LWP [Raza14]	32 32 [Raza14]	5 ?	<1	No	ScratchPad ScratchPad	249 401
OpenRISC derivative	32 32 [Kiefer15]	3 ?	0.5	No	Shared Cache Shared Cache	3300 3300
App. Spe.	- -	- -	- -		- -	- -
	[Wang10] [Choi13] [Podobas14] [Lebedev10] [Vestias14] [Jose15]					

Vector processing is not commonly used in FPGAs, although there are several remarkable examples of them. For instance [Li03] uses 95 vector units that achieve an IPC level of approximately 31, [Yiannacouras08] uses 16 vector units achieving an approximately IPC level of 6. Although it initially can be though as a great idea to increase IPC, there are few codes that exhibit the regularity needed to take profit of the peak performance offered by a vector architecture.

The same problem is found with VLIW architectures, since they usually suffer from the limited ILP of the code to execute. [Jones04] studies some benchmarks considering an infinite number of executing units finding that the effective IPC is still below 2. Similar results can be found in [Stevens10] where the speedups coming from increasing the issue width of the VLIW processor are modest, and its best speedup is obtained by using specialized coprocessors and replicating cores.

Most of the systems analyzed in Table 19 use a Harvard architecture. In some cases caches are used and in other cases scratchpad memories are preferred. Scratchpad memories are simpler and faster than caches but limit the generality of the systems because the number of applications that can fit in them is severely reduced. On the other hand, they provide a perfectly predictable access time, which is usually a good feature to estimate WCET in hard real-time systems.

2.7.3. System Frequency

It is important to stress that frequency has a complex relation with energy efficiency. When synthesizing a circuit, system designers usually specify some frequency requirements. If the requirements are very easy to meet the synthesis tool can focus on optimizing the area and power consumption for the design, producing a smaller netlist with less transistors (N), and probably smaller transistors as well, decreasing their capacitance (C). On the other hand, if the frequency goal is hard to achieve, the effort will probably be spent in creating redundant and larger transistors. Frequency will have a negative impact during circuit synthesis because higher frequencies will produce larger N and C which are negatively correlated with G in equation (1.17). As detailed by [Aitken14] this effect is even exacerbated with new technology nodes. This is shown in Figure 41 where energy efficiency is presented in terms of MHz/mW when the same design is synthesized targeting a given target frequency. It is also interesting to realize how new nodes are particularly energy efficient working at low frequencies.

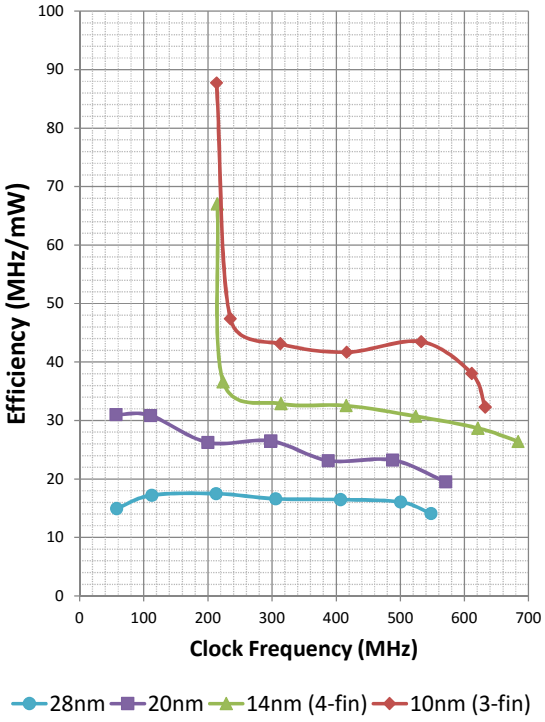


Figure 41 Efficiency measured in MHz/mW as a function of the target frequency in multiple synthesis of the same processor design. Image extracted from [Aitken14]

On the other hand, it is obvious that performance is linearly dependent on frequency (f_{clk}), but, once the circuit is synthesized, it is not so obvious that G also has a positive correlation with f_{clk}. The answer is also in the equation (1.17), and should be understood by the following reasoning. Performance is linear with frequency, and dynamic power as well. Since they appear in the numerator and denominator of (1.16) they should cancel themselves, but this has ignored static power. If we take it into account, we see that static power must be integrated during the execution of the task. The longer it takes the execution, the more power will be integrated. So higher frequencies will cause shorter integration times of the static power, thus reducing the energy efficiency.

Given this dual nature of the relation between f_{clk} and energy efficiency, I analyze the clock frequency values reported in the literature (see Table 20). Most designs use frequencies below 150 MHz. Since FPGAs use predefined logic blocks, synthesis tools do not have the same degrees of freedom like in full custom ASIC design. This would explain why frequency has not evolved much during the last decade. However, latest advances seem to begin to allow a jump on attainable maximum clock frequencies, for instance [Baklouti14] reaches the 328 MHz with a recent 5SGXEA7 Stratix V device, and Altera promises to double the attainable frequency in their new Stratix 10 devices.

Table 20 Main system's clock frequency

Freq Range (Mhz)	Reference
?	[Mplemenos08] [Wang08] [Yan09] [Lee09] [Chen11] [Jin05]
< 50	[Kondo13] [Vestias14] [Kiefer15]
50-74	[Li03] [Huerta05] [Hung05] [Hubner05] [Lehtoranta05] [Kulmala06] [Tumeo07] [Bao09] [Fernandez10] [Tumeo10] [Castells11] [Castells12] [Podobas14] [Castells15]
75-99	[Salminen05] [Choi13] [Rashtchi14] [Han12] [Pitter08] [Krasnov07]
100-149	[Ravindran09] [Dykes07] [Huerta07] [Khan09] [Stevens12] [Plumbridge13] [Plumbridge14] [Giefers10] [Kornaros10] [Wang10] [Freitas07] [Martina02]
150-250	[Lebedev10] [Jing13] [Jose15]
>250	[Raza14] [Baklouti14]

2.7.4. Number of processors replicated

The number of soft-cores that a design can embed depends on the resource usage of the soft-core (see Table 19) and the available resources of the hosting FPGA (see Table 18). Table 21 shows the number of processors of the systems reported in the literature.

Table 21 Number of sof-core processors per design

#Cores	Reference
<6	[Hubner05] [Lehtoranta05] [Kulmala06] [Freitas07] [Huerta07] [Tumeo07] [Dykes07] [Yan09] [Lee09] [Bao09] [Kornaros10] [Castells12] [Chen11] [Han12] [Jing13] [Raza14]
8-10	[Martina02] [Huerta05] [Hung05] [Salminen05] [Pitter08] [Wang10] [Tumeo10] [Stevens12] [Kondo13] [Kiefer15] [Plumbridge13]
11-20	[Jin05] [Krasnov07] [Ravindran09] [Fernandez10] [Castells11] [Plumbridge14] [Rashtchi14] [Jose15]
23-50	[Wang08] [Khan09] [Giefers10] [Lebedev10] [Vestias14] [Baklouti14]
51-100	[Mplemenos08] [Li03]
>100	[Choi13] [Podobas14] [Ben14] [Castells15]

As could be expected systems with a larger number of cores can be built as devices increase their logic density. An exception is [Li03], but the reason for that is that it uses a vector architecture with 95 small arithmetic units that I have classified as being "cores". It is worth mentioning that my 128-core system is the system with a higher number of standard 32 bits cores on the list. [Ben14] replicates 1024 modified PacoBlaze processors.

2.7.5. Application domains

The applications found in the literature are in many different domains (see Table 22). From pure number crunching to biomedical or industrial applications.

Table 22 Application domains of the multiprocessing soft-core designs in the literature

Domain	Reference
Number	[Huerta05] [Pitter08] [Fernandez10] [Castells11] [Vestias14] [Baklouti14] [Jose15]
Crunching	[Martina02] [Lee09] [Kondo13] [Krasnov07] [Pitter08] [Vestias14] [Podobas14] [Castells15] [Kiefer15] [Giefers10] [Lebedev10] [Choi13]
Multimedia	[Lehtoranta05] [Kulmala06] [Tumeo07] [Yan09] [Khan09] [Fernandez10] [Tumeo10] [Kornaros10] [Jing13]
Cryprography	[Li03] [Huerta05] [Plumbridge13] [Plumbridge14] [Raza14]
Industrial	[Dykes07] [Castells12] [Pitter08]
BioMedical	[Mplemenos08] [Kornaros10] [Stevens12]
System Testing	[Hung05] [Huerta07] [Salminen05]
Automotive	[Hubner05] [Khan09]
Communications	[Jin05] [Ravindran09] [Rashtchi14]
Database	[Wang10]
Space	[Han12]
Militar	[Chen11]
None	[Freitas07] [Wang08] [Bao09]

2.7.6. Energy Efficiency

Surprisingly the authors tend to stress the performance achieved by multiprocessing reconfigurable systems without much analysis of their energy efficiency. This is surprising because, generally (as shown in Table 17), the best benefit from using FPGAs is their energy efficiency, not only their performance.

In fact, there are few works that report power consumption at all (see Table 23). In the reported works energy efficiency comes from different sources. In [Martina02] a relatively high number is the product of working with 16 bits datapath, a MAC unit that allows to reach 1.9 IPC, and 8 cores. One could argue that a 16 bit datapath is not comparable to a 32 bit datapath, but this is essentially the good point of FPGAs, that we can change the architecture to fit our application requirements. So, reducing the number of bits of the datapath is a perfect valid option. In a similar way, other proposals suggest to implement a subset of the ISA depending on the application to run [Yiannacouras06]. In [Castells12] the energy efficiency level is reached by replicating 4 cores and implementing custom instructions to reach the IPC to 3 while working with a low frequency. On the other hand [Raza14] replicates very simple processors combined with coprocessors, which allow an IPC close to 1, with a high frequency resulting on the highest observed energy efficiency value. Finally I present a system replicating 128 cores

with a modest frequency and a relatively high power consumption. Although performance is higher than previous systems, energy efficiency is lower because I do not use any method to increase the IPC of individual cores.

Table 23 Energy efficiency of multiprocessing reconfigurable systems found in the literature

Reference	Device	Power (Watts)	IPC	Cores	Freq (Mhz)	GOPS	GOPS/Watt
[Martina02]	XCV1000	0.231	1.9	8	89	1.35	5.8
[Castells12]	EP4CE22	0.360	3	4	50	0.60	1.6
[Raza14]	XC7K325T	0.102	1	3	289	0.86	8.5
[Castells15]	EP4SGX530	4.500	1	128	50	6.40	1.4

After the analysis of previous results, it is apparent that there is still the need for more research to analyze the energy efficiency levels that many-soft-cores should be able to provide, by combining parallel architectures and custom logic. On the following chapters I will try to address this need, although there still will be much work to be done in the future.

3. Building Many-Soft-Core Processors

The main motivation to build many-soft-core MPSoCs is to achieve a high energy-efficient factor, without scarifying the flexibility and generality of a "mainly" software-based approach. As studied in the previous chapter, the energy efficiency starting point for a soft-core processor is rather low, around 15 times worse than a standard processor. But on the contrary, soft-core processors have an architectural flexibility not present in hard-cores. I am proposing to take profit of this key feature to improve the OPC as much as possible to increase the system performance and energy efficiency factors. Additional performance gains can be reached with parallel architectures.

The resulting systems would include special hardware to be more energy efficient addressing application specific workloads, but they could still be used by general applications.

This chapter will focus in describing how to build the hardware infrastructure needed by many-soft-core MPSoCs.

3.1. Building blocks

The productivity gap was closed by reusing hardware blocks. Thus, I am proposing to reuse big hardware blocks like processors, coprocessors, and all the required infrastructure to make the resulting system easily programmable. Reusable hardware blocks (also called IP blocks, or IP cores) are sometimes pieces of HDL code that can be directly instantiated on an FPGA design, but a lot of times are given as the product of hardware generators, which allow tuning to specify some parameters before the HDL is created.

In the following subsections I will present some of the IP cores I have used in this thesis.

3.1.1. MIOS Soft-Core processor

Altera provides the NIOS processor, a 32 bit RISC processor highly optimized implementation addressing the particularities of the FPGA environment. NIOS is, in fact, created by a generator software that allows to configure a multitude of architectural parameters. I will describe it in depth in the following section. Nevertheless, NIOS source code is not available, and unlike Microblaze, which has open source ISA compatible processors like SecretBlaze [Barthe11], it does not have an open source alternative.

Hence, I decided to create my own NIOS ISA compatible processor for educational purposes, and to have more design freedom to test new architectural ideas. I called my processor “MIOS”.

The instruction set of the NIOS processor contains three types of instructions, Immediate (I), Register (R), and Jump (J). I-Type instructions contain 2 register references, a 16 bit immediate value and an operand (see Table 24). They are usually used to perform an operation between the register specified in *A* and the immediate value specified in *Imm16*. The result is then stored into the register specified in *B,* as shown in (3.1).

$$Reg\ B \leftarrow\ operation(Reg\ A, Imm16) \tag{3.1}$$

Table 24 Bit fields of I-Type instructions

31–27	26–22	21–6	5–0
Reg A	Reg B	Imm16	Opcode

R-Type instructions contain 3 register references (see Table 25). A large number of operations of the NIOS ISA are implemented as R-Type instructions. Since the five bits of the opcode are not enough to encode them all, R-Type instructions use additional 6 bits identified as Opcode Extension to encode them. The rest 5 bits are used optionally used as an immediate value.

$$Reg\ C \leftarrow\ operation(Reg\ A, Reg\ B, [Imm5]) \tag{3.2}$$

Table 25 Bit fields of R-Type instructions

31–27	26–22	21–17	16–11	10–6	5–0
Reg A	Reg B	Reg C	Opcode Ext.	Imm5	Opcode

Finally J-Type instructions use an immediate value of 26 bits to encode memory addresses (Table 26). Only *jmp* and *call* operations use this type of encoding.

Table 26 Bit fields of J-Type instructions

31–6	5–0
Imm26	Opcode

MIOS implements all the NIOS ISA, but have significant differences with the later. MIOS does not have separated buses for instruction and data memory streams (like

in Harvard architectures). A single bus to access memory is used, so it has a Von-Neumann architecture. The processor pipeline has three stages: Fetch, Decode, and Execute (see Figure 42).

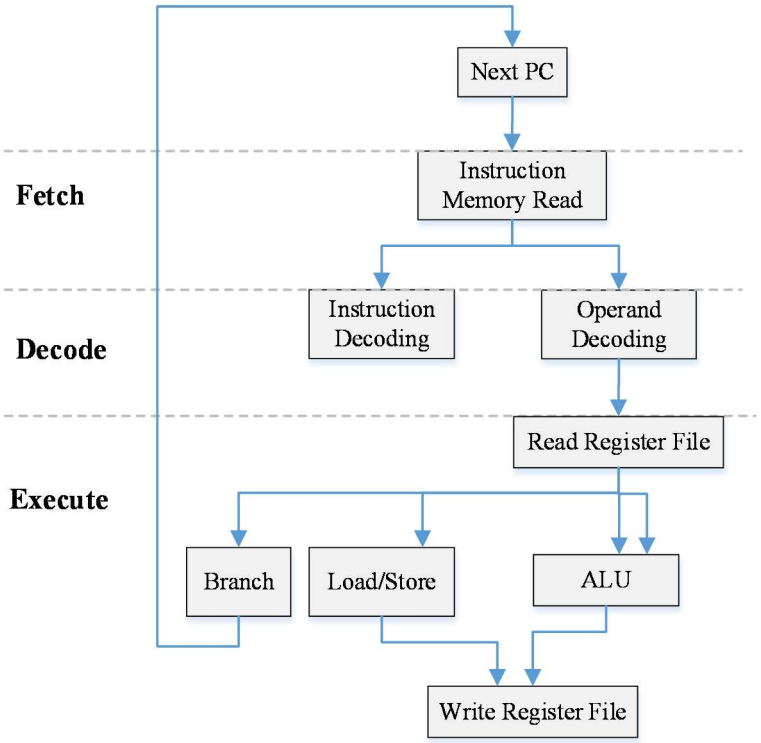


Figure 42 MIOS 3 stage pipeline

The Fetch stage fetches instructions directly from memory, without using any instruction cache. The instructions are fetched every cycle unless a data memory operation is executed in the execution stage. When that happens, all the pipeline is freezed until the data memory operation is completed and a new instruction can be fetched. Since data memory instructions are not executed every cycle the penalty is not that high. MIOS does not use branch prediction.

The Decode stage basically decodes the fetched instruction into the *opcode*, the register references (up to 3 in the R-Type instructions), and the immediate values, either 26, 16, or 5 bit long. The *opcode*, is in fact further decoded to provide a one-hot signal for every instruction of the repertory.

The Execute stage contains all the execution units, i.e. an Arithmetic Logic Unit (ALU), a Branch Unit and a data memory Load/Store Unit. The register file is embedded in the Execute stage, so register operands are fetched during the same cycle when result is written to the target register. With this design, it is no possible to have instructions in

the pipeline that cannot be executed because a target register has not been written yet. Hence, it simplifies the pipeline, avoiding the dependency checking and bypass multiplexors needed in longer pipelines. The register file is implemented with real registers, and not with memory as other implementations. Some arithmetic instructions (like *mult* and *div*) take several cycles to complete. In these cases, like in load/store operations, which also take several cycles to complete, the pipeline is frozen until the results are available, or the memory operation has finished.

MIOS is designed in 5188 lines of Java code using the JHDL framework [Bellows98]. Since it is totally compatible with NIOS ISA, the toolchain based on GNU tools provided by Altera's NIOS can be used to generate executable files for MIOS. The power of JHDL is shown with its verification capabilities. JHDL allows to visualize signal traces and interactively interact with the schematic view of all the circuit hierarchy. In addition, complex test-benches and custom visualizers can be built that extract and display information of the HDL model in the most convenient way. As an example, I created a visualizer of the processor pipeline (see Figure 43). With this visualization, one can easily see how the instructions progress through the processor pipeline. For every pipeline stage the program counter, the instruction in its hexadecimal form and a disassembly version of the instruction is shown. When the memory map of the application is provided, the disassembly output substitutes memory addresses by function names.

When the pipeline is frozen due to an arithmetic, or memory operation, it is signaled in the bottom-left part of the window. When a pipeline stall occurs, the pipeline stages are signaled as invalid and they are shown in red color.

Figure 43 MIOS pipeline visualizer

I provide another visualizer to shown the state of the processor. Thus, allowing an easy debug at the assembly level (see Figure 44). In this window, the value of all the registers of the processor is displayed, as well as the instruction stream being executed. Some buttons interact with the simulation framework to allow to do the typical “Step Into”, “Step Over”, and “Run to Cursor” debugging actions. The application symbol map can be generated with the *objdump* GNU tool and provided to the framework to enhance the visualization of the instruction stream with function names. The call-stack is not shown, but the currently executed function is presented instead.

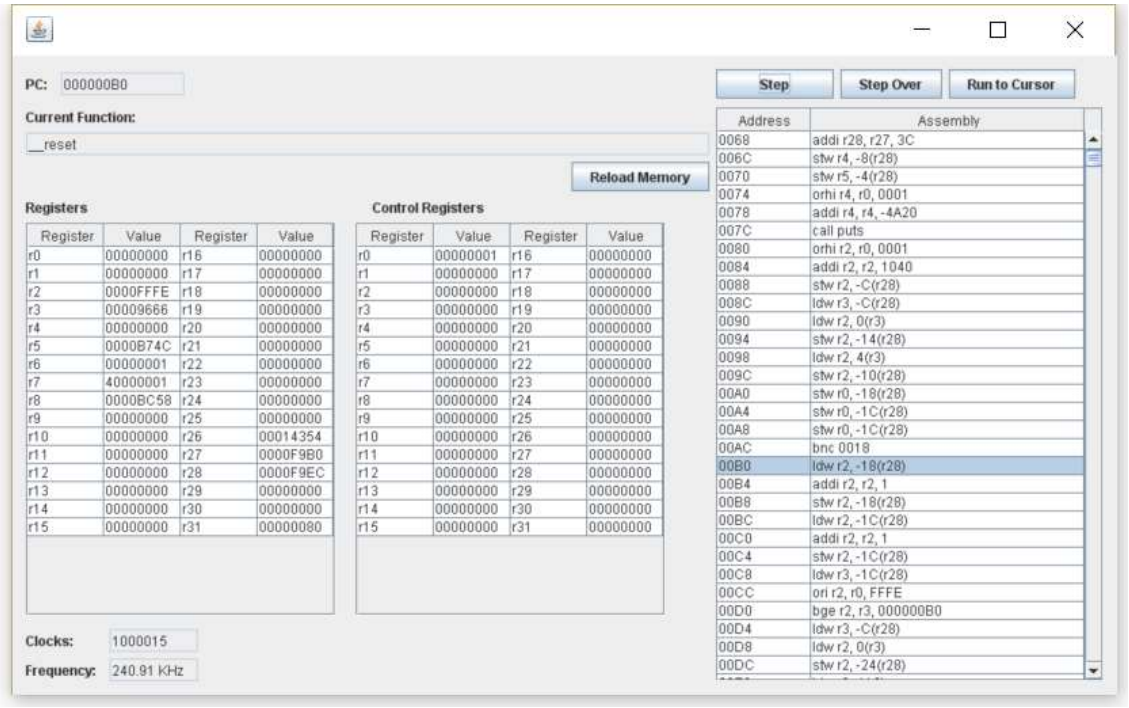


Figure 44 Processor details visualizer

Following the JHDL coding style, the MIOS source code is not a static HDL code, but rather a highly parameterizable generator framework. Instruction subsets can be implemented, as suggested in [Yiannacouras06], and latency of some pipelined operations like *mult* and *div* can be modified.

The performance of MIOS is low because of several factors. First the maximum clock frequency is not very high due to the complexity of the execution stage. Although synthesis results inform that other units could run at 300 MHz on the Altera Cyclone IV FPGA family, the *Execute* stage reduces significantly this number to 60 MHz. On the other hand, the IPC is also low. The simulation of simple application shows that 54% of clock cycles are spent in the frozen state because of the multi-cycle memory and

arithmetic operations. In addition, 13% of bubbles are inserted in the pipeline due to the lack of a branch prediction unit. The resulting effect is that the IPC is 0.33.

Table 27 Synthesis results of MIOS for the Cyclone IV FPGA family

Parameter	Value
LUTs	5890
FFs	3070
f_{max}	60 MHz
P_{dyn}	104.12 mW

Combining the results shown in Table 27 obtained with the Altera tools for synthesis, timing analysis and power estimation, I rewrite (1.16) as (3.3). With this, I compute the peak theoretical efficiency of MIOS processor in a Cyclone IV device considering that all the necessary circuits and peripherals that would be needed in a real scenario consume 0 Watts. Although this is unrealistic scenario, by isolating the processor element, I will be later able to compare its relative efficiency against the Altera NIOS processor. The efficiency factor of MIOS is 0.19 GOPS/Watt.

$$G_{MIOS} = \frac{0.33 \cdot 0.060\ GOPS}{0.104\ Watts} = 0.19\ GOPS/Watt \tag{3.3}$$

3.1.2. NIOSII Soft-Core processor

As already mentioned, Altera's soft-core offering is NIOSII. Like MIOS, NIOS is not a monolithic piece of HDL code, but the product of a highly configurable generation process. All possible NIOSII instantiations have in common a Harvard architecture and a subset of the implemented instruction set. So, in fact, NIOSII already implements ISA subsetting. The most "economical" version of the processor is NIOSII/e does not support some instructions, such as *mult,* and *div*. They must be emulated by software.

The parameter selection to control the processor generation process is done in the QSys tool, an Altera tool included in their Quartus II IDE that superseded the former SOPC tool. The tool lets the designer choose values for several dozens of parameters. Users can select if caches are used, their size, how to implement various arithmetic operations, the size of the branch prediction table, and many other parameters that affect the operation of the system. In order to have a simple perspective of how efficient NIOS is, I instantiate two minimal systems, by isolating the processor elements in order to measure some of their parameters. First I created a NIOSII/e processor, which do not

support *mult* and *div* operations and does not have any branch prediction logic, and no caches. I assume that NIOSII/e IPC is approximately 0.12 [Altera15b]. This system would identify the lower bound of the performance achievable by the NIOSII processor. Then I created a NIOSII/f processor with instruction and data caches of 2 KB each, with full hardware support for arithmetic operations and branch prediction with 256 entries. Although caches could be bigger and more entries could be used by branch prediction, I use the minimal settings and still assume that the IPC that I will get is close to 1. This would identify somehow the upper bound for performance, and energy efficiency. Then, I synthesized both systems with Altera's Quartus II to measure the maximum frequency with Altera's Time Quest, and to estimate the dynamic power consumption of the processors with Altera's Power Play. For that, I measured power at 60 MHz first, in order to have a quick reference value to compare NIOSII with the power consumption running at MIOS's maximum frequency. The results are shown in Table 28.

Table 28 Synthesis results of NIOSII/e and NIOSII/f for the Cyclone IV FPGA family

Parameter	NIOSII/e	NIOSII/f
LUTs	1466	3114
FFs	841	1858
Memory Bits	10240	44800
DSP Elements	0	8
f_{max}	174 MHz	165 MHz
P_{dyn} at 60 MHz	14.67 mW	35.27 mW
P_{dyn} at f_{max}	41 mW	94.04 mW
IPC	0.12	1
G_{NIOS}	0.512	1.75

It is interesting to see that both NIOS designs are much more efficient than MIOS. The reason for this high efficiency is detailed in [Ball07]. Many optimizations were done to increase the performance and reduce the power consumption of NIOSII. In NIOSII/f some arithmetic functions are implemented as DSP blocks. Not only the multiplication, but also shift and rotate operations. This reduces the number of logic needed by the ALU dramatically. Another optimization is done at the register file. In MIOS, the register file is implemented as a collection of registers. The power consumed by the register file alone in MIOS is 33.38 mW. That is a huge number, considering that NIOSII/e total consumed

power is 14mW. The difference is that NIOSII implements the register file as embedded memory, rather than registers like in MIOS.

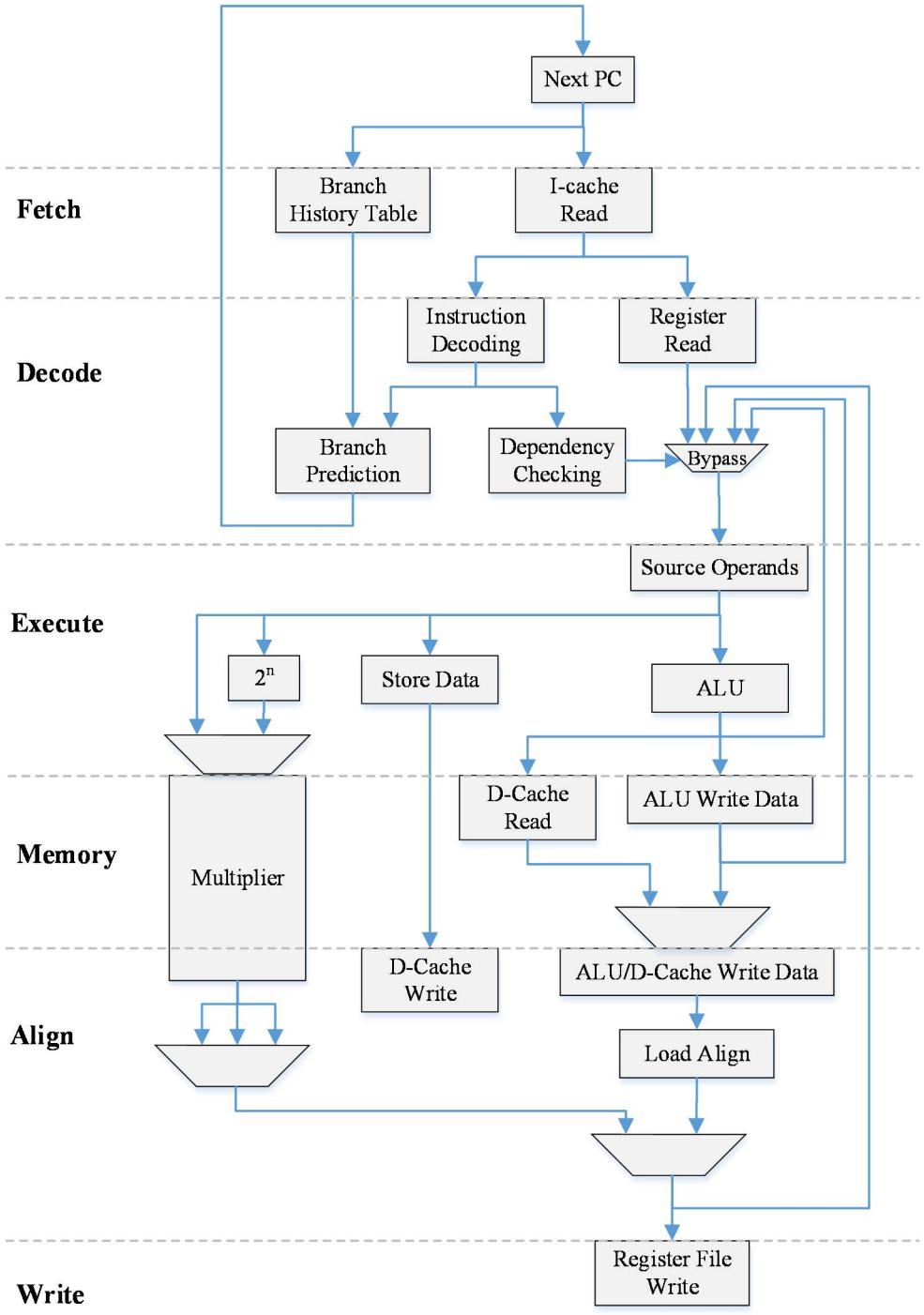


Figure 45 NIOSII/f pipeline

On the other hand, NIOSII/f is a more complex processor than MIOS. It includes cache memories, and supports tightly coupled memories as well. Memory protection and virtual memory are optional through MPU and MMU units. The virtual memory support

implemented in the MMU includes translation look-ahead buffers. Debug support is provided through JTAG.

A high frequency of operation is achieved by using a six stage pipeline (see Figure 45), and the use of the embedded multipliers. To reduce the impact of branches a dynamic branch prediction is used, avoiding the high number of pipeline bubbles seen in MIOS. But as the number of pipeline states is increased there is a probability that an incoming instruction in the decode stage needs to read the value of a register that is targeted by the instructions already executing on the execute stage. Dependency checks and bypass circuits must be included to manage these issues. The result of all those elements result on an IPC close to 1.

Altera proposes two methods to be able to increase the IPC achieved by a NIOS system. Attaching a coprocessor to the Avalon Bus, or implementing a custom-instruction. Custom Instructions (CIs) are a method to implemented extensions to the processor ALU (see Figure 46).

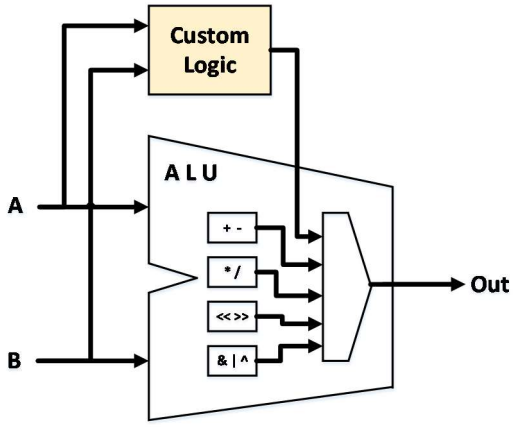


Figure 46 NIOSII Custom Instruction integration

Custom logic implemented as CIs must conform to the interface of an ALU functional unit, i.e. having a maximum of two operands and a result. The instruction to invoke CIs is an R-Type instruction that can use an embedded immediate value of 8 bits to specify the custom instruction to execute. Thus, a NIOSII processor supports up to 256 different custom instructions.

3.1.3. Performance Counter

Measuring the elapsed time of the tasks executed by the system is the main method to estimate the performance of the system. Hence, a good time reference is needed. By good I mean having high resolution and low granularity. Resolution is determined by the units of measure used to express the time value. In most computing systems time

references are expressed as integer numbers. So, the resolution is given by the unit representing this integer value.

Granularity is determined by the minimum distance between two different time measurements. Although time references can be represented with very high resolution, time references can be sampled at different sampling rates. Low sampling rates give coarse grain time references, and higher sampling rates give fine grain time references.

In a digital sequential circuit, such as a processor, the maximum resolution needed for time measurements is given by the frequency of the system clock. There is no point of having better resolution references, because everything happens in multiples of clock cycles.

Altera provides two IP cores that can be used to measure time: *altera_avalon_timer* (Figure 47) and *altera_avalon_performance_counter* (Figure 48).

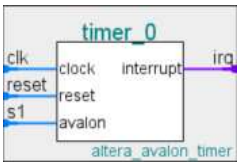


Figure 47 Altera timer IP

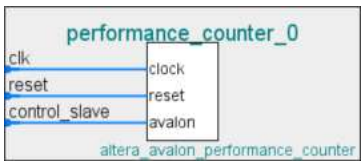


Figure 48 Altera performance counter IP

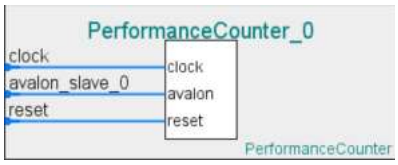


Figure 49 My Performance counter

Timer core is an Avalon Slave IP core that includes a programmable counter. It has several modes of operation, but the usual mode is to program the counter so that it generates an Interrupt when it reaches the specified period and automatically resets to count again. This is used to periodically generate interrupts at a certain frequency. The core is programmed to generate interrupts every 10 ms by default. An ISR routine in the NIOS HAL software layer increments a counter. The C runtime can access this tick counter with the clock function, and can use the CLOCKS_PER_SEC macro to convert the tick count into clocks. This method limits the granularity of the clock because sampling frequency cannot be increased much since a software routine must be executed every clock tick, thus becoming a bottleneck.

The other available core is also an Avalon Slave IP core that implements a 64 bit counter that is automatically increased every clock cycle. Additionally there are several sections that can take snapshots of the counter. This is provided to reduce the number of software cycles needed to process snapshots of the high resolution clock reference. However the number of sections is limited a low number, not being useful for a global method to take time references.

I created a simple performance counter similar to the Altera performance counter, but without any section and without a counter to collect the number of invocation like Altera does.

This results on a resource reduction as shown in Table 29. It is a 64 bits automatically incremented counter. Since the Avalon bus is 32 bits wide, two bus read operations must be performed to obtain the complete counter value.

Table 29 Synthesis results of Altera's performance counter and my performance counter on EP4SGX530

	Altera's Perf. Cnt.	My Perf. Cnt.
LUTs	303	69
FFs	226	64

A 64 bits counter running at a 100 MHz will overrun after 58 centuries of operation. This seems more than enough time period for any need. On the other hand, a 32 bit counter running at 100 MHz will overrun after 43 seconds. Time references are usually taken to compute the elapsed time between two points in time. If we know that the period is below 43 seconds we can work with just 32 bits, avoiding many cycles to fetch the high part of the counter and to compute the difference between two 64 bits numbers.

3.1.4. Floating Point Units

Soft-core processors do not usually include a Floating-Point Unit (FPU), although it can generally be attached to them by using specific extension interfaces. In Altera NIOSII, for instance, it is often done through the CI mechanism [Altera10]. In LEON3, through special floating point unit interface or a generic coprocessor interface (as described in [Gupta04]). In Cortex M1, it can be done through a bus interface [Joven11].

The IEEE standard for binary floating-point Arithmetic (IEEE-754) [Stevenson85] is the most widely-used standard (since 1985) for floating-point computation, and it is followed by many processors, compilers and custom hardware FPU implementations. The standard defines formats for representing floating-point numbers in single and double precision (i.e. including zero and denormal numbers, infinities and *NaNs*) and special values together (such as ±1 or ±0). The number of floating-point operations in a typical application is generally low. However, they usually appear in hot execution paths having an important contribution to application bottlenecks.

Altera FPUs

Altera has two generations of single precision floating-point custom instruction units. The first generation (FPH1, Figure 50) supports four operations, add, sub, mult and div in a multi-cycle custom instruction. Although is the unit that supports less instructions

the LUT and FF resources it uses is the largest compared with the other analyzed units (see Table 30). This could be the result of no use of memory and moderated use of DSP elements. The second generation (FPH2, Figure 51) implements two custom instruction units. One for the instructions that can be implemented as combinational functions, and another for multi-cycle instructions. This allows to better integrate the combinational instructions into the processor pipeline avoiding to freeze the processor. This unit supports 24 instructions, adding comparisons, conversions and square root function. The LUT and FF resource usage is almost ¼ of the FPH1, although it makes extensive use of memory and more DSP resources.

Figure 50 Altera FPH1 IP core

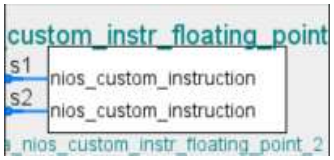


Figure 51 Altera FPH2 IP core

Figure 52 MikeFPU IP core

MikeFPUs

Instead of using Altera floating point units I use MikeFPU [MikeFPU] a simple floating-point custom instruction unit implemented by Michael Schoeggl using Altera's floating point macro-functions (Figure 52). The good point of MikeFPU is that it is simple, its resource needs are moderate and source code is available. Compared with FPH2 it lacks support for max, min, and *sqrt* instructions. Max and min functions can be easily implemented with comparisons, and *sqrt* is not very common in most source codes.

Table 30 Comparison between the synthesis results on EP4SGX530 of Altera's FPH1, FPH2, and MikeFPU floating point units

	FPH1	FPH2	MikeFPU
LUTs	4063	1337	1134
FFs	4033	658	1250
Memory bits	0	15360	4608
DSP elements	4	9	20
instructions	4	24	16
abs		✓	✓
neg		✓	✓
max		✓	
min		✓	
compare		✓	✓
add	✓	✓	✓
sub	✓	✓	✓
mult	✓	✓	✓
div	✓	✓	✓
convert		✓	✓
sqrt		✓	

Shared FPU

The availability of the floating-point unit source code allows the following simple observation. The floating-point custom instruction is divided in N different independent pipelines that will be heavily underutilized. If $L_{pipeline}$ is the mean pipeline length of the functional units, and f_{clk} the system frequency, the theoretical peak performance of the unit would be given by (3.4).

$$FLOPS_{peak} = N \cdot f_{clk} \tag{3.4}$$

But the custom instruction design allows to execute just one instruction simultaneously, so just one unit of N will be active, and just one stage of the pipeline will be effectively used when executing an instruction. Furthermore, floating point instructions are typically just a subset of the instructions that a processor execute. If we let P_{FPins} to be the probability of executing a floating point instruction, the average floating point operations executed will be determined by (3.5).

$$FLOPS_{avg} = \frac{f_{clk} \cdot P_{FPins}}{L_{pipeline}} \tag{3.5}$$

The underutilization factor would be given by the expression 3.5. As an example, if we had an application with $P_{FPins} = 10\%$ using the MikeFPU that has $N = 7$ and $L_{pipeline} = 4.4$ the underutilization factor would be 308.

$$\frac{FLOPS_{avg}}{FLOPS_{peak}} = \frac{P_{FPins}}{N \cdot L_{pipeline}} \tag{3.6}$$

Superscalar processors increase the efficiency of floating point unit utilization by analyzing data dependencies and independently enqueuing operations to functional units so that the pipelines can be better filled. As they also fetch multiple instructions at every cycle, the probability of having floating point instructions is also increased. The result is that they are usually designed to be close to peak performance in floating point intensive code.

I propose to increase its utilization by sharing the same floating point unit among a number of processors. This is not a new idea, shared FPU designs were introduced by several processor manufacturers like IBM [Meltzer99] [Kahle04], AMD [Oberman99], and Texas Instruments [Dao00]. However the limited number of processors able to include in a single chip by that time did not raised the interest to study the scalability of such solution. Kumar's work introduce the topic [Kumar04]. Nowadays the capacity of integration is high enough to justify a deeper analysis of that subject as we try to do here. But many-core IC designs are usually taking advantage of regularity in order to ease large IP blocks reuse. Such a large block could contain a processor with some cache/scratch-pad memory, a FPU and a network routers and lateral connections to allow tiled compositions. Tiled designs are much more convenient than irregular ones since the tile can be replicated as many times as needed with not much extra design effort.

Since many-cores are usually created without a single target application in mind, the optimal sharing factor of FPUs among processors is difficult to predict. A worst case analysis would suggest a low order of sharing, for instance 1 FPU for every 2 or 4 processors, as in previously cited examples.

On the contrary many-soft-cores have two different features that motivate the potential sharing among larger number of cores. First, in FPGAs there are fewer

incentives for tile regularity. Second, the reconfigurability allows designing architectures tailored to specific applications. In this case, for instance, an application that is not intensive in floating-point operations could increase the level of sharing because collisions accessing the FPU would be very unlikely. These observations were already made in the RAMP Blue many-soft-core project, and conveniently described in [Krasnov07]. However, their design introduces some unnecessary overhead because each floating-point operation requires two processor instructions and again not much attention is given to its scalability.

One of my goals is to allow simultaneous access to the different operation units. An access collision will not occur when two processors access the FPU, but when both processors are accessing the very same operational unit of the processor. In the 2 CPUs scenario if a collision happens one CPU will be given immediate access to the functional unit while the other will have to wait to next clock cycle to queue its operands into the pipeline of the functional unit. So the penalty for the waiter will be just one clock cycle, not the whole functional unit latency. If there are more processors competing for the unit they will contribute to increase the latency linearly with the number of competitors. To design a shared floating point unit (as illustrated in Figure 53), the access to functional units must be granted by an arbiter that redirects the operands towards each functional unit by using a crossbar switch. In the absence of collisions n operand sets can be delivered to functional units per cycle. After the functional units another crossbar must redirect the results to the respective processors.

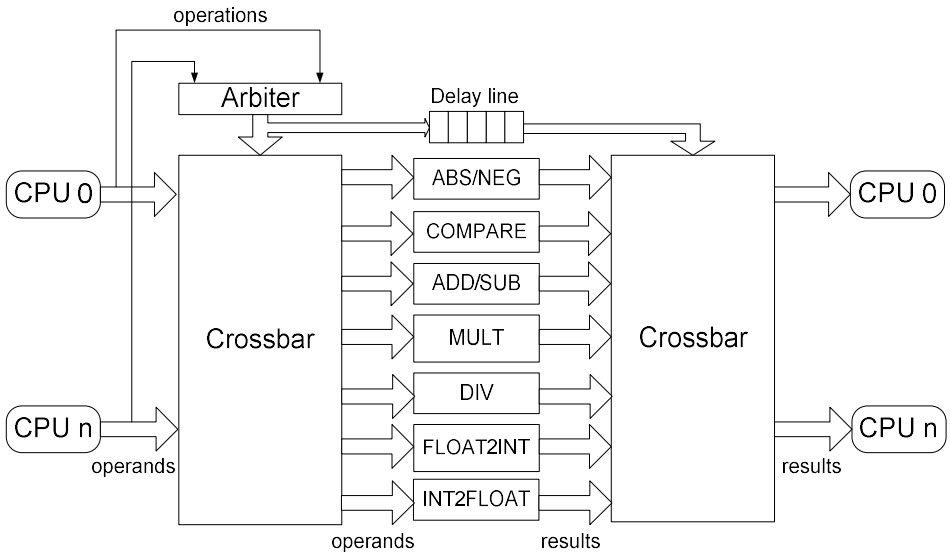


Figure 53 High level design of the shared FPU. Multiple functional units have different latencies. A CPU can only be performing a single FP operation at a certain time, but several CPUs can share the same functional unit.

A parameterizable shared FPU design was created with JHDL and compared with the previous MikeFPU design. The design can be scaled up to be used with any given number of processors. I synthesized the design for an increasing number of processors and compared the resource usage with the resources used when replicating the MikeFPU as many times as processors used.

A more detailed analysis of the resource costs is presented in Table 31. Both, the shared FPU and the Multiple FPU designs have a linear progress with the number of processors, but the former has a much smaller slope than the latter (see Figure 54).

The shared version starts with an important number of resources for 2 processors. In this case, the main contributions are from the floating-point functional units. Obviously, the aim is to share those functional units, when number of processors increase, the additional costs come from bigger arbiters, bigger crossbars and bigger delay lines. It is not that we are having longer delay lines, but that we are saving wider crossbar decision matrices.

Table 31 Detailed resource usage of shared FPU vs. multiple FPUs after synthesis for EP2S15F484C3 device

Processor	**Shared FPU**		**Multiple FPUs**	
Ports	***LUTs***	***FF***	***LUTs***	***FF***
2	3529	1557	6798	2938
4	3657	1701	13596	5876
6	3987	1745	20394	8814
8	4109	1839	27192	11752
10	4213	1933	33990	14690
12	4377	2027	40788	17628
14	4459	2121	47586	20566
16	4590	2215	54384	23504
18	4674	2314	61182	26442
20	4820	2410	67980	29380
22	4993	2496	74778	32318
24	5039	2590	81576	35256
26	5158	2684	88374	38194
28	5228	2778	95172	41132
30	5365	2871	101970	44070
32	5495	2965	108768	47008

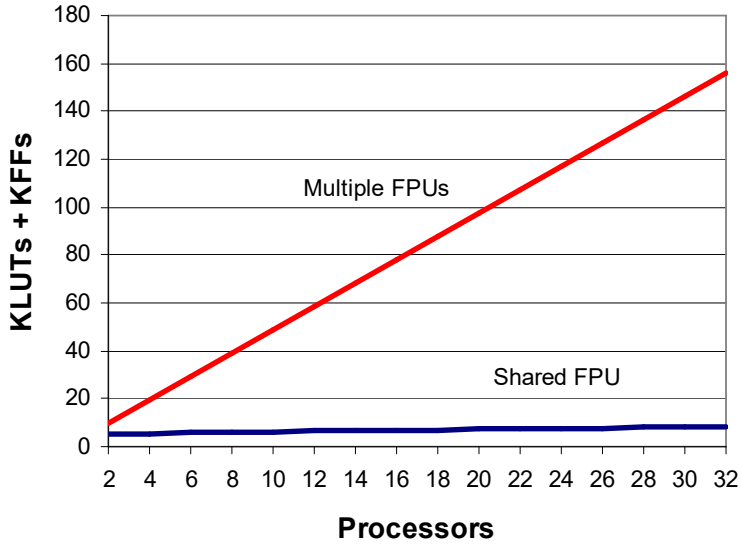


Figure 54 Resource usage (LUTs+FFs) of shared FPU (dark blue) vs. non shared FPU (red) after synthesis for Altera EP2S15F484C3 device

The area savings are extraordinary as the number of processors increase. However, sharing a resource increases the fan-out of some logic cells and increases the length of some combinational logic paths, which obviously have the undesired effect of increasing the signal delay and reducing the maximum frequency at which the circuit can operate. This effect is clearly shown in Figure 55, which depicts how the maximum frequency of operation is reduced as the number of processors keep increasing.

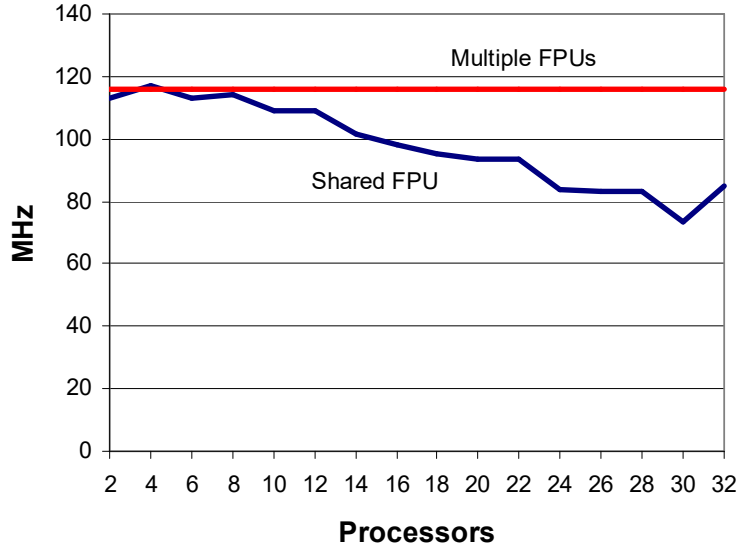


Figure 55 Maximum frequency of operation of the shared FPU (dark blue) vs multiple versions of simple FPU (red) after synthesis for Altera EP2S15F484C3 device

3.1.5. Interconnection Networks

FPGA manufacturers promote to interconnect IP cores using multiplexed buses (see Figure 56). In this case, multiplexed does not mean that addresses and data are shared and multiplexed in time, but it means that the bus is implemented with multiplexors rather than tri-state logic gates. A multiplexed design allows simultaneous transactions in multi-master designs if no collisions occur.

The bus principle relies in the fact that a bus master can place read and write operations to any address of the system memory map. Slaves connected to the bus must have a starting address, and a size span so that masters can correctly address them.

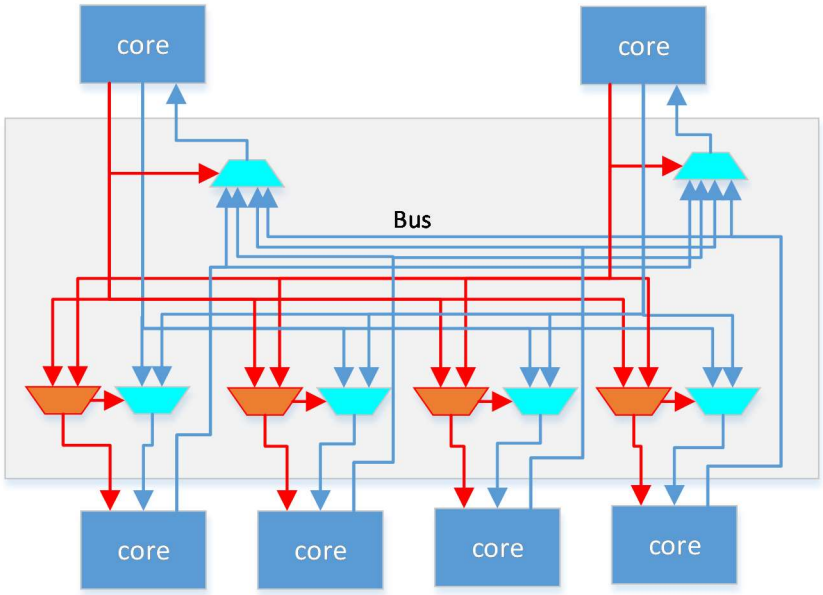


Figure 56 Multiplexed Bus

Altera uses the Avalon bus to interconnect the system elements. Networks on chip (NOCs) were proposed to overcome the scalability limitations of bus-based systems in complex SoCs [Dally01] [Benini02]. NOCs are often more energy efficient than other interconnects, are highly scalable and allow boosting design productivity by reusing large blocks like processors, and network elements. NOCs evolved from all the work previously done by the HPC community in highly parallel architectures [Duato03] [Dally04].

Networks connect IP cores (typically CPU cores) by links and routing circuits. When all routing elements are connected to IP cores we have a Direct Networks. When there exist routing elements in intermediate layers that are not connected to any IP core we have Indirect Networks.

Networks can be classified by their network topologies. The topology is the physical arrangement of routers, links and cores. Some example network topologies are shown in Figure 57-Figure 64.

Direct Network Topologies

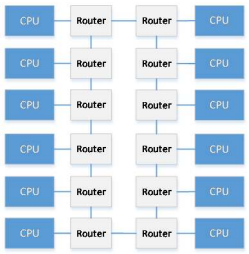


Figure 57 Ring Topology

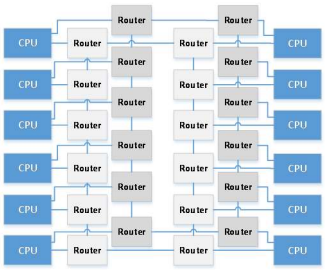


Figure 58 Double Ring Topology

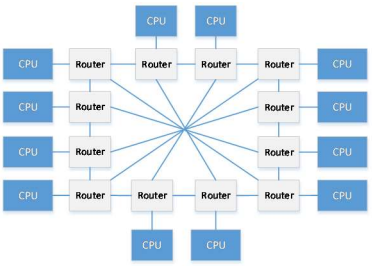


Figure 59 Spidergon Topology

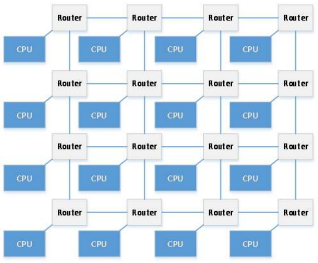


Figure 60 Mesh Topology

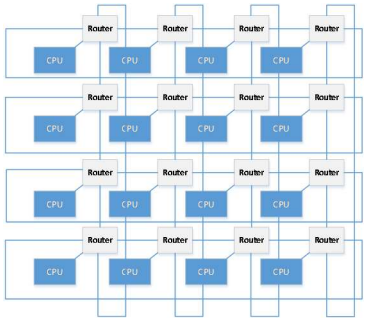


Figure 61 Torus Topology

Indirect Network Topologies

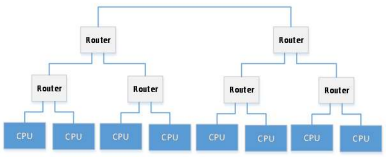


Figure 62 Tree Topology

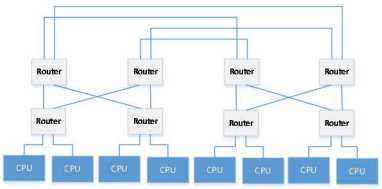


Figure 63 Fat-Tree Topology

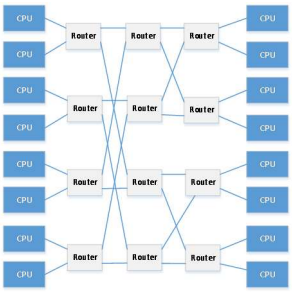


Figure 64 Butterfly Topology

A hop is defined as the number of steps over the network elements that a transmission unit has to perform to go from the transmitting endpoint to the receiving endpoint. Spidergon, and trees topologies focus on reducing the number of hops traversed when transmitting between two any given endpoints. Reducing the number of hops reduces the communication latency.

Other topologies, like mesh and torus, can increase the necessary number of hops to communicate various endpoints, but on the other hand, they are better in terms of regularity, which eases their implementation and eventually allows higher levels of scalability.

Switching strategy is also an important parameter of NOCs. The main classification is between circuit switching and packet switching. In circuit switching a channel is first established by a signaling protocol. When available, the channel is completely devoted to that communication, whether or not there is information to transmit. When no more communication is needed, the channel is tear-down by signaling protocol. Figure 65 illustrates a communication between node N1 to node N4, going through nodes N2 and N3. Green boxes represent signaling information; white boxes the transmitted payload; red patterned bars represent that the channel is busy with another communication; a transparent red patterned bar is used to indicate that another party is requesting to use the channel but it is unable to do it because this ongoing transmission; and finally the black arrows represent the acknowledgement responses from receiving endpoints.

In **Circuit Switching** the channel is blocked by communicating parties, so if another party wants to use it it must wait until channel is tear-down. Circuit switching networks can be implemented with low resource needs, but their blocking nature is a usual cause of inefficiency, often solved by over dimensioning the number of channels.

To avoid blocking channels for a long period of time, packet switching networks divide the information to transmit into packets that are transmitted individually. Different flavors of packet switching differ in how they manage storage in the intermediate network nodes.

In **Store & Forward** (see Figure 66), a packet arriving to an intermediate node is completely stored before forwarding it to the next destination. On the other hand, in **Virtual Cut-Through** (see Figure 67) the packet is forwarded as soon as possible but if channel contention appears packets must be stored like in Store & Forward. The high number of resources needed to store several packets on the routing elements make these methods unpractical for systems on chip.

The alternative solution is the **Wormhole** strategy (Figure 68), which subdivides the packets into smaller flits, which become the unit of transmission. Router storage is dimensioned to be able to store just few flits per channel. If no congestion occurs the

transmitting endpoint injects the packet into the network like in previous schemes. However, if congestion happens the packet is only partially injected, and all the path to reach the congestion point is blocked by the different flits of the packet trying to progress. Although collisions can occur and channels can be blocked for some time due to limited amount of storage in the intermediate network elements, the resource needs of this switching scheme is moderate, and in practice is the most used switching method in NOC design.

In order to reduce the number of resources needed by network routers I proposed another switching strategy, which I named "**Ephemeral Circuit Switching**" [Castells06] (see Figure 69). In this scheme, payload is embedded with signaling. After the circuit is just established it is automatically tear-down because the payload has already been delivered along with the signaling information. For small packets, this is the lowest possible latency switching strategy. Moreover, the sender has feedback of the successful reception of the packet. On the contrary, its drawback is its low throughput.

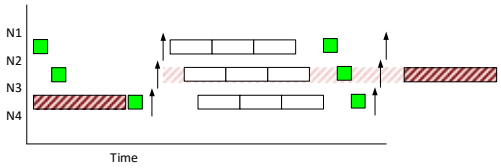


Figure 65 Circuit Switching

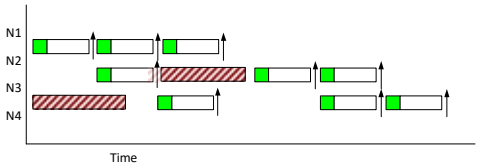


Figure 66 Store & Forward

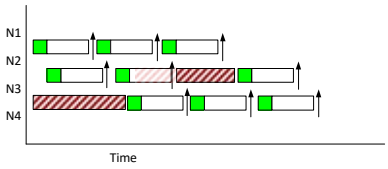


Figure 67 Virtual Cut-Through

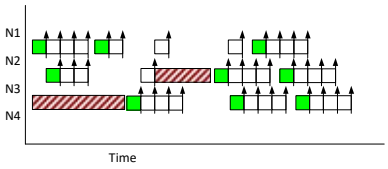


Figure 68 Wormhole Switching

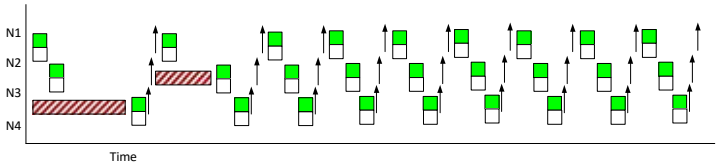


Figure 69 Ephemeral Circuit Switching

Besides topologies and switching schemes, there are a lot of other network parameters, like routing strategies, flow control, packet layout, virtual channels, dead-lock and live-lock avoidance, etc. Moreover, the traffic pattern play a crucial role to determinate whether the NOC is adequate for the system. If traffic pattern is fixed beforehand we could create an optimal network addressing that traffic pattern. However, traffic pattern is impossible to predict in general purpose computing (GPC) systems since

different applications exhibit very different network activity and GPC systems are always ready to host new applications. This is more feasible in embedded systems with few application scenarios. But as application scenarios can be very different, they could led to a large amount of different network designs best adapted to each scenario.

Tools for building NOCs: NocMaker

NOC design tools are not only needed to design few general purpose NOCs but to design a large number of application specific networks. NOC design tools should be able to create the network and determine system metrics but also to validate it, identify design errors and system bottlenecks.

Being a logic circuit, NOCs can be designed by building HDL models and analyzing them. This strategy is followed by tools like xPipes [Bertozzi04], NoCGen [Chan04], xeNoC [Joven08], and others.

The drawback to work with a detailed HDL model is their low simulation speed. Higher abstraction models based on SystemC and TLM design level can offer faster simulation speeds (like in [Talwar08]), but then it is difficult to extract area information, and synthesis is more complex and too much dependent on the quality of synthesis tools.

To address this issue some frameworks (like xPipes) choose to maintain several models at different levels of abstraction, so that simulation is fast and synthesis is more deterministic. But this is non-desirable since maintaining models synchronized is difficult and error prone.

In order to measure the network characteristics it is important that the circuit is stimulated appropriately. Hence, stimulation traffic must be created to go through the network. Statistical traffic pattern generators have been widely used for this (e.g. NoCSim [Jantsch06], Pande [Pande05]) but tend to be very different from traffic patterns observed in real systems. It is more convenient to work with real traffic patterns produced by real processors.

An alternative to inject realistic traffic patterns is to use processor simulation platforms executing applications that inject traffic to the network, like in MPARM [Mahadevan05], or by simulating abstract CPU models like in [Bouchhima05].

To make analysis more complex, traffic characteristics can vary depending on the location arrangements of the computing elements on the network. Tools like SunFloor [Murali06] and Arteris [Lecler11] addresses the need for such mapping phase.

I created my own NOC design tool called NocMaker [Castells09]. The main focus of NocMaker was to create synthesizable designs, and that the tool provided a rich set of verification and validation features not common in HDL design tool-flows.

As described before, the NOC design space is a multidimensional space. In some dimensions only a few discrete values are possible, but other dimensions have a large number of possible values.

NocMaker can actually create a small subset of all the possible designs. NocMaker creates a detailed HDL model of all the network elements to obtain accurate information of the build systems. It can also be seen as a complex HDL generator based on the JHDL framework as depicted in Figure 70. All the generator parameters are passed to the generator as an object instance of the *NocDesignSpacePoint* Java class (NDSP, in short). NDSPs can be serialized as XML files (see code below) or can be created interactively through a Wizard process as shown in Figure 71

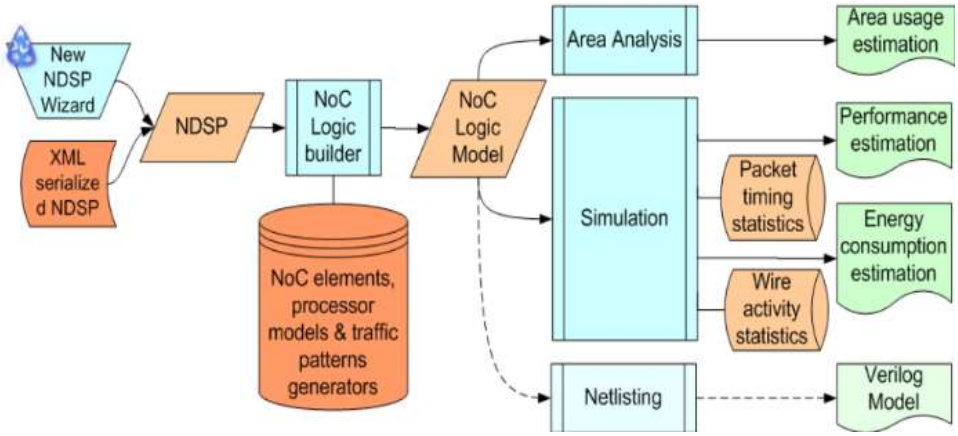


Figure 70 NocMaker process flow used to extract metrics from design space points

```
<org.cephis.nocmaker.model.NoCDesignSpacePoint>
<topology>MESH</topology><removeInputLocalQueue>false</removeInputLocalQueue>
<removeOutputLocalQueue>false</removeOutputLocalQueue>
<queueLength>0</queueLength><routingAlgorithm>XY</routingAlgorithm>
<switchingMode>EPHEMERAL_CIRCUIT_SWITCHING</switchingMode>
<flowControlMethod>FOUR_PHASE_HANDSHAKE</flowControlMethod>
<trafficPattern>PERFECT_SHUFFLE</trafficPattern>
<processorType>ABSTRACT_32_BITS</processorType>
<packetSourceAddressBits>8</packetSourceAddressBits>
<packetDestinationAddressBits>8</packetDestinationAddressBits>
<packetMinPayloadBits>16</packetMinPayloadBits>
<packetMaxPayloadBits>16</packetMaxPayloadBits>
<channelWidth>32</channelWidth><injectionRatio>0.1</injectionRatio>
<injectedBytes>0</injectedBytes><dyadRouting>false</dyadRouting>
<meshWidth>4</meshWidth><meshHeight>4</meshHeight>
</org.cephis.nocmaker.model.NoCDesignSpacePoint>
```

No matter how the NDSP is created, it is used by the building process to create all the necessary JHDL logic elements. The NDSP object is passed among the different elements of the circuit hierarchy as they are instantiated, so that appropriate circuits are created to respond the NDSP requirements. The model, can then be exported to Verilog, analyzed to get early estimations of area consumption, or simulated to get performance and power estimations. Those estimations are often used to find an optimal design (or trade-off) satisfying application requirements.

Figure 71 Some of the NoCMaker Wizard screens to create a simple Mesh Network

JHDL encourages the use of the structural level of design. Designers normally instantiate very simple logic elements like and gates and registers to build large blocks, which can be instantiated again to build even larger blocks, and so on and so forth until the whole system is built. The result of this process is a hierarchy of logic circuits whose leaves are primitive logic elements. The number of these primitive elements is limited

and they originally do not contain any information of area usage. Area usage is usually counted as number of Logic Elements (LEs) for FPGA devices and number of gates for ASICs. NocMaker estimates area usage by assigning a number of logic elements (LEs) or an equivalent gate count to each primitive logic element. Then, the area usage of a NOC design is computed by summing up the resource count of all the leaves of the structural hierarchy (see Figure 72).

This kind of measurement is simplistic and should not be considered an exact prediction of the real area occupancy of the final design because synthesis tools perform resource trimming of unnecessary logic. Moreover, FPGA devices have very different LE architectures. So from vendor to vendor, or even from device to device they can embed a varying amount of logic. Nevertheless, for architectural design space exploration, it is only required that the relations found in early area estimations of circuits match with the relations of their real synthesized versions.

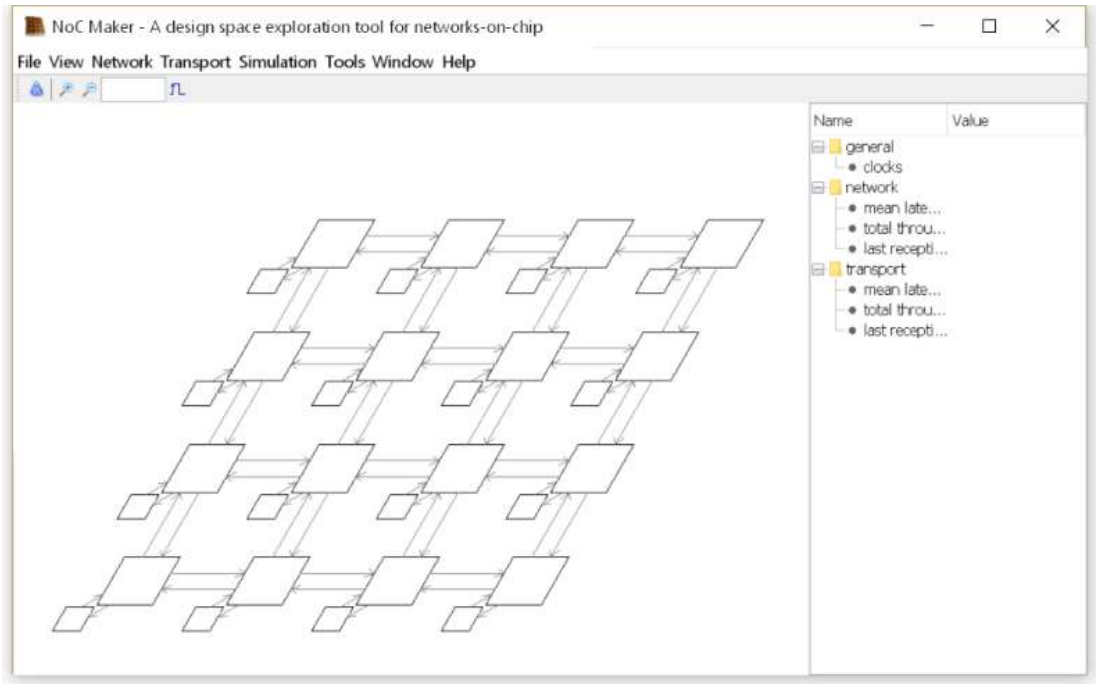


Figure 72 High level view of a 4x4 mesh created in NocMaker (left), and its resource usage estimation (right)

A comparison between the resource estimation and the real resource needs of the different routers of a simple NOC is shown in Figure 73. The results have been normalized to eliminate the effect of the disparity between the absolute value of the estimated and synthesis results. This graph shows that, although resource estimations can

be far from the real observed resource needs, they still maintain the same proportion with a maximum error around 15%, which I consider quite reasonable.

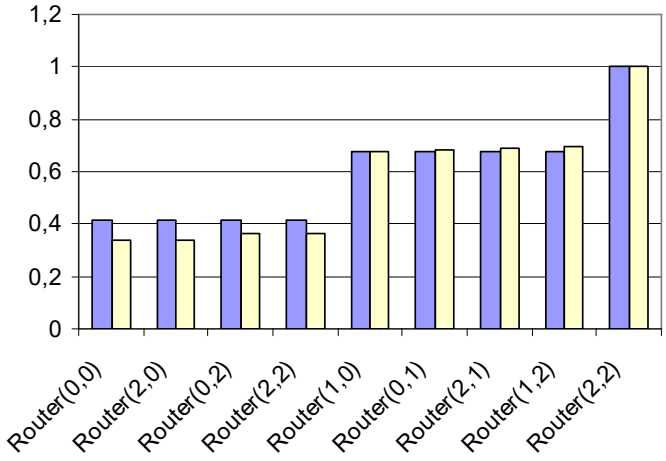


Figure 73 Predicted (blue) and real (yellow) logic elements usage for different routers of an Ephemeral Circuit Switching NoC with a 3x3 mesh topology. The values have been normalized to the largest elements

NOC building parts are mainly Routers and Network Adapters (NAs). These circuits can be fairly simple and easy to independently verify. However when several modules are combined to form a network on chip verification becomes very complex. A simple 4x4 mesh NOC, for instance, has 80 communication links among routers. If each link has 32 data bits and two control bits the number of wires that interconnect the routers is 2720. Analyzing the behavior of such a system at the inter-router level with classical waveform analysis becomes impossible as, after grouping 32 data bits, you still have 240 waveforms to look at. Moreover, the network dynamics is crucial in the appearance of errors. The problems that can arise can involve several flits at several different routers, or could be triggered when a very specific traffic pattern has occurred at a certain region of the NOC. It is mandatory to be able to mix classic HDL verification methods (like waveforms) with high level network analysis tools. NocMaker provides high level visualizations, but I will cover them in the following chapter. On the other hand, NocMaker uses other techniques like hardware assertions and the preexisting JHDL facilities such as the circuit browser, the interactive schematic view (Figure 74), which presents all the values of the circuit wires in real time during simulation.

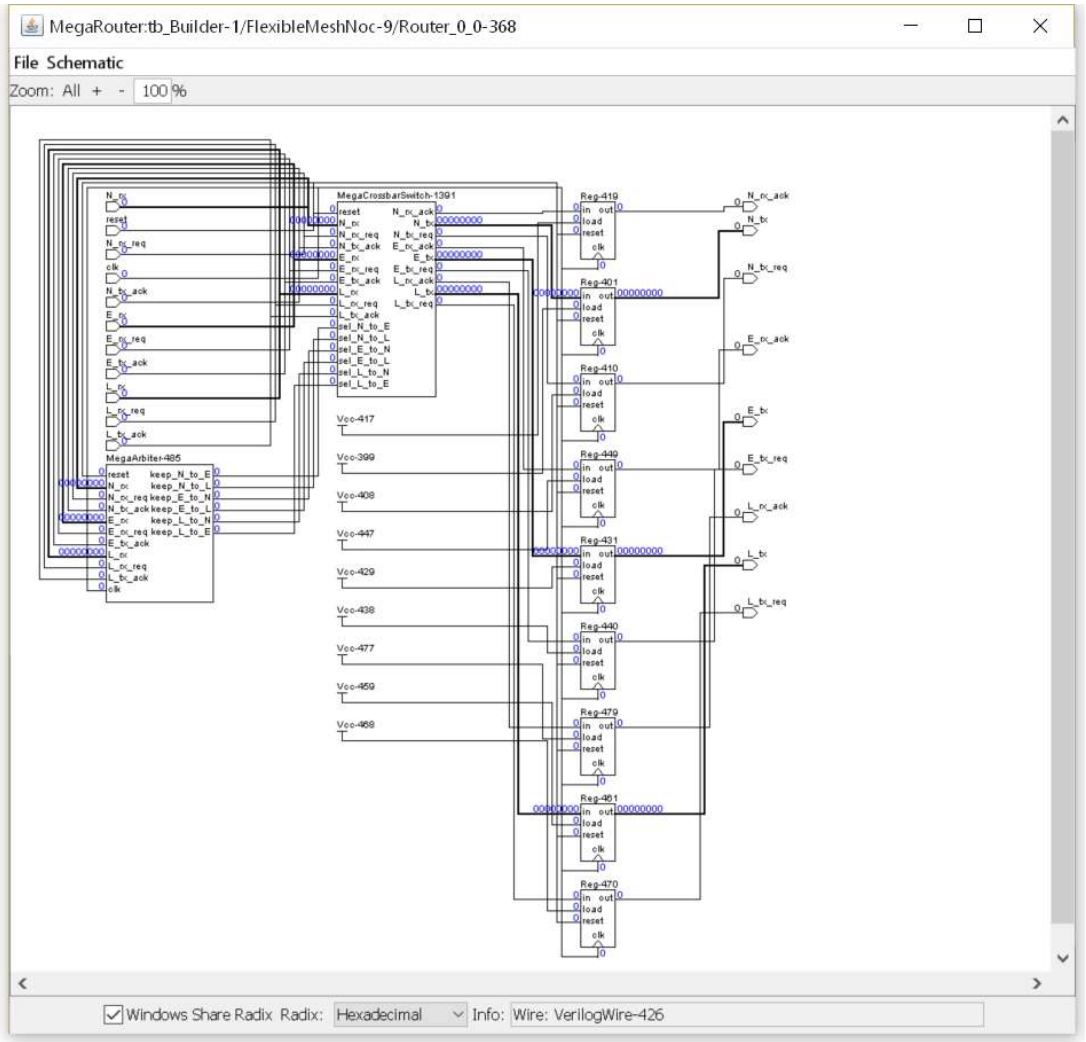


Figure 74 JHDL interactive schematic view of a router. In the schematic the values of each wire is annotated and is updated in realtime as the clock advances.

Combining all these techniques a designer can detect problems at a high level and successively go to deeper levels of detail to identify the final causes and correct them. The resulting design can be automatically exported to Verilog to be later synthesized by FPGA synthesis tool-chains. Several networks have been synthesized using this process for Xilinx and Altera FPGAs. Table 32 shows some example synthesis results of different 4x4 mesh NOCs implemented in NocMaker and synthesized in FPGAs.

Table 32 Resources for different NoCs designed in NocMaker implementing a 4x4 Mesh topology with different switching strategies. Both, area estimation and synthesys results for the EP1S80F1508C5 device are reported.

Design	**NocMaker** *estimated resources (LUTs+FFs)*	**FPGA**		
		LUTs	***FFs***	***Total resources***
Wormhole Switching	31760	16592	6640	23232
Ephemeral Circuit Switching	8020	5488	2468	7956

3.1.6. Network Adapters

Network adapters are used to connect an IP core (like a processor) to a network. Obviously network adapters must comply with the protocol used by the network. NocMaker, already creates the network adapters to interface Microblaze and NIOS soft-core processors with the created NOCs.

NOCs can be designed to avoid dead-locks [Duato93], but sometimes dead-locks can be introduced by message dependencies on the higher layers of the communication stack as described in [Murali06b]. A usual technique to avoid message-dependent deadlocks is to use multiple networks, such as used in [Volos12]. Network adapters must play an important role, especially if they have to make the use of several networks transparent to higher layers. In fact, a detail that is not sufficiently stressed from my point of view in NOC research is the possible roles of the network adapter in the communication stack.

In many shared memory CMPs (like [Kwon15]) the NOC is part of the memory hierarchy, and memory read and write transactions are encapsulated over the underlying transport and network layers of the communication stack (see Figure 76). NOCs usually implement up to the network layer of the ISO stack. In distributed memory architectures the network adapter usually implements the adaptation between a transport layer implemented in software and the network layer provided by the NOC (like in [Matilainen11]). It is worth to mention that any given architecture favors one programming model or the other, but it does not completely determine it. In fact, a shared memory programming model is possible on a distributed memory machine (as described in [Basumallik07]), and a message passing programming model is possible on a shared memory machine. Actually, most Message Passing Interface (MPI) implementations handle intra-node communications through shared memory (as described in [Tang00]).

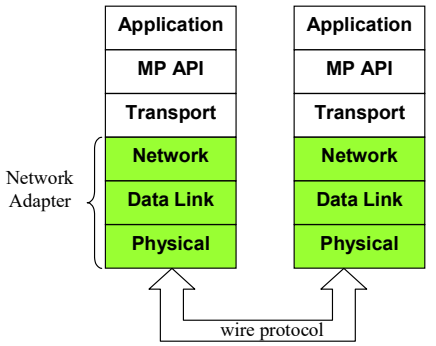


Figure 75 Standard NOC communication protocol stack

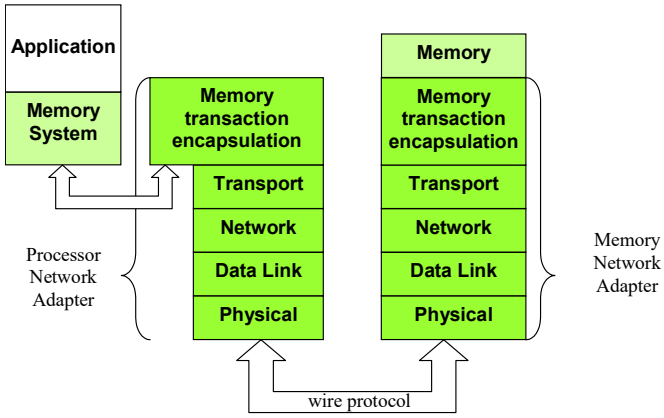


Figure 76 Memory transactions encapsulation over NOC communication protocol stack

Network adapters can be even more complex if the communication system provides services of higher level layers. In [Fernandez14] the NOC and the adapters provide support for MPI primitives. In [Joven13] the network and the adapters provide support for QoS.

As shown in Figure 77 , in memory transactions over NOC encapsulation network adapters must be coupled transparently in the memory hierarchy, as part of the cache structure, or serving as a bus bridge. In any case, the programmer is not necessary aware of the existence of the NOC as bus transactions are transparently translated to packet communications. On the other hand, in non-transparent NOCs, networking primitives must be explicitly invoked by programmers, either directly or through intermediate system libraries such as message passing APIs. In this case network adapters can be coupled to processors either as core connected to the processor bus (as shown in Figure 78) or as part of the processor functional units (as shown in Figure 79).

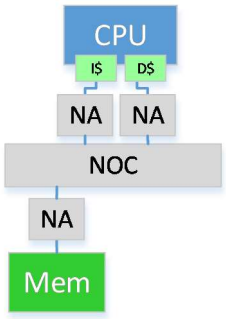


Figure 77 NAs in memory transaction encapsulation over NOC

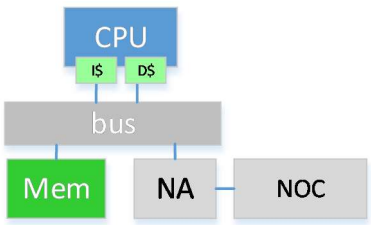


Figure 78 Bus attached NAs

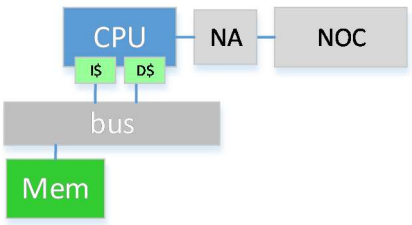


Figure 79 Tightly coupled NAs

In NIOS tighter coupled NAs can be implemented by using custom instructions. In Microblaze a tight coupling can be achieved by using FSL interfaces. The benefit of a tighter coupling is a latency reduction.

Latency measures the time between message generation and message arrival. Point to point latency depends on the distance between the communicating endpoints, often measured as number of hops, but also on the architecture of the different NOC modules. It is interesting to understand how latency and throughput influence the time to complete the running application. Although total network throughput if often the most considered metric, latency can be the dominant factor that determines performance for a number of application. For instance, in a request/response transaction, if the packet is short, the round trip delay of the transaction is dominated by latency. A big latency causes a big round trip delay whereas a low bandwidth does no deteriorate it significantly, as shown in Figure 80. In the case b of this example, although a higher bandwidth allows a lower injection time ($IT_b < IT_a$) the higher latency ($L_b > L_a$) dominates, thus producing a bigger roundtrip delay ($RT_b > RT_a$).

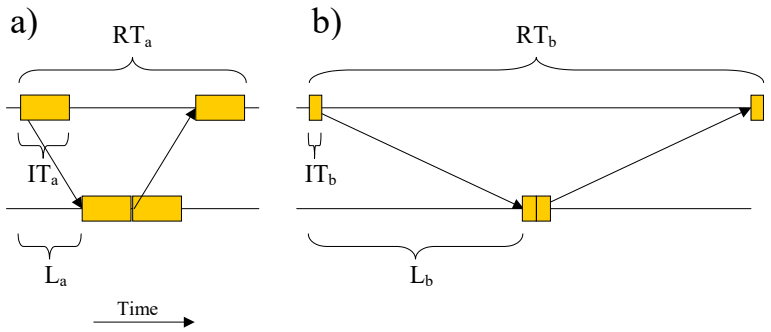


Figure 80 Roundtrip delay of request/response transactions with (a) low bandwidth and low latency and (b) high bandwidth and high latency

Other illustrative situations are found in stream processing applications. For instance, an audio communications system, where audio samples are produced at a certain interval and are processed by various modules in a dataflow pipeline. A usual goal in such a system is to reduce the total system delay. As shown in Figure 81, in case b a bigger latency ($L_b > L_a$) contributes to a large total delay ($TD_b > TD_a$) whereas a high bandwidth reduces the injection time ($IT_b < IT_a$), but as a consequence it only reduces the time window in which processing can occur but does not affect significantly the total delay. Again latency is the dominant factor.

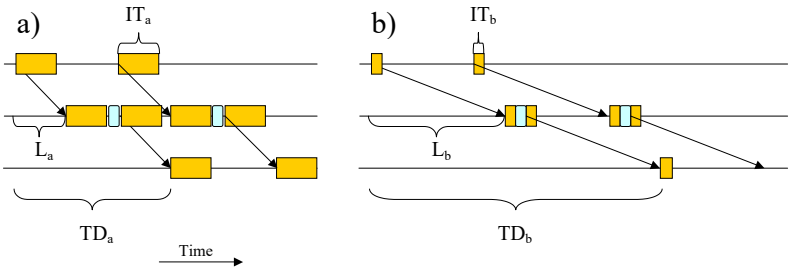


Figure 81 Total delay in stream processing applications with (a) low bandwidth and low latency and (b) high bandwidth and high latency

Reducing latency is important, but, in absence of congestion, what are the sources of latency? There are four possible sources of latency, the links, the routers, the NAs and the system software. Much research is devoted to try to reduce the link delay and the routers delay. System software can have a big influence in latency but this is more usually approached by application specific analysis. Network adapters are the other source of latency. As described in [Henry92] there are several contributions to latency that are related with the NA. First, NAs are usually attached to the processor bus, and a bus transaction usually involves several clock cycles to complete. Second, the communication with the NAs is encapsulated a non-negligible amount of device drivers code that has to be executed at every message transfer. This is a considerable cost when data to transmit is short. Finally, doing some polling on a status register is a usual technique to wait for the occurrence of an expected event, like a received packet. From the processor perspective this supposes a conditional branch that causes stalls in a simple processor pipelines, with the corresponding performance loss.

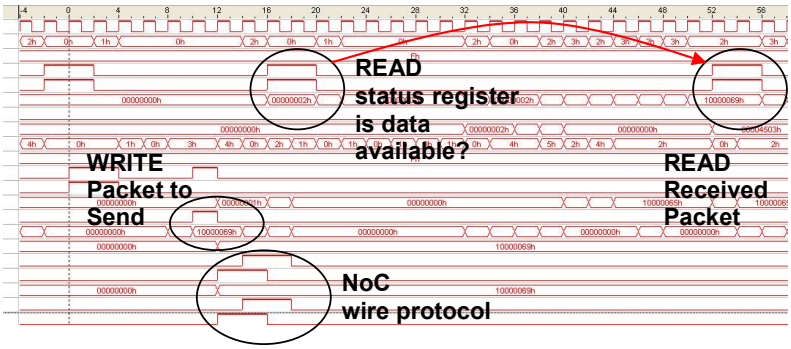


Figure 82 Waveform of the bus transactions at the sender and receiver processor, and the NoC wire protocol. Notice the overhead introduced by software (polling the status register) in a short message transmission over a NoC

An empiric test of the mentioned problems is shown in Figure 82. In this case I show a short packet transfer over a NOC between two NIOS II processors that use a bus-

based NA. Notice that the sending processor injects the message in just two cycles and the NOC needs four additional cycles to transmit it to the receiving node. However, twenty cycles are wasted between the successful polling of the status register to sense the incoming message and the actual ejection. Considering these results, it seems reasonable to firstly optimize the NA design before paying attention the network architecture.

In order to reduce the latency I propose to extend the instruction set of the processor to add specific communication instructions that help to offload the functions of a message passing programming framework. A similar approach is used by [Henry92] [Lu03].

The proposed new instructions are listed in Table 33. They try to be general enough to be applicable to different network topologies, switching methods, and network adapter architectures.

Table 33 Proposed new instructions

Mnemonic	OPERANDS	Result
setdsaddr	A=address	-
setsraddr	A=address	-
istxready	-	1 = ready, 0 = otherwise
txtto32	A=packet	-
txdst	A=payload	-
tx32	A=flit	-
tail32	A=tail flit	-
isda	-	1=data available, 0 = otherwise
istail	-	1=is tail flit, 0 = otherwise
waitda	-	-
rxsize	-	packet size
rx32	-	flit

The *setsrcaddr* and *setdstaddr* are used to inform the network adapter of the source and destination addresses of the next packet to transmit. Payload can be injected using *txdst* instruction. This approach allows to avoid the software overhead of adding the message headers at each packet. On the other hand, *txto32* is used to send a whole packet of 32 bits containing the header and payload. This instruction only has sense in Ephemeral Circuit Switching with a packet size of 32 bits, longer packets in Wormhole switching would not use it.

Packets in wormhole switching would use tx32 to inject flits of a packet and tail32 to inject the last flit of the packet. The first flit is not explicitly identified as it can be automatically detected with the following call to *tx32* after a *tail32*. This avoids using an additional instruction to specify the header. All sending instructions *txdst*, *tx32*, and

tail32 are blocking to avoid polling for the availability of the transmit channel, although a *istxready* instruction is also available to allow a non-blocking operation. Support for multi-cycle custom instructions is necessary to allow blocking operation. Although not covered in this work blocking instructions could be used to gate the processor clock until data is received in order to save energy.

On the receiving part, *rx32* is used to receive a complete packet or flit in a blocking way. In ephemeral circuit switching it receives a whole packet. In wormhole switching if the network adapter stores the entire incoming packet the processor could ask for the size of the incoming packet with *rxsize* instruction and perform multiple rx32 instructions to eject the packet. If the NA does not store the whole packet *istail* instruction should be used in a loop to sense the tail. To wait until some incoming data is received either the blocking *waitda* instruction or the non-blocking *isda* can be used.

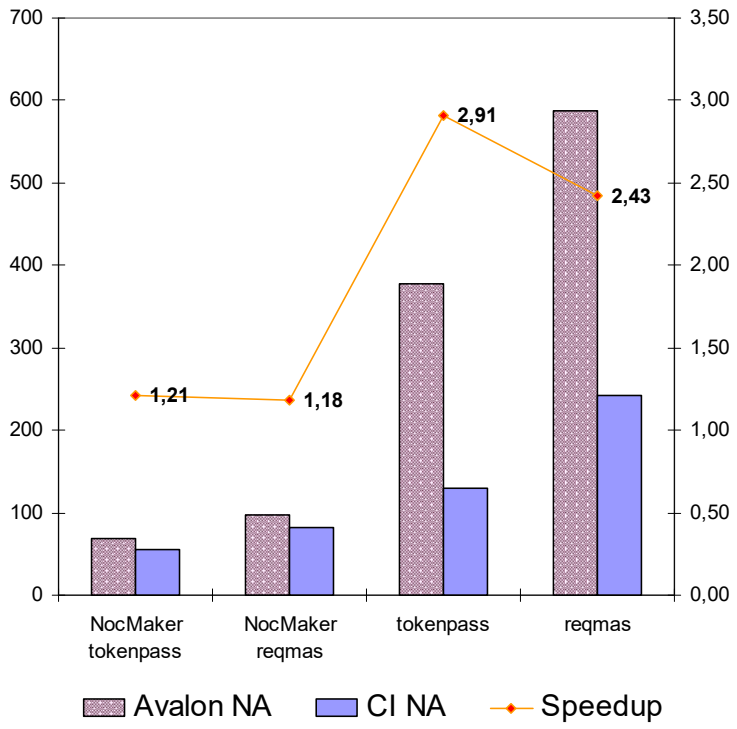


Figure 83 Performance gains due to latency reduction

The impact in latency reduction can be up to a factor of 3 if we compare the same system using a bus attached NA vs. a custom instruction implementation on NIOS II. Figure 83 shows the obtained results when running two micro-benchmarks (*tokenpass* and *reqmas*) on a small NOC-based MPSOC. The *tokenpass* benchmark consist of a short message (token) that is being sent from a node to the next one until the token returns to

the original sender. The *reqmas* benchmark consists of a request and response transaction that a master node sequentially performs to all their slaves.

3.2. Architectures

In chapter 2 I already described some architectural methods that try to increase the operations per second achieved by a processor. Another option to achieve the same goal is to replicate processors. Ideally a number of processors could be replicated as much as it is allowed by the available resources, and the performance would be multiplied by the number of the processors of the system in case of optimal implementation.

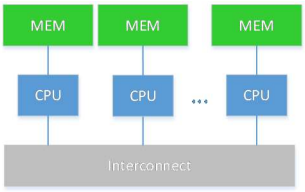
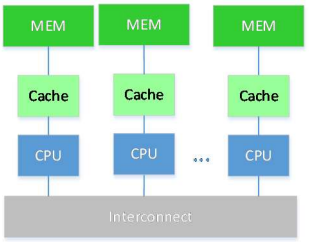
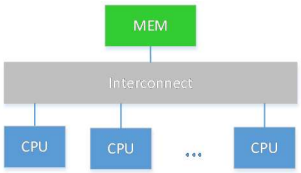
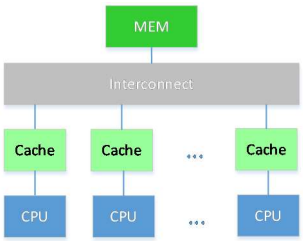
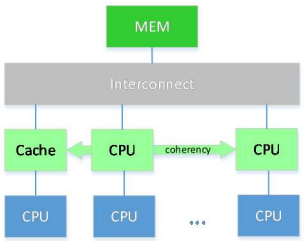
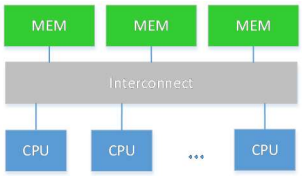
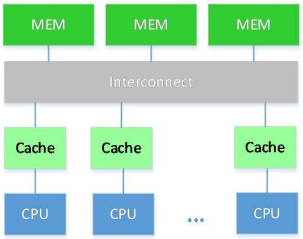
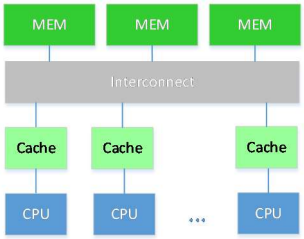
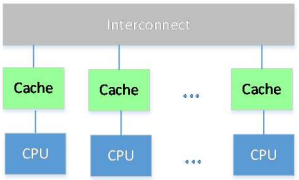
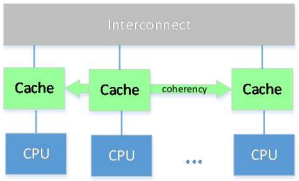
However, there are two important drawbacks that limit the performance improvement of this approach for any application. First, parallel processors must share some resources, and this is a source of contention that will limit the ideal linear speedup. Second, applications cannot always be transformed from sequential code perfectly parallel code. Often, parallel applications still have portions of sequential code and, as observed by [Amdahl67], those parts eventually limit the potential speedups of the system when executed in a parallel computer. Its application to parallel programming is as follows, let $F_{parallel}$ be the fraction of parallel code, and N the number of processors. The speedup factor S_{total} of parallel execution is determined by (3.7).

$$S_{total} = \frac{1}{\left(1 - F_{parallel}\right) + \frac{F_{parallel}}{N}} \tag{3.7}$$

Parallel architectures would fall in the MIMD classification in the Flynn's Taxonomy, but the number of possible designs is huge. Multiprocessors are composed by assembling processors, caches, memories, and interconnection networks in different possible architectures.

One of the crucial design options is how the memory is accessed by the different processors, and whether it shared or not. When memory is shared, it is important to decide if its access latency must be uniform across all the system or not, or even if a notion of shared memory is needed although it actually does not exist. With these considerations I made the following classification:

Table 34 Different multiprocessor architectures by their memory architecture and cache use

	No Cache	Independent Caches	Coherent Caches
NORMA			n/a
UMA			
NUMA			
COMA	n/a		

NORMA: NO Remote Memory Access. Usually called distributed memory, or message passing architectures. In this kind of architectures no memory is shared among processors. Each processor has its own private memory and cooperation is achieved by explicit communication calls.

UMA: Uniform Memory Access. Some network designs can enforce the exact same latency for all memory banks. This simplifies the logical view of memory, which can be seen as a simple block in contrast with what happens in NUMA.

NUMA: Non Uniform Memory Access. When memory is shared among a large number of processors it is often split in several memory segments that are connected to all processors through a network. It is usually difficult to maintain the same access latency

from all the processors to all memory blocks. In NUMA, applications must consider data location to obtain optimal performance.

COMA: Cache Only Memory Architecture. Local memory is used only as cache.

Another important aspect is whether cache memories are present, and whether they maintain coherency. Maintaining coherency is an important overhead for the memory system. If we combine the previous classifications we get a number of possible multiprocessor organizations.

3.3. Combining building blocks to create architectures

The building blocks presented in section 3.1 are often created by complex circuit generators, but they always result in an HDL code that can be integrated more or less easily to create even more complex parallel systems.

Nevertheless, as the number of modules can be of the order of hundreds and the number of connections can easily grow to the order of thousands, coding the instantiation and connection of all the modules of such a complex system in classic HDL languages like VHDL or Verilog can be a daunting task. More modern HDL languages like SystemVerilog can slightly alleviate it, by introducing the concept of interfaces to group several related input and output ports. However the numbers are still excessive, and can easily cause coding errors.

This process can be improved by using high level architecture creation tools provided by FPGA manufacturers. Altera, for instance, provide QSys, which supersede the former SOPC design tool that was addressing the same process. In QSys designers can integrate IP blocks and connect them graphically. Connections are based on interfaces, not signals, simplifying the visualization and minimizing the probability of introducing design errors.

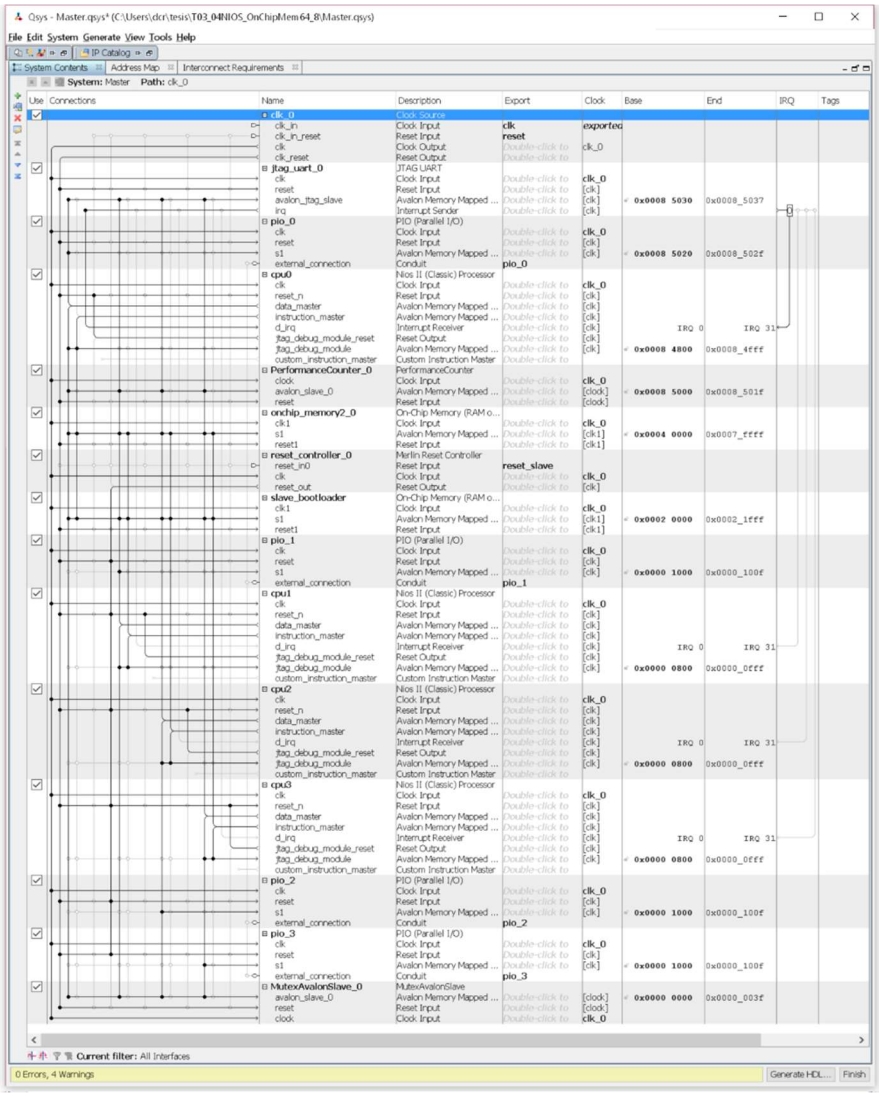


Figure 84 QSys interface for a 4 NIOSII multiprocessor

Figure 84 depicts the QSys user interface to describe an MPSoC system with 4 NIOSII cores sharing an On-Chip Memory, a *PerformanceCounter*, and a *Hardware Mutex*. Although the system is not very complex the user interface to describe all system elements and connections is considerably big. For bigger systems, like the 16 core MPSoC shown in Figure 85, the user interface is already limiting the capacity to glimpse the connectivity of the system and debug for potential errors.

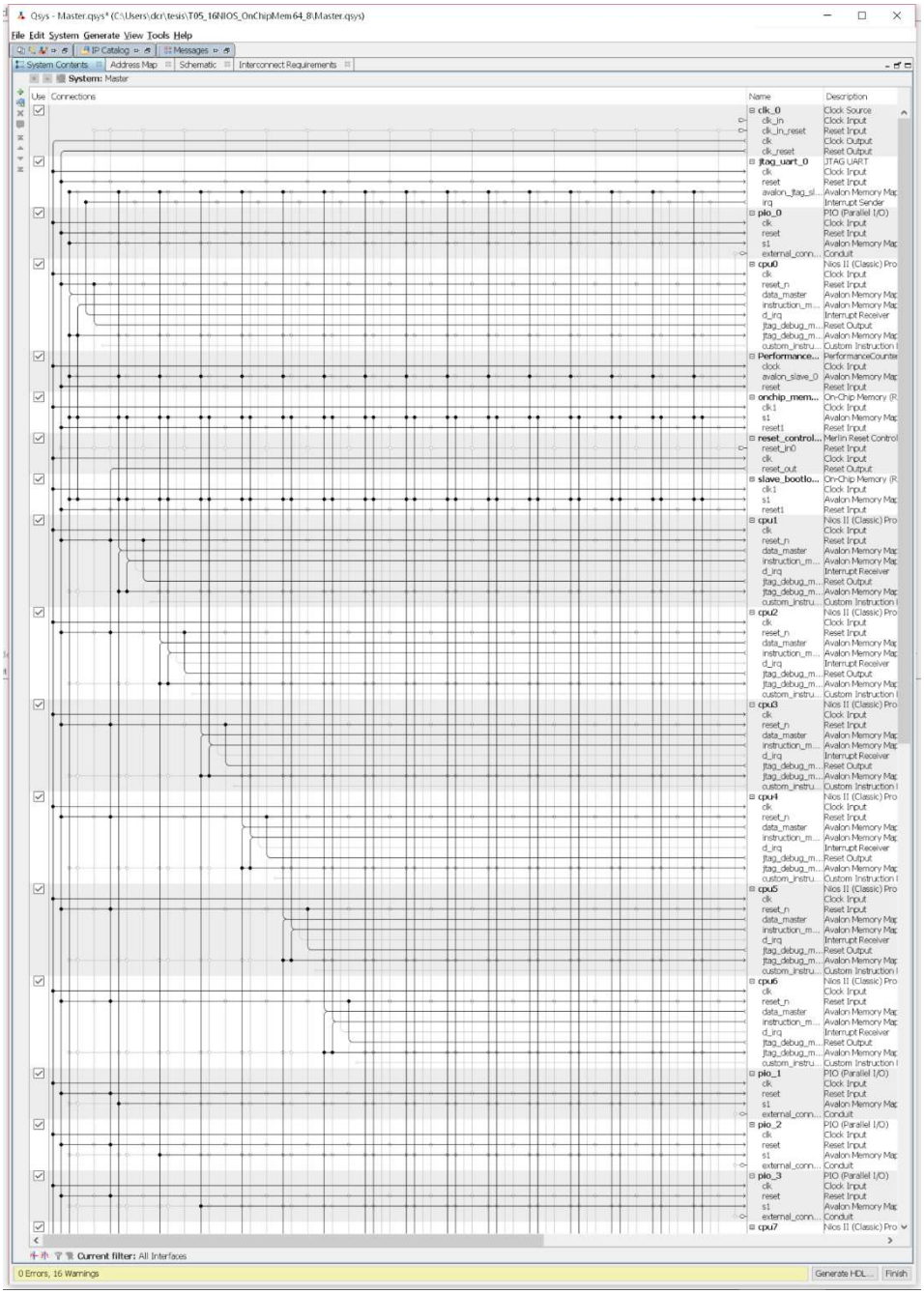


Figure 85 QSys multiprocessor system with 16 NIOS II processors

QSys allows working with hierarchies of elements. That could allow to describe some systems in a graphical way, and still, with a manageable complexity. However, I propose a custom generator approach to define the complete system with simple parameters. The toolchain is depicted in Figure 86.

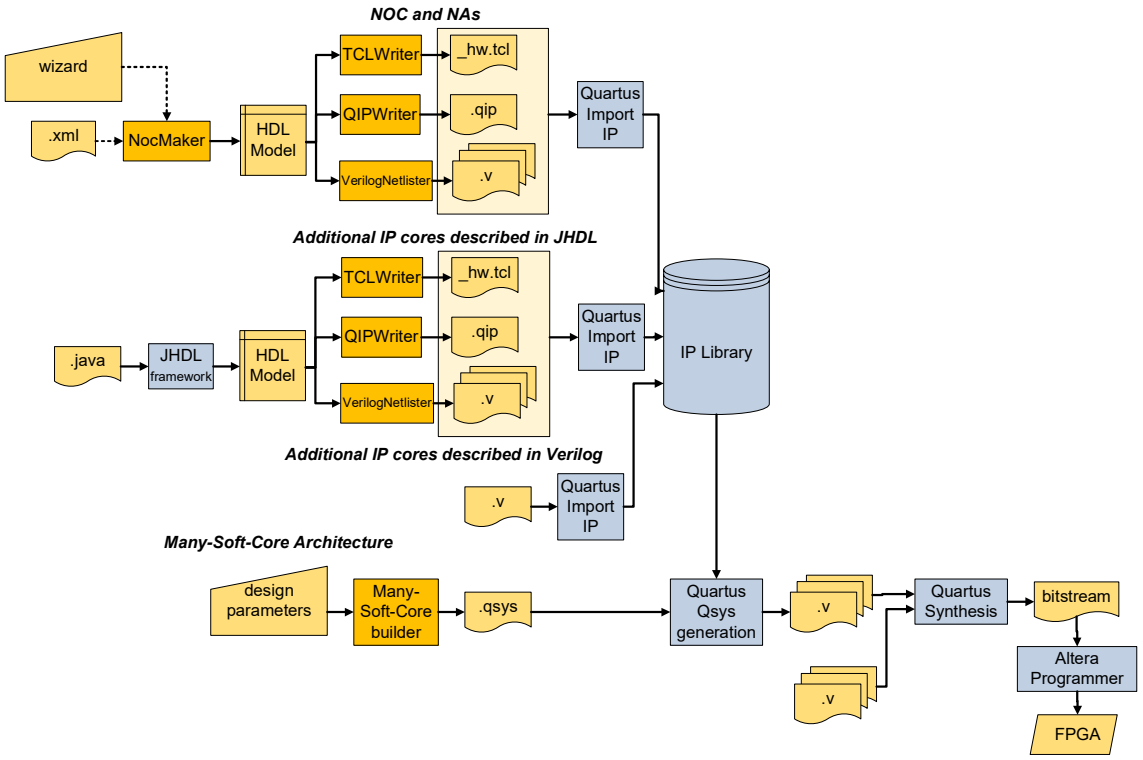


Figure 86 Many-soft-core architecture building toolchain

The main part of the system is the *Many-Soft-Core Builder* application (see Figure 87). This is a Java application that I have developed that generates a QSys file with the system properties selected in the user interface. This allows to define QSys files with a large number of elements without being limited by the QSys graphical user interface, which is not enough efficient when working with a large number of elements.

The generated QSys file uses IP cores from the Altera IP library. Some of the cores (like UARTs and Memories) are already provided by Altera, but other IP cores are imported from external IP core providers. External IP cores can be provided as Verilog or VHDL files, but I usually develop my cores with JHDL for easier verification and I export them into Verilog. Actually, JHDL did not originally support exporting designs to Verilog. I implemented this feature, and the automatic creation of QIP and TCL files for easier integration of modules in Quartus.

The NOC and NAs are created by NocMaker. When exporting to Verilog, NocMaker creates an IP block for the NOC and potentially multiple IP blocks for NAs depending on the characteristics of the NOC. This is so because some NOC designs can reuse the same NA IP for all the nodes, while in other cases the NA is specific for every node.

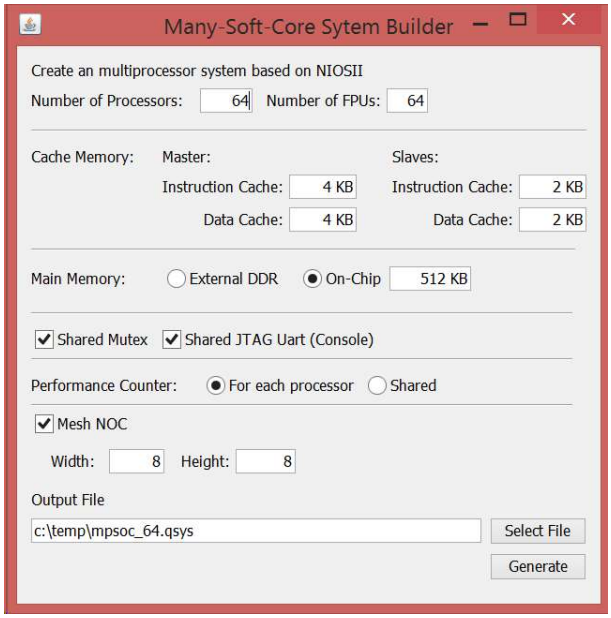


Figure 87 Many-Soft-Core builder user interface

The resulting IPs are incorporated to the Quartus IP repository and later referenced by the QSys file created by the Many-Soft-Core builder tool. Quartus translates the QSys file into multiple HDL files during the synthesis process, and they are combined with other HDL files that could be part of the project to define additional blocks of the system to address the particularities of the FPGA board in which the system is running. The synthesis process creates a bitstream file that is programmed into the FPGA for its execution.

3.4. Conclusions

Many-soft-core platforms are created by combining many IP blocks to build parallel architectures. FPGA vendors already provide IPs for most of the required blocks. For instance, Altera offers the QSys tool to easily instantiate elements from a large library of IP (proprietary o third party). The main element that is replicated when creating a many-soft-core is the processor.

In this chapter I presented the details of the Altera NIOSII processor and the methods provided for its customization to increase of the performance and energy efficiency. Also, I introduced the alternative MIOS processor, which is ISA compatible with NIOSII, is also presented. Having full access to the source code of the processor gives the flexibility to implement ISA subsetting or other architectural changes that could impact the energy efficiency of the system.

I analyzed the different possible floating-point support options. Single precision FPUs use as many resources (if not more) as a NIOSII processor. I demonstrated that those resources are highly underutilized and that better efficiency could be achieved by sharing them among a number of processors.

QSys automatically creates the necessary interconnect among the instantiated IPs, although not many parameters of the created interconnect can be tuned. I described NocMaker, a tool to create many possible different NOC designs that can be used to interconnect processors by an alternative mean. NOCs can be implemented with different topologies, routing algorithms, switching strategy, etc. I introduced a particular switching strategy, "Ephemeral Circuit Switching", which I used to minimize latency in NOC designs. Processors are connected to the NOC through NAs. I described different implementations of NAs stressing the benefits of tightly coupling NAs to processors.

The lack of cache coherency in commercial tools limits the types of parallel architectures that can be build.

Finally I described the limitations of graphical-based design tools to create a many-soft-core platforms and proposed the Many-Soft-Core Builder application that I will use in the rest of the thesis to create highly scalable systems. The tool is able to create non-coherent cache UMA, NORMA, or a hybrid UMA-NORMA by combining shared memory with message passing support over NOC.

4. Software Development Process for Many-Soft-Cores

The expected benefit of using Many-Soft-Cores instead of creating specific hardware is that designs will be generic and more easily reused for several applications, and that programmability will be easier. But that would not be possible without the support for convenient parallel programing models.

The main focus of this chapter consists in providing effective parallel programming models for them, so that those benefits can be reached.

4.1. Parallel Programming Development Process

The development of parallel code is much focused on obtaining the best possible performance that a parallel platform can provide. It usually starts with a sequential code that has to be transformed into a parallel program to be later compiled to the target parallel platform. Performance information has to be collected and analyzed to detect bottlenecks in order to guide changes to achieve potential improvements. The development process is highly iterative (see Figure 88), and it is usually complex one due to the usually huge amount of information to analyze. The techniques used to reduce the execution time need a deep understanding of the underlying hardware architectures.

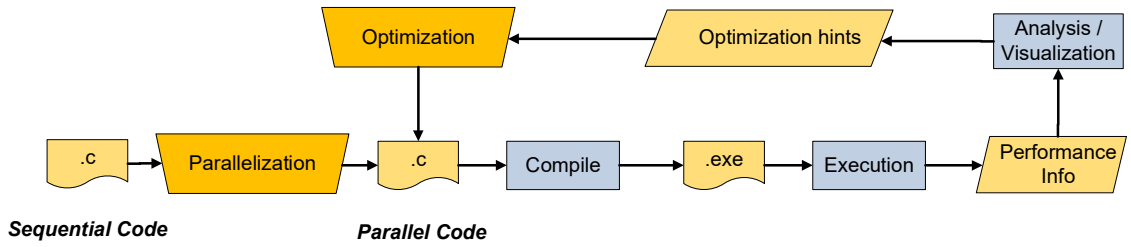


Figure 88 Parallel Programming Development Process

4.1.1. Sequential code analysis

Parallelization is often understood as rewriting a sequential program source code into one using a parallel programming language or API for its execution in an architecture containing multiple processing units, i.e. architectures falling in the SIMD and MIMD classification of the Flynn's taxonomy. Most often, for SIMD architectures, it is not

required to rewrite the source code as parallelization is usually performed by compilers, and programmers do not need to be aware of it. For vector architectures parallelization is synonymous of vectorization (like as used in [Lorenz05]), and in VLIW it is often just called VLIW compilation (like in [Stümpel98]).

Unless optimization is done at the assembly language, parallelization for SIMD architectures is seldom a manual process. Compilers have an important role in this case. Performance can vary depending on the optimizations selected by compilation flags like presented in [Inglart08]. Tools like MAQAO [Djoudi05] can be used to determine the quality of the binary code generated by the compiler.

However, In MIMD architectures, parallelization is usually a complex manual process that might involve complex source code refactoring. Thus, the goal of parallelization is to transform the application to execute it using N processors and reduce its execution time by a factor N. Before investing too much time in this process, some observations must be made:

1. In software, a Pareto Principle often occurs [Jensen08]. Most of the execution time is used to execute a small portion of the code. This implies that not too much effort should be devoted to parallelize the whole application, since most of the time is spent on just portions of it. Those are usually called hot-spots or bottlenecks.
2. The achievable speed-up can be limited by the non-parallelizable (sequential) part of the code, as stated by Amdahl's law [Amdahl67].
3. Data dependence can limit the parallelization factor as operations cannot start on processors if they are still waiting for results of other processors.
4. Due to the implementation details of the memory hierarchy data locality can have a great impact on performance [Kennedy14].

Before manually parallelizing a sequential application, it is often required to analyze its performance in order to detect its hot-spots, so that optimization is focused on improving the performance of the detected bottlenecks.

Commonly, data access must be carefully analyzed. It is interesting to determine what are the existing dependencies and the locality of data accesses.

There are some tools that help in both aspects of this analysis, like Intel's Vtune [IntelVt], Valgrind [Nethercote07], Acumem's Threadspotter [Acumem], Critical Blue's Prism [CritBlueP], VectorFabrics' Pareon [VectorFabricsP], etc. The principle behind

these tools is to collect information about the instructions executed and the memory positions accessed. They usually combine static analysis of the code to predict performance and data accesses, with dynamic analysis to have better insights of the runtime behavior of the applications.

Once bottlenecks, dependencies, and data access details are obtained, sequential code can be manually parallelized to transform it into some kind of parallel description.

On the other hand, some parallelizing compilers that can do this tasks semi-automatically. Some assistance is usually needed to determine the hot-spots and techniques to use.

4.1.2. Parallel Programming Languages and APIs

When parallel architectures, which were able to execute several instruction streams concurrently, were widespread the next challenge became how to efficiency describe those multiple instruction streams. The basic ideas were already sketched almost forty years ago in many works, like [Critcklow63][Hansen70] [Hoare76] [Wirth77] [Hoare78][Birrell87] [Cooper88] and many more. The basic principle of most approaches is that several units of execution exist. Proposals differ in how they call these units of execution (either tasks, processes, or threads), in the resources associated with them, their communication methods, and how its execution is scheduled.

The dominant approaches were derived from 1) the notion of communicating sequential processes explicitly through communication primitives [Hoare78] and 2) the notion of independent processes working with a shared state accessible from processes that can coordinate by using special synchronization primitives [Cooper88].

From those initial works, there have been a lot of proposals trying to rise the abstraction layer and ease the description of parallel workloads. New programming languages have been addressing parallelism from the start, like Java, Erlang, Chapel, Cilk, Haskell, StreamIt, SmallTalk, UPC, Scala, F# and many more. They support previous concepts in different ways but still have not been widely adopted by the HPC community.

[Diaz12] identify Threads, OpenMP and MPI as pure parallel programming models. The pure parallel programming model concept is used to illustrate how these programming models adapt naturally to the possible underlying architectures. Threads and OpenMP to shared memory architectures, and MPI to distributed memory architectures.

Nevertheless, since programming with threads is complex and error prone, the HPC de-facto standards are OpenMP, and MPI. They both have good support for gcc.

Using standards is also a good way to reuse all the knowledge and tools that the community has created for decades, and this goes in favor of increasing the, productivity factor by reducing the time needed to learn new languages, but also by being able to reuse all the available tools.

MPI provides a communication library, and a runtime system to execute parallel applications on multiple processors. MPI is in fact a specification defined by the MPI Forum, and there are many conformant implementations like OpenMPI and MPICH. It advocates for explicitly embedding communication primitives in the source code, making it hard to read and maintain.

On the other hand OpenMP advocates for a pragma based approach making the compiler responsible for the efficient parallelization of the application. A fairly easy way of parallelizing is by using OpenMP compiler directives. With this approach the compiler takes care of managing the threads needed for the code to work in parallel.

4.1.3. Performance Analysis

After first parallelizing an application it is unlikely that a speedup factor of N is achieved running on N processors, or even if this almost achieved for a certain low value of N, it is not generally applicable for an increasing value of N. The speedup factor can be plotted as a function of N, we call it scalability profile. A generic goal is to get an almost linear scalability profile as shown in Figure 89, but due to many possible reasons a bad scalability profile is often shown. An optimization process is needed to refactor the parallel source code in a way that a scalability profile as close as possible to the ideal linear speedup is achieved. In some uncommon cases even super-linear speedup is achieved. This is usually the result of improving data locality when partitioning the workload among the collaborating processors.

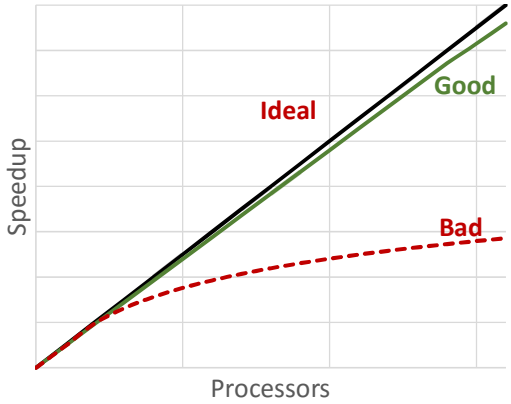


Figure 89 Scalability profile

The optimization process, often called performance tuning, is an iterative process in which the performance is measured and analyzed to decide how to modify the application source code to increase the performance.

Collecting and analyzing performance information is crucial to optimize the application, but what information do we need?

One could measure total task time and make assumptions about the reasons that limit performance. Then one should modify the code to address the assumed reasons and execute and measure again to validate them. If one suspects that the performance is driven by a limited number of factors that can have a discrete number of values, one could try to follow a brute force approach and measure all the design space, or a representative subset. Although it is not used to optimize a parallel application, the approach of measuring total task time is used in [Silva08] similarly to find optimal points in a design space. Nevertheless, total task time is an indirect measure of the real reasons that block performance. It is neither useful to identify Hot-spots.

A slightly better picture of the performance determinants can be obtained by collecting the total amount of time spent in every function of the application, and the number of invocations of them. This information can be used to effectively detect hot-spots and avoid optimizing non critical parts of the code, which majority as described by the Pareto Principle. Collecting aggregated information of function invocations and time spent in function is called profiling. There are two techniques to do profiling: Sampling and Instrumenting.

In sampling based profiling (Figure 90) a timer is programmed to get the value of the program counter (PC) periodically. With the PC value and the application symbols table the profiler infer the name of the function being called at every sample. By collecting all the measures over the execution time the profiler can get the number of invocations and the time spent in each function. A good things of this method is that it can work with the binary file, so no modifications to the source code are needed. On the other hand, if sampling frequency is low the overhead introduced by profiling can be very low. This, of course, can go against the accuracy of the measurements, since events could be even missed. If sampling frequency is too high, the overhead can be excessive and modify the normal program behavior. Sampling based profiling in many tools like Intel's Vtune and Microsoft Visual Studio 2015, and GNU gprof, to name a few.

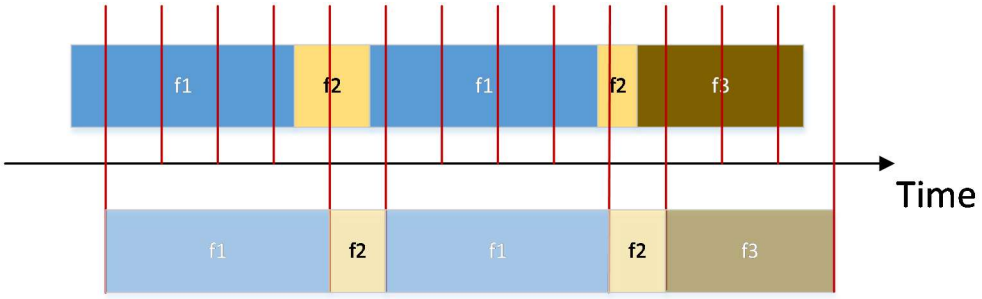


Figure 90 Sampling Profiling. The upper bar represents the execution of functions as time progresses on the application under test. The lower bar represents the perceived execution of functions. Notice that the sampling process introduces some error.

In instrumentation based profiling (Figure 91) the compiler introduces hooks at the prolog and the epilog of every function to collect time and invocation information. The aggregated information for every functions is usually maintained in memory during runtime by the profiling code added by the compiler and flushed to disk at application exit. The resulting information is analyzed and presented by another tool at the end of the profiling session.

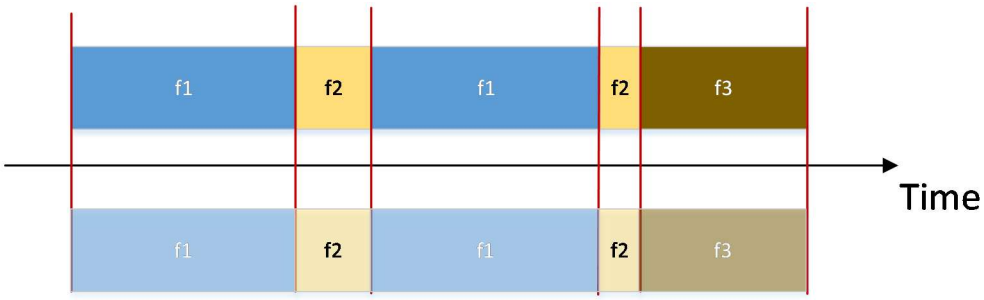


Figure 91 Instrumentation Profiling. . The upper bar represents the execution of functions as time progresses on the application under test. The lower bar represents the perceived execution of functions. In this case no error is introduced by sampling period.

Application profiling is usually understood as the process of gathering histogram information from the functions of an application. The usual observed metrics are the number of function calls and the time spent inside every function. Profilers can be based on periodically sampling and collecting the value of the program counter or either instrumenting the entry and exit of every function. Sampling profiling has usually lower overhead although it also has lower accuracy. Accuracy vs. overhead if a tradeoff controlled by the sampling frequency as depicted in Figure 92. The blue boxes represent the time spent in the function that collects the necessary information. When sampling frequency is low (b) the original execution time of the application (a) is only slightly incremented, but the profiler never knows that function f2 is called. So the results will be

misleading. On the other hand if sampling frequency is incremented (c) the profiler will know that f2 is called but the execution time will be substantially affected.

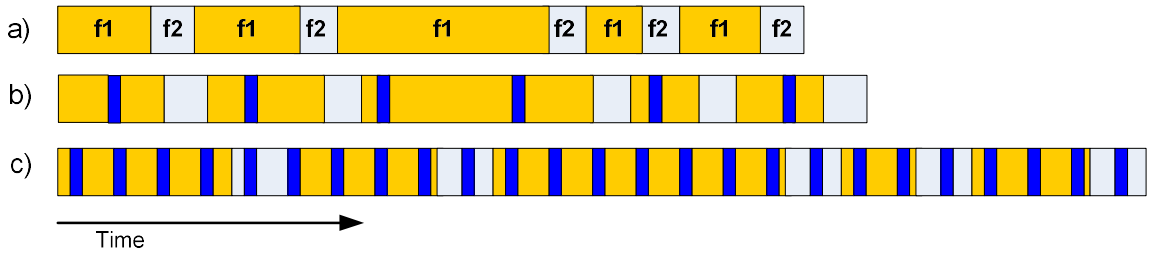


Figure 92 Tradeoffs between Accuracy and Overhead in Profiling. a) Original application execution. b) Profiling with low sampling frequency. c) Profiling with high sampling frequency.

Profiling helps in identifying hot-spots, but still do not provide enough information to completely identify the causes of bottlenecks. Some bottlenecks could be caused by data dependencies, or by sequential parts dominating over parallelized code, as described by Amdahl's law. These behaviors would not be captured by profilers, as they work on aggregated information.

A way to catch them is to record the absolute timestamps of every function call at prolog and epilog. Data dependencies can be found, at some extend, by analyzing the details of messages exchanged among processors in MPI applications, and in calls to synchronization functions for OpenMP applications. Compiler instrumentation can be used to automatically insert hooks that record this information in memory and later flush it to disk. This method is often known as tracing, and it is implemented by tools like VampirTrace [William09] and Score-P [Knüpfer12].

But sometimes, with this information, it is still not possible to understand the source of performance penalties, an even more insight is necessary. This happens, for instance, if the bottlenecks are caused by the memory hierarchy and too many cache missed occur, or it could also happen because of too many processor stalls, or a limit on the functional units of the processor. To measure this details it is usually needed to collect the values of special processor counters that collect this kind of information, when possible. This information can be also included in activity traces.

All this potentially huge amount of information can be later analyzed. This activity is known as Post-mortem trace analysis, and it is a usual technique used by the HPC community to optimize parallel applications. Tools like and Vampir [Nagel96] [Brunst01], Paraver [Pillet95], TAU [Bell03] are commonly using this approach to perform performance analysis on very-large systems with thousands of processors.

In order to reduce the amount of information produced in tracing, there is usually the option to manually instrument the code to select just the functions of interest. With this method the overhead can also be minimized but some resolution is lost.

4.2. Shortcomings of the development process for soft-core multiprocessors

Although FPGA providers provide the means to build multiprocessors they still lack a complete toolchain that allows to follow the iterative process depicted in Figure 88 effectively. There is also a fundamental difference on the resources available in typical systems using soft-core processors and the resources available to standard processors used in HPC, and this differences also influence the features offered by toolchains. Computers using standard processors have large storage in forms of memory and disks and powerful user interaction methods, either via human interface devices or via networking devices and remote terminals. The large storage allows to include complex operative systems like GNU/Linux, and complete software development toolchains covering all the steps previously described.

On the other hand, soft-core based systems are often resource limited, having less memory and no disks. They usually do not support operative systems and user interaction is limited. Development cannot be done from the same system but it must be done from a host computer that interacts with the reconfigurable system.

This forces to use cross-compilation to develop applications for soft-cores, i.e. the compilation process is done in a host computer with a compiler emitting a stream of instructions of the target soft-core ISA. The executable must be downloaded to the target system and debugging is done remotely. This limits the visibility of the executable program.

But this is not the most limiting factor, in fact some embedded systems using standard processors are also developed under similar constraints.

4.2.1. Lack of parallel programming support

Altera and Xilinx provide a development toolchain based on GNU tools, but they still do not support OpenMP or MPI. Soft-cores are usually used to run embedded applications with no operative system or with minimal runtime systems like eCos [Massa03] or Xilkernel [Saha13]. This small runtime systems (also called kernels)

provide threads, but limited to a one processor scenario. Lack of threads for SMP architecture is partly motivated as well by the usual lack of cache coherency support.

To make it worse, manufacturers are not promoting the Single Program Multiple Data (SPMD) paradigm. When implementing shared memory systems, Altera, for instance, promotes to completely divide the memory among the processors. Figure 93 depicts such division for a system with two processors. The Reset section contains the code executed on processor boot. The Code section contains the instructions of the application. The Heap section contains the objects dynamically allocated, and finally Stack section contains the local function variables, that are allocated and unallocated as functions are called and exited.

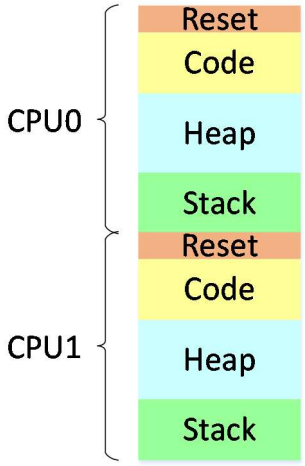


Figure 93 Memory division adopted by Altera for Soft-Core shared memory multiprocessor

This division can be reasonable in Multiple Program Multiple Data scenarios, i.e. when cores execute totally different applications, but there would be few reasons to share the same memory. If all CPUs should execute the same code Reset and Code sections (which are read-only) would be replicated as many times as processors in memory, and the Heap space available to a processor allocating memory would be limited to a the subsection of the allocating processor, ignoring the available Heap space on other processors Heaps. This is completely inefficient.

4.2.2. Lack of appropriate Performance Analysis tools

A large number of applications implemented on FPGAs described in academia focus on performance of the design related to other possible implementations, comparing the Total Execution Time (TET) to determine the best implementation. This is even the case for works based on soft-core processors. Surprisingly there is little work on more detailed performance analysis for soft-core systems to understand the ultimate roots that

determine the performance. There are some very notable exceptions ([Curreri10] [Koehler11]) but do not focus on the specific multi-soft-core and many-soft-core specific issues.

Compared with typical HPC performance analysis techniques FPGA provide both some constraints and some additional features that can be used to get information of the system performance.

Logic Analyzer based analysis

A quite rudimentary approach available to FPGAs is to obtain time information by asserting a signal that can be traced externally by a digital logic analyzer. This is a very old idea ([Sun99]), but with the proliferation of embedded logic analyzers inside FPGA devices it became possible without the need of an external logging device ([Altera07]). However the limitation on the signals that can be traced, and the size of the trace buffer, does not make it a convenient way to analyze a typical complex application. On the other hand, if the application is partitioned in subsets that are analyzed separately by this technique the number of re-synthesis and run steps increases making it an extremely time consuming process.

Profiling

The GNU toolchain includes the gprof profiler, a sampling profiler which is usually available to popular soft-cores. Performance analysis using gprof is a common technique to get performance information from soft-cores [Altera-AN391] [Tong07]. It consists in a three step process, first the application if instrumented by adding the `-pg` compilation flag in gcc. When the `-pg` flag is enabled, the compiler inserts a call to a function that will keep the count of every function invocation, and registers a timer callback function that will record the program counter at every sampling period. Second, the application is executed and its execution generates an output file containing all the collected information. Since soft-cores do not usually have a file system, the collected information is typically transferred automatically through JTAG by the `nios2-elf-download` tool as described in [Altera-AN391].

And third, the output file is analyzed in a host computer by a visualization tool to present all the information to the developer. In case of the Altera toolchain, this application is called `nios2-elf-gprof`.

Sampling based profiling can be very misleading if the sampling frequency is very low, because there is a high probability to miss the execution of short functions. On the

other hand, if the sampling frequency is very high, it can produce a high overhead causing a significant alteration on the usual application behavior.

Transparent profiling

An alternative approach to reduce the overhead is to monitor the program counter (PC) with specific additional hardware. Shannon et al. in [Shannon04] described a method to continuously monitor the value of the PC. In that work, a hardware module includes a low address and high address register, comparison logic and a counter. When the PC value is between the low and high value the counter is incremented. Those registers are typically programmed with the address boundaries of a function. In addition, the circuit can be replicated as many times as functions you have to measure. Although this approach is limited to just a small number of functions the lack of overhead is appealing.

Functional simulation

A large body of research has been devoted to performance modeling based of high level of abstraction descriptions (for instance [Monton07] [Posadas04] [Posadas11] [Böhm10]). Starting with RTL hardware simulation one can try to speed up the system simulation by using higher level models of the same circuits, or one can start directly from high level design to get to the hardware implementation in an iterative process. This, of course, comes at losing accuracy (see Figure 94). Why is simulation preferred to execution on those works? The argument is that hardware design is a costly process and it is desirable to start software integration as soon as possible to minimize the Time-to-Market. To do it, functional models can be quickly described to avoid postponing the development of software.

Simulation type	Speed	Accuracy
Functional execution	100000	
Timed native co-simulation	10000	
Timed binary translation	1000	
ISS (instructions)	100	
ISS (cycle accurate)	10	
Pin accurate	1	

Figure 94 trade-off between speed and accuracy (from [Posadas11])

This makes sense for ASIC design and for the design of FPGA applications where custom hardware must be developed from scratch, or when it is too expensive to replicate the hardware platform. However, for many FPGA based systems the premise of the

hardware not being available is more difficult to hold given the large number of IP Cores, the availability of soft-core processors, and the low cost of some FPGA platforms. So, in those cases, building a complex virtual platform does not make sense, since building the actual platform is easier, and, then, it is more convenient to measure timings directly actual platform to have full accuracy.

Virtual platforms can give a lot of information, however they suffer several drawbacks. If simulation is done at a low level (RTL) or at a combination of to include the existing descriptions of hardware IP Cores, the simulation speed is slow. Also, as mentioned before, if binary translation or functional simulation is used simulation is greatly accelerated but then time accuracy is lost and some development effort is needed to provide IP cores as functional descriptions.

Moreover, to analyze the interaction with some real physical (non-virtual) device and determine if the real-time constraints are met, virtual platforms cannot be generally used.

4.3. Memory Access Pattern Analysis new Proposal

One of the fundamental roadblocks to parallelize applications is data dependency. When an operation depends on data that has to be previously computed we have to delay execution of the operation until operands are available. Interdependencies between variables are terribly common in procedural programming languages that were conceived to be executed sequentially like C/C++, Java, and many others. A common code structure that has been always addressed as an easy candidate to parallelize are bounded loops. Some of such loops can be easily executed in parallel if they do not exhibit dependency.

The following code, shows a simple loop.

```
for (int i=0; i < 10; i++)
{
   a[i] = k * b[i];
}
```

In this case all ten iterations could be executed simultaneously because iterations do not exhibit any dependency.

The following code is slightly more complex as introduces dependency.

```
for (i = 0; i < 4; i++)
  for (j = 0; j < 4; j++)
    z[i+j] += x[i] * y[j];
```

In this case the iterations should be executed sequentially because iterations need data that are produced in former iterations. The operations that are executed between the operands are irrelevant, any operation would produce the same data dependency.

Figure 95 plots the iteration space of the former loop in a graphical way the data dependencies between iterations as arrows.

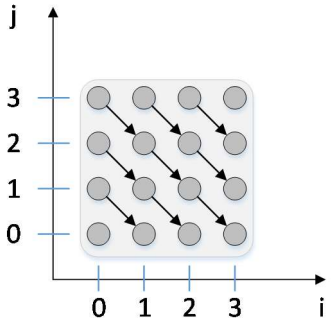


Figure 95 Plot of the iteration space of former code and the data dependencies they exhibit (from [Zinenko15])

If either the inner or the outer loop did not had dependencies it could be parallelized. This could be done, for instance, with OpenMP *#parallel omp for* pragma. But since it has dependencies in both dimensions it cannot be done.

However, as explained in [Zinenko14] [Zinenko15], applying the polyhedral model, the loop can be transformed into an equivalent code. And the loop iteration space is now represented as Figure 96.

```
for (i = 0; i < 7; i++)
  for (j = max(0,i-3); j <= min(3,i); j++)
    z[i] += x[i-j] * y[j];
```

Although there are still dependencies, they occur just on the j iterations (the inner loop). And now the outer loop can be parallelized with safe.

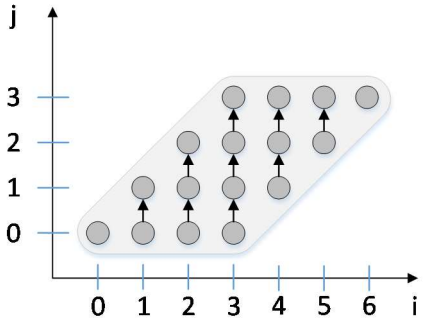


Figure 96 Plot of the iteration space of the transformed code using polyedral model and the resulting data dependencies (from [Zinenko15])

Memory hierarchy can be serious bottlenecks to scalability. Cache sizes can influence the parallelism that a system can achieve. If the data accessed by different cores do not reuse data from caches and has constantly go main memory, the penalties of cache misses can make the efforts to paralyze the application worthless. Cache misses can be measured with hardware counters (if available) or can be simulated. Hardware counters can be measured by tools like Intel VTune Amplifier XE as described in [Gromova12], but this method is limited to x86 based systems that support Performance Monitoring Unit (PMU) hardware counters and is useless for a soft-core cross-compilation toolchain. The later approach is followed by cache-grind, a tool included in the Valgrind framework [Nethercote07] to profile the cache behavior by simulating two levels of cache. Identifying that the rate of cache misses is higher than expected has an important value, although it cannot be straightforward to easily understand why it happens.

Simple codes can be relatively easy to understand and to decide if they can be parallelized, however, in complex code, including conditional clauses and function invocations, determining data dependency can be really hard.

A more detailed analysis of the memory access pattern would be needed to understand how memory is accessed by applications. This could help in identifying dependencies in complex code and help to determine why cache misses occur.

I propose a new method to analyze the memory access pattern exhibited by applications. The idea has similarities with the proposed by [Brewer88], [Corina10], and more recently in [Subotic14].

4.3.1. Proposed Method

I propose to record the memory read and write operations of a subset of the variables of the application under test during its normal execution. In my proposal the programmer instruments the code, either by introducing some pragmas in the code to identify the section and variables to be analyzed, or by manual instrumentation. Then he executes the program. This creates a memory access trace, which is later analyzed by a visualization tool. The process is illustrated in Figure 97. The visualization tool shows the data access pattern of the application and potentially detects if there are dependencies that prevent loops from being unrolled.

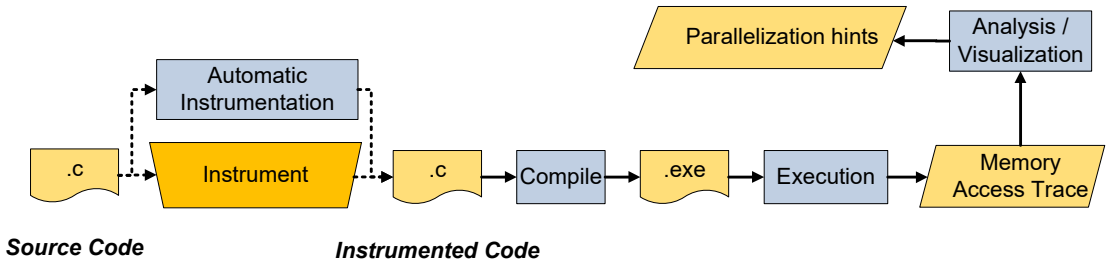


Figure 97 Proposed process for data access pattern analysis

Manual instrumentation

I propose the following functions to determine the loop context, i.e. to identify when a loop starts and ends, and when every iteration starts and ends. Each loop is given a name, and each iteration is based on a variable that takes a constant value during the iteration. If loops do not follow this rule, they cannot be analyzed.

```
void mem_trace_loop_start(char* loop);
void mem_trace_loop_end(char* loop);

void mem_trace_iter_start(char* var, int v);
void mem_trace_iter_end(char* var, int v);
```

The idea is that we can later identify which accesses are performance during any iteration of any loop. Loops will be unequally identify by a name, and iterations inside a loop will be identified by the value of its iteration variable.

Variable characteristics need to be known before recording their accesses. The systems needs to know their name, their dimensions, and their place in memory. By knowing the total size of the variable (that is, the number of dimensions, and size of each dimension), the base memory position, and the type size of variables, it is possible to track exactly which variable elements are accessed just by recording the addresses of memory accesses.

```
void mem_trace_def_array1d(char* v, int d1);
void mem_trace_def_array2d(char* v, int d1, int d2);
void mem_trace_def_array3d(char* v, int d1, int d2, int d3);
void mem_trace_def_mem(char* var, void* ptr, int typeSize, int varSize);
```

To trace the memory accesses the following functions are used.

```
void mem_trace_read(char* var, void* idx);
void mem_trace_write(char* var, void* idx);
```

Besides previous functions, we provide a method to disable logging, and setting alias. This is useful if function calls are present inside the analyzed loops.

```
void mem_trace_enable(int v);
void mem_trace_set_alias(char* v, char* w );
```

Automatic Instrumentation

Manually instrumenting the code can be a tedious process. Another option is to automatically instrument the interesting loops. I propose to use the source to source compiler developed in [Saa16] to transform the original code so that tracing functions are inserted. I use pragmas to identify the interesting loops.

The instrumentation is done automatically by the use of a S2S compiler based on BSCs Mercurium framework [Balart04] that includes traces on the original code that later will be used as log traces on execution. Mercurium gives us a S2S compilation infrastructure aimed at fast prototyping and supports C and C++ languages. This platform is mainly used in the Nanos environment to implement OpenMP but since it is quite extensible it has been used to implement other programming models or compiler transformations, providing the S2S compiler with an abstract representation of the input source code: the Abstract Syntax Tree (AST) having then easy access to source code structure representation, the table of symbols and the context of these.

To automatically introduce login function to an input source code, the implemented tool requires a source code marked using a created new pragma directive that is dedicated to describe the information needed to analyze inside the marked block. In our case, the marked blocks will contain always a *for* loop.

The created directive is named `analyze_access_pattern`, and this could be completed using two clauses `var` and `iter`. We show an example of the use this new directive using both clauses on Table 35. The `var` clause is used to determine the target variables in which we suspect that could appear a data dependency (marked on red) and, on the other hand, `iter` clause will determine the iterators on which we will target the access log(marked on blue).

Having the variables of interest and the iterators that define our loop for to check on the annotated new pragma directive, we traverse the AST detecting memory accesses to these and including the proper log functions in the points on which we detect the aforementioned uses. The result of that S2S compilation process can be seen on Table 35, on the left we illustrate an example of an input code and on the right there is the generated

source code. A particular case to consider is when S2S tool detects a function call inside the annotated pragma block, in that case the tool will do an inline transformation before traversing the AST.

Table 35 Example of an automatically instrumented code

Input Code

```
double alpha, A[N][N], B[N][N];
int main(int argc, char** argv) {
  int i, j, k, n = N;
  init_array();
  //  B := alpha*A'*B, A triangular
  #pragma analyze_access_pattern var(B) iter(i,j)
  for (i = 1; i < n; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < i; k++)
        B[i][j] += alpha * A[i][k] * B[j][k];
  return 0;
}
```

Transformed Code

```
double alpha, A[N][N], B[N][N];
int main(int argc, char** argv) {
    int i, j, k, n = N;
    init_array();
    mem_trace_def_array2d("B", N, N);
    mem_trace_def_mem("B", &B, sizeof(double), sizeof (B));
    mem_trace_loop_start("loop1");
    for (i = 1; i < n; i++) {
        mem_trace_iter_start("i", i);
        mem_trace_loop_start("loop2");
        for (j = 0; j < n; j++) {
           mem_trace_iter_start("j", j);
           for (k = 0; k < i; k++) {
              mem_trace_read("B", &B[j][k]);
              mem_trace_read("B", &A[i][j]);
              mem_trace_read("A", &A[i][k]);
              mem_trace_write("B", &B[i][j]);
              (B[i][j] += alpha * A[i][k] * B[j][k]);
           }
           mem_trace_iter_end("j", j);
        }
        mem_trace_loop_end("loop2");
        mem_trace_iter_end("i", i);
    }
    mem_trace_loop_end("loop1");
    return 0;
}
```

Execution

When the application is instrumented, it can be executed to obtain its memory access pattern trace. Logging the memory accesses produces an overhead. Moreover it also generates a great demand for memory when visualizing the accesses. Here, an observation must be made. Quite often the data access pattern is determined by the algorithmic nature of the application rather than the size of the input data. For instance, to understand how a matrix multiplication access pattern occurs it is not necessary to work with 1000x1000 matrices, analyzing a 10x10 can provide the same kind of information that allows an effective parallelization strategy, although it will provide totally different information about data locality.

Working with smaller workloads is highly encouraged, although it is not always possible.

Data analysis and Visualization

Before visualizing the data, information is aggregated so that it can be usefully presented. The first principle is that all memory access happening in the same iteration are aggregated. There is no point in looking at individual read and write operations. Developers are interested in detecting which memory positions are accesses in different loop iterations.

The visualization tool keeps information about if read or write operations have happened in each variable. In case of arrays, it keeps this information for each element of the array. This information is aggregated for every iteration of the loop.

Each iteration is identified by a keyword formed by by the name of the loop and the value of the iteration value. If a hierarchy of loops are analyzed the keyword is formed by the combination of each iteration identifier.

Before visualizing the trace, some analysis of the logs are made to detect the read-only variables. Read-Only variables can be usually ignored if the analysis is focused on detecting parallelism potential, but no if there is interest on detecting the memory accesses that could generate a bottleneck.

The visualization tool let the user play back the application iterations and detect if dependencies are found. It allows to select which loop is analyzed. When a loop is selected, all the memory accesses happening inside each iteration are aggregated. Memory positions receiving only Read accesses are displayed in blue. Memory positions

receiving only write accesses are displayed in red. Memory positions receiving both Read and Write accesses are displayed in orange.

When data dependency check is enabled, memory positions that are written in one iteration and read or written by another iteration of the same loop are marked in yellow to inform that there are memory collisions in them.

Example: Triangular Matrix Multiplication

I take, for instance, triangular matrix multiply from Polybench, instrument it and execute it to generate the memory trace file for later analysis. The code defines the size of the matrix in a constant N. For the analysis I put N to 20, so that it reduces the amount of collected data, and eases the visualization of it.

```
// loop1
for (i = 1; i < n; i++){
   // loop2
   for (j = 0; j < n; j++) {
      for (k = 0; k < i; k++) {
         B[i][j] += alpha * A[i][k] * B[j][k];
      }
   }
}
```

When running the visualization tool and selecting to detect data dependency in loop1 (see Figure 98), there are positions of the B marked in yellow because they are read in an iteration and were previously written in other previous iterations. As the iteration variable i is bigger, the number of read after write (RAW) collisions increases. With this simple analysis it is clear that loop 1 is no directly parallelizable.

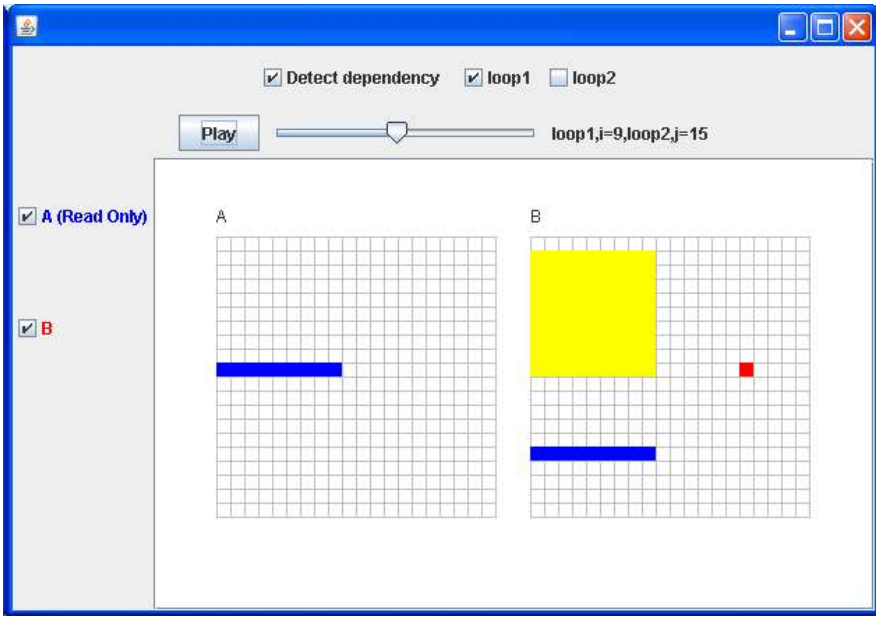


Figure 98 Analysis of the data dependency of trmm in loop1.

I run the analyzer again focusing on loop2 (see Figure 99). In this case collisions still occur. But they only occur when j is equal to i. When j is lower than i, the memory accesses are done to compute the values that go in the first positions of the i row. And those values are not written in this i iteration. When j is higher than i the computed values are written for the last positions of the i row that are no read.

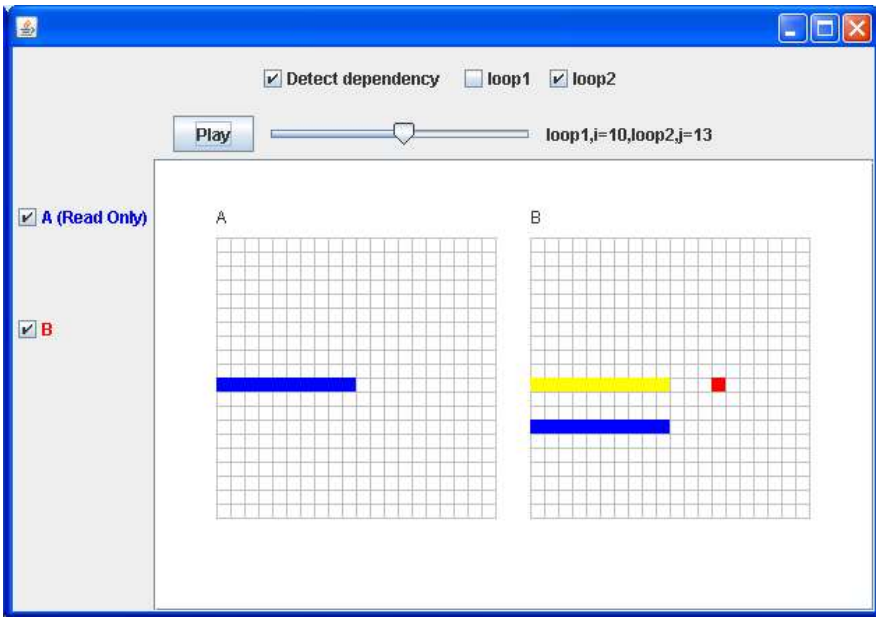


Figure 99 Analysis of the data dependency of trmm in loop2.

Seeing this behavior the code can be restructured to create three inner loops with the objective to isolate the line where collisions occur and be able to parallelize the other loops

```
// loop1
for (i = 1; i < n; i++) {
   // loop2
   for (j = 0; j < i; j++) {
      for (k = 0; k < i; k++) {
         B[i][j] += alpha * A[i][k] * B[j][k];
      }   }
   // loop3
   for (j = i; j <=i; j++) {
      for (k = 0; k < i; k++) {
         B[i][j] += alpha * A[i][k] * B[j][k];
      }   }
   // loop4
   for (j = i+1; j < n; j++) {
      for (k = 0; k < i; k++) {
         B[i][j] += alpha * A[i][k] * B[j][k];
      }   }   }
```

After this change *loop1* is still showing dependency, but now *loop2*, and *loop4* have no dependency and can be parallelized.

I execute it on a Bullion machine, but now for N = 1000, and the maximum speedup factor achieved is 12 for 30 cores (see Figure 100).

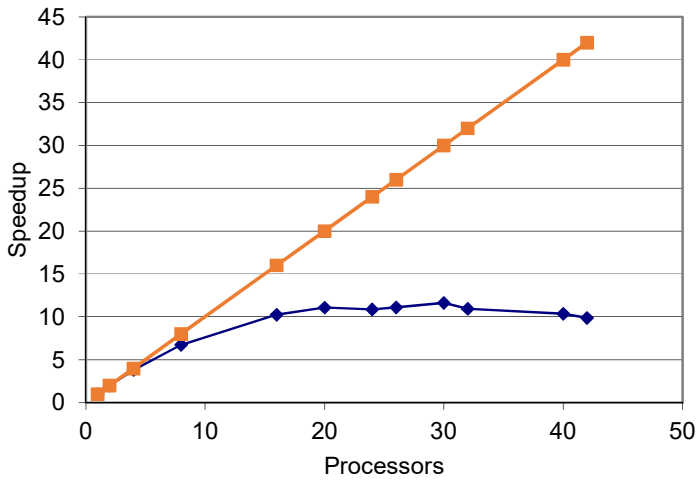


Figure 100 Speedup factor achieved in triangular matrix multiplication after memory access pattern analysis

4.4. Supporting programming models in Many-Soft-Cores

In order to effectively exploit the power of the parallel architectures it is important to focus on the methods and techniques that help programming such Many-soft-core

systems. The main goal must be to offer programming models and languages to the programmer that raise the productivity, in terms of executable code written per time, effectiveness, in terms of performance achieved per time spent, and obtain a good scalability, understood as the ability to increase the performance of the application by adding processors to share the workload.

It would be desirable that programming models would hide the hardware complexity of the systems as much as possible, while taking into account that many-soft-core systems will probably be heterogeneous because custom hardware could be created just in certain processors if needed.

Since commercial soft-core toolchains are based in gcc it would be reasonable to use a parallel programming framework with good support for gcc.

I believe that the MPI programming model is the best option for many-soft-core systems as it offers scalability for distributed memory systems, where usually OpenMP is not easyly used, and can be even more scalable in multi-core processors [Mallón09].

Message Passing Interface is the standard de facto used in distributed memory systems, like HPC clusters, for communication among processors.

MPI promotes data locality, which usually goes in favor of scalability. MPI is a real option to program highly parallel and scalable many-soft-cores. Furthermore, the portability and extensibility of MPI API make it easy to be tailored to many-soft-cores, and MPI API is a very well-known programming model for the programmer community.

I propose to support MPI and Threads as a starting point. Since OpenMP can be based on threads I propose to port it in the future.

4.4.1. MPI support

The lack of cache coherency and the lower complexity of distributed memory reconfigurable multiprocessors vs. shared memory reconfigurable multiprocessors have promoted some initiatives to adapt message passing libraries to multi-soft-cores and many-soft-cores. Table 2 presents information for the two most popular implementation of the MPI standard and some embedded MPI implementations. It is obvious that an adaptation is needed. OpenMPI and MPICH are not suitable for many-soft-core systems due to the prohibitive size of the libraries. Moreover, OpenMPI and MPICH support around 300 standard MPI primitives, which are very seldom used.

Table 36. Memory footprint of standard MPI implementations

	Availability	Library Size	Supported Functions
OpenMPI [Gabriel04]	Open	25 MB	300
MPICH [Gropp99]	Open	7 MB	300
TDM-MPI Saldana06]	Propietary	9 KB	11
SoCMPI [Mahr08]	Propietary	11-16 KB	6 - 18
RAMSoC-MPI [Göhringer10]	Propietary	37 KB	18

TMD-MPI, SoC-MPI and RAMPSoC-MPI implementations are lightweight solutions for embedded domain. SoC-MPI can be configured with up to 18 MPI functions using 16Kb of memory size for the library, while TMD-MPI uses 9Kb for 11 MPI functions. RAMPSoC-MPI has 18 MPI commands implemented, and uses 37Kb for the MPI library.

The drawback of these implementations is that they are proprietary and addressing specific systems. Hence, they are not easy portable to other platforms.

I re-implemented ocMPI as an open-source library available in https://github.com/davidcastells/ocmpi. It is an evolution of the work done in [Joven10] and [Fernandez14]. Unlike some other implementations of MPI, it can be ported to other platforms and architectures. OcMPI is similar to other embedded implementations in terms of number of supported primitives and library size. It has been developed in ANSI C to minimize the footprint of the library. It can be compiled for many processors that support the gcc tool-chain, like NIOS II, MicroBlaze, ARM, and x86. On ocMPI a minimal selection of standard MPI functions are selected. Table 37 shows the main functions of a minimal working configuration of ocMPI.

Using these functions many MPI applications can be developed and other more complex MPI functions (like collective communication primitives) can be implemented by invoking this simple ones.

Table 37. Minimal set of functions of ocMPI

Function	Description
`MPI_Init`	Initializes MPI execution environment
`MPI_Finalize`	Terminates MPI execution environment

MPI_Comm_rank	Determines the rank of the calling process in the communicator
MPI_Comm_size	Determines the size of the group associated with a communicator
MPI_Send	Performs a basic send (blocking send)
MPI_Recv	Performs a basic receive (blocking receive)
MPI_Wtime	Returns the time on the calling processor

MPI_Init and MPI_Finalize are management primitives. MPI_Init, sets up the MPI environment and, as in the homonymous standard MPI function, no other ocMPI function can appear previously, and MPI_Finalize finalizes the execution environment and no other ocMPI function can appear after MPI_Finalize is called. Additionally, MPI_Comm_Size and MPI_Comm_rank are management primitives. The relevance of those functions is explained in the following sections. MPI_Send and MPI_recv are point-to-point communication primitives that implement the basic blocking send/receive primitives. In section IV these functions can be seen with more details. Finally, MPI_Wtime primitive can be used for time measurement.

Process Identification

Within MPI environment a group of processors can exchange information through a communicator. There can be multiple communicators involving different processors. For simplicity ocMPI only supports one communicator involving all the processors of the system, known as COMM_WORLD in MPI terminology.

Each processor can be source and/or destination of messages. Processors are identified within a communicator using a unique variable called rank. In MPI context, each processor can know its own rank and the number of processors that participate in the communicator, which is denoted as size. This knowledge is obtained by invoking the MPI_Comm_size and MPI_Comm_rank functions respectively.

The way of determine the mapping of the ranks in the system is free to the MPI library implementer, so it can be done in different ways. The determination of the rank of the processors can be done statically at compile time, or dynamically at runtime. In case a dynamic approach is used, a manager entity is needed to centralize the assignment of ranks to each processor.

In any case a processor identifier is needed. Since the rank is a logic identifier that could be distributed during runtime, we need an invariable physical identifier for each

processor. The NIOSII processor has a bank of control registers, register number 5 (*cpuid*) was specially created for this mission. In this case a simple C implementation of `MPI_Comm_rank` can just call the macro `NIOS2_READ_CPUID(x)` to obtain its value.

However, this is not the only physical processor identifier we could have. If the many-soft-core contains a NOC with NAs, having a hardcoded physical address, that physical network address could be used as physical processor identifier.

A typical example of this approach would be a many-soft-core system with a 2D mesh NOC with NAs having hardcoded network addresses. In this case, a certain processor would become the Master (rank 0), could be in charge to set the rank of each other processor in the environment by sending network packets to other processors during the initialization phase started with the invocation of `MPI_Init`..

Message Delivery

The most basic communication primitives of MPI that allow to exchange information between two processors are `MPI_Send` and `MPI_Recv`. When using `MPI_Send` the rank of the destination process must always be informed. However, when using `MPI_Recv` the rank of the source process is not mandatory, one can inform the parameter `ANY_SOURCE` if it is interested in receiving any packet regardless of its originator. From the programmer point of view, the memory architecture of the system is completely transparent when using ocMPI and it's the ocMPI implementation itself that carry out the task of using the properties of the system in the most efficient way. Therefore, depending of the memory architecture the inner implementation of the communication library will be different.

Two are the main options for the memory architecture of the many-soft-core:.

ocMPI for shared memory architectures

Several options appear when trying to implement ocMPI over shared memory architecture; however these options can be gather in two main options: implement a single queue where the communication must happen, or create several queues.

When a single queue is created the writers leave the messages on the first free position available on the queue and, when receive is posted, the reader analyzes the headers of the messages to find the whished message. This implementation can be an option when the application performed is not intensive in communication or in messages.

When implementing several queues, three options appear. The first one, implementing a single queue for each receiver, in that case, the sender's access to the specific queue of the receiver to leave the message, and it's the duty of the receiver to analyze the queue and find the message wanted. In the case of receiving from `ANY_SOURCE` the complexity is similar because it is the same queue that has to be analyzed.

The second option is to implement a queue, not in the receivers but, in the senders. Using that solution, the sender writes always on the same queue and it is the receiver that access different queues depending on the source of the message, and, also, searches for the wanted message or messages. In the case of receiving from `ANY_SOURCE` the receiver has to analyze all the queues from all the senders, which makes it quite an impractical option.

Finally, the third option is to create a matrix of queues where each couple of sender/receiver has a dedicated queue. In that solution, it is not necessary to analyze the messages to find the wanted one/ones, since the use of such queues is deterministic and, therefore, all the messages inside a queue have the same source and destination. In the case of receiving from `ANY_SOURCE` the receiver has to analyze just the queues of the matrix where it is the destination.

Regarding the method chosen, all the queues must be accessed from different points at the same time. That implies that all the queues must be synchronized and protected. That can be done using both, software or hardware solutions. For software solutions a mutual exclusion algorithm can be implemented, such as a Lamport's bakery-based algorithm [Lamport74]. For hardware solutions, a hardware mutex can be used. NIOS-II tools already includes a Hardware mutex IP-Core.

In addition, the implementer has to decide what to post to the queues. Either pointer to messages or complete messages. The ocMPI implementation only post messages pointers into the queues, and bypasses the data cache by using direct I/O processor instructions such as *ldwio* and *stwio*.

ocMPI for distributed memory architectures

For distributed memory architectures the implementation of `MPI_Send` and `MPI_Recv` functions use the network to transmit messages, therefore, it is not necessary to implement any additional communication mechanism as happened for shared memory architectures.

However, it is necessary to implement a delivery protocol to synchronize the sender and the receiver. This may be expressed as a choice between two different options: using eager protocol or using rendezvous protocol [Rashti08].

In eager protocol, the sender sends messages regardless of the state of the receiver. Therefore, implementing eager protocol requires also the implementation of a software transport layer at the receiver point to be able to multiplex several streams from several sources.

Eager protocol could achieve better results in terms of performance in some applications, however, since messages are sent regardless of the state of the receiver, incoming messages whose `MPI_Recv` has not been invoked must be stored within a buffer at the destination until a receive call is posted by the MPI application. Such situation becomes worst in a multipoint communication scenario where this protocol has a great probability of going out of order at the delivery of the messages emitted from several sources. This behavior implies the need of several buffers in the destination nodes to order the messages coming from each source.

The need of buffers in the reception nodes creates an overhead at managing the memory copies, which implies penalizations in execution time due to these copies, and in extra memory space request for the implementation of such buffers, which are a serious handicap for the usual lack of memory resources of distributed many-soft-core systems.

In rendezvous protocol, there is a global flow control synchronizing all the system. The use of this protocol solves the problem of needing extra buffer space to copy incoming messages in the reception node, and also removes the extra time required to manage the copies and organize the source of incoming messages because these messages are now arriving in order. However, rendezvous protocol introduces its own overhead. This overhead is produced by the short signaling messages that must be sent to synchronize the supply and demand of messages. These signaling messages require time to be generated in the transfer node and to be processed in the receiver node, in the software layer. Moreover, this time required by the software level is increased by the time spent to transfer the message through the network, resulting a relevant total latency. If the data packets are small, the overhead is considerable, because the creation and delivery of small signaling messages add several software instructions for a rather simple process.

Initialization

MPI usually is described as a programming model adapted to the Single Program Multiple Data computer architecture. In order to increase productivity it is desirable to have a single program that is distributed to all the processors for its execution. We started with the assumption that the many-soft-cores that we describe execute a single application which must be loaded from flash.

This was a reasonable assumption because Altera already provides support to execute an application stored in flash on its NIOSII processor. The way Altera supports this is a little tricky. To be able to boot from Flash the processor has to contain an EPCS controller attached to the bus. Then, the programmer has to instruct that the reset address of the processor is in the EPCS device. Actually what happens is that a small memory is inserted into the system and the actual reset address of the processor is pointing there. That memory contains a boot-loader that, using the EPCS device, loads the program from flash and then jumps to its initial starting address.

This boot process cannot be easily generalized to all the processors in a many-soft-core. First because it is usually only one processor that has access to the EPCS controller. Second because the compiled program knows the address of the stack pointer, and if all processors load the same program the stack pointer will be the same and all processors will collide their stacks.

If only one processor (let's call it master) has access to the EPCS controller it must be responsible to transfer the program to the rest of the processors and then make them boot. In the case of a shared memory architecture there is no need to transfer the program because the slave processors can directly access to the code memory that has already been loaded by the master processor. In this case the slave processors have to modify the Stack Pointer to avoid collisions with other processors. In fact the algorithm to assign different stack pointer to the processors must be embedded into the application initialization function, and must use a physical identifier of the processor such as the *cpuid* control register.

In addition, the master processor has to control the reset of all the slaves to be able to start their execution once the program has been loaded into memory.

On distributed architectures there is no risk of stack collisions, so stack pointer modification could be possibly avoided. However the program is no longer accessible from the slave processors, so it must be transferred to them through networking primitives. This means that slave processors need to have a small boot-loader

implemented in fixed memory (like the EPCS boot-loader provided by Altera). But in this case the slave boot-loader has to wait for the master to send the program by using networking primitives and then transfer execution to its starting address.

4.4.2. Multithreading Support

On shared memory many-soft-cores multithreading is possible. The architectures that I can build do not support cache coherency. Multithreading code assumes that shared memory is consistently viewer by all processors. On the absence of cache coherency an alternate method to support multithreaded is by using un-cacheable regions should be used like proposed in [Mattson10]. If used indiscriminately that would limit the scalability of the application. It should only be used on shared variables.

Many-soft-cores are determined by having a large number of processors. High performance applications tend to benefit from each processor when there is a mapping between one thread per processor. If only one thread per processor is provided, this simplifies the implementation because no preemption must be implemented. In addition, it also minimizes the use of memory because only a stack per processor must be maintained.

The implementation is very simple, just the three functions listed in Table 38 are provided. When the system starts each slave processors execute a boot-loader, which initializes the stack pointer and waits until a thread is assigned to the processor. When `thread_create` is called it is assigned to the first idle processor and its state is change to START. When the boot-loader detects the START state it changes the state to RUNNING and actually calls the thread function. When the thread function invokes the `thread_exit` function the thread state is changed to DEAD. The thread states are illustrated in Figure 101.

Table 38 thread functions

`thread_create`	Creates a thread that will run the provided function in an idle processor.
`thread_exit`	Terminates a thread
`thread_join`	Waits until a thread has finished

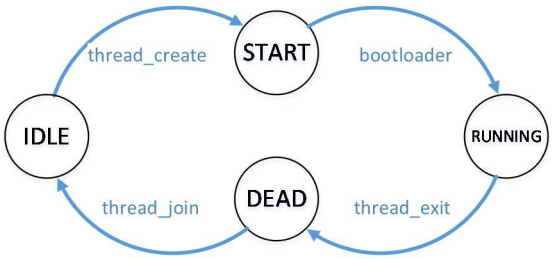


Figure 101 Thread states

Figure 102 Memory organization for Multithreading support in a NIOS multi-soft-core with 2 processors

In a multi-soft-core system I always assign a special role to CPU0. It is the CPU the boots from the standard reset address in main memory as provided by the Altera toolchain. The rest of the CPUs boot from a small separate ROM memory containing the boot-loader. The master is responsible to determine the stack organization of the system and it is the only allowed to allocate memory in the global heap. After boot all the slaves can execute functions from the Code section if they are instructed to do so. Hence, no code is replicated, heap space is global and data from any region in memory can be shared among all processors. Figure 102 depicts the memory organization of a multithreaded system with two processors.

4.5. Proposals of Novel Performance Estimation Methods

Compared with HPC systems or generic desktop multicore processors many-soft-core processors could offer more visibility of the system state, since software can be debugged and analyzed with similar techniques and hardware details can also be observed through embedded signal analyzers and event specific hardware can be created to increase the visibility of certain parts of the system. But in practice this is not the case due to several factors. The typical limited resources available to reconfigurable systems is a drawback. Trace analysis in HPC systems often involves the generation of trace files of several Giga bytes, and this usually over the storage capacity found in many reconfigurable systems. Moreover, this amount of information has to be transmitted to a host computer for its analysis, the communication link between the host computer and the

reconfigurable system can easily become a bottleneck. Additionally, although deeper lower level analysis can be performed in reconfigurable platforms, it usually involves the re-synthesis of the designs, which is usually a time consuming process that make it impractical as a method to use frequently.

Binary compatibility is another aspect that allows faster iterations in HPC than in reconfigurable systems. Applications for HPC systems can be compiled and executed in several platforms. A development team can implement applications on several desktop computers to validate functional requirements and measure performance and test them further on remote systems. This binary compatibility is not found in reconfigurable systems. The design has to be tested on the specific platform for that it was build. When working with a development team, the access to the platform can become a bottleneck in itself. This problem can be addressed by building a model of the system that can be executed on the host machine.

4.5.1. Performance estimation from a communication centric perspective

Given that the processors that constitute a many-soft-core must cooperate to achieve a certain task, they must exchange information either by sharing variables in memory, or by exchanging messages. The latency and bandwidth of these communications can determine the total performance of the system, and can easily become a bottleneck that limits application scalability. Either shared memory or messaging is used, Networks-on-chip are often behind the hardware infrastructure needed for this communication to happen. The efficiency of a NOC design can be greatly influenced by the traffic pattern exhibited by the application using it. So before creating it it would good to test it with the expected traffic patterns, so that changes can be made on the topology, switching scheme, flow control, buffering, packet size, or other possible parameter.

With this goal in mind I contributed to create j2empi [Joven09], an add-on software to NocMaker that allow to use a MPI-like application description to get performance information of the underlying network. The whole process workflow is depicted in Figure 103.

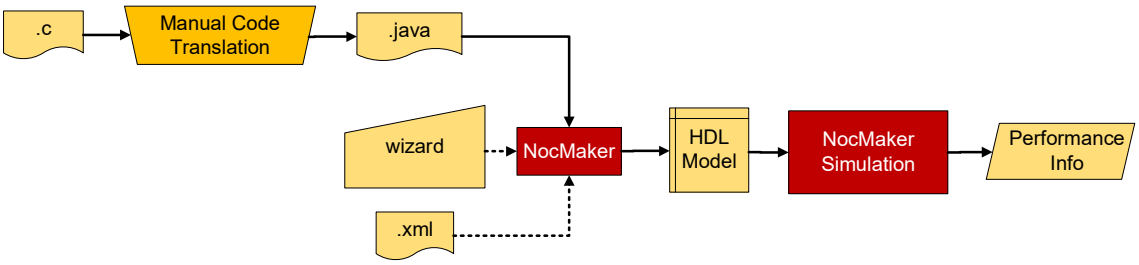


Figure 103 j2empi workflow

A C MPI application has to be manually translated to Java using the MPI primitives implemented by *j2empi* (see Table 39). Then the application can be used as a communication pattern generator for any NOC design described in NocMaker. As explained in chapter 2, NocMaker builds an HDL model of the NOC and the NAs, but uses an abstract functional model for processors. Processing time in processors can be defined by explicitly annotating the *j2empi* application with primitives to consume cycles of the abstract processors. Otherwise, all computing between two interactions with the NAs are virtually executed in just one cycle. NocMaker already provides some visualization of the network performance.

Table 39 Supported MPI functions in j2empi

Types of MPI functions	Ported MPI functions
Management	`MPI_Init`, `MPI_Finalize`, `MPI_Finalized`, `MPI_Initialized`, `MPI_Comm_size`, `MPI_Comm_rank`, `MPI_Get_processor_name`, `MPI_Get_version`
Profiling	`MPI_Wtick`, `MPI_Wtime`
Point-to-point communication	`MPI_Send`, `MPI_Recv`, `MPI_SendRecv`
Collective communication	`MPI_Broadcast`, `MPI_Gather`, `MPI_Scatter`, `MPI_Barrier` `MPI_Reduce`, `MPI_Scan`

To illustrate the methodology a simple MPI mandelbrot application is translated from C to Java using j2empi and used as a traffic pattern generator to evaluate a simple 4x4 mesh network. The simulation infrastructure can be interactively controlled to advance the clock an arbitrary number of cycles (see Figure 104). The control flow of the links is illustrated by a color code, and established connections in routers are also shown. It is assumed that every processor only runs a thread, so only a MPI operation can be executed on each processor at any instant on time. Aggregated performance information is also presented, for each communication stack layer.

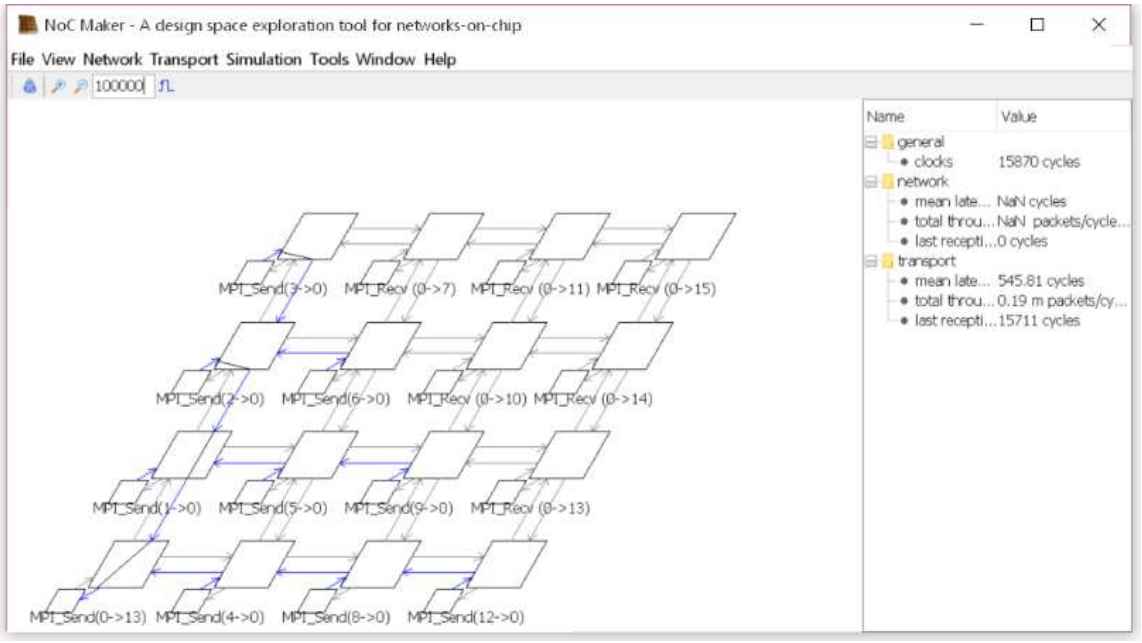


Figure 104 NocMaker visualizer that interactive shows how networks packets progress through the network, and collects performance information

NocMaker can also visualize a time line with the transport layer packets being transmitted. Figure 105 shows the time line for the former mandelbrot application. Notice that small packets show a smaller latency while bigger packets have bigger latency.

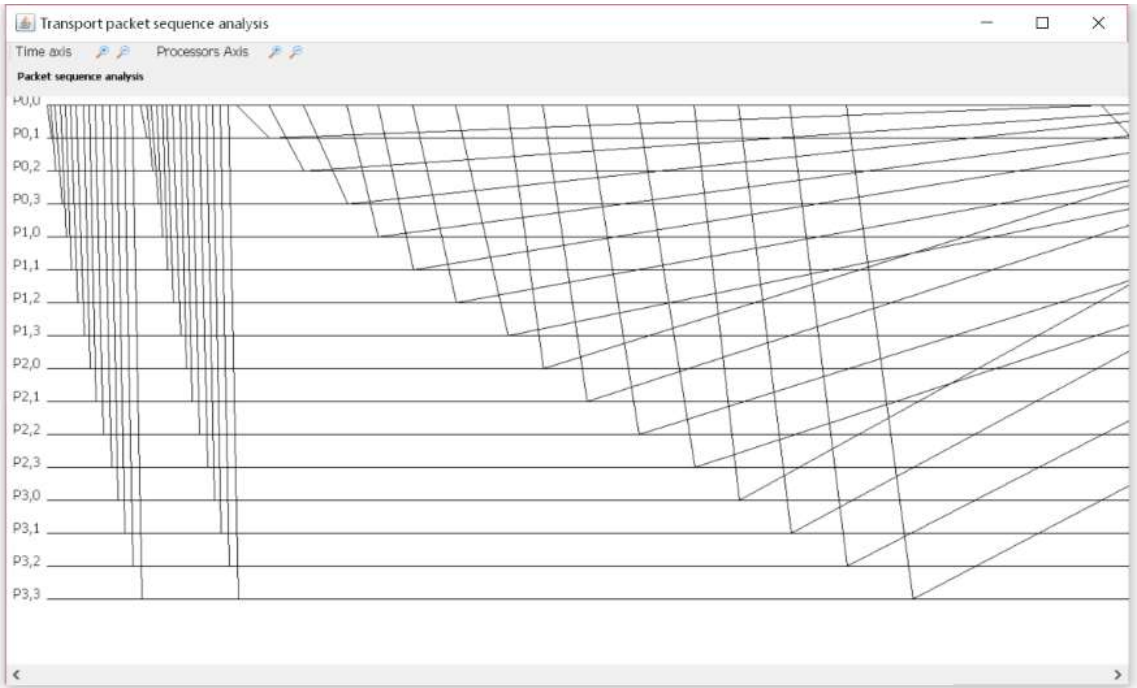


Figure 105 NocMaker transport packet visualizer

4.5.2. Transparent instrumentation in Virtual Prototyping Platforms

Virtual prototyping platforms can be used to do functional validation or performance estimation of applications targeting embedded systems. This is a very popular method to test applications for embedded systems and mobile devices. For mobile devices they are usually called emulators [AndroidEmu].

Virtual prototypes use a *host* computer that runs a virtual machine supporting the Instruction Set of the target system. The hardware resources of the platform are simulated and they are potentially implemented using the host computer features. In mobile device emulators, for instance, the wireless internet access of the mobile devices is provided by using the Internet connectivity of the host. Similarly the internal SD-Card storage is implemented as a subdirectory of the host machine.

When virtual prototypes are used for functional validation the time accuracy is usually ignored. In this case the performance of the emulation of platform can be maximized by using techniques that minimize the time used to execute the instructions of the target machine on the host. The most successful technique is dynamic binary translation [Bellard05], where the executable is dynamically translated from target ISA to host ISA as it is executed. Since the the architecture of the host can be totally different than the target, the time characteristics of the application using this technique are completely altered.

Performance estimation of many-soft-core applications requires time accuracy, so typical binary translation is usually discarded. An alternative is to complement binary translation with a performance model of the target platform like done in [Böhm10]. Another alternative is to work with Instruction Set Simulators (ISSs) that offer a better time accuracy with some time penalty.

Trace generation in ISS

An ISS takes the executable generated for the target system together with a model of the hardware present on the target system, and then interprets every instruction of the target application and emulates its behavior considering the hardware model, as shown in Figure 106. Interpret based ISSs are slower than compiled ISSs but they have been used to simulate multi-core processors, like in [Silvano11]. However the kind of performance analysis done when using this simulators are often very coarse grain.

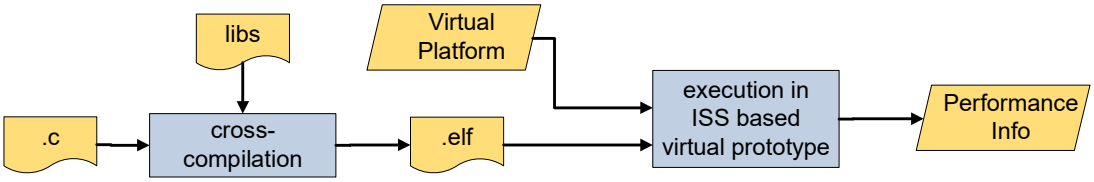


Figure 106 performance analysis in ISS-based virtual prototypes

In [Castells10] I proposed to implement transparent instrumentation in ISSs to obtain much more detailed performance information of a multi-soft-core system. I combine a detailed HDL model of the hardware system together with multiple ISSs. A diagram of the architecture is depicted in Figure 107.

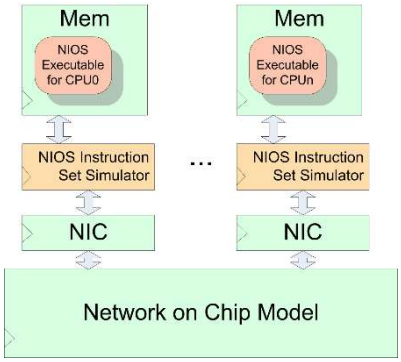


Figure 107 Logic design of a multi-soft-core virtual prototype

The multiple ISSs intercept the execution of certain instructions of the target ISA to automatically produce traces. By transferring the responsibility to inject traces to the ISS the time characteristics of the application under test is unaltered. Typically intercepting the execution of the call and ret instructions would be equivalent to what compiler automatically instrumentation do. Figure 108 shows how the transparent ISS-based instrumentation is performed.

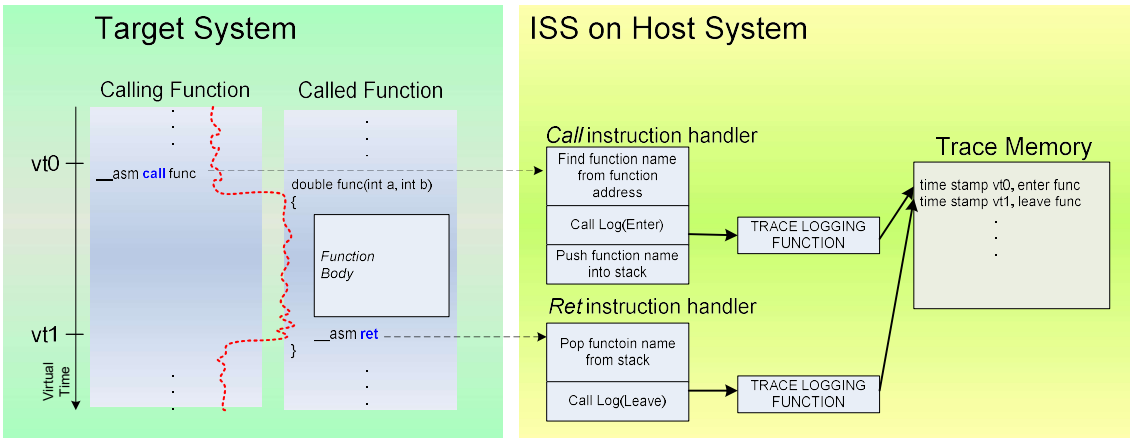


Figure 108 transparent instrumentation method

The benefits of using transparent instrumentation in virtual prototypes are that we can overcome two of the typical issues faced in performance analysis. One, the potential excessive overhead caused by instrumentation. Here it is totally eliminated because we spend *host* time (instead of *target* time) to produce the traces. Two, we are not constrained by the limited memory or communication resources of the *target* system. So we can generate huge trace files into the host hard disk for later analysis.

Those trace files can be later visualized with tools like Vampir and after human analysis the application bottlenecks can be found and solved to get new optimized versions of the applications. In the example depicted in Figure 109 we show how the visualization of the event traces shed light about the inefficiencies of the communication primitives used by implementation of a parallel version of the Mandelbrot application. The orange and pink color bars show the periods of time spent in communication libraries. In this case the slave processors are wasting a lot of time in waiting for new messages, while the master processor is busy composing new messages through the communication stack (light blue color). Hence, no computation overlap is happening and very low parallel efficiency is achieved. After the optimization of the communication primitives, computation is overlapping in slave processors and a much higher parallel efficiency is achieved.

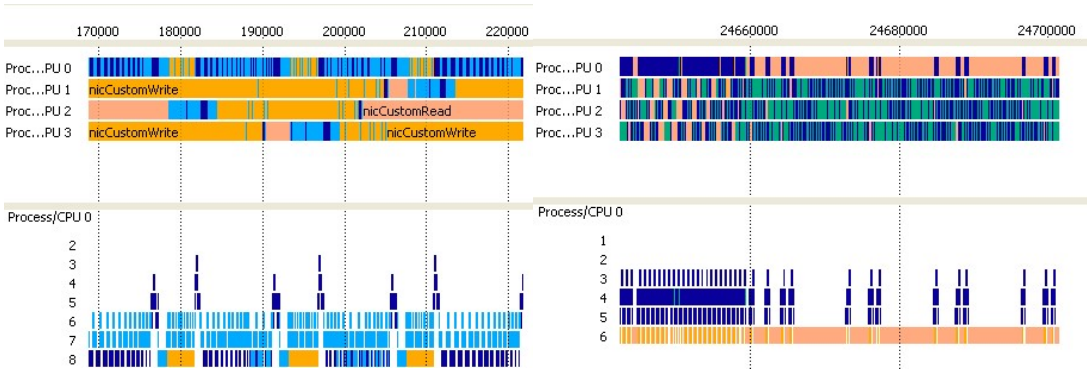


Figure 109 Performance optimization process of a parallel Mandelbrot application. Left) trace visualization of the initial version. Right) trace visualization of the optimized version

The concept of transparent instrumentation from the ISS can also be applied to obtain other kind of information from the processor. In [Hubert07] authors describe memtrace, an extension to the ARM ISS ARMulator that allows capturing the memory accesses of an application.

Trace generation in Native Simulation

A different type of virtual platforms can be build using native simulation. In native simulation, the application source code is annotated with the time and/or energy that a piece of code is estimated to consume. During execution, the annotations are aggregated enabling to estimate the execution time and power consumption of the complete simulated system. It is possible to use manual annotation, or automatic annotation. The annotation resolution can be coarse grain (function level, for instance) or fine grain (instruction level). Obviously fine grain annotations will be more accurate.

All the process is done in the host computer where the simulation is performed (see Figure 110). Virtual platforms based on native simulation allow the embedded software development process to be started even before the HW platform is completely defined because a limited number of high-level HW-platform parameters are needed.

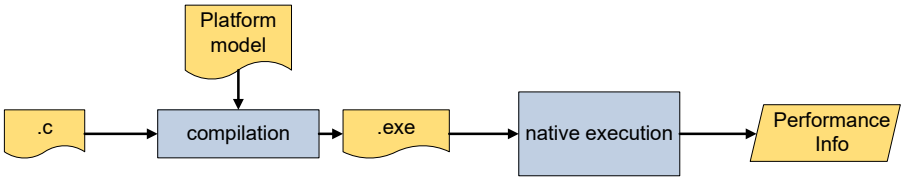


Figure 110 Process of native execution

In [Castells14] I propose to extend the ScoPE [Posadas11] native simulation framework to introduce transparent instrumentation. ScoPE supports modeling parallel systems, especially addressing shared memory architectures with OpenMP support. In ScoPE the annotation of the code with the performance details is done during compilation. Given a platform model, the compiler analyzes the instructions to execute and automatically inserts time and power annotations. Determining the time required to execute an instruction and the source code level is challenging since compilation can optimize the code with different techniques producing different codes with different time and power consumption characteristics. To estimate this value SCoPE automatically compiles the code to target ISA and analyze the resulting machine instructions with a time and power model of every instruction of the ISA. SCoPE also models the cache memories of the target system statistically to determine the time consumed by memory access instructions.

In SCoPE, virtual time is advanced by the annotations introduced in the application code during the compilation flow. If a section of code is not annotated with this time information it is eventually seen as happening in a zero time delta.

I benefit from this property by introducing trace generation calls at function prolog and epilog but avoiding to complement these tracing calls with time annotations.

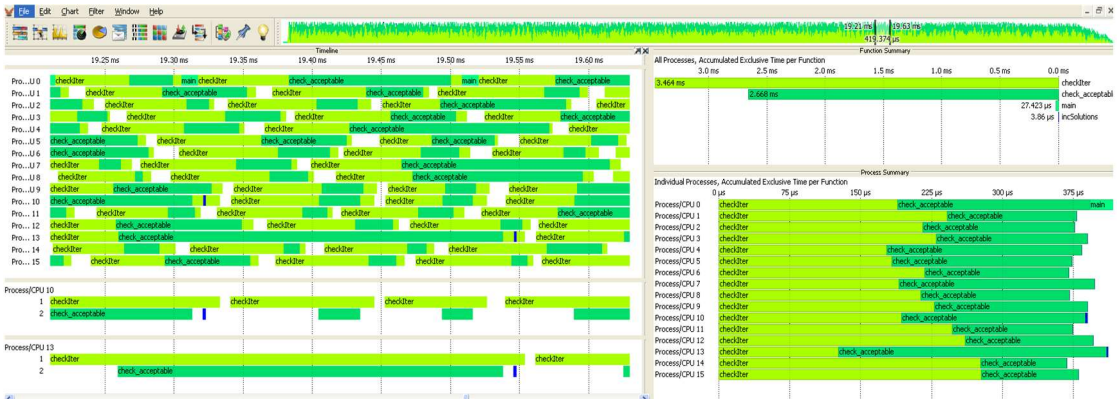


Figure 111 Trace visualization of n-queens application on a 16-core many-core processor virtual platform

To test the system I use two applications. The first one is n-queens, a simple application which computes the different solutions of placing a number of n queens in a chess board of n by n tiles in such a way that they not threaten each other.

This application is not a typical embedded workload but it is convenient because it can be simply parallelized by using simple OpenMP pragma directives and does not require accessing external files through I/O interfaces.

We test the execution of the application with n=5 in different virtual platforms with different number of cores. The used cores are equivalent to arm926tnc as they support its instruction set. Trace generation adds a small overhead when flushing trace data to disk, but in all virtual platforms the execution of the application takes less than one second of the host time. In terms of target time the application execution time depends on the virtual platform in use. For instance in 16-core platform it takes 27 ms to execute. But this time is no effected at all by the time devoted to trace logging since this only consumes host time.

Figure 111 shows how trace visualization tools present the collected information. On the top right hand side of the picture we can find a report of the execution time by application function. This is the kind of information that we could get from a profiler. The left top panel shows the dynamics of the application and all its threads. Below we can show how the call-stack of each thread is progressing. In the example we just show the call-stack from processor 10 and 13.

The second application analyzed is a JPEG encoder, a typical embedded workload. There are several implementations of the image compression algorithm. I have used [PJPEGENC]. Some operations like color conversion, DCT transform, and quantization can be performed in parallel since there is no data dependency among different blocks of the image. However Huffman encoding must be serialized because of several data dependencies of the algorithm.

Table 40. Execution time for JPEG on 16-core virtual platform

	Host	**Target**
Platform	Intel XEON E5620	Virtual arm926tnc
Cores	4	16
Clock Frequency	2.40 GHz	470 Mhz
Execution Time	18ms	26ms

As in the previous example, different Hardware configurations are tested with the same application. The visualization of the traces (as shown in Figure 112) give a clear idea of why the application does not scale well above a certain number of cores (typically 5 as explained in [Castells10b]). After the parallelization of the first loop the sequential part of the algorithm (Huffman coding, and I/O) dominates.

In this case, the simulation of the application running on a 16-core target system is executed in 18ms, in host's system time (see Table 40). Thus, target expected execution time is bigger than the simulation time, even with the overhead of trace generation.

Figure 112 Visualization of the traces generated for the JPEG application in a 16-core many-core

4.6. Conclusions

The materialization of many-soft-core hardware architectures is supported by FPGA vendors in a much higher degree than is the software development for them. Their strategic investment to address the HPC market is centered in OpenCL based compilation toolchains that allow creating application specific designs. Thus, the result is that the programmability of multiprocessors using soft-cores is not equivalent to that of a typical multiprocessor system. Their main limitations are: 1) reduced visibility and controllability of cross-compilation tool-chains which are very common in the embedded domain and 2) lack of SPMD programming model. Instead, they promote a MPMD model, which require implementing as many applications as processors.

To circumvent this limitation, I implemented the support for MPI and threads, making possible to execute a single program on all replicated processors. Since I can create different types of architectures, like UMA and NORMA, the implementation of the MPI runtime makes use of the features of the underlying platform. On the other hand, threads are only supported on shared memory architectures. Nevertheless, their support is an intermediate step towards a future support for OpenMP as the de-facto standard for shared-memory parallel programming.

Another roadblock for parallel programming is the lack of effective performance analysis methods. FPGA vendors support either low-level or simple performance analysis tools, like embedded logic analyzers or profilers. However, for successful optimization of parallel applications, a more detailed analysis is required.

I added support for trace generation so that post-mortem trace analysis using HPC tools (like Vampir) can be performed, and the code can be iteratively improved until it reaches the required scalability level. Trace logs can be generated in real platforms, but they can also be produced by virtual platforms. I presented a novel technique (Transparent Instrumentation) to generate traces in virtual platforms without modifying the time characteristics of applications.

Finally I complemented the tool-chain with a data access-pattern analysis tool, which used on sequential code (prior to parallelization) provides better understanding about its parallelism potential.

5. Case Studies

On this chapter I will present some case studies that demonstrate the techniques presented in former chapters.

5.1. Laser Controller

In [Castells12] I presented a case study in which several methods are combined to implement an industrial laser controller system. Laser processing is increasingly used in the manufacturing and packaging of several goods. Faster systems are always in demand to increase the productivity of manufacturing plants without compromising the quality of the processes. In many systems, the laser light beam is continuously irradiating but a mechanical system is used to block or divert the beam to apply it in a proper way over the treated surfaces to obtain the desired results. These mechanical parts often become the bottleneck of such systems, preventing them to increasing their productivity. As a result, the laser beam is typically blocked during 75% of the time.

A way to increase laser marker systems productivity is parallelism. Multiple laser markers could be replicated to multiply the number of processing units per minute. However, the laser light emitter is one of the most expensive parts of the system, and it seems unwise to replicate it when it is already heavily underutilized. A better solution is to replicate the final optical control system and time multiplex the laser light source as presented in [Vila12]. This was the principle of the system implemented in [Castells12].

The design of an embedded system to control such machine imposed some challenging requirements such as the hard real time control of the different heads while managing a shared resource (the laser beam) in a synchronized way. The need to implement fast motion control loops is not the only challenging part that motivates the use of reconfigurable hardware. It is also the very accurate control of the laser activation and the adaptation to different galvanometer control protocols such as the XY2-100 and SL2-100. The galvanometer control protocols are based on serial transmission with a 2 MHz clock, and require to periodically transmit the position values for every axis every 10μs. The laser activation must be controlled at the microsecond resolution.

While the control of a single head itself is already too demanding to be implemented by a single microcontroller, the situation is worse when it comes to control four heads simultaneously.

The laser marking system is used to mark cork stoppers and is based on rotary systems that make the cork stoppers rotate while a laser beam is irradiated on their surface depending on the design to be marked. The laser beam is shared among multiple heads to increase the utilization factor of the beam as shown in Figure 113.

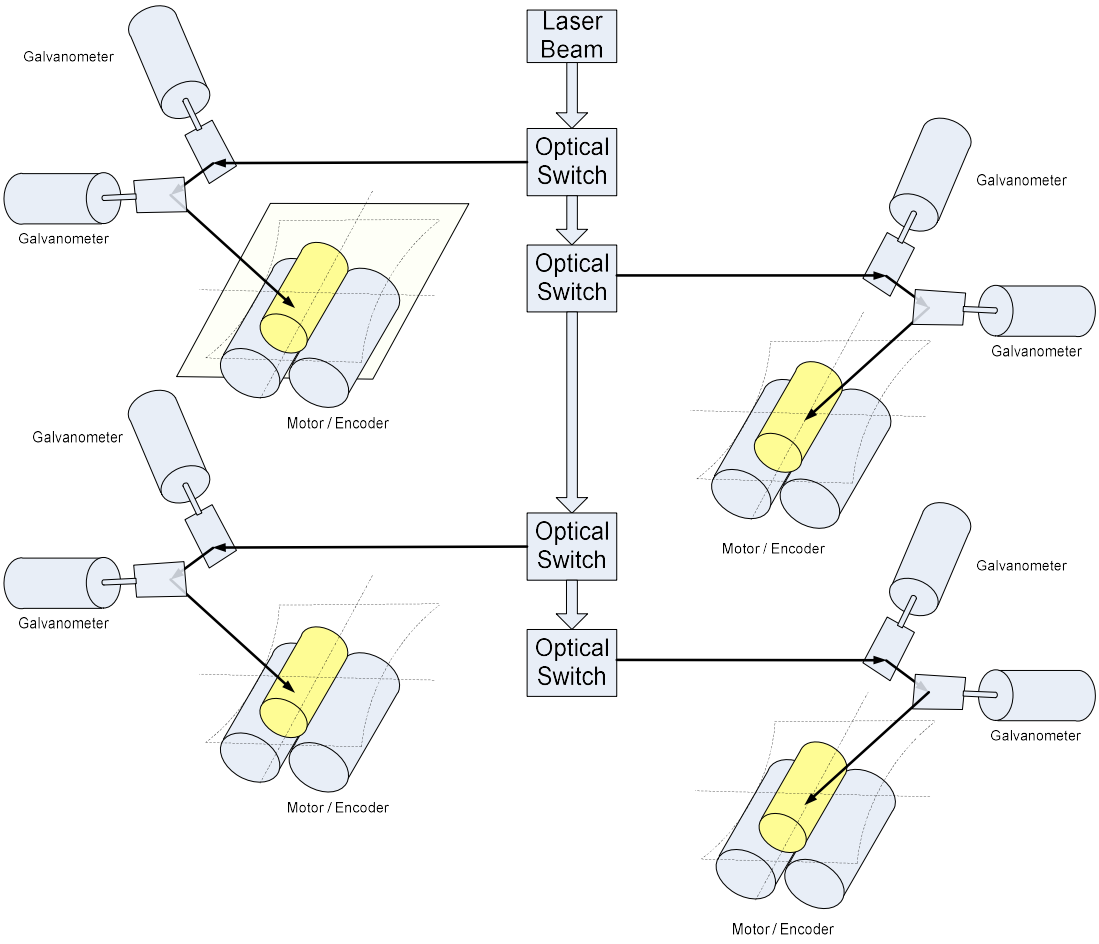


Figure 113 Laser Marking Machine Diagram

In each head, the beam is diverted by two galvanometers that make it possible to cover a 2D surface. In order to be able to mark all the surface of the cork stopper the stopper is rotated by a motor. Due to the slow response of the motor an encoder is used to give feedback of the actual position of the motor rotator.

The laser beam is shared among the different marking heads. The idea is that the different heads cooperate to time multiplex the usage of the beam. So, in principle, only a single head would be using the beam at a given time. The diversion of the beam to the appropriate head is controlled by some optical switches.

Marking is achieved by pulsing the laser during a short period of time. Although it is variable, depending on different factors the pulse period to mark a black dot can be

around 40us. Shorter periods produce lighter grey dots. This is used surpass the limitation of marking black and white images and allow to mark grey scale images.

I decided to build a multi-soft-core system based on NIOSII processor and prototype it in a low-cost Terasic DE0 Nano board (see Figure 114), which is connected to another PCB that interfaces with the sensors and actuators of the industrial machine. The communication with an external computer is done through USB.

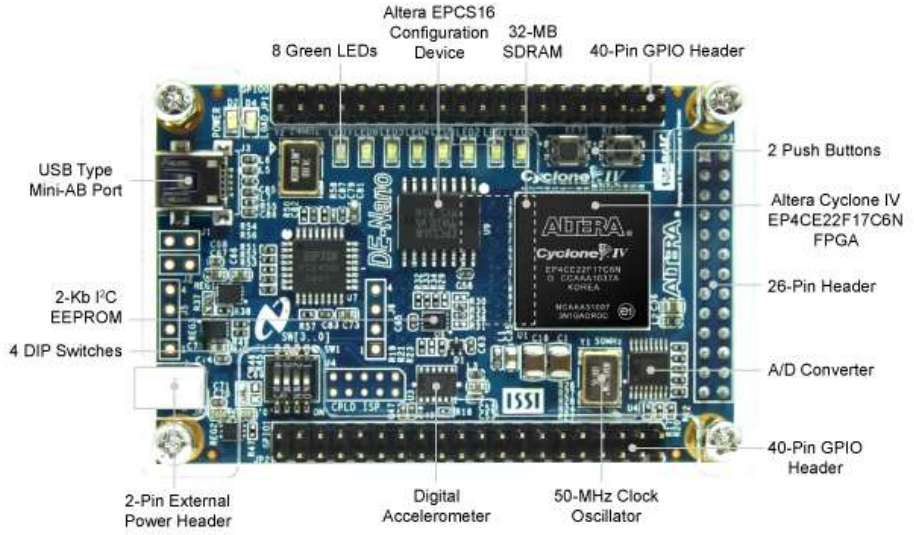


Figure 114 Terasic DE0-nano board

5.1.1. Design of the Real-Time critical parts

I started with analyzing the hard real time requirements of the system. The most sensible part of the system is the control of the laser pulse. This is a safety critical requirement, because if the pulse is too long it is easy to set fire to the system. A 40µs pulse produces a black dot, and shorter pulses produce lighter colors. 8 bits are used to represent the greyscale of a pixel in an image, so the pulse period is divided by 2^8 to obtain the resolution at which pulse period should be varied to produce grey scale colors. In this case a difference of 156.25 ns in pulse length would produce a different color level from the possible 256.

It could be challenging to control those short periods of time by an embedded processor running a RTOS for just a head, it's even worst considering that four simultaneous heads could be active. Popular RTOS have a latency of some µs [Hambarde14].

To address that challenge, I created a hardware unit to create a pulse and attach it to the processor by extending the instruction set of the processor. In contrast with a bus

attached device, the custom instruction interface ensured that no extra latency would be inserted due to bus arbitration.

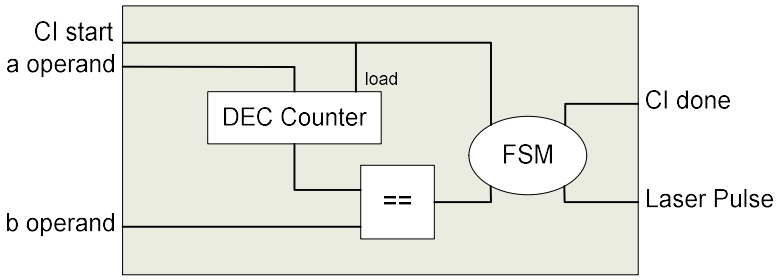


Figure 115 Pulse Generation Custom Instruction

The Laser pulse was controlled by a 32 bit decrement counter that decrements at every clock cycle. At first the pulse is asserted, and when the counter value reaches a specified value the pulse is de-asserted. The design is depicted in Figure 115.

Moreover, the custom instruction was implemented in an asynchronous fashion with the goal to avoid blocking, if possible, to let the processor progress with computing when performing a pulse operation. In case the unit is busy and several pulse operations are performed, the unit becomes blocked until it finishes the operation (see Figure 116). For instance, this allows calculating new pixel coordinates during the pulse generation.

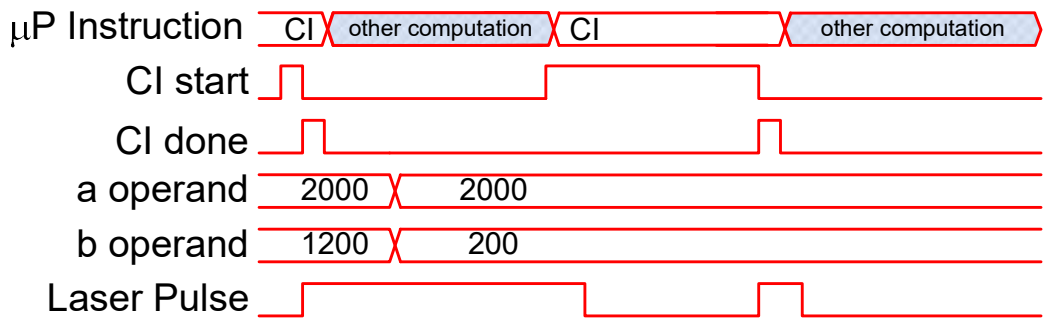


Figure 116 Asynchronous Custom Instruction Timing. First CI starts immediately and the processor thinks that it has been completed (as done signal is asserted). Actually the operation is still taking place, since it has to complete the 2000 period informed in the operation parameters. So when the second CI is invoked it must wait until the previous operation finishes to start its processing. This allow the processor to work on other computation between consecutive CI invocations.

Simultaneously to the laser activation, the laser beam has to be diverted by an external galvanometer system, which is controlled through the XY2-100 serial protocol. This protocol enforces to continuously update the beam position, and the serial protocol has to drive a clock signal of 2 MHz.

Another Custom Instruction controls the position of the heads through the galvanometers. The heads are controlled by commercial drivers which are controlled by the XY2-100 communication protocol. The instruction updates some internal registers

and a FSM is responsible to continually send the values of X and Y to the galvanometers. The design is depicted in Figure 117.

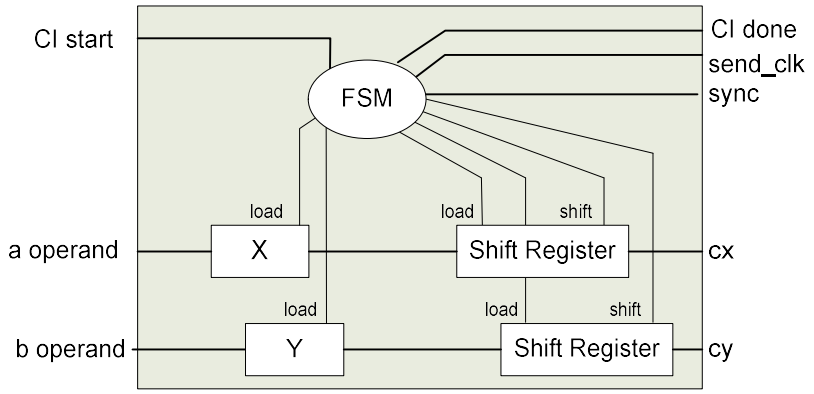


Figure 117 Position Control Custom Instruction

The motor has a slow response, and initially it was controlled by a parallel I/O. However an accurate control of the actual position is needed to ensure a good quality of the marking. So a feedback loop was created by the use of an encoder. The encoder can give information about the actual position of the motor by reading the value of marks into a rotating disk. There are two concentric set of marks A and B, that can be interpreted as a continuous digital sequence 00, 01, 11, 10. To convert the encoder readings to a certain position one can periodically sample the values A and B and when observing a change determine if the position was increased or decreased depending on the direction of the sequence. For instance, if the last reading was a 01 and the new reading is 11 it can be determined that the position was increased, otherwise if the new reading is 00 position was decreased.

A position decoder can be designed with simple logic that stores the previous state and use so simple combinational circuit to compares it with current one and provide two signals to increment or decrement a counter that will store the position into a register (as shown in Figure 118). This register is used by the software to correct the coordinates sent to the galvanometers which have a faster response. Following this approach, marking can be performed simultaneously to rotation, increasing the speed of the marking process.

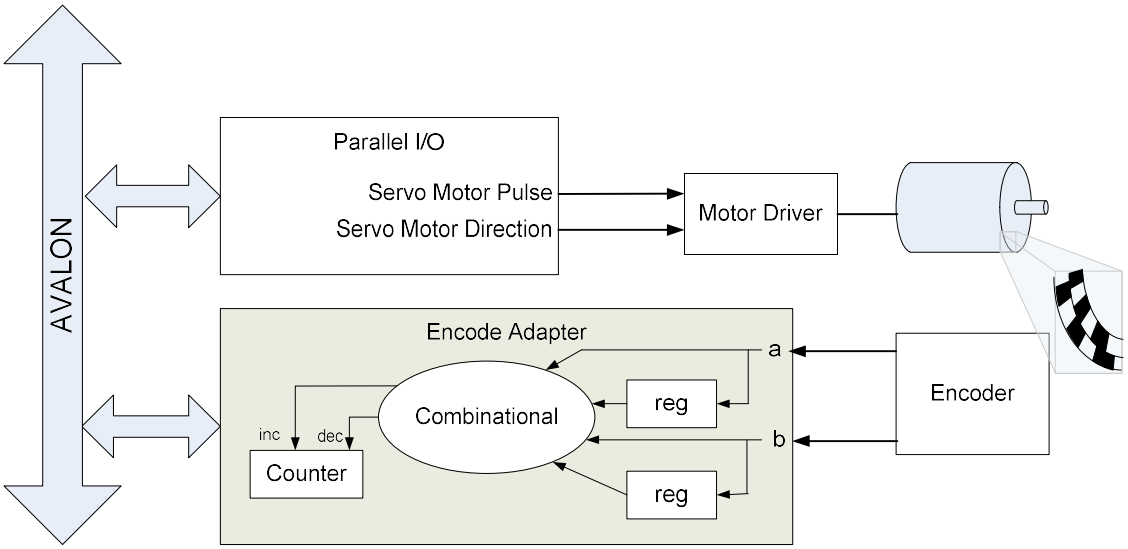


Figure 118 Motor control elements

5.1.2. Multi-soft-Core design

The controller had to perform 4 tasks that cooperate by using a multiplexed resource. Each task was isolated in a processor. A processor was, in fact, part of a tile that includes the necessary specific hardware to control the laser head. The full system consisted on four replicated tiles to control the four independent heads. In addition to the encoder adapter and the custom instructions each tile has some additional modules. There were some inputs to be able to handle various alarms. A 64 bit performance counter was used to get accurate time measurements during the operation.

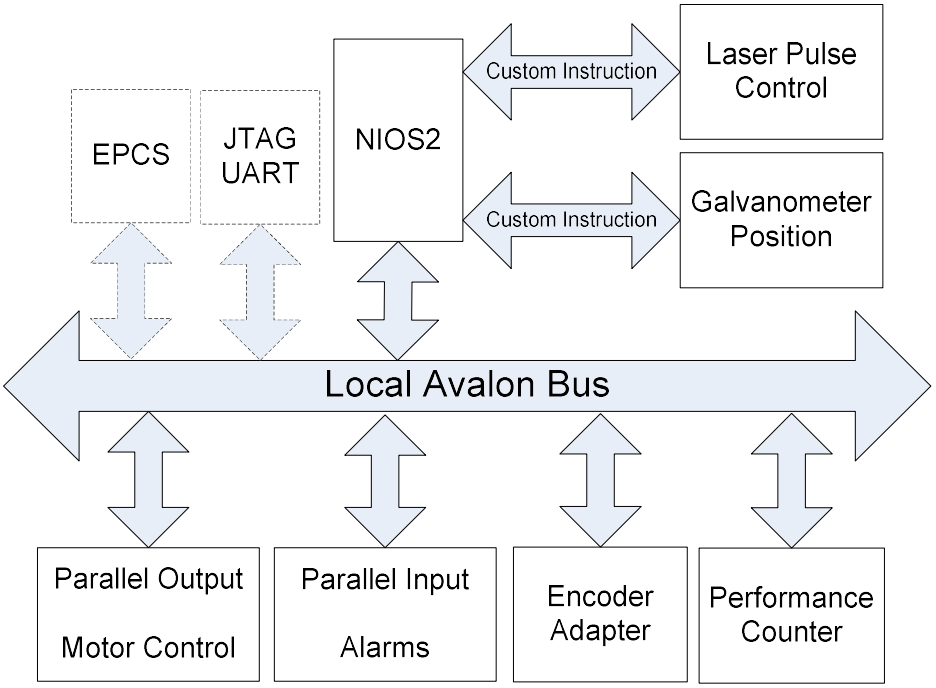


Figure 119 Tile design

The first CPU, which we call Master had some extra modules. An EPCS interface to read the configuration device, which is important in the boot-loading process as we will later detail. A JTAG UART to communicate with a host computer through the JTAG channel implemented through the USB connector. The whole system is depicted in Figure 120.

To build the multi-processor, four tiles were connected through a shared bus that also connected to several shared devices as shown in Figure 120. First the SDRAM controller, which gives access to external SDRAM memory, where all the programs and data will be stored. An additional On-Chip memory was used to share frequently used information and implement a message passing mechanism. Finally a mutex device was used to implement critical sections in the code.

The independent laser pulse signals that were produced in the custom instructions were also combined to be sent to the laser light controller. The combination was done with a simple *OR* gate because software was responsible to make them not activate simultaneously.

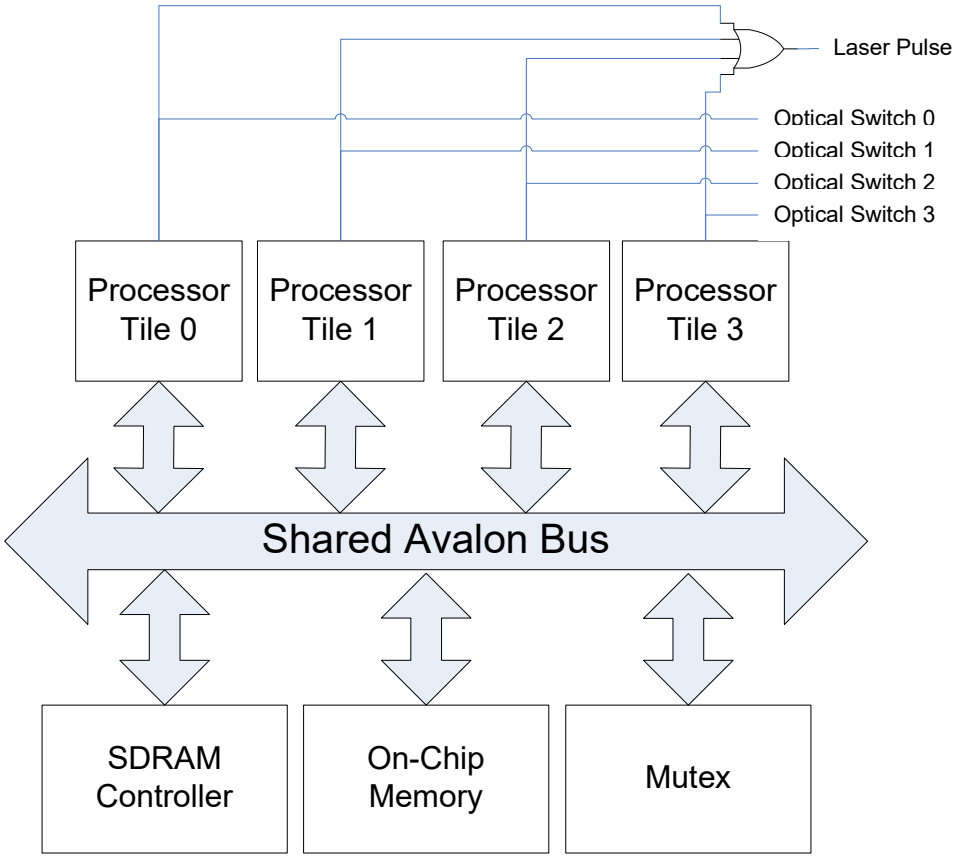


Figure 120 Multi-soft-core design

The marking application was described in MPI. The ocMPI library was implemented over shared memory using a mailbox approach. Using the SPMD instead of the MPMD approach supported by Altera is not only a way to simplify development and minimize errors, but it is also an important way to reduce executable size enabling to store the complete application in the flash device used to boot the system.

The system communicates with an external computer through the JTAG UART channel. Although the master processor controls the channel and receives all the commands from the host, the console commands can be targeted to other processors other than the master. The master processor redirects those commands by using the message passing facilities of the system. In order to have the ability to diagnose the correct operation of all the systems connected to the controller, a set of test commands was included. Other commands were used to download images and change operational parameters, like pulse period, or marking dimensions. Finally, there were some commands devoted to control the marking process for all the marking heads.

5.1.3. Functional Validation and Performance Analysis support

Functional validation was done at various levels. First, unit testing was done for modules implemented in hardware. Complex stimuli can be generated in the JHDL environment and cycle accurate performance information is collected. In Figure 121 a waveform to validate the correct operation of the XY2-100 generation module is shown.

Figure 121 Validation of the XY2-100 serial protocol module with interactive simulation in JHDL

After unit testing was done for all basic modules, system integration and verification was done at HDL level. Since software was part of the system, the verification could be done by using an ISS or a processor model at the HDL level.

To validate the integration of the different modules in a tile, I created a JHDL module combining the module that generates the laser pulses and the XY2-100 protocol to control the galvanometer system, the encoder module, and a MIOS processor and a memory containing a test application developed with NIOSII toolchain. The interactive simulation framework is shown in Figure 122. The drawback of simulation was speed, but also lack of real stimuli. As soon as the tile unit was validated, the system could be tested in the real hardware. Figure 123 shows the same system marking a metal surface.

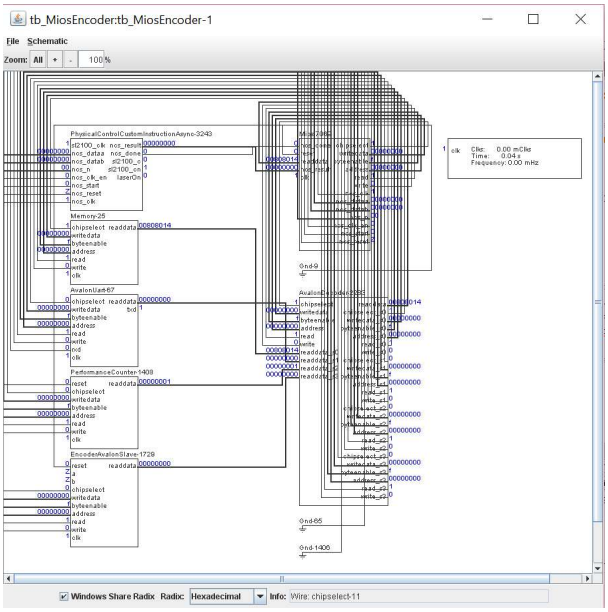


Figure 122 Tile system validation

Figure 123 Validation in real platform

But being a complex system, validation did not end at this level. To reach high volume productivity it was necessary to control that the marking process was executed so that it minimized the dependence on the slowest parts of the system, which were the mechanical parts: galvanometers, and servos. To analyze and optimize the performance of the marking process, the system can dump a trace of all the operations it performs at a very low level. This trace is sent to an external computer through the JTAG UART device that works over the USB cable of the board.

I developed a software application to analyze the trace and visualize the marking process so that errors and potential improvements could be easily detected for further optimization. The application allows to replay the trace and observe the result marking (see Figure 124), and the evolution of the values sent to the actuators to control the

galvanometer positions and the laser activation (Figure 125). This tool is not only important for validation, but for performance optimization as well. It is important to stress that the first source of inefficiency is often algorithmic, so it is much valuable to identify the algorithmic bottlenecks.

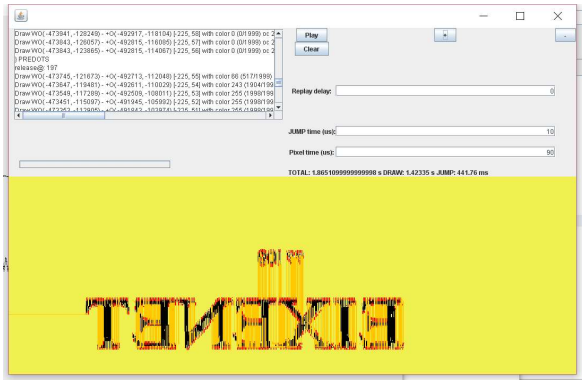


Figure 124 Trace replay application

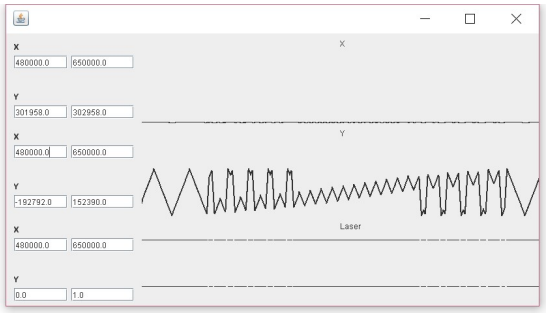


Figure 125 Actuator signals to control X and Y position of the galvanometers, and Laser activation

The real-time operation was extensively tested. Both, by using signal-tap analysis and also by external signal analyzer. Figure 126 depicts how an external signal analyzer was used to measure how the optical switches (F0, F1, F2, F3 in negative logic) are coordinated to share the laser beam during a laser process.

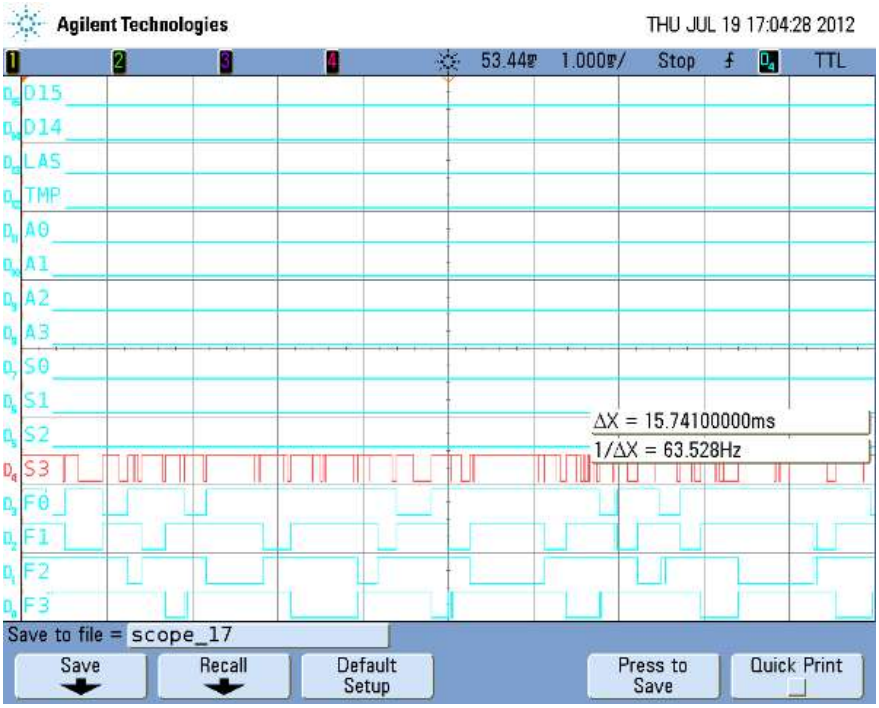


Figure 126 External signal analysis of the real-time behavior

Trace based Performance Analysis was also supported. I introduced automatic compiler instrumentation. This is enabled by the `-finstrument-functions` flag in gcc, which inserts a calls to a tracing function in the prolog and epilog of every function unless it is annotated with the `no_instrument_function` attribute.

My implementation of the tracing function just save a simple log containing the time captured from the performance counter, and the instrumented function address. Due to the limited space for logging, I provided enable/disable logging functions to have a fine grain control of when traces are generated. Captured traces could be downloaded to the host by a console command. The download process had to gather all the traces produced by different processors before sending them to the host.

In the host, the traces were converted into a standard format like OTF with a tool I created (named `ConvertTraceToOTF`) that converts the time, address pairs into the appropriate format as defined by the OTF format. As part of this process, the map-file of the application must be parsed to be able to get the function names related to the observed function addresses.

The ability to get and visualize trace files from the real system operation greatly simplified the optimization process. Figure 127 shows a Vampir screenshot where it can be seen how the laser beam is correctly shared between the four processors. The top panel shows a combined view of the four processors activity, while the panels below visualize the call stack of each processor. We can see that laser beam usage (in blue color) is not overlapped, so the system is properly synchronized. The lock acquisition function is shown in red, so while other functions are shown in green and pink. We can see that processors can do some computation, and not only block waiting for the shared resource.

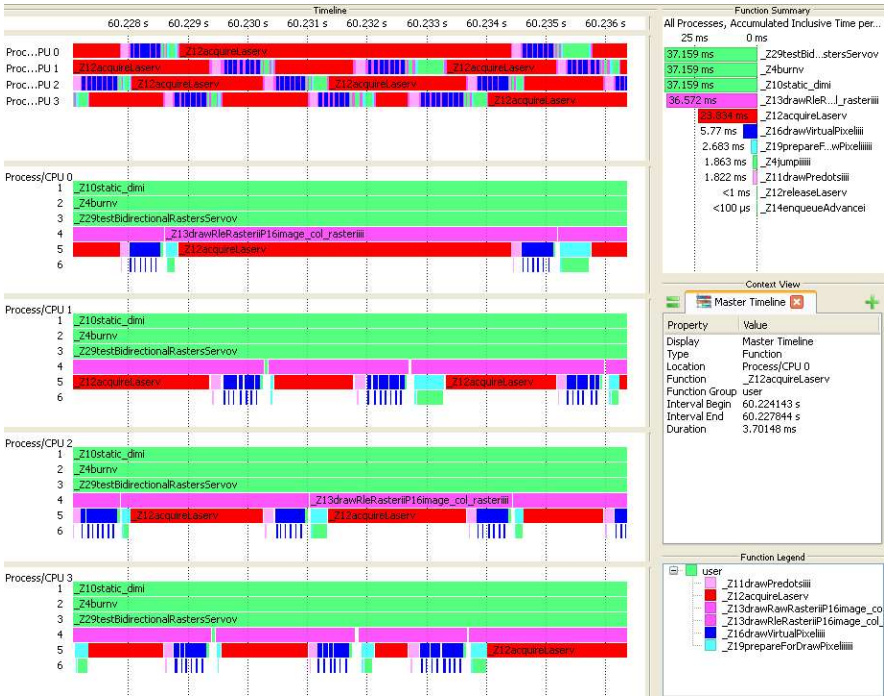


Figure 127 Visualization of system traces in Vampir

After the optimization performed in several iterations using performance analysis, we reached a 92% laser beam utilization rate.

5.1.4. Implementation and Results

The design was synthesized for the Cyclone IV EP4CE22F17C6 device, and the whole system was implemented in the Terasic DE0-Nano prototyping board. A brief summary of the synthesis results is presented in Table 41.

Table 41 Synthesis results for the EP4CE22F17C6 device

Property	Design	Device Total
LUTs	14619	22320
FFs	10398	22320
Memory bits	257704	608256
f_{max}	62.98 MHz	

The four core multi-soft-core processor with its custom hardware units fitted into the Cyclone Device, using 14 KLEs of the 22 KLEs available, giving an occupancy rate of 63%. The internal memory consumption was 42%, which is mainly devoted to processor caches, since main memory is implemented in the external SDRAM. Table 42 gives the resource consumption details for each block of the system.

Table 42 Detailed resource usage by block

Name	LUTs	FFs
NIOSII Processor 0	2214	1663
NIOSII Processor 1	2192	1643
NIOSII Processor 2	2194	1645
NIOSII Processor 3	2198	1644
Custom Instruction 0	365	357
Custom Instruction 1	365	354
Custom Instruction 2	366	355
Custom Instruction 3	369	356
PWM Slave 0	186	96
PWM Slave 1	179	96
PWM Slave 2	186	96
PWM Slave 3	186	96
Performance Counter 0	132	64
Performance Counter 1	132	64
Performance Counter 2	132	64
Performance Counter 3	132	64
Encoder Adapter 0	73	68
Encoder Adapter 1	78	68
Encoder Adapter 2	80	68
Encoder Adapter 3	89	68
JTAG UART	146	104
Mutex	56	36
EPCS Controller	117	115
SDRAM Controller	301	212
Other	3860	1336
TOTAL	**14619**	**10398**

An important aspect of this industrial system is cost. The prototyping board used to implement the design costs around 60€. The final custom PCB containing a low-cost Cyclone device can be manufactured for the same price or less. This system supports 4 simultaneous scan heads. The price per head would be less than 15€. To the best of my knowledge there is no scan head controller with this low cost. Some offerings can go from few hundred dollars to 1 thousand dollars, for just one head. So the achieved price efficiency is estimated to be about an order of magnitude with respect to alternative solutions.

5.2. Scaling up to many-soft-core processors

In [Castells15] I presented another case study to test the ability to scale up the methodology up to a large number of soft-core processors. Thus, creating a 128 core many-soft-core system based on 32 bit commercial soft-cores supporting MPI. To my

knowledge, the biggest such system ever build in a single FPGA. There are some works with more processors. In [Ben14] they embed up to 1024 modified PacoBlaze processors in a single device to demonstrate the scalability of the Distributed Memory Machine concept, a UMA architecture without caches where the memory is distributed in various nodes and accessed through an indirect network that performs hashing on the address. In this case, a SPMD multithreading programming model is used, but programs are limited in size to be able to be stored in a small instruction ROM memory. In RAMP Blue [Krasnov07] the complete system has up to 1008 cores, but they are embedded in 84 FPGA devices, and each device only embeds 12 soft-cores.

Figure 128 Terasic DE4 board

The 128 core system is implemented on a Terasic DE4 board containing a EP4SGX530 device (see Figure 128). The system created with the tools presented in Chapter 3. In fact, I used the tools to create a number of systems with an increasing number of processors to find the highest number of processors that could be embedded in the used FPGA. Replicating a tile containing a NIOSII processor with 4KB instruction and data caches and a FPU, the limit was found to be 64. Figure 129 shows the resource usage percentage per design. The 64 cpu+fpu exhaust the number of DSP elements of the FPGA, and when FPUs have to be implemented with standard logic (LUTs+FFs) the resource consumption skyrockets. A 128 cpu+fpu design is not able to fit into the EP4SGX530 device as the synthesis flow reports a resource usage of 169% ot the total device. If FPUs are not used a system with 128 core must reduce the cache sizes from 4KB to 2KB to fit in the device. Although the reported total memory bits consumption is less than half of the device capacity, in fact the designs exhaust all the M144K and M9K

memory blocks. The unused capacity is in MLABs, which are small memory elements distributed in the device.

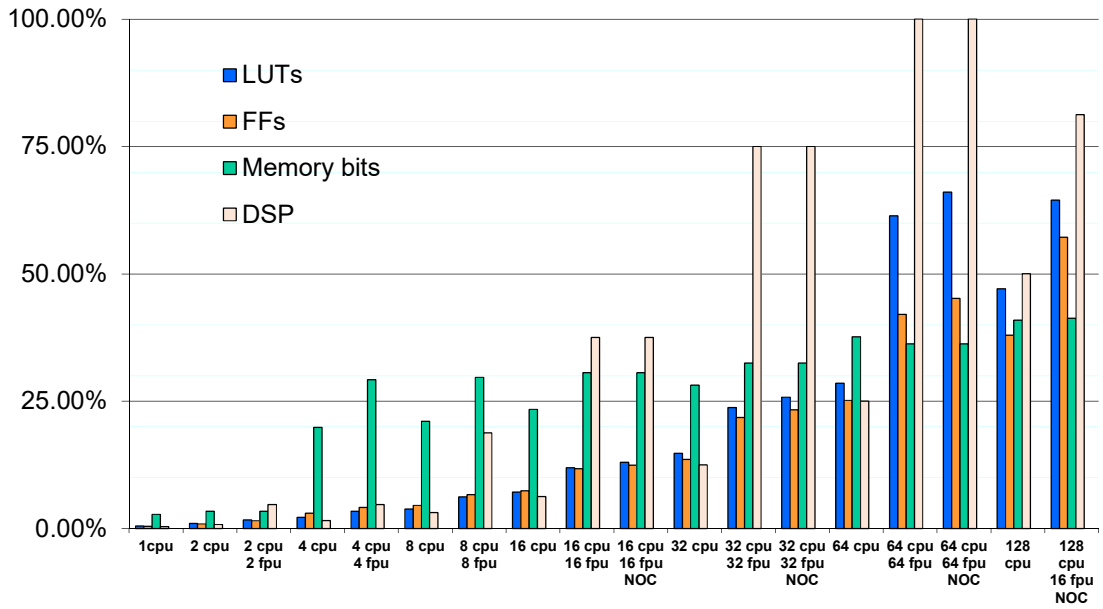


Figure 129 Percentage of the resource usage (LUTs, FFs, Memory bits, and DSP elements) of various multiprocessor designs synthesized for EP4SGX530

An important factor that can determine the success of a development process is the iteration speed, i.e. how fast a change is implemented, and tested in the system. Modern development practices encourage Agile methodologies [Dahlby04] that focus on frequent and fast iterations. In this context it is convenient to evaluate the time consumed in synthesizing the many-soft-core designs. Figure 130 shows the time used for the different steps of the synthesis process to synthesize the previous multiprocessor designs. Notice that the time scale is logarithmic, so synthesis process can actually go to several hours and even to the day scale for large complex designs. Software iterations are much faster in comparison, usually in the minutes scale. The obvious implication is that evolving and optimizing software is much faster than optimizing the architecture.

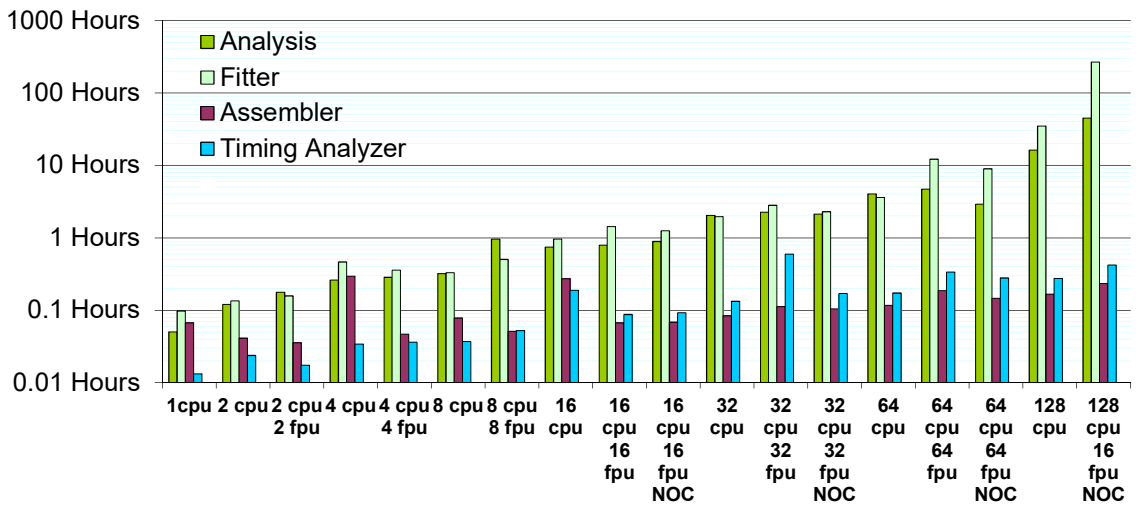


Figure 130 Time (log scale) consumed in the different steps of the synthesis process to synthesize different multiprocessor designs.

Table 43 Resource details of the synthesis of various multiprocessor designs for EP4SGX530

Design							Fmax
CPUs	FPUs	NOC	LUTs	FFs	Memory bits	DSP	(MHz)
1			2072	1870	587840	4	145
2			4144	3739	717120	8	128
2	2		7098	6477	718144	48	118
4			9368	12785	4217216	16	74
4	4		14288	17565	6204928	96	62
8			16175	19343	4468352	32	73
8	8		26273	28290	6304000	192	71
16			30335	31572	4970624	64	71
16	16		50631	49775	6502144	384	73
16	16	4x4 mesh	55239	52823	6502144	384	73
32			62834	57767	5975168	128	60
32	32		100896	92681	6898432	768	65
32	32	8x4 mesh	109662	99097	6898432	768	65
64			121184	106867	7984256	256	59
64	64		260904	178577	7702848	1024	55
64	64	8x8 mesh	280651	191847	7702848	1024	52
128			200029	161307	8686336	512	52
128	16	16x8 mesh	273884	243064	8763024	832	52

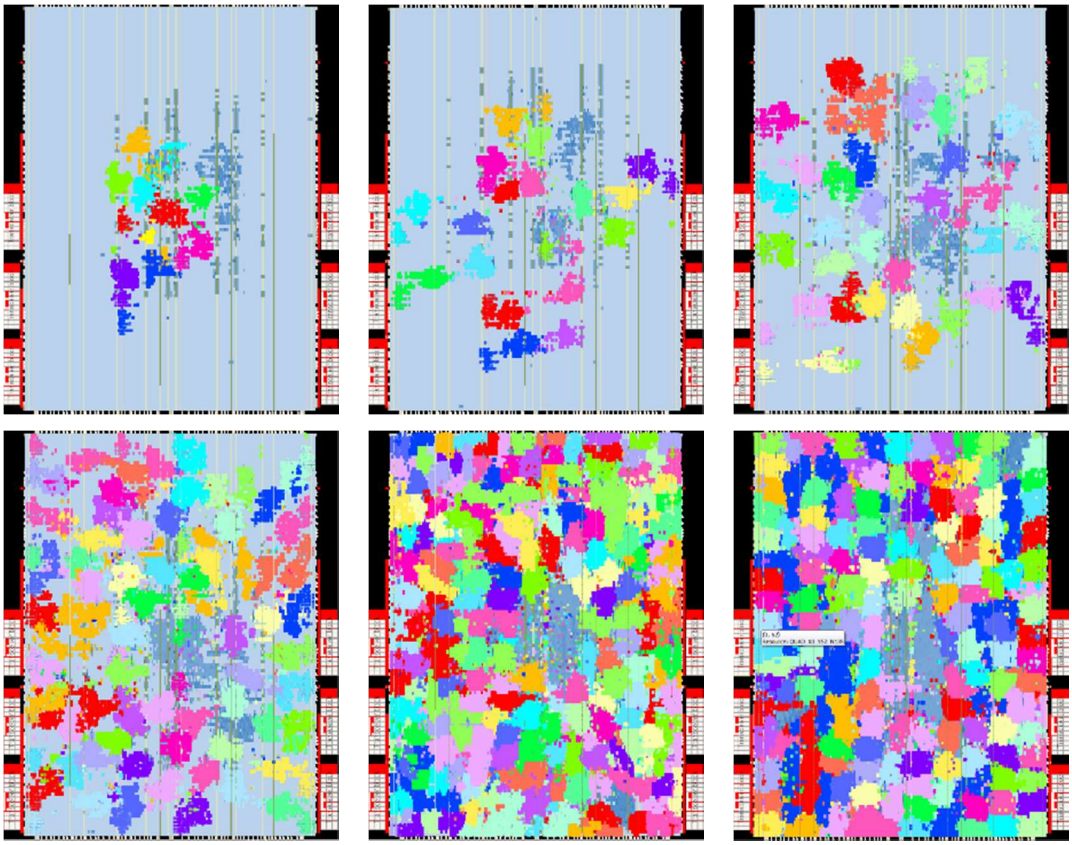


Figure 131 Floorplan of the synthesis results for 4,8,16,32,64, and 128 core multiprocessors on EP4SGX530.

Table 43 shows the resource usage details shown in Figure 129, corresponding to various multiprocessor designs. The system has a shared memory architecture with uniform memory access and caches, but also a distributed architecture is implemented by connecting the processors through another interconnection network. The processor interconnect is implemented in NocMaker. Figure 132 shows the design of the 16x8 mesh NOC for the 128-core many-soft-core. All the NOCs build for the previous designs have a 2D mesh topology using ephemeral circuit switching, XY distributed routing, and 4 phase handshaking flow control. Processors are attached to the NOC through custom instruction NAs. So, they architecture could be considered and hybrid UMA+NORMA with independent caches. The interconnect connecting the processors with the main memory is a standard QSys system provided by Altera.

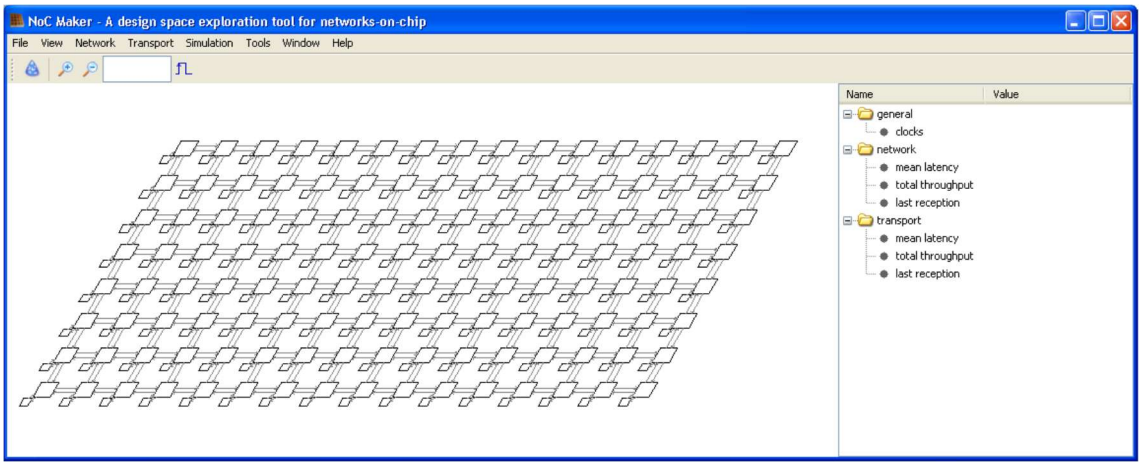


Figure 132 NocMaker view of the NOC build for the 128-core many-soft-core

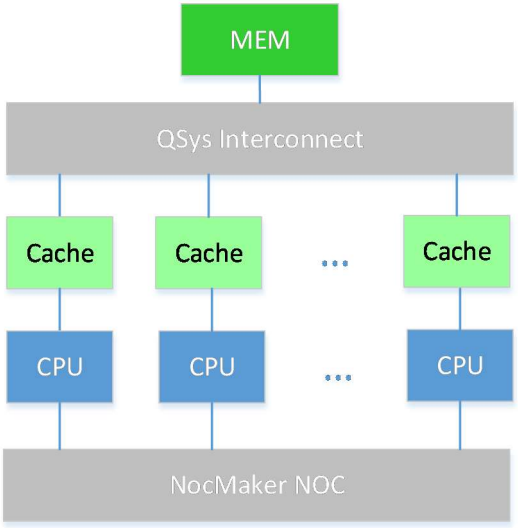


Figure 133 Hybrid UMA+NORMA many-soft-core architecture

This architecture allows an easy implementation of the SPMD paradigm, since a single program is stored in memory and each processor has its own stack. The MPI adaptation library (ocMPI) can be implemented using the shared memory or the NOC. The performance can be very different. ocMPI over shared memory introduces a lot of overhead to manage the queues needed to exchange messages among processes. If the size of the packet is large, the bandwidth to main memory can mitigate the high latency introduced by the software stack. On the other hand the ephemeral circuit switching NOC minimizes latency, but sacrifices bandwidth. Figure 134 and Figure 135 show the performance of the token-pass MPI micro-benchmark using shared memory or NOC respectively. The micro-benchmark measures the time that a token is returns to the master node after being passed from node to node until no more nodes are found in the

communication. The test is repeated for an increasing number of nodes participating in the communication. Notice that the total latency for NOC-based communication starts in the order of tens of picoseconds, while shared memory based communications start in the order of tens of milliseconds.

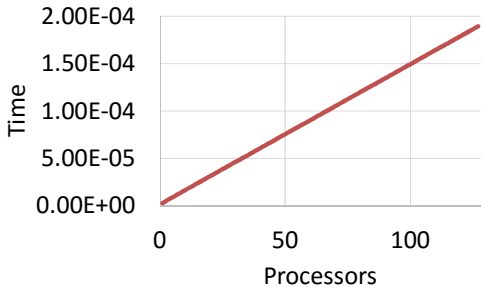


Figure 134 Elapsed time of token-pass micro-benchmark using ocMPI over NOC

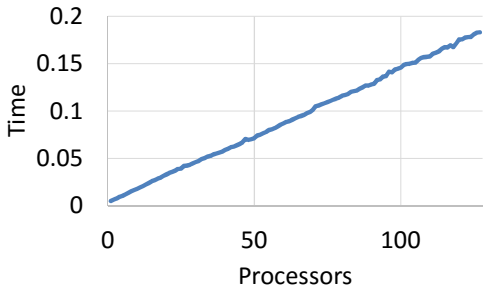


Figure 135 Elapsed time of token-pass micro-benchmark using ocMPI over shared memory

Two MPI applications are tested to validate that the system effectively can scale up for some workloads. The first application is an n-queens application, which computes the number of queens that can be placed in an n x n chess-board without attacking each other. The second application is PI computing, which computes the value of PI by Monte Carlo simulation.

The n-queens application fits very well to the platform, and benefits ideally from having a large number of processors (see Figure 136). This is so because the application is dominated by computation over communication and also because the amount of code to execute fits in the small instruction caches of all processors, and also because memory accesses are limited to small arrays that are also accessed from the data cache. The result is that, even the platform has an apparent bottleneck in the shared bus, when application locality benefits from cache a good scalability can be achieved.

On the other hand the PI application scales very poorly, only up to factor x 2. In this case the code executed by slaves includes calls to functions (like rand) and does not fit in the instruction cache. This provokes cache misses in all processors, which then congest the main bus to fetch instructions from the main memory. Moreover, cache misses are also happen in data caches.

To solve the issue, a simple optimization can be done to compact the code so that the hot-spot of the program can fit into the instruction cache of the processors. Instead of calling rand function, we create a simple and small pseudo-random number generation

and implement it as a macro. We also use the register C/C++ variable modifier to try to avoid accessing main memory during the hot-spot.

The result of this optimization is depicted in Figure 137. The obtained scalability profile is almost linear up to 100 processors.

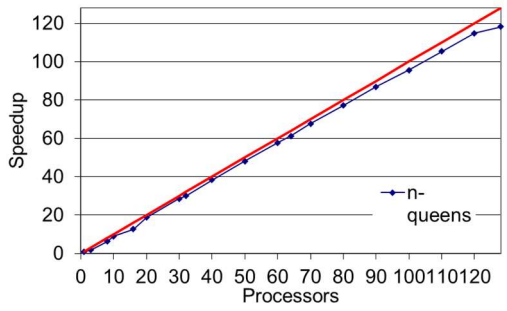


Figure 136 Scalability Profile of n-queens

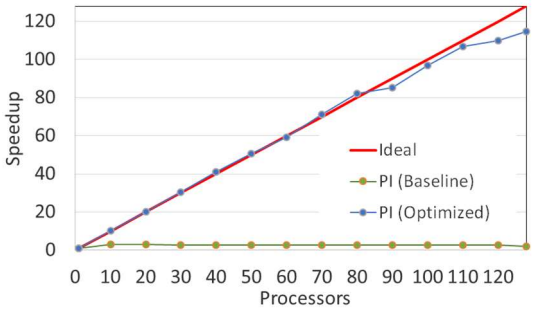


Figure 137 Scalability profile of PI computing. Baseline version shows a poor scalability, but optimized version has a good profile.

5.3. Increasing Energy Efficiency

In chapter 2 we observed that soft-cores can be, in the best case, an order of magnitude less energy efficient than modern low power processors. On the other hand, they offer more flexibility to try to increase the number of operations per cycle, thus allowing to increase the energy efficiency. In this section I will explore how to increase the energy efficiency of the designs by exploiting from the flexibility offered by the FPGA.

5.3.1. Mandelbrot

Computing the Mandelbrot Set [Mandelbrot13] is often used to test scalability of a parallel system (like in [Choi13]). The Mandelbrot set is created by the mathematical iterative formula described in (5.1) and (5.2), where Z_n and c are complex numbers. A value of c belongs to the mandelbrot set (also called Mandelbrot sea, see Figure 138) if after iterating the formula Z_n never scapes outside the circle of radius 2 (5.3). It can be graphically represented as Figure 138. A software implementation cannot test an infinite number of iterations, so typically a maximum iteration value is defined to build the set in reasonable time. Points that do not belong to the Mandelbrot set can escape from the circle of radius 2 at different iterations. The escape iteration is often used to produce multicolor representations of the Mandelbrot set (Figure 139).

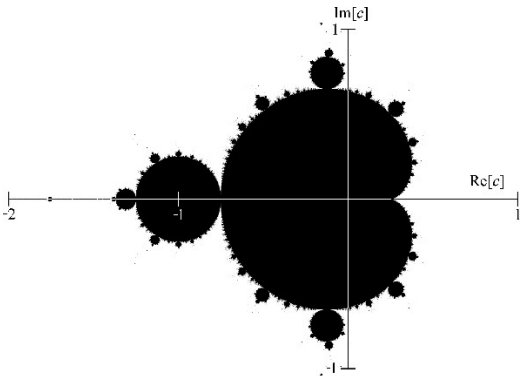


Figure 138 Mandelbrot Set in the complex number space. Black points belong to the set.

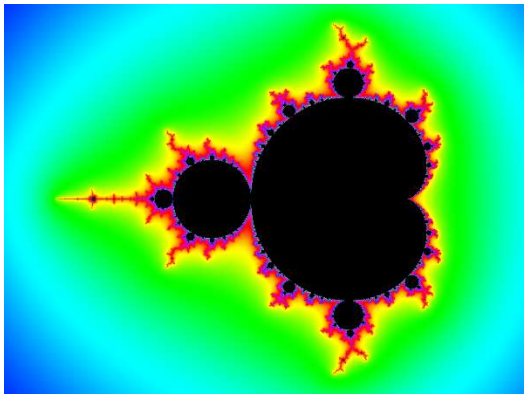


Figure 139 Multicolor representation of the Mandelbrot Set

$$Z_0 = 0 \tag{5.1}$$

$$Z_{n+1} = {Z_n}^2 + \mathrm{c} \tag{5.2}$$

$$c \in M \Leftrightarrow \lim_{n\to\infty} |Z_n| < 2 \tag{5.3}$$

Computing the Mandelbrot Set is, what it is called, an embarrassingly parallel problem. It exhibits an uncommon level of data parallelism because all different values of c can be tested independently. But points belonging to the set need more computation than the rest of the points. This variability motivates to use load balancing strategies to effectively parallelize it.

Table 44 Results of Mandelbrot Set application for a 640x480 frame and 1000 iterations

	i75500U	NIOSII
F (MHz)	2900	175
Time (seconds)	0.131	289
P_{dyn}(Watts)	8.4	0.6
FPS/Watt	0.90	0.005

I code a sequential version of the simple application, and test the performance and power consumption on a laptop computer and a NIOSII running at 175MHz that I build on the EP4SGX530 FPGA device. I compute a 640x480 frame with a maximum iteration of 1000. The results are shown in Table 44.

To analyze the reasons of the bad performance achieved on the NIOSII I use an ISS-based virtual platform to test the code. The platform supports transparent instrumentation, so a detailed analysis can be made without altering the time characteristics of the program. As shown in Figure 140 the *computePoint* function, which should be the most active function is not so active, instead computing is dominated by floating point emulation functions. This is reasonable as the design does not have an FPU.

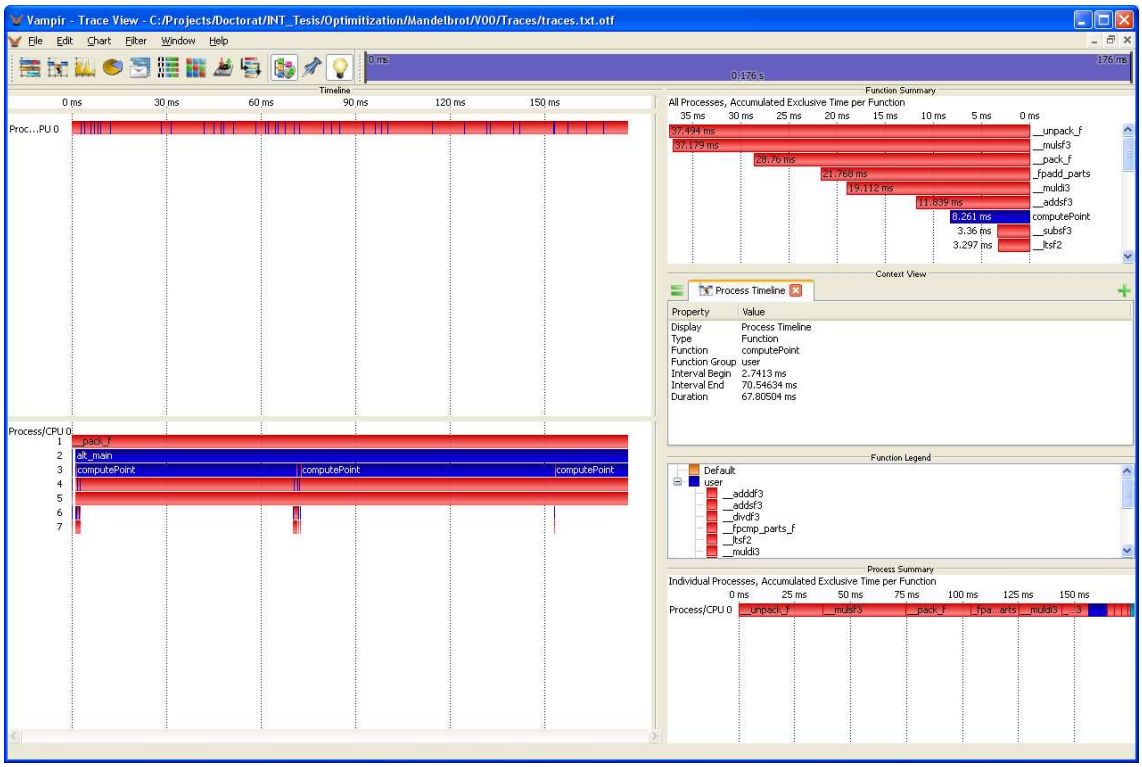


Figure 140 Analysis of the Mandelbrot application running on a NIOSII virtual platform using transparent instrumentation and visualization in VAMPIR

I synthesize a new design adding a MikeFPU and recompile the code to make use of it. I modify the virtual platform to mimic the same architecture, and execute on it the compiled code so that traces are generated. As shown in Figure 141 the floating point emulation calls are remove, but now the most time consuming function is still not *computePoint*, but the function I was using to store the results. By a simple optimization I remove this bottleneck and obtain a new version that I measure again.

The change does not alter the performance on the NIOSII system without FPU support as it was dominated by floating point emulation functions. It slightly improve the performance of the laptop system. The performance of the NIOSII with the FPU is improved by a factor x15 (see Table 45). But still the energy efficiency of the NIOSII+FPU system is 20 times worse than that of the laptop system.

Table 45 Results of optimized Mandelbrot Set application for a 640x480 frame and 1000 iterations

	i75500U	NIOSII+FPU
F (MHz)	2900	175
Time (seconds)	0.125	19.09
P_{dyn}(Watts)	8.4	1
FPS/Watt	0.95	0.052

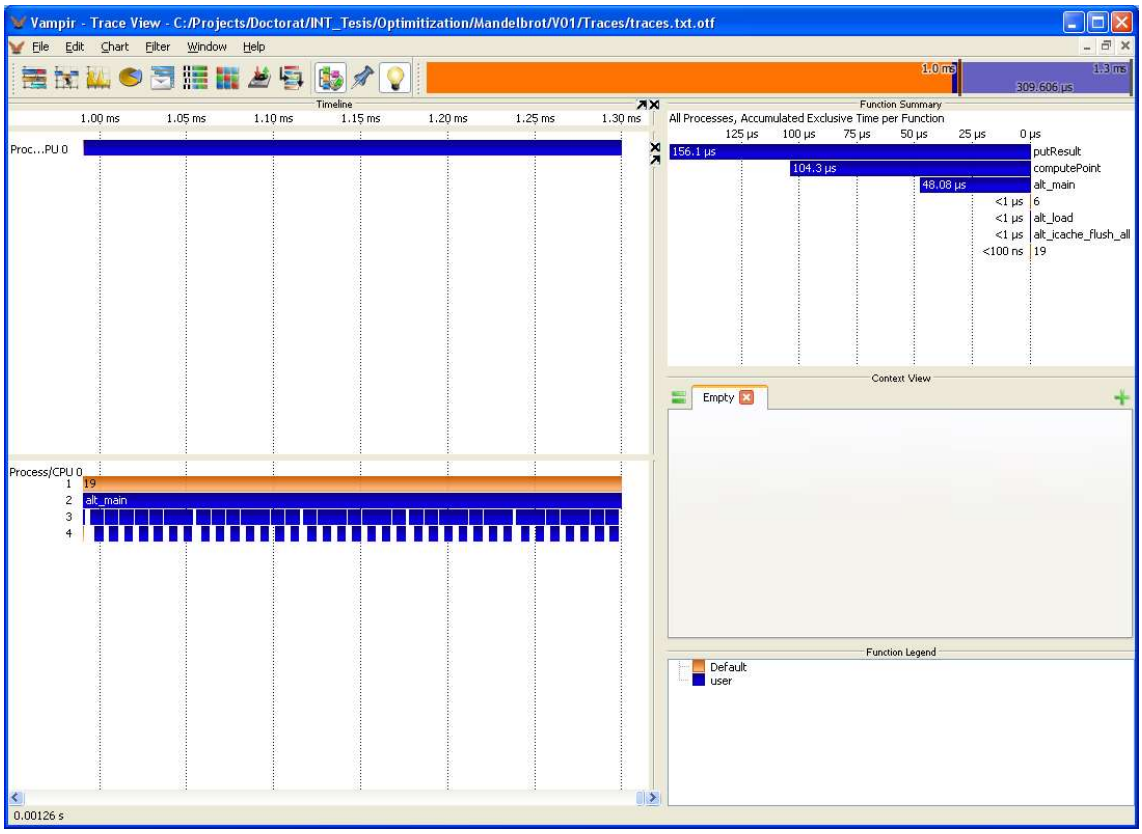


Figure 141 Analysis of the Mandelbrot application running on a NIOSII+MikeFPU virtual platform using transparent instrumentation and visualitzation in VAMPIR

Can the energy efficiency of the single core NIOSII system be raised anymore? Actually, the equation (5.2) is implemented with the following code.

```
float x = x0; // x co-ordinate of pixel
float y = y0; // y co-ordinate of pixel

int iter = 0;
int colour;

while ( (x*x + y*y < (2*2))  &&  (iter < max_iteration) )
{
    float xtemp = x*x - y*y + x0;
    float ytemp = 2*x*y + y0;
    x = xtemp;
    y = ytemp;
    iter++;
}
```

For every iteration 6 floating point multiplications, 3 floating point additions, and 2 integer additions are executed. A specific hardware can be build that implements this operations in parallel when possible, thus, reducing the latency introduced by the processor.

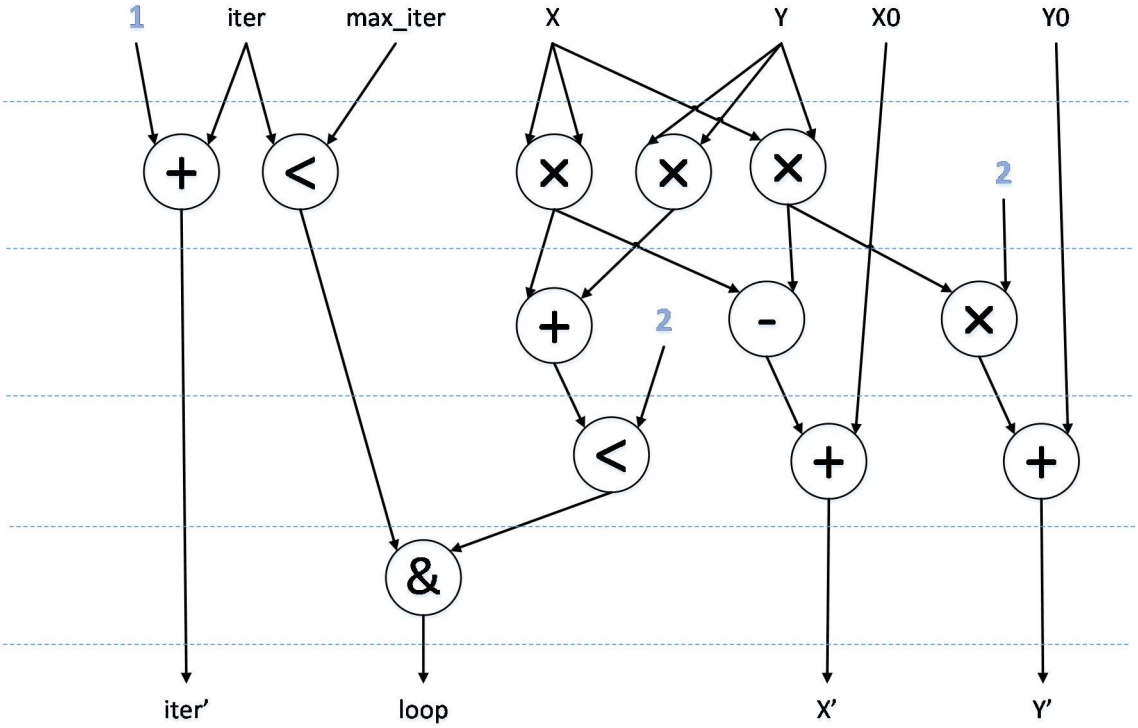


Figure 142 Data dependency graph of the computation performed in the loop iteration

Figure 142 shows the data dependency graph of the operations performed in the *computePoint* loop. The sequential execution of the loop uses 12 operations, 8 being floating point. In a completely parallelized implementation of the loop iteration the latency would be dominated by the 3 floating point operation steps needed to produce X' and Y'. Hence, we could expect, at least, a 8/3 speedup factor.

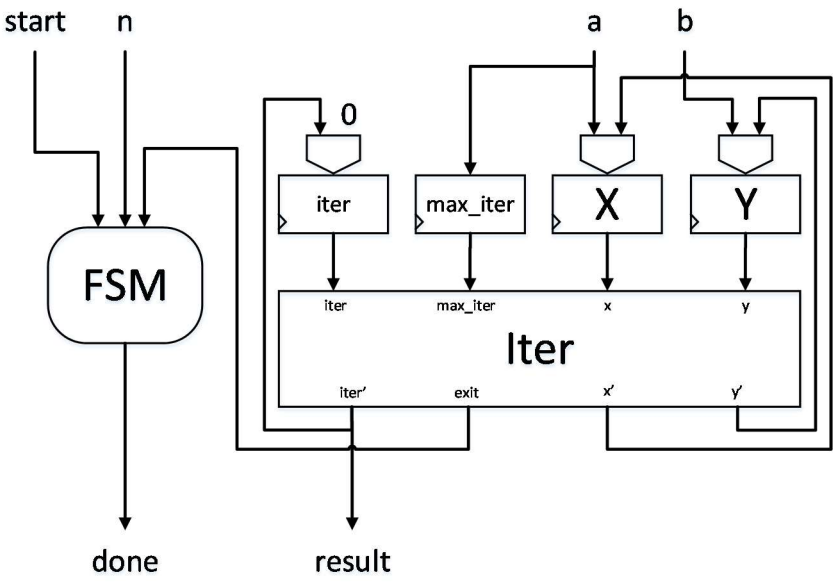


Figure 143 design of the Custom Instruction

The operations performed in the loop iteration can be parallelized, but unfortunately the loop cannot be unrolled as there is dependency between the values of X and Y among different iterations.

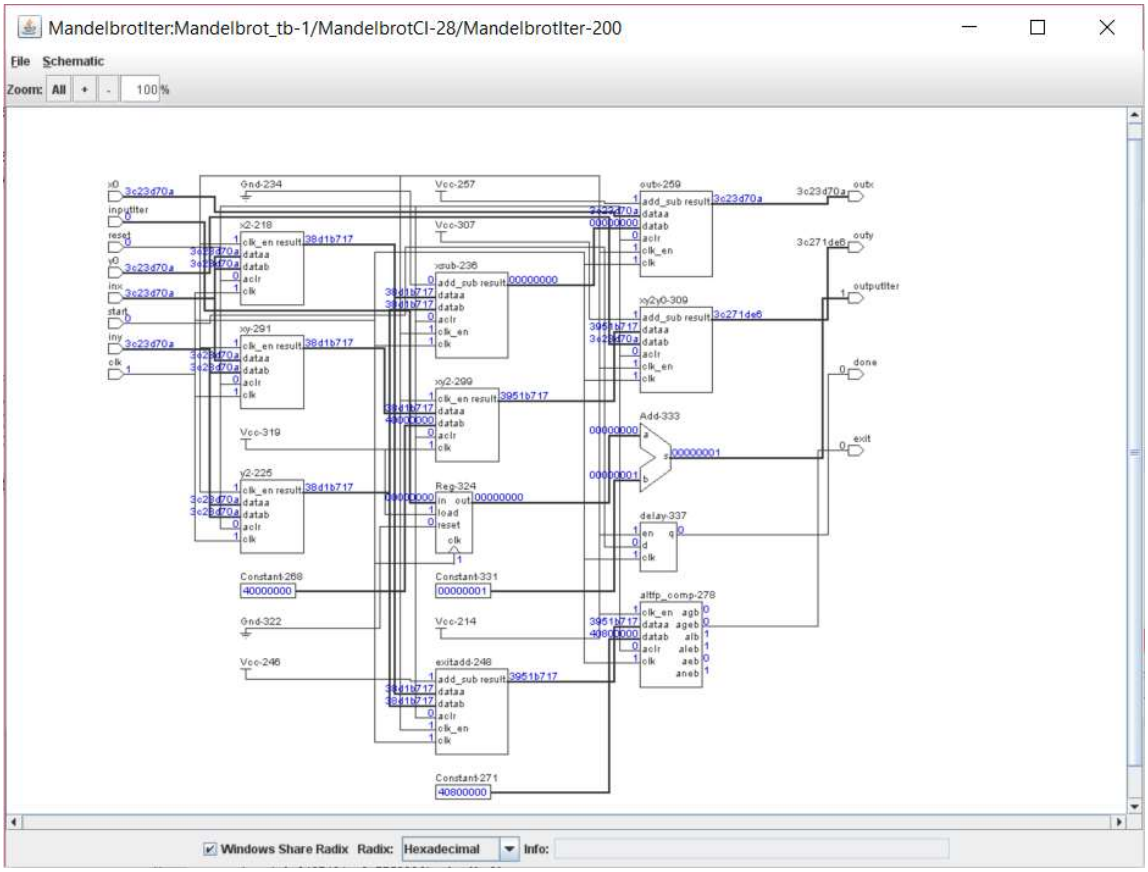


Figure 144 JHDL schematic view of the iteration module, which is part of the Mandelbrot custom instruction

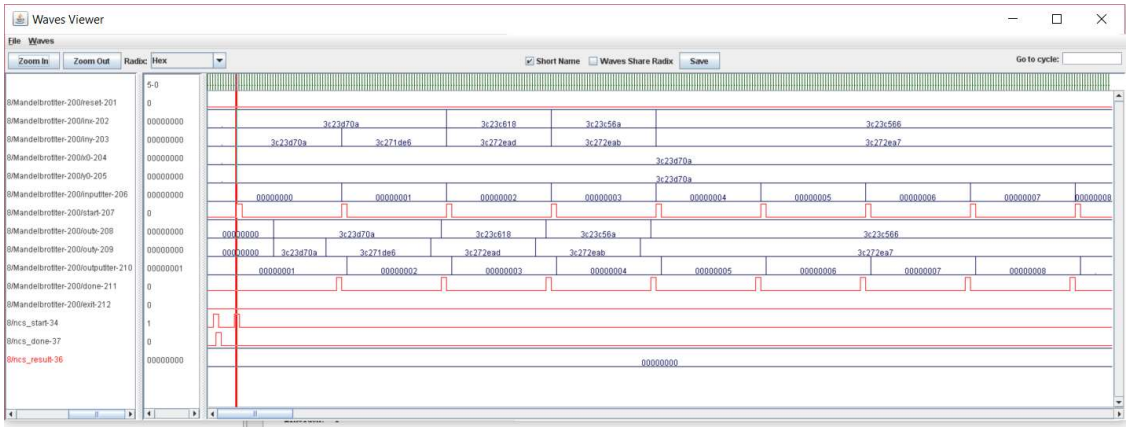


Figure 145 Waveform to validate the mandelbrot custom instruction

After this optimization, the performance is improved by a factor x4.7 vs. the NIOSII+FPU architecture and a factor x72 vs. the optimized NIOSII implementation.

Table 46Results of Mandelbrot computing using the custom instruction

	NIOSII+Custom Instruction
F (MHz)	175
Time (seconds)	4.01
P_{dyn}(Watts)	1.2
FPS/Watt	0.20

Unfortunately it still does not reach to the same energy efficiency level than the i75500U. This is so because we cannot benefit from increasing the parallelism that could be at the loop level due to the data dependency.

5.3.2. Primer Numbers

Floating point support for soft-core processors is not implemented as part of the data path of the processor, but as an external logic that has to be accessed through custom instructions. Since the floating point units have a significant latency due to their pipelined design, on a soft-core processor, they are underutilized most of the time, whereas in a modern superscalar processor the pipeline is optionally filled by independent instructions on queue. This penalty is one of the reasons why Mandelbrot set application is more energy efficient on laptop processor.

Integer computing should draw better results as integer units are part of soft-core data paths. Computing the first prime numbers from 1 to 10^6 is another embarrassingly parallel application that can be used to test the methodology presented in this thesis. A brute force approach can be used to determine the primes. There exist alternative more efficient algorithms, but the algorithm used can be illustrative of how energy efficiency can be obtained using many-soft-cores. Basically, all numbers in the range must be evaluated for being prime. The function to determine if a number is prime can be described with the following code:

```
int isPrime(int v)
{
   int i;

   if (v >= 0 && v <= 3)
      return 1;

   if (v % 2 == 0)
      return 0;

   for (i = 3; (i*i)  < v; i++)
   {
      if (v % i == 0)
         return 0;
   }

   return 1;
}
```

Indeed, the application has a lot of parallelism, and it can be easily parallelized using OpenMP as shown in the following code.

```
#pragma omp parallel for private(i,n) shared(primes)
for (i = 2; (i*i) < n; i++)
{
   if (isPrime(i))
   {
      #pragma omp atomic
      primes++;
   }
}
```

I execute the application on the i75500U processor to get its performance and energy efficient. The OpenMP code is compiled in gcc with full optimization (-O3) and executed on the processor, which has support for 4 threads. The minimum execution time is 22.35 seconds when using 8 threads. Thus, the OpenMP parallel implementation achieves an speedup factor of 2.3 with respect to the sequential implementation. Table 47 shows the detailed results.

The estimated MOPS/Watt is calculated aggregating the number of operations done in each iteration of the loop inside `isPrime`. The best energy efficiency obtained

in the i75500U is 27.7 MOPS/Watt using 6 threads (see Figure 146). The energy efficiency increases with the number of threads but not more than a factor 1.5 with respect to the sequential version.

Table 47 Performance results of OpenMP implementation for the 10^6 first primes on the i75500U platform

Threads	Time (seconds)	P_{dyn}(Watts)	Estimated MOPS/Watt
1	51.42	10.7	17.571
2	39.77	15.6	15.582
3	31.33	14.6	21.135
4	27.45	15.3	23.018
5	25.63	15.6	24.179
6	23.89	14.6	27.716
7	22.92	16.6	25.409
8	22.35	16.6	26.057

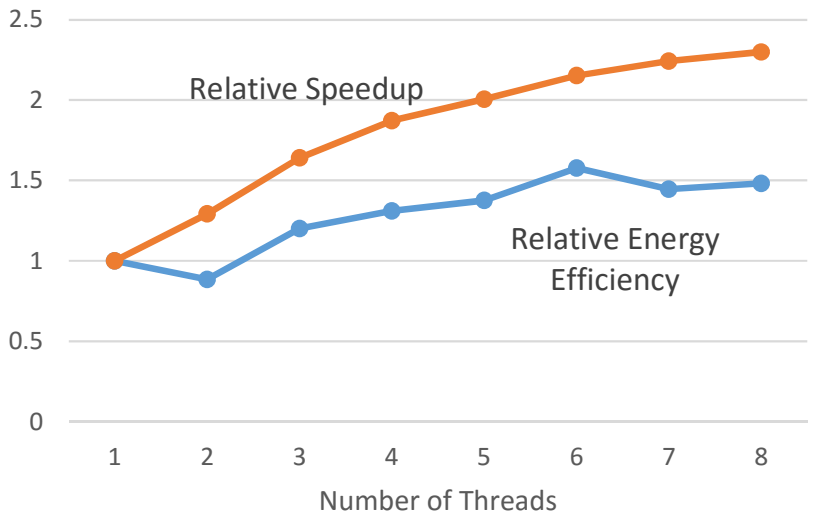


Figure 146 Relative Performance and Energy Efficiency of computing the 10^6 first primes using an OpenMP application on i75500U

To analyze the performance of a soft-core based solution, I build a simple NIOS II system running on 60 MHz. A reduced clock frequency is used to avoid modifying the clock frequency as the design is iteratively modified, so that an easier comparison can be made. The single thread application is compiled using the gcc tool-chain and full optimization. The system is synthesized for the EP4SGX530 FPGA. Table 48 shows the

performance and efficiency results. As expected, the performance of the system is very poor, the execution time is 630 times worse than the laptop version, but the energy efficiency is just an order of magnitude worse.

Table 48 Performance results of computing the 10^6 first primes on NIOSII running at 60 MHz on EP4SGX530

Time (seconds)	P_{dyn}(Watts)	Estimated MOPS/Watt
14093	0.28	2.45

Analyzing the source code of the application we can detect opportunities to create custom hardware that could allow us to increase the instruction parallelism, and improve performance and energy efficiency. Figure 147 shows the data flow graph of the instructions executed in the loop of the `isPrime` function, which is obviously the hot-spot of the application.

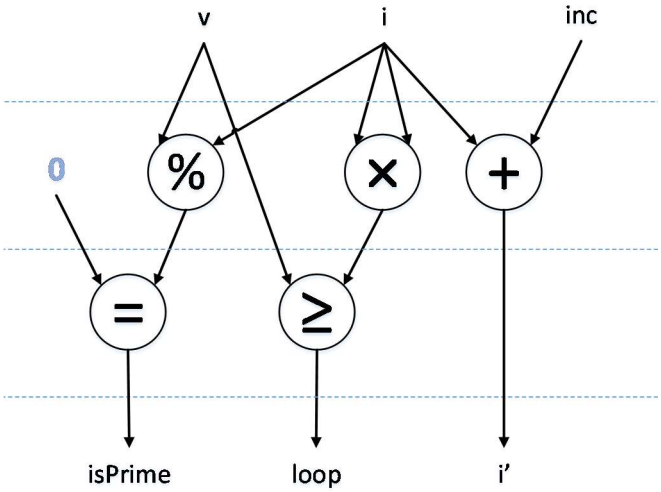


Figure 147 loop iteration data flow in prime number calculation

To try to improve the performance of the `isPrime` function I propose to implement the whole loop on a custom instruction that receives the number to check whether it is prime as an argument, and returns a boolean value informing the result.

I implement the logic described in Figure 147 as a logic block in JHDL which I call it *Iter*. Similarly, I implement a NIOS II custom instruction with the design shown in Figure 148. The operation is very simple: the value to check is passed as the *a* argument to the custom instruction and it is stored in register *v*. An *i* register is initialized with value 2 and it is passed to the *Iter* module, which keeps incrementing it at every iteration. If *v* is divisible by *i* the *isPrime* signal is asserted. If not, the loop signal indicates wether a

next iteration should be performed with the next value of *i*. A simple FSM controls the data path.

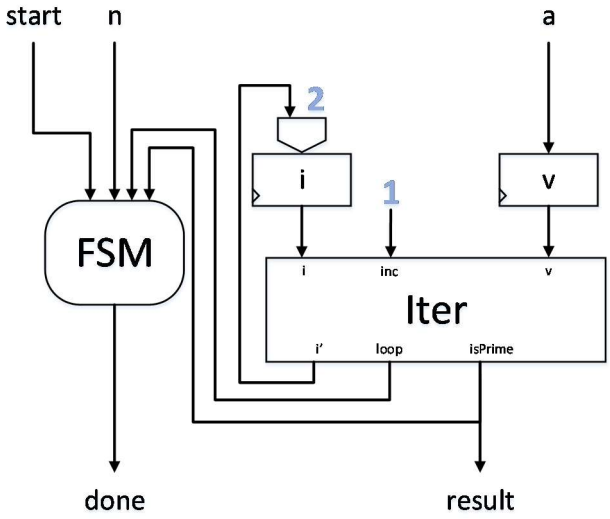


Figure 148 Custom Instruction Design for prime number checking

The design is synthesized and integrated into the former NIOS II system. The application must be modified to take profit of the new custom instruction. The results of the new design are shown in Table 49. The new system shows a speedup of x7 approx. vs. the previous one, while the energy consumption is almost the same. This increases the energy efficiency to a similar to that of the laptop system. Although the performance is much lower, 274 times worse.

Table 49 Performance results of computing the 10^6 first primes on NIOSII with custom instruction running at 60 MHz

Time (seconds)	P_{dyn}(Watts)	Estimated MOPS/Watt
1879	0.3	17.146

Unlike the Mandelbrot case, the loop of prime number computing can be unrolled nicely since there is no data dependencies between iterations. Unrolling means to execute concurrently some of the different iterations simultaneously. This could be done by replicating the iter loop units and assigning different values of *i* to each one. But, the nature of the algorithm allows further optimization. Is not necessary to do the check for the loop bounds in every iteration, by doing it in blocks the behavior is not altered and some hardware resources can be saved. To implement this idea it a new optimized unit without the hardware resources to check the bounds is done (see Figure 149).

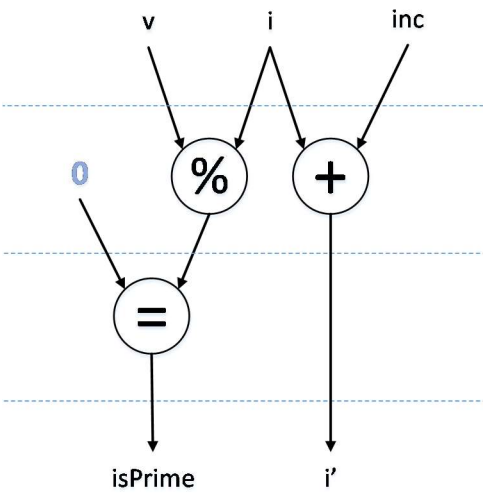


Figure 149 Data flow of the operations done in the optimized loop iteration module

Figure 150 depicts the new custom instruction, which combines the former *iter* unit with the new optimized one to save some resources. A system with 10 units is build following this strategy. Like in previous design, the system is easily implemented and verified in JHDL. As an example, in Figure 151 the interactive schematic view of the optimized iteration module is shown.

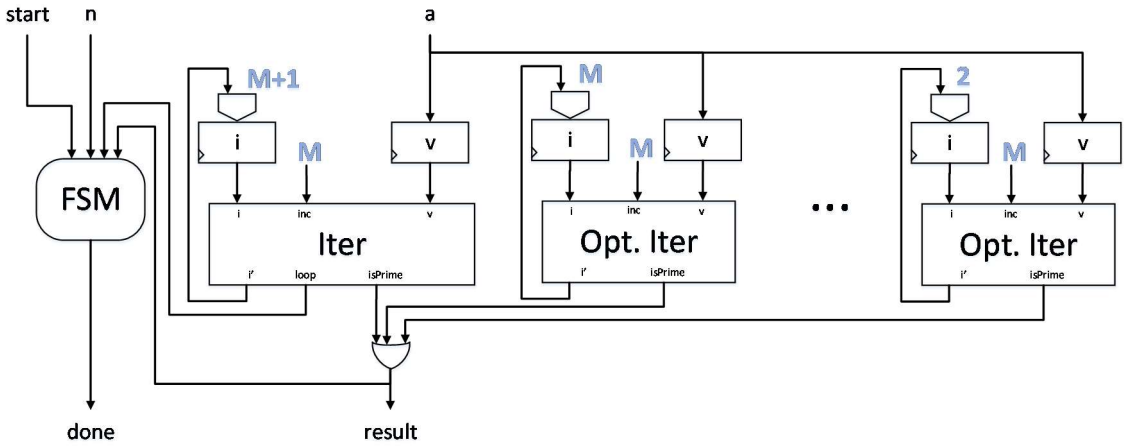


Figure 150 MultiUnit Custom instruction

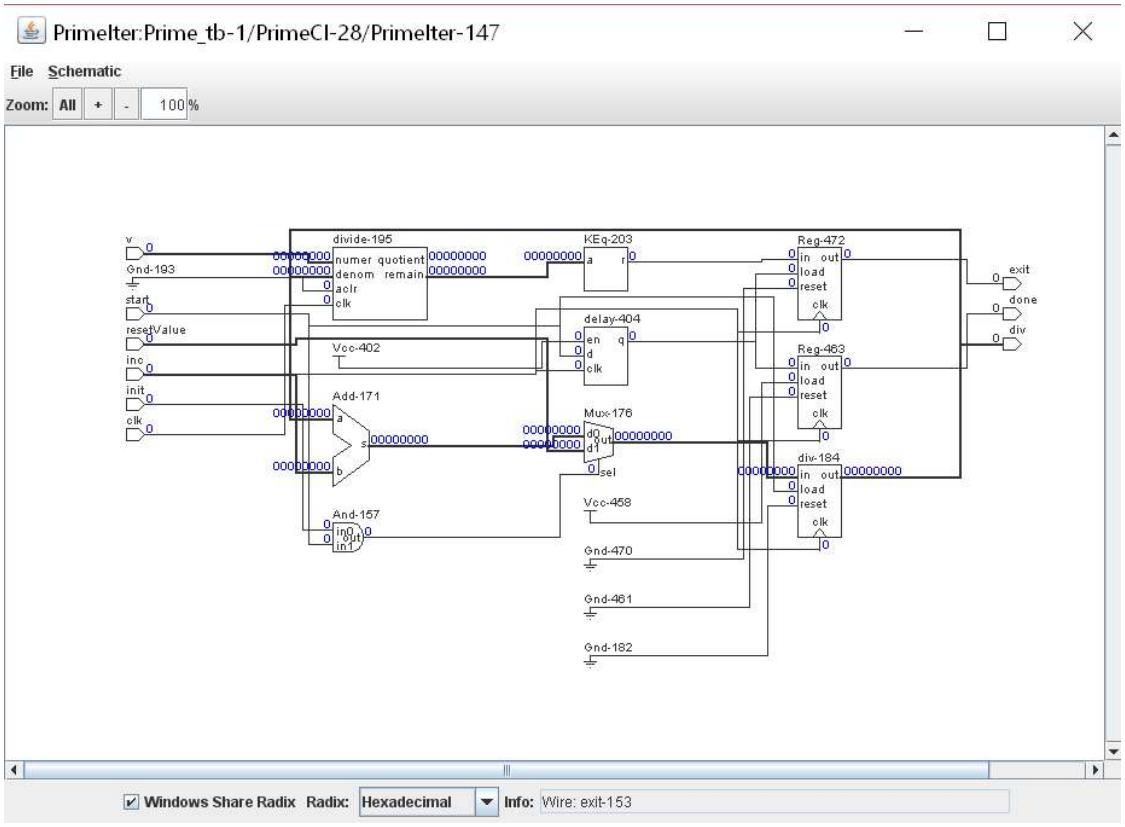


Figure 151 Prime Number Iteration Module, which is part of the Custom Instruction

The new design is synthesized an included in the NIOSII system. The application must be modified again to use the new custom instruction. The results of the new design are shown in Table 50. Power consumption is slightly increased, but performance is increased by almost a factor x10. This, boost the energy efficiency to a higher level than that of the laptop.

Table 50 Performance results of computing the 10^6 first primes on NIOSII with a custom instruction containing 10 units running at 60 MHz

Time (seconds)	P_{dyn}(Watts)	Estimated MOPS/Watt
188.19	0.4	128.42

The system is still underutilized, since the divider units have a latency of 5 cycles due to its pipelined design, but just one unit of the pipeline is active per cycle. In addition, some additional cycles are used by the control unit. Is it possible to increase the utilization of the dividers pipeline?

Again, the algorithmic nature of the problem allows it. The value of i could be increased every cycle without the need to wait for the completion of the previous modulo operation. All the iterations are done to find if any number give a zero remainder. So it does not matter if we detect it with some latency, or that we lose the reference of the number that produced the result. Thus, a new value of i can be injected in every unit every cycle. The latency of the detection is increased but it can be easily handled by the control unit.

After implementing this change the system has to be synthesized again and the application has to be modified and recompiled to use the new custom instruction. The results are show in Table 51. Again, the increase in power consumption is minimal since we are just adding a small amount of logic to use the resources more efficiently. The speedup factor vs. the previous version is almost x6, which is in line with the number of cycles that were waster before. The performance efficiency factor is boosted again by a similar factor.

The system has a comparable performance to the laptop, but its energy efficiency is 23 times better that the best implementation with 8 threads on the multicore standard processor.

Table 51 Performance results of computing the 10^6 first primes on NIOSII with custom instruction containing 10 pipelined unit running at 60 MHz

Time (seconds)	P_{dyn}(Watts)	Estimated MOPS/Watt
31.63	0.5	611.27

To continue increasing the performance of the system, more parallel units could be added to the custom instructions. There is also the possibility to increase the pipeline levels of the divider. The presented designs use a pipeline of 5 stages, and this certainly limits the maximum frequency of the system. It is reasonable to increase the pipeline levels up to 35 for higher frequency operation. Having this long latency would increase the opportunity to both increase the frequency and still benefit from the new levels of pipeline.

Another option to increase performance is to replicate the processor with its custom instructions and implement a parallel version of the application. The initial computer application was parallelized using OpenMP. Since there is not support for

OpenMP yet in the architectures presented in this thesis I use threads on an 8-core multi-soft-core that includes the best performing custom instruction.

After synthesis, compilation and execution on the platform, the results shown in Table 52 are obtained.

Table 52 Performance results of computing the 10^6 first primes on a multiprocessor having 8 NIOSII with custom instruction containing 10 pipelined unit running at 60 MHz

Threads	Time (seconds)	P_{dyn}(Watts)	Estimated MOPS/Watt
1	31.63	0.75	407.51
2	15.55	0.9	690.76
3	10.32	1	936.75
4	7.57	1.1	1160.95
5	6.35	1.2	1268.67
6	5.27	1.35	1358.81
7	4.53	1.4	1524.33
8	3.97	1.5	1623.39

The best performance achieved is 5.6 times better than that of the computer, but the energy efficiency is 58 times larger. Moreover the scalability profile for the application is almost ideal (Figure 152). Given the theoretical background presented in chapter 2, it can be surprising that the energy efficiency increases with the number of threads since the number of active transistors should have a linear relation with the number of used processors. This is true, but we must also consider shared resources that are basic part of the architecture. They contribute to power consumption whatever number of processors are executing. Obviously, this contribution will have less impact as number of processors keep increasing.

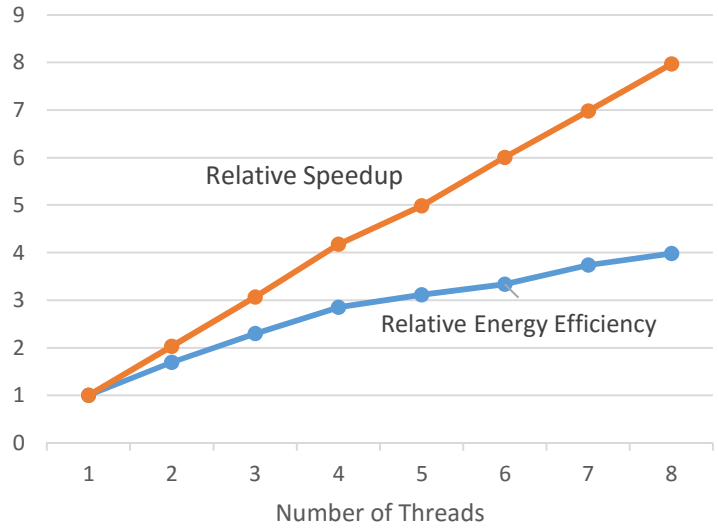


Figure 152 Relative Performance and Energy Efficiency of computing the 10^6 first primes using a multithreaded application on a multi-soft-core processor containing 8 NIOSII with coprocessors.

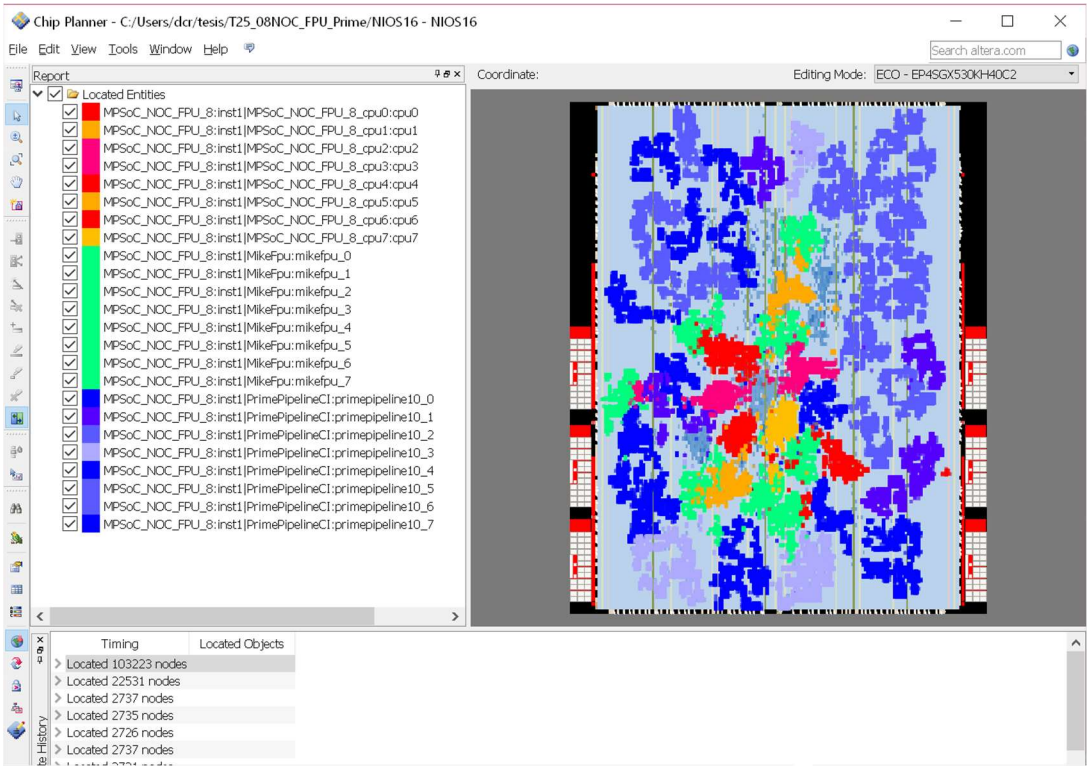


Figure 153 FPGA layout of the synthesis of the 8-core prime number computing multi-soft-core for EP4SGX530

This case study was tested up to 8 cores, but (if more time had been available) more processors could have been added as the FPGA occupation was approximately 26%. The resource usage is detailed in Table 53. The custom instruction units consume more resources than the soft-cores themselves. This can easily be seen in Figure 153, in which CIs have been shown in blue colors.

If we revisit one of my goals, it was to create application-specific energy efficiency systems that could be simple to program, and that could still be used for general

purpose processing. The prime-number computing system reaches a high efficiency ratio and still offers a general purpose platform.

Table 53 Resource details of the synthesis of the 8-core prime number computing multi-soft-core for EP4SGX530

Element	LUTs	FFs	Memory bits	DSP
CPU 0	1419	1600	63104	4
CPU 1	1200	1305	20224	4
CPU 2	1201	1306	20224	4
CPU 3	1202	1306	20224	4
CPU 4	1202	1306	20224	4
CPU 5	1202	1306	20224	4
CPU6	1204	1306	20224	4
CPU 7	1205	1306	20224	4
FPU0	1343	1254	4608	20
FPU1	1340	1254	4608	20
FPU2	1340	1254	4608	20
FPU3	1340	1254	4608	20
FPU4	1340	1254	4608	20
FPU5	1343	1254	4608	20
FPU6	1343	1254	4608	20
FPU7	1340	1254	4608	20
CI0	10474	1817	0	0
CI1	10474	1817	0	0
CI2	10478	1817	0	0
CI3	10471	1817	0	0
CI4	10471	1817	0	0
CI5	10472	1817	0	0
CI6	10472	1817	0	0
CI7	10502	2140	0	0
Other	6811	6773	9528320	0
TOTAL	111190	42401	9769856	192

5.4. Conclusions

In this chapter, I used several cases studies to validate that many-soft-core platforms can be implemented and used effectively for certain application domains.

Hard real-time systems are one of them. In that context, FPGAs can better adapt to real-time requirements by implementing specific hardware when needed. Alternative platforms like GPGPUs, DSPs or CPUs, do not have this choice.

On the other hand, the task isolation approach simplifies the required WCET analysis making it easier to meet real-time constraints.

The number of processors that can be implemented in a FPGA is only limited by its resources. I demonstrated the implementation of complete application with 128 cores

as an outstanding result according to the state of the art, much in line with my original simple estimation.

Finally I have proven that, although not generally reaching the best performance, many-soft-core platforms can be a good vehicle to increase energy efficiency of certain HPC workloads. This can be achieved by just coding a limited part of HDL code to boost the energy efficiency ratio, and then scaling up the design by using parallel architectures.

6. Conclusions and Future Directions

6.1. Conclusions

Reconfigurable devices provide a flexible platform to experiment with different computing architectures. The microelectronics industry still foresee ways to continue increasing the transistor density for some years, at least, up to the few nanometer nodes (7nm, 5nm). However the access to new nodes will increase even more the NRE costs. The ability to be generally enough to be used in many applications and systems will be a necessity for the chips using that technology. In this context, Reconfigurable systems can play a growing important role, since they design complexity is limited due to its regularity. Moreover their structure could absorb better the expected decrease of yield (number of good devices per total manufactured device) in future nodes. Actually reconfigurable device manufactures have a recent history of being the second customers of silicon foundries after major microprocessor manufacturers.

Future FPGAs will probably increase the capacity to the order of the millions of Logic Cells while reducing their energy consumption. Current FPGAs can already embed more than 100 soft-core processors. Although the energy efficiency of a soft-core is, at best, an order of magnitude lower than that of a modern low power microprocessor, the architectural flexibility provided by these platforms can be used to close that gap and beat alternative architectures for certain applications.

In this thesis I have proposed a new methodology for building highly scalable multiprocessors based on standard soft-core processors. To make it possible I have complemented the synthesis toolchains of device manufacturers with several tools to cover the gaps that were not still addressed in commercial toolchains. I have created the NocMaker tool to build different NOC and NA designs that can be integrated in FPGA-based MPSoCs. A lot of effort has been made to allow NocMaker designs be easily imported in Quartus II toolchain.

Given the complexity of systems containing hundreds of IP cores, I have shown that working with graphical interfaces is not practical to describe the connection among these blocks. The use of higher level tools is proposed so that this complexity is hidden to make design possible. I have developed such a tool and demonstrated that can it be

used to build large systems like the challenging and top state-of-the-art 128 core many-soft-core processor.

FPGA manufacturers are promoting the use of high level synthesis with the use of OpenCL language to build application specific architectures trying to maximize performance and energy efficiency, especially for the HPC domain. I advocate that building many-soft-cores can be a more flexible alternative because, once synthesized and running on the device, the processors are not bound to a single application, but can be used as general purpose devices. In this context software development plays a crucial role. I identified the missing elements on the soft-core software tool chains necessary to build and optimize parallel applications on many-soft-core architectures. Namely, support for parallel programming models, and support for advanced performance analysis.

To close this gap I evolved the ocMPI library and added support for various delivery strategies. I also provided a method to allow a SPMD approach, avoiding the need to recompile the same source code for several devices. With similar techniques I provide support for threads.

By analyzing many potential workloads to execute in the created platforms I realized that there is a lack of tools that help identifying the parallelism potential of generic sequential applications. I contributed with a new tool that can be useful to detect data dependencies that could block the scalability of applications.

What applications would benefit from reconfigurable multiprocessors? My argument is that there are, at least, two scenarios where they could be applicable. Hard real-time systems and energy efficient HPC workloads.

Electronic systems are increasingly intermingled with mechanical or physical systems, giving birth to the recent concept of cyber-physical systems. Embedded systems taking part on such systems are regularly used in complex control loops where hard-real time requirements are usually found. RTOS have been frequently used to tackle the problems found trying to meet the requirements. RTOS can try to efficiently remove the uncertainties introduced with multitasking OSs, but as multicore processors are widespread, the memory hierarchies are increasingly complex, and the level of resource sharing rises, the level uncertainty in modern processors is surging. The large number of processors in many-soft-core systems can be used to map one task per processor. This task isolation model can remove the uncertainty produced by multitasking. On the other hand the hardware flexibility gives the freedom to try to further reduce the uncertainty produced by resource sharing or memory hierarchies. This simplification can also

produce a benefit in terms of economic cost (an order of magnitude in my presented example).

Another scenario where many-soft-cores are valuable is in the quest for energy efficiency in the HPC domain. It is well known that FPGAs can be more energy efficient that GPGPUs and standard processors for certain applications. But high levels of efficiency are frequently reached after a timely HDL coding process, or lately by using high level synthesis languages and related tools (like OpenCL). These processes create application specific solutions. In designs coded with HDL, the performance driver is often a small piece of hardware implementing an optimized data path, which is replicated many times and is controlled by an additional logic which is responsible for feeding data to the datapath. Then, the development effort of the surrounding control logic is bigger that the performance driver block. In these cases it seems easier to reuse a many-soft-core processor with its parallel programming support to distribute the data among processors and to implement the performance driver modules as coprocessing units of the cores.

I demonstrate that both ideas are feasible. First I applied the methodology to create a Laser controller system with hard real time safety-critical requirements. In this context task isolation and ad-hoc hardware removes the uncertainties of the system, achieving a remarkable performance, and a remarkable reduction of economic cost. Second, I proved that large many-soft-core systems can be built, and that reconfigurable processors can exploit their flexibility in many ways to provide higher levels of energy efficiency without wasting much development time on the control parts that do not determine the performance of the system.

6.2. Future directions

This work tickles the surface of many interconnected domains: Hardware design, computer architecture, compiler architecture, runtime support, performance analysis, application mapping. When thinking on opportunities to evolve this work I feel like in the middle of an open Blue Ocean.

On the hardware level there are many things to be explored in many fronts. On computer architecture I have the opportunity to evolve the MIOS processor to explore new ideas. MIOS uses more resources than NIOSII and it has a worse performance, however I have full flexibility to introduce any change. A potential improvement to MIOS would be to link the communications activity with a clock gating support. Meaning that processors in a many-soft-core reconfigurable platform could be clock-gated until some

work is sent to them to process and, when completed, return to the sleep state. This could have an impact on the energy efficiency of the system.

On the other hand multilevel or shared caches could be implemented to reduce the cache misses for big many-soft-cores and could allow implementing up to 256 processors in the same device used in this work.

Other hardware evolutions could be done in the network and network adapter design. I have shown that the level of overhead introduced by software stacks in message passing architectures is considerable. Most of the overhead is related to the management of the packet storage. Working with simpler primitives, focused on reducing storage needs, would minimize those overheads and provide a performance boost.

Many networks are still not supported by NocMaker, like most indirect networks. It would be very interesting to add support for those networks so that they could be integrated in future MPSoC designs. Indirect networks could offer better performance for certain applications. NocMaker is also limited to the build message passing adapters. This is reasonable for distributed memory systems but limits its applicability to adapters that work with the memory transaction encapsulation approach, which is necessary for embedding the NOC in a shared memory system.

On a similar way, Altera allows specifying the latency used by system interconnect used to access memory mapped devices. Further research should be made to compare how the increase the latency of the bus affects the performance of a many-soft-core system.

Better integration between the Many-soft-core builder and NocMaker could be done. By now NOCs have to be pre-created in NocMaker, and then manually imported in many-soft-core builder. A better integration would allow to speedup the architecture creation.

Another potential improvement to the many-soft-core tool would be to automatically create a virtual prototyping platform from the description generated in the tool. Other methods to create virtual prototyping platforms with transparent instrumentation could be explored, like binary patching or binary translation.

On the programming models domain, an obvious improvement would be to include support for OpenMP and MCAPI, promoted by the Multicore Association.

One of the arguments of the thesis is that we can build an application specific system that still has some general purpose support. Compared with other FPGA based methodologies (like OpenCL toolchains), the benefit could be to avoid to reconfigure the

FPGA device or to allow simultaneous execution of several workloads. This should be tested and verified.

Finally, more applications should be tested to demonstrate the ability of the platform to reach high levels of energy efficiency. In this case a multi-device analysis comparing CPUs, GPGPUs, and FPGAs would be necessary, considering the effort to program each solution.

References

[Acumem] Acumem: Acumem Threadspotter. http://www.roguewave.com/products/threadspotter.aspx

[Aitken14] Aitken, R.; Yeric, G.; Cline, B.; Sinha, S.; Shifren, L.; Iqbal, I. & Chandra, V. Physical design and FinFETs *Proceedings of the 2014 on International symposium on physical design,* **2014**, 65-68

[Altera07] Altera Corporation, "Design Debugging Using the SignalTap II Embedded Logic Analyzer", Mar. **2007**

[Altera10] Altera Nios II Processor Reference Handbook *Document Version: v10.1,* **2010**

[Altera-AN391] Altera Profiling Nios II Systems Application Note AN-391-3.0 **2011**

[Altera12] Altera. "Reducing Power Consumption and Increasing Bandwidth on 28-nm FPGAs", WP-01148-2.0, White Paper **2012**.

[Altera15] Stephen Lim, "Expect a Breakthrough Advantage in Next-Generation FPGAs", WP-01199-1.1 White Paper **2015**

[Altera15b] Altera, "NIOS II Core Implementation Details", NII51015 **2015**

[AltiumTsk] http://techdocs.altium.com/display/FPGA/TSK3000A

[Amdahl67] Amdahl, Gene M. "Validity of the single processor approach to achieving large scale computing capabilities." *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, **1967**.

[Amsler12] Amsler, C. The effects of hardware acceleration on power usage in basic high-performance computing *Kansas State University,* **2012**

[AndroidEmu] http://developer.android.com/tools/help/emulator.html

[Anghel11] Anghel, I.; Cioara, T.; Salomie, I.; Copil, G.; Moldovan, D. & Pop, C. Dynamic frequency scaling algorithms for improving the CPU's energy efficiency *Intelligent Computer Communication and Processing (ICCP), 2011 IEEE International Conference on,* **2011**, 485-491

[ArmM1] http://www.arm.com/products/processors/cortex-m/cortex-m1.php

[Arnold09] Arnold, B. Shrinking possibilities *IEEE Spectrum,* **2009***, 4*, 26-28

[Auth12] Auth, C.; Allen, C.; Blattner, A.; Bergstrom, D.; Brazier, M.; Bost, M.; Buehler, M.; Chikarmane, V.; Ghani, T.; Glassman, T. & others A 22nm high performance and low-power CMOS technology featuring fully-depleted tri-gate transistors, self-aligned contacts and high density MIM capacitors *VLSI Technology (VLSIT), 2012 Symposium on,* **2012**, 131-132

[Bai04] Bai, P.; Auth, C.; Balakrishnan, S.; Bost, M.; Brain, R.; Chikarmane, V.; Heussner, R.; Hussein, M.; Hwang, J.; Ingerly, D. & others A 65nm logic technology featuring 35nm gate lengths, enhanced channel strain, 8 Cu interconnect layers, low-k ILD and 0.57 $m 2 SRAM cell *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International,* **2004**, 657-660

[Balart04] Balart, J.; Duran, A.; Gonzalez, M.; Martorell, X.; Ayguade, E. & Labarta, J. "Nanos mercurium: a research compiler for openmp". *In Proceedings of the European Workshop on OpenMP , volume 8*, **2004**.

[Ball07] Ball, J. Designing Soft-Core Processors for FPGAs *Processor Design, Springer,* **2007**, 229-256

[Baklouti14] Baklouti, M. & Abid, M. Multi-Softcore architecture on FPGA *International Journal of Reconfigurable Computing, Hindawi Publishing Corp.,* **2014***, 2014*, 14

[Barthe11] Barthe, Lyonel, et al. "Optimizing an open-source processor for FPGAs: A case study." *Field Programmable Logic and Applications (FPL), 2011 International Conference on*. IEEE, **2011**.

[Barthe11b] Barthe, Lyonel, et al. "The secretblaze: A configurable and cost-effective open-source soft-core processor." *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE, **2011**.

[Basumallik07] Basumallik, Ayon, Seung-Jai Min, and Rudolf Eigenmann. "Programming distributed memory sytems using OpenMP." *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, **2007**.

[Bao09] Bao, Y. & Brorsson, M. An Implementation of Cache-Coherence for the Nios II™ Soft-core processor *2nd Swedish Workshop on Multi-core Computing,* **2009**

[Bell03] Bell, Robert, Allen D. Malony, and Sameer Shende. "ParaProf: A portable, extensible, and scalable tool for parallel performance profile analysis." *Euro-Par 2003 Parallel Processing*. Springer Berlin Heidelberg, **2003**. 17-26.

[Bellard05] Bellard, Fabrice. "QEMU, a Fast and Portable Dynamic Translator." *USENIX Annual Technical Conference, FREENIX Track*. **2005**.

[Bellows98] Bellows, P. & Hutchings, B. JHDL-an HDL for reconfigurable systems *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on,* **1998**, 175-184

[Ben14] Ben Asher, Yosi, et al. "1K manycore FPGA shared memory architecture for SOC." *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. ACM, **2014**.

[Benini02] Benini, L. & De Micheli, G. Networks on chips: a new SoC paradigm *Computer, IEEE,* **2002***, 35*, 70-78

[Bernat12] Bernat, G.; Colin, A. & Petters, S. M. WCET analysis of probabilistic hard real-time systems *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE,* **2002**, 279-288

[Bertozzi04] D. Bertozzi, et al. . "Xpipes: A Network-on-Chip Architecture for Gigascale Systems-on-Chip". IEEE Circuits and Systems Magazine, v.4(2), pp. 18-31, **2004**

[Birrell87] Birrell, Andrew, et al. *Synchronization primitives for a multiprocessor: A formal specification*. Vol. 21. No. 5. ACM, **1987**.

[Böhm10] Böhm, I.; Franke, B. & Topham, N. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator Embedded Computer Systems (SAMOS), 2010 International Conference on, **2010**, 1 -10

[Bohr11] Bohr, M. The evolution of scaling from the homogeneous era to the heterogeneous era *Electron Devices Meeting (IEDM), 2011 IEEE International,* **2011**, 1-1

[Bouchhima05] Bouchhima, Aimen, et al. "Using abstract CPU subsystem simulation model for high level HW/SW architecture exploration." *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*. ACM, **2005**.

[Brewer88] Brewer, O.; Dongarra, J. & Sorensen, D. “Tools to aid in the analysis of memory access patterns for FORTRAN Programs” *Parallel Computing, Elsevier,* **1988***, 9*, 25-35

[Brunst01] Brunst, Holger, et al. "Performance optimization for large scale computing: The scalable VAMPIR approach." *Computational Science-ICCS 2001*. Springer Berlin Heidelberg, **2001**. 751-760.

[Castells06] Castells-Rufas, David, Jaume Joven, and Jordi Carrabina. "A validation and performance evaluation tool for ProtoNoC." *System-on-Chip, 2006. International Symposium on*. IEEE, **2006**.

[Castells09] Castells-Rufas, D., et al. "NocMaker: A cross-platform open-source design space exploration tool for networks on chip." *INA-OCMC Workshop, Paphos, Cyprus*. **2009**.

[Castells10] Castells-Rufas, David, et al. "MPSoC performance analysis with virtual prototyping platforms." *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. IEEE, **2010**.

[Castells10b] Castells-Rufas, D.; Joven, J. & Carrabina, J. “Scalability of a Parallel JPEG Encoder on Shared Memory Architectures” *2010 39th International Conference on Parallel Processing*, **2010**, 502-507

[Castells11] Castells-Rufas, D.; Fernandez-Alonso, E.; Carrabina, J. & Joven, J. Sharing FPUs in many-soft-cores *Field-Programmable Technology (FPT), 2011 International Conference on,* **2011**, 1-6

[Castells12] Castells-Rufas, D.; Vila-Closas, O. & Carrabina, J. Design of a multi-soft-core based Laser Marking controller *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on,* **2012**, 1-6

[Castells14] Castells-Rufas, D.; Carrabina, J.; González de Aledo Marugán, P. & Sánchez Espeso, P. “Fast Trace Generation of Many-Core Embedded Systems with Native Simulation” *In proceeding of: High Performance Energy Efficient Embedded Systems (HIP3ES 2014),* **2014**

[Castells15] Castells-Rufas, D. & Carrabina, J. 128-core Many-Soft-Core Processor with MPI support *Jornadas de Computación Reconfigurable y Aplicaciones (JCRA),* **2015**

[Chan04] Chan, J. et al. "NoCGEN: A Template Based Reuse Methodology for Networks on Chip Architecture". In: VLSI’04. **2004**, pp 717-720

[Chapman03] Chapman, Ken. "Picoblaze 8-bit microcontroller for virtex-e and spartan-ii/iie devices." *Xilinx Application Notes* (**2003**).

[Chen11] Chen, H.; Yin, L. & Peng, G. Implementation of multi-core embedded system on compound guidance system *Electronics, Communications and Control (ICECC), 2011 International Conference on,* **2011**, 4348-4351

[Chi15] Chi, C. C.; Alvarez-Mesa, M.; Bross, B.; Juurlink, B. & Schierl, T. SIMD Acceleration for HEVC Decoding *Circuits and Systems for Video Technology, IEEE Transactions on,* **2015***, 25*, 841-855

[Choi13] Choi, J.; Brown, S. & Anderson, J. From software threads to parallel hardware in high-level synthesis for FPGAs *Field-Programmable Technology (FPT), 2013 International Conference on,* **2013**, 270-277

[Cichowski12] Cichowski, P.; Keller, Jö. & Kessler, C. Modelling power consumption of the Intel SCC *The 6th Many-core Applications Research Community (MARC) Symposium,* **2012**, 46-51

[Cooper88] Cooper, E. and Draves, R. C Threads. Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June, **1988**.

[Corina10] Corina, M. “Quantitative analysis and visualization of memory access patterns” *TU Delft, Delft University of Technology,* **2010**

[Coupé12] Cheryl Coupé, “Bandwidth Demands Drive FPGA/PLD Market” **2012** http://eecatalog.com/fpga/2012/01/27/bandwidth-demands-drive-fpgapld-market/

[CritBlueP] Critical Blue. Prism, http://www.criticalblue.com/

[Curreri10] Curreri, J.; Koehler, S.; George, A.; Holland, B. & Garcia, R. Performance analysis framework for high-level language applications in reconfigurable computing ACM Transactions on Reconfigurable Technology and Systems (TRETS), ACM, **2010**, 3, 5

[Czechowski14] Czechowski, K.; Lee, V. W.; Grochowski, E.; Ronen, R.; Singhal, R.; Vuduc, R. & Dubey, P. Improving the energy efficiency of big cores *Proceeding of the 41st annual international symposium on Computer architecuture,* **2014**, 493-504

[Cragon80] Cragon, H. G. The elements of single-chip microcomputer architecture *Computer, IEEE,* **1980**, 27-41

[Critcklow63] Critcklow, A. J. "Generalized multiprocessing and multiprogramming systems."*Proceedings of the November 12-14, 1963, fall joint computer conference*. ACM, **1963**.

[Dahlby04] Dahlby, Doug. "Applying agile methods to embedded systems development."*Embedded Software Design Resources* 41 (**2004**): 1014123.

[Dalay03] Dalay, B. Accelerating system performance using SOPC builder *System-on-Chip, 2003. Proceedings. International Symposium on,* **2003**, 3-5

[Dally01] Dally, W. J. & Towles, B. Route packets, not wires: On-chip interconnection networks *Design Automation Conference, 2001. Proceedings,* **2001**, 684-689

[Dally04] Dally, W. J. & Towles, B. P. Principles and practices of interconnection networks *Elsevier,* **2004**

[Dao00] Dao, Tuan Q., and Donald E. Steiss. "Shared floating-point unit in a single chip multiprocessor." U.S. Patent No. 6,148,395. 14 Nov. **2000**.

[Davis11] Davis, R. I. & Burns, A. A Survey of Hard Real-time Scheduling for Multiprocessor Systems *ACM Comput. Surv., ACM,* **2011***, 43*, 35:1-35:44

[Dennard74] Dennard, R. H.; Rideout, V.; Bassous, E. & Leblanc, A. Design of ion-implanted MOSFET's with very small physical dimensions *Solid-State Circuits, IEEE Journal of, IEEE,* **1974***, 9*, 256-268

[Deval15] Deval, A.; Ananthakrishnan, A. & Forbell, C. Power management on 14 nm Inteltextregistered Core- M processor *Low-Power and High-Speed Chips (COOL CHIPS XVIII), 2015 IEEE Symposium in,* **2015**, 1-3

[Dhrystone] Weicker, R. P. Dhrystone: a synthetic systems programming benchmark *Communications of the ACM, ACM,* **1984***, 27*, 1013-1030

[Diaz12] Diaz, Javier, Camelia Munoz-Caro, and Alfonso Nino. "A survey of parallel programming models and tools in the multi and many-core era." *Parallel and Distributed Systems, IEEE Transactions on* 23.8 (**2012**): 1369-1386.

[Djoudi05] Djoudi, Lamia, et al. "Maqao: Modular assembler quality analyzer and optimizer for itanium 2." *The 4th Workshop on EPIC architectures and compiler technology, San Jose*. **2005**.

[Donofrio09] Donofrio, D.; Oliker, L.; Shalf, J.; Wehner, M. F.; Rowen, C.; Krueger, J.; Kamil, S. & Mohiyuddin, M. Energy-efficient computing for extreme-scale science *Computer, IEEE,* **2009**, 62-71

[Dorta10] Dorta, Taho, et al. "Reconfigurable multiprocessor systems: a review."*International Journal of Reconfigurable Computing* 2010 (**2010**): 7.

[Duato93] Duato, José. "A new theory of deadlock-free adaptive routing in wormhole networks." *Parallel and Distributed Systems, IEEE Transactions on* 4.12 (**1993**): 1320-1331.

[Duato03] Duato, J.; Yalamanchili, S. & Ni, L. M. Interconnection networks: An engineering approach *Morgan Kaufmann,* **2003**

[Dykes07] Dykes, J.; Chan, P.; Chapman, G. & Shannon, L. A Multiprocessor System-on-Chip Implementation of a Laser-based Transparency Meter on an FPGA *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on,* **2007**, 373-376

[Ellis85] Ellis, John R. *Bulldog: A compiler for VLIW architectures*. Yale Univ., New Haven, CT (USA), **1985**.

[Esmaeilzadeh11] Esmaeilzadeh, H.; Blem, E.; Amant, R. S.; Sankaralingam, K. & Burger, D. Dark silicon and the end of multicore scaling *Computer Architecture (ISCA), 2011 38th Annual International Symposium on,* **2011**, 365-376

[Feng03] Feng, W.-C. Making a case for efficient supercomputing *Queue, ACM,* **2003***, 1*, 54

[Fernandez10] Fernandez-Alonso, E.; Castells-Rufas, D.; Risueno, S.; Carrabina, J. & Joven, J. A NoC-based multi-soft-core with 16 cores *Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on,* **2010**, 259 -262

[Fernandez14] Fernandez-Alonso, E. Offloading Techniques to Improve Performance on MPI Applications in NoC-Based MPSoCs *Engineering School. Universitat Autònoma de Barcelona.,***2014**

[Flynn72] Flynn, Michael J. "Some computer organizations and their effectiveness."*Computers, IEEE Transactions on* 100.9 (**1972**): 948-960.

[Foster13] Foster, H. D. Why the Design Productivity Gap Never Happened *Proceedings of the International Conference on Computer-Aided Design, IEEE Press,* **2013**, 581-584

[Freitas07] Freitas, H.; Colombo, D.; Kastensmidt, F. L. & Navaux, P. Evaluating Network-on-Chip for Homogeneous Embedded Multiprocessors in FPGAs *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on,* **2007**, 3776-3779

[Gabriel04] Gabriel, Edgar, et al. "Open MPI: Goals, concept, and design of a next generation MPI implementation." *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Berlin Heidelberg, **2004**. 97-104.

[Gaisler03] Gaisler, Jiri. "The LEON-2 processor user's manual." *Version* 1 (**2003**): 14.

[Gaisler07] Gaisler, Jiri, et al. "GRLIB IP core user's manual." *Gaisler research* (**2007**).

[Gayasen04] Gayasen, A.; Tsai, Y.; Vijaykrishnan, N.; Kandemir, M.; Irwin, M. J. & Tuan, T. Reducing leakage energy in FPGAs using region-constrained placement *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays,* **2004**, 51-58

[George07] George, V.; Jahagirdar, S.; Tong, C.; Smits, K.; Damaraju, S.; Siers, S.; Naydenov, V.; Khondker, T.; Sarkar, S. & Singh, P. Penryn: 45-nm next generation Inteltextregistered core™ 2 processor *Solid-State Circuits Conference, 2007. ASSCC'07. IEEE Asian,* **2007**, 14-17

[Giefers10] Giefers, H. & Platzner, M. A Triple Hybrid Interconnect for Many-Cores: Reconfigurable Mesh, NoC and Barrier *Field Programmable Logic and Applications (FPL), 2010 International Conference on,* **2010**, 223-228

[Goodman83] Goodman, J. R. Using cache memory to reduce processor-memory traffic *ACM SIGARCH Computer Architecture News,* **1983***, 11*, 124-131

[Gong08] Gong, N.; Guo, B.; Lou, J. & Wang, J. Analysis and optimization of leakage current characteristics in sub-65nm dual V t footed domino circuits *Microelectronics Journal, Elsevier,* **2008***, 39*, 1149-1155

[Gray00] Gray, J. Building a RISC system in an FPGA *Circuit Cellar Magazine,* **2000**, 117

[green500] http://www.green500.org/

[Gropp99] Gropp, William, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press, **1999**.

[Göhringer10] Göhringer, Diana, et al. "Message passing interface support for the runtime adaptive multi-processor system-on-chip RAMPSoC." *Embedded Computer Systems (SAMOS), 2010 International Conference on*. IEEE, **2010**.

[Göhringer14] Göhringer, Diana. "Reconfigurable Multiprocessor Systems: Handling Hydras Heads--A Survey." *ACM SIGARCH Computer Architecture News* 42.4 (**2014**): 39-44.

[Gromova12] Gromova, V. Optimize Data Structures and Memory Access Patterns to Improve Data Locality *Intel Guide for Developing Multithreaded Applications,* **2012**

[Guan01] Guan, D.; Yu, S.; Liang, C. & Wang, X. MPEG-2 TS generate system and its implementation with FPGA *ASIC, 2001. Proceedings. 4th International Conference on,* **2001**, 510-513

[Gupta04] Gupta, H. Integration of a Floating Point Unit with the Leon Processor *Indian Institute of Technology Delhi,* **2004**

[Hamada09] Hamada, T.; Benkrid, K.; Nitadori, K. & Taiji, M. A Comparative Study on ASIC, FPGAs, GPUs and General Purpose Processors in the O(N^2) Gravitational N-body Simulation *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on,* **2009**, 447-452

[Hambarde14] Hambarde, Prasanna, Rachit Varma, and Somesh Jha. "The Survey of Real Time Operating System: RTOS." *Electronic Systems, Signal Processing and Computing Technologies (ICESC), 2014 International Conference on*. IEEE, **2014**.

[Han12] Han, Z.; Wang, J. & Zeng, Y. Qsys NOC-based MPSOC design for LAMOST Spectrographs *SPIE Astronomical Telescopes+ Instrumentation,* **2012**, 84513D-84513D

[Hansen70] Hansen, Per Brinch. "The nucleus of a multiprogramming system. "*Communications of the ACM* 13.4 (**1970**): 238-241.

[Hennessy81] Hennessy, J.; Jouppi, N.; Baskett, F. & Gill, J. MIPS: a VLSI processor architecture *Springer,* **1981**

[Hennessy11] Hennessy, John L., and David A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, **2011**.

[Henry92] Henry, Dana S., and Christopher F. Joerg. "A tightly-coupled processor-network interface." *ACM SIGPLAN Notices*. Vol. 27. No. 9. ACM, **1992**.

[Hoare76] Hoare, C. A. R. *Parallel programming: an axiomatic approach*. Springer Berlin Heidelberg, **1976**.

[Hoare78] Hoare, Charles Antony Richard. "Communicating sequential processes."*Communications of the ACM* 21.8 (**1978**): 666-677.

[Howard10] Howard, J.; Dighe, S.; Hoskote, Y.; Vangal, S.; Finan, D.; Ruhl, G.; Jenkins, D.; Wilson, H.; Borkar, N.; Schrom, G. & others A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International,* **2010**, 108-109

[Hubert07] Hubert, H.; Stabernack, B. & Wels, K.-I. Performance and Memory Profiling for Embedded System Design Industrial Embedded Systems, **2007**. SIES '07. International Symposium on, 2007, 94 -101

[Hubner05] Hubner, M.; Paulsson, K. & Becker, J. Parallel and Flexible Multiprocessor System-On-Chip for Adaptive Automotive Applications based on Xilinx MicroBlaze Soft-Cores *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International,* **2005**, 149a-149a

[Huerta05] Huerta, P.; Castillo, J.; Mártinez, J. & Lopez, V. A microblaze based multiprocessor soc *WSEAS transactions on circuits and systems,* **2005***, 4*, 423-430

[Huerta07] Huerta, P.; Castillo, J.; Martinez, J. & Pedraza, C. Exploring FPGA Capabilities for Building Symmetric Multiprocessor Systems *Programmable Logic, **2007**. SPL '07. 2007 3rd Southern Conference on,* **2007**, 113-118

[Hung05] Hung, A.; Bishop, W. & Kennings, A. Symmetric multiprocessing on programmable chips made easy *Design, Automation and Test in Europe, 2005. Proceedings,* **2005**, 240-245 Vol. 1

[Inglart08] Inglart, N.; Niar, S.; Cohen, A. “Hybrid performance analysis to accelerate compiler optimization space exploration for in-order processors.” In 2nd*Workshop on Statistical and Machine learning approaches applied to ARchitectures and compilaTion (SMART)*, Gothenburg, Sweden, January **2008**

[IntelVt] Intel: Intel Vtune Amplifier XE. www.intel.com/software/products/vtune

[Islam10] Islam, R.; Sabbavarapu, A.; Patel, R.; Kumar, M.; Nguyen, J.; Patel, B. & Kontu, A. Next generation Inteltextregistered ATOM™ processor based ultra low power SoC for handheld applications *Solid State Circuits Conference (A-SSCC), 2010 IEEE Asian,* **2010**, 1-4

[ITRSXX] International technology roadmap for semiconductors (ITRS) *Semiconductor Industry Association,* **20XX**

[Iwai09] Iwai, H. Roadmap for 22nm and beyond *Microelectronic Engineering, Elsevier,* **2009***, 86*, 1520-1528

[Jan12] Jan, C.-H.; Bhattacharya, U.; Brain, R.; Choi, S.-J.; Curello, G.; Gupta, G.; Hafez, W.; Jang, M.; Kang, M.; Komeyli, K. & others A 22nm SoC platform technology featuring 3-D tri-gate and high-k/metal gate, optimized for ultra low power, high performance and high density SoC applications *Electron Devices Meeting (IEDM), 2012 IEEE International,* **2012**, 3-1

[Jantsch06] Jantsch A. Nocsim: A NoC Simulator. School of Information and Communication Technology, Royal Institute of Technology, Stockholm, version 0.4 alpha edition, **2006**.

[Jensen08] Jensen, David W. "Developing System-on-Chips with Moore, Amdahl, Pareto, and Ohm." *Electro/Information Technology, **2008**. EIT 2008. IEEE International Conference on*. IEEE, 2008.

[Jin05] Jin, Y.; Satish, N.; Ravindran, K. & Keutzer, K. An Automated Exploration Framework for FPGA-based Soft Multiprocessor Systems *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, ACM,* **2005**, 273-278

[Jing13] Jing, F.; xia null, L. J.; Long, M. & He, C. Feature level fusion of SAR and optical image and MPSoC IMPLEMENTATION *Radar Conference 2013, IET International,* **2013**, 1-6

[Jones04] Jones, A.; Hoare, R.; Kourtev, I.; Fazekas, J.; Kusic, D.; Foster, J.; Boddie, S. & Muaydh, A. A 64-way VLIW/SIMD FPGA architecture and design flow *Electronics, Circuits and Systems, 2004. ICECS 2004. Proceedings of the 2004 11th IEEE International Conference on,* **2004**, 499-502

[Jose15] Jose, W.; Neto, H. & Vestias, M. A Many-Core Co-Processor for Embedded Parallel Computing on FPGA *Digital System Design (DSD), 2015 Euromicro Conference on,* **2015**, 539-542

[Joven08] Joven, J., et al. "xENoC-an experimental network-on-chip environment for parallel distributed computing on NoC-based MPSoC architectures." *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*. IEEE, **2008**.

[Joven09] Joven, Jaume, et al. "NoCMaker & j2eMPI A Complete HW-SW Rapid Prototyping EDA Tool for Design Space Exploration of NoC-based MPSoCs."*IEEE/ACM Design, Automation and Test in Europe*. **2009**.

[Joven10] Joven Murillo, Jaume, and Jordi Carrabina i Bordoll. *HW-sw components for parallel embedded computing on noc-based mpsocs*. Universitat Autònoma de Barcelona,, **2010**.

[Joven11] Joven, J. et al. "Hw-sw implementation of a decoupled fpu for arm-based cortex-m1 socs in fpgas." *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*. IEEE, **2011**.

[Joven13] Joven, Jaume, et al. "QoS-driven reconfigurable parallel computing for NoC-based clustered MPSoCs." *Industrial Informatics, IEEE Transactions on* 9.3 (**2013**): 1613-1624.

[Kahle04] Kahle, James Allan, and Charles Roberts Moore. "Shared execution unit in a dual core processor." U.S. Patent No. 6,725,354. 20 Apr. **2004**.

[Kennedy14] Kennedy, Ken, and Kathryn S. McKinley. "Optimizing for parallelism and data locality." 25th Anniversary International Conference on Supercomputing Anniversary Volume. ACM, **2014**.

[Kestur10] Kestur, S.; Davis, J. D. & Williams, O. Blas comparison on fpga, cpu and gpu *VLSI (ISVLSI), 2010 IEEE computer society annual symposium on,* **2010**, 288-293

[Khan09] Khan, J.; Niar, S.; Saghir, M.; El-Hillali, Y. & Rivencq-Menhaj, A. Trade-Off Exploration for Target Tracking Application in a CustomizedMultiprocessor Architecture *EURASIP Journal on Embedded Systems,* **2009***, 2009*, 21-pages

[Kiefer15] Kiefer, G.; Seider, M. & Schaeferling, M. ParaNut-An Open, Scalable, and Highly Parallel Processor Architecture for FPGA-based Systems *Embedded world Conference,* **2015**

[Knüpfer12] Knüpfer, Andreas, et al. "Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir." *Tools for High Performance Computing 2011*. Springer Berlin Heidelberg, **2012**. 79-91.

[Koehler11] Koehler, S.; Stitt, G. & George, A. D. Platform-aware bottleneck detection for reconfigurable computing applications ACM Trans. Reconfigurable Technol. Syst., ACM, **2011**, 4, 30:1-30:28

[Koliaï13] Koliaï, Souad, et al. "Quantifying performance bottleneck cost through differential analysis." *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, **2013**.

[Kondo13] Kondo, M.; Nguyen, S.; Hirao, T.; Soga, T.; Sasaki, H. & Inoue, K. SMYLEref: A reference architecture for manycore-processor SoCs *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific,* **2013**, 561-564

[Kornaros10] Kornaros, G. A soft multi-core architecture for edge detection and data analysis of microarray images *Journal of Systems Architecture,* **2010***, 56*, 48 – 62

[Kozyrakis02] Kozyrakis, C. & Patterson, D. Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture,* **2002**, 283-293

[Krasnov07] Krasnov, A.; Schultz, A.; Wawrzynek, J.; Gibeling, G. & Droz, P. RAMP Blue: A message-passing manycore system in FPGAs *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on,* **2007**, 54-61

[Kulmala06] Kulmala, A.; Lehtoranta, O.; Hämäläinen, T. D. & Hännikäinen, M. Scalable MPEG-4 encoder on FPGA multiprocessor SOC *EURASIP journal on Embedded Systems, Hindawi Publishing Corp.,* **2006***, 2006*, 8-8

[Kumar04] Kumar, R.; Jouppi, N. & Tullsen, D. Conjoined-Core Chip Multiprocessing *Microarchitecture, 2004. MICRO-37* ***2004****. 37th International Symposium on,* , 195-206

[Kuon07] Kuon, I. & Rose, J. Measuring the gap between FPGAs and ASICs *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, IEEE,* **2007***, 26*, 203-215

[Kwon15] Kwon, W.-C. & Peh, L.-S. A Universal Ordered NoC Design Platform for Shared-memory MPSoC *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, IEEE Press,* **2015**, 697-704

[Lamport74] Lamport, Leslie. "A new solution of Dijkstra's concurrent programming problem." Communications of the ACM 17.8 (**1974**): 453-455.

[Lebedev10] Lebedev, I.; Cheng, S.; Doupnik, A.; Martin, J.; Fletcher, C.; Burke, D.; Lin, M. & Wawrzynek, J. MARC: A many-core approach to reconfigurable computing *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on,* **2010**, 7-12

[Lecler11] Lecler, Jean-Jacques, and Gilles Baillieu. "Application driven network-on-chip architecture exploration & refinement for a complex SoC." *Design Automation for Embedded Systems* 15.2 (**2011**): 133-158.

[Lee09] Lee, S.-h. The Design and implementation of parallel processing system using the Nios II embedded processor *Journal of the Korea Society of Computer and Information, Korean Society of Computer Information,* **2009***, 14*, 97-103

[Lehtoranta05] Lehtoranta, I.; Salminen, E.; Kulmala, A.; Hannikainen, M. & Hamalainen, T. A parallel MPEG-4 encoder for FPGA based multiprocessor SoC *Field Programmable Logic and Applications, 2005. International Conference on,* **2005**, 380-385

[Leow13] Leow, Y. K.; Akoglu, A. & Lysecky, S. An Analytical Model for Evaluating Static Power of Homogeneous FPGA Architectures *ACM Transactions on Reconfigurable Technology and Systems (TRETS), ACM,* **2013***, 6*, 18

[Li03] Li, S. Y.; Cheuk, G. C.; Lee, K.-H. & Leong, P. H. FPGA-based SIMD processor *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03),* **2003**, 267

[LM32] LatticeMico32 Website: http://www.latticesemi.com/

[Lorenz05] Lorenz, Juergen, et al. "Vectorization techniques for the Blue Gene/L double FPU." *IBM Journal of Research and Development* 49.2.3 (**2005**): 437-446.

[Lu02] Lu, Y.-H.; Benini, L. & De Micheli, G. Dynamic frequency scaling with buffer insertion for mixed workloads *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, IEEE,* **2002***, 21*, 1284-1305

[Lu03] Lu, Zhonghai, and Raimo Haukilahti. "NOC application programming interfaces: high level communication primitives and operating system services for power management." *Networks on chip*. Kluwer Academic Publishers, **2003**.

[Lu07] Lu, S.-L. L.; Yiannacouras, P.; Kassa, R.; Konow, M. & Suh, T. An FPGA-based Pentium® in a complete desktop system *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays,* **2007**, 53-59

[Lv09] Lv, M.; Guan, N.; Zhang, Y.; Deng, Q.; Yu, G. & Zhang, J. A survey of WCET analysis of real-time operating systems *Embedded Software and Systems, 2009. ICESS'09. International Conference on,* **2009**, 65-72

[Mahadevan05] Mahadevan, S. et al., "A network traffic generator model for fast network-on-chip simulation," in Proc. DATE, Mar. **2005**, pp. 780-785.

[Mahr08] Mahr, Philipp, et al. "SoC-MPI: A flexible message passing library for multiprocessor systems-on-chips." *Reconfigurable Computing and FPGAs, 2008. ReConFig'08. International Conference on*. IEEE, **2008**.

[Mallón09] Mallón, Damián A., et al. "Performance evaluation of MPI, UPC and OpenMP on multicore architectures." *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Berlin Heidelberg, **2009**. 174-184.

[Mandelbrot13] Mandelbrot, Benoit. *Fractals and chaos: the Mandelbrot set and beyond*. Vol. 3. Springer Science & Business Media, **2013**.

[Martina02] Martina, M.; Molino, A. & Vacca, F. FPGA system-on-chip soft IP design: a reconfigurable DSP *Circuits and Systems, 2002. MWSCAS-2002. The 2002 45th Midwest Symposium on,* **2002***, 3*, III-196-III-199 vol.3

[Martins15] Martins, M.; Matos, J.; Ribas, R.; Reis, A.; Schlinker, G.; Rech, L. & Michelsen, J. Open Cell Library in 15nm FreePDK Technology *Proc. of the Int'l Symp. on Physical Design (ISPD),* **2015**

[Massa03] Massa, Anthony J. *Embedded software development with eCos*. Prentice Hall Professional, **2003**.

[Matas97] Matas, B. & DeSubercausau, C. Memory, 1997: Complete Coverage of DRAM, Sram, EPROM, and Flash Memory IC's *Integrated Circuit Engineering Corp.,* **1997**

[Matilainen11] Matilainen, Lauri, et al. "Multicore Communications API (MCAPI) implementation on an FPGA multiprocessor." *Embedded Computer Systems (SAMOS), 2011 International Conference on*. IEEE, **2011**.

[Mattson10] Mattson, Timothy G., et al. "The 48-core SCC processor: the programmer's view." *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, **2010**.

[Meenderinck09] Meenderinck, C. & Juurlink, B. (When) Will CMPs Hit the Power Wall? *Euro-Par 2008 Workshops-Parallel Processing,* **2009**, 184-193

[Meltzer99] Meltzer, David. "Single chip multiprocessor with shared execution units." U.S. Patent No. 5,987,587. 16 Nov. **1999**.

[MikeFPU] Schoegg, M. Configurable FPU *http://www.alterawiki.com/wiki/Configurable_FPU*

[Mistry07] Mistry, K.; Allen, C.; Auth, C.; Beattie, B.; Bergstrom, D.; Bost, M.; Brazier, M.; Buehler, M.; Cappellani, A.; Chau, R. & others A 45nm logic technology with high-k+ metal gate transistors, strained silicon, 9 Cu interconnect layers, 193nm dry patterning, and 100% Pb-free packaging *Electron Devices Meeting, 2007. IEDM 2007. IEEE International,* **2007**, 247-250

[Monton07] Monton, M.; Portero, A.; Moreno, M.; Martinez, B. & Carrabina, J. "Mixed SW/systemC SoC emulation framework" Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on, **2007**, 2338-2341

[Moore75] Moore, G. E. & others Progress in digital integrated electronics *IEDM Tech. Digest,* **1975***, 11*

[Murali06] Murali, S., et al. "Sunfloor: Application-Specific Design of Networks-on-Chip."*Poster presentation at University Booth at the Design, Automation and Test in Europe Conference and Exhibition*. **2006**.

[Murali06b] Murali, Srinivasan, et al. "Designing message-dependent deadlock free networks on chips for application-specific systems on chips." *Very Large Scale Integration, 2006 IFIP International Conference on*. IEEE, **2006**.

[Mplemenos08] Mplemenos, G. & Papaefstathiou, I. MPLEM: An 80-processor FPGA Based Multiprocessor System *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on, IEEE Computer Society,* **2008***, 0*, 273-274

[Nagel96] Nagel, Wolfgang E., et al. "VAMPIR: Visualization and analysis of MPI resources." (**1996**).

[Nakatani89] Nakatani, Toshio, and Kemal Ebcioğlu. ""Combining" as a compilation technique for VLIW architectures." *ACM SIGMICRO Newsletter*. Vol. 20. No. 3. ACM, **1989**.

[Nalamalpu15] Nalamalpu, A.; Kurd, N.; Deval, A.; Mozak, C.; Douglas, J.; Khanna, A.; Paillet, F.; Schrom, G. & Phelps, B. Broadwell: A family of IA 14nm processors *VLSI Circuits (VLSI Circuits), 2015 Symposium on,* **2015**, C314-C315

[Natarajan14] Natarajan, S.; Agostinelli, M.; Akbar, S.; Bost, M.; Bowonder, A.; Chikarmane, V.; Chouksey, S.; Dasgupta, A.; Fischer, K.; Fu, Q. & others A 14nm logic technology featuring 2 nd-generation FinFET, air-gapped interconnects, self-aligned double patterning and a 0.0588 $m 2 SRAM cell size *Electron Devices Meeting (IEDM), 2014 IEEE International,* **2014**, 3-7

[Nethercote07] Nethercote, Nicholas, and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." *ACM Sigplan notices*. Vol. 42. No. 6. ACM, **2007**.

[Oberman99] Oberman, S. Floating point division and square root algorithms and implementation in the AMD-K7TM microprocessor *Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on,* **1999**, 106-115

[Packan09] Packan, P.; Akbar, S.; Armstrong, M.; Bergstrom, D.; Brazier, M.; Deshpande, H.; Dev, K.; Ding, G.; Ghani, T.; Golonzka, O. & others High Performance 32nm Logic Technology Featuring 2 nd Generation High-k+ Metal Gate Transistors *Electron Devices Meeting (IEDM), 2009 IEEE International,* **2009**, 1-4

[Pande05] Pande, P. P. et al., "Performance evaluation and design trade-offs for network-on-chip interconnect architectures," IEEE Trans. Comput., vol. 54, no. 8, pp. 1025–1039, Aug. **2005**.

[Podobas14] Podobas, A. Accelerating Parallel Computations with OpenMP-Driven System-on-Chip Generation for FPGAs *Embedded Multicore/Manycore SoCs (MCSoc), 2014 IEEE 8th International Symposium on,* **2014**, 149-156

[Pantelopoulos12] Pantelopoulos, S. & Brokalakis, A. "Deliverable D2. 4 - Parallel Hardware System Specifications", from Project No: FP7-ICT- 247615, **2012**

[Parulkar08] Parulkar, Ishwar, et al. "OpenSPARC: An open platform for hardware reliability experimentation." *Fourth Workshop on Silicon Errors in Logic-System Effects (SELSE)*. **2008**.

[Patterson85] Patterson, D. A. Reduced instruction set computers *Communications of the ACM, ACM,* **1985***, 28*, 8-21

[Pillet95] Pillet, Vincent, et al. "Paraver: A tool to visualize and analyze parallel code."*Proceedings of WoTUG-18: Transputer and occam Developments*. Vol. 44. mar, **1995**.

[Pitter08] Pitter, C. & Schoeberl, M. Performance evaluation of a java chip-multiprocessor *Industrial Embedded Systems, 2008. SIES 2008. International Symposium on,* **2008**, 34-42

[PJPEGENC] https://github.com/davidcastells/pjpegenc

[Plumbridge13] Plumbridge, G. & Audsley, N. Programming FPGA based NoCs with Java *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on,* **2013**, 1-6

[Plumbridge14] Plumbridge, G.; Whitham, J. & Audsley, N. Blueshell: A Platform for Rapid Prototyping of Multiprocessor NoCs and Accelerators *SIGARCH Comput. Archit. News, ACM,* **2014***, 41*, 107-117

[Posadas04] Posadas, H.; Herrera, F.; Sanchez, P.; Villar, E. & Blasco, F. System-level performance analysis in SystemC Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings, **2004**, 1, 378 - 383 Vol.1

[Posadas11] Posadas, H.; Real, S. & Villar, E. M3-SCoPE: Performance Modeling of Multi-Processor Embedded Systems for Fast Design Space Exploration Multi-Objective Design Space Exploration of Multiprocessor SOC Architectures: The Multicube Approach, Springer Verlag, **2011**, 19

[Rashtchi14] Rashtchi, V. & Nourazar, M. A MULTIPROCESSOR Nios II IMPLEMENTATION OF DUFFING OSCILLATOR ARRAY FOR WEAK SIGNAL DETECTION *Journal of Circuits, Systems, and Computers, World Scientific,* **2014***, 23*, 1450054

[Rashti08] Rashti, M. & Afsahi, A. Improving communication progress and overlap in MPI Rendezvous protocol over RDMA-enabled interconnects 22nd International Symposium on High Performance Computing Systems and Applications (HPCS 2008), **2008**, 95-101.

[Ravindran09] Ravindran, K.; Satish, N.; Jin, Y. & Keutzer, K. An FPGA-based soft multiprocessor system for IPv4 packet forwarding *Field Programmable Logic and Applications, 2005. International Conference on,* **2005**, 487-492

[Raza14] Raza, M. & Azeemuddin, S. Multiprocessing on FPGA using light weight processor *Electronics, Computing and Communication Technologies (IEEE CONECCT), 2014 IEEE International Conference on,* **2014**, 1-6

[Renbi09] Renbi, A. & Lindh, L. Power and energy efficiency evaluation for HW and SW implementation of nxn matrix multiplication on Altera FPGAs *Proceedings of the 6th FPGAworld Conference,* **2009**, 45-51

[Roberts09] Roberts-Hoffman, K. & Hegde, P. ARM cortex-a8 vs. intel atom: Architectural and benchmark comparisons *Dallas: University of Texas at Dallas,* **2009**

[Saa15] Saa-Garriga, Albert, David Castells-Rufas, and Jordi Carrabina. "OMP2MPI: Automatic MPI code generation from OpenMP programs." *In proceeding of: High Performance Energy Efficient Embedded Systems (HIP3ES 2015),* (**2015**).

[Saa16] Saà-Garriga, A. Automatic Source Code Adaptation for Parallel and Heterogenous Platforms. *Universitat Autònoma de Barcelona,* **2016**

[Sakran07] Sakran, N.; Yuffe, M.; Mehalel, M.; Doweck, J.; Knoll, E. & Kovacs, A. The implementation of the 65nm dual-core 64b merom processor *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International,* **2007**, 106-590

[Saha13] Saha, Simanto, Anandaroop Chakrabarti, and Rajesh Ghosh. "Exploration of Multi-thread Processing on XILKERNEL for FPGA Based Embedded Systems."*Control Systems and Computer Science (CSCS), 2013 19th International Conference on*. IEEE, **2013**.

[Saldana06] Saldana, Manuel, and Paul Chow. "TMD-MPI: An MPI implementation for multiple processors across multiple FPGAs." *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on*. IEEE, **2006**.

[Salminen05] Salminen, E.; Kulmala, A. & Hämäläinen, T. D. HIBI-based multiprocessor SoC on FPGA *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on,* **2005**, 3351-3354

[Schoeberl03] Schoeberl, M. JOP: A Java optimized processor *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops,* **2003**, 346-359

[Seki08] Seki, N.; Zhao, L.; Kei, J.; Ikebuchi, D.; Kojima, Y.; Hasegawa, Y.; Amano, H.; Kashima, T.; Takeda, S.; Shirai, T. & others A fine-grain dynamic sleep control scheme in MIPS R3000 *Computer Design, 2008. ICCD 2008. IEEE International Conference on,* **2008**, 612-617

[Shannon04] Shannon, L. & Chow, P. Using reconfigurability to achieve real-time profiling for hardware/software codesign Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, **2004**, 190-199

[Silva08] Silva-Filho, Abel G., and Sidney ML Lima. "Energy consumption reduction mechanism by tuning cache configuration usign NIOS II processor." *SOC Conference, 2008 IEEE International*. IEEE, **2008**.

[Silvano11] Silvano, Cristina, et al. "Multicube: Multi-objective design space exploration of multi-core architectures." *VLSI 2010 Annual Symposium*. Springer Netherlands, **2011**.

[Stevens10] Stevens, D. & Chouliaras, V. LE1: a parameterizable VLIW chip-multiprocessor with hardware pthreads support *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on,* **2010**, 122-126

[Stevens12] Stevens, D.; Chouliaras, V.; Azorin-Peris, V.; Zheng, J.; Echiadis, A. & Hu, S. BioThreads: A Novel VLIW-Based Chip Multiprocessor for Accelerating Biomedical Image Processing Applications *Biomedical Circuits and Systems, IEEE Transactions on,* **2012**, *6*, 257-268

[Stevenson85] Stevenson, D. & others IEEE 754-1985 IEEE Standard for Binary Floating-Point Arithmetic *20pp, IEEE, July,* **1985**

[Stümpel98] Stümpel, Esther, Michael Thies, and Uwe Kastens. "VLIW compilation techniques for superscalar architectures." *Compiler Construction*. Springer Berlin Heidelberg, **1998**.

[Subotic14] Subotic, Vladimir, et al. "Automatic exploration of potential parallelism in sequential applications." Supercomputing. Springer International Publishing, **2014**.

[Sun99] Sun, T.S.J. and Leu, D.R. Software performance analysis using hardware analyzer. Patent US 5903759. **1999**.

[Sugavanam13] Sugavanam, K.; Cher, C.-Y.; Gunnels, J. A.; Haring, R. A.; Heidelberger, P.; Jacobson, H. M.; McManus, M. K.; Paulsen, D.; Satterfield, D. L.; Sugawara, Y. & others Design for low power and power management in IBM Blue Gene/Q *IBM Journal of Research and Development, IBM,* **2013**, *57*, 3-1

[Talwar08] Talwar, B. et al. "A System-C based Microarchitectural Exploration Framework for Latency, Power and Performance Trade-offs of On-Chip Interconnection Networks", *First International Workshop on Network on Chip Architectures* (NoCArc 2008), Lake Como, Italy, **2008**.

[Tandon11] Tandon, James. "The openrisc processor: open hardware and linux." *Linux Journal* 2011.212 (**2011**): 6.

[Tang00] Tang, Hong, Kai Shen, and Tao Yang. "Program transformation and runtime support for threaded MPI execution on shared-memory machines." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22.4 (**2000**): 673-700.

[Taylor13] Taylor, M. B. A landscape of the new dark silicon design regime *Micro, IEEE, IEEE,* **2013**, *33*, 8-19

[Tian09] Tian, X. & Benkrid, K. Mersenne Twister Random Number Generation on FPGA, CPU and GPU *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on,* **2009**, 460-464

[Tian10] Tian, X. & Benkrid, K. High-Performance Quasi-Monte Carlo Financial Simulation: FPGA vs. GPP vs. GPU *ACM Trans. Reconfigurable Technol. Syst., ACM,* **2010**, *3*, 26:1-26:22

[Tong07] Tong, J. & Khalid, M. A Comparison of Profiling Tools for FPGA-Based Embedded Systems Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on, **2007**, 1687 -1690

[top500] http://top500.org/

[Trimberger15] Trimberger, S. M. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology *Proceedings of the IEEE, IEEE,* **2015**, *103*, 318-331

[Trefethen10] Trefethen, A. E. Extreme Computing: Challenges, Constraints and Opportunities *Fujitsu HPC Users Group Meeting,* **2010**

[Tumeo07] Tumeo, A.; Monchiero, M.; Palermo, G.; Ferrandi, F. & Sciuto, D. A Design Kit for a Fully Working Shared Memory Multiprocessor on FPGA *Proceedings of the 17th ACM Great Lakes Symposium on VLSI, ACM,* **2007**, 219-222

[Tumeo10] Tumeo, A.; Regazzoni, F.; Palermo, G.; Ferrandi, F. & Sciuto, D. A Reconfigurable Multiprocessor Architecture for a Reliable Face Recognition Implementation *Proceedings of the Conference on Design, Automation and Test in Europe, European Design and Automation Association,* **2010**, 319-322

[Valentini13] Valentini, G. L.; Lassonde, W.; Khan, S. U.; Min-Allah, N.; Madani, S. A.; Li, J.; Zhang, L.; Wang, L.; Ghani, N.; Kolodziej, J. & others An overview of energy efficiency techniques in cluster computing systems *Cluster Computing, Springer,* **2013***, 16*, 3-15

[Vangal08] Vangal, S. R.; Howard, J.; Ruhl, G.; Dighe, S.; Wilson, H.; Tschanz, J.; Finan, D.; Singh, A.; Jacob, T.; Jain, S. & others An 80-tile sub-100-w teraflops processor in 65-nm cmos *Solid-State Circuits, IEEE Journal of, IEEE,* **2008***, 43*, 29-41

[Vasudevan10] Vasudevan, V.; Andersen, D.; Kaminsky, M.; Tan, L.; Franklin, J. & Moraru, I. Energy-efficient cluster computing with FAWN: Workloads and implications *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking,* **2010**, 195-204

[VectorFabricsP] Vector Fabrics. Pareon, http://www.vectorfabrics.com/products

[Vestias14] Véstias, Má. & Neto, H. A Many-Core Overlay for High-Performance Embedded Computing on FPGAs *arXiv preprint arXiv:1408.5401,* **2014**

[Vila12] Vila-Closas, O. "Method and device for rotational marking." U.S. Patent No. 8,319,810. 27 Nov. 2012.

[Volos12] Volos, Stavros, et al. "CCNoC: Specializing on-chip interconnects for energy efficiency in cache-coherent servers." *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*. IEEE, **2012**.

[VonNewmann45] Von Newmann, J. First draft of a report on the EDVAC **1945**

[Wang08] Wang, Z. & Hammami, O. External DDR2-constrained NOC-based 24-processors MPSOC design and implementation on single FPGA *Design and Test Workshop, 2008. IDT 2008. 3rd International,* **2008**, 193-197

[Wang10] Wang, Q.; Jiang, W.; Xia, Y. & Prasanna, V. A message-passing multi-softcore architecture on FPGA for Breadth-first Search *Field-Programmable Technology (FPT), 2010 International Conference on,* **2010**, 70-77

[Waterman11] Waterman, Andrew, et al. "The risc-v instruction set manual, volume i: Base user-level isa." *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* (**2011**).

[Weerasekera08] Weerasekera, R. System Interconnection Design Trade-offs in Three-Dimensional (3-D) Integrated Circuits *KTH,* **2008**

[Wiecek82] Wiecek, C. A. A case study of VAX-11 instruction set usage for compiler execution *ACM SIGARCH Computer Architecture News,* **1982***, 10*, 177-184

[Wilkes49] Wilkes, M. V. & Renwick, W. The EDSAC-an electronic calculating machine *Journal of Scientific Instruments, IOP Publishing,* **1949***, 26*, 385

[William09] William, Thomas, et al. "Enhanced performance analysis of multi-core applications with an integrated tool-chain." *Parallel Computing* (**2009**).

[Wilton04] Wilton, S. J.; Ang, S.-S. & Luk, W. The impact of pipelining on energy per operation in field-programmable gate arrays *Field Programmable Logic and Application, Springer,* **2004**, 719-728

[Wirth77] Wirth, Niklaus. "Modula: A language for modular multiprogramming." *Software: Practice and Experience* 7.1 (**1977**): 1-35.

[Wong11] Wong, H.; Betz, V. & Rose, J. Comparing FPGA vs. Custom Cmos and the Impact on Processor Microarchitecture *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, ACM,* **2011**, 5-14

[Xilinx06] Xilinx, I. A. R. R. "Microblaze processor reference guide." *reference manual* 23 (**2006**).

[Yan09] Yan, L.; Dongsheng, L.; Duoli, Z.; Gaoming, D.; Jian, W.; Minglun, G.; Haihua, W. & Luofeng, G. Performance evaluation of the memory hierarchy design on CMP prototype using FPGA *ASIC, 2009. ASICON '09. IEEE 8th International Conference on,* **2009**, 813-816

[Yiannacouras06] Yiannacouras, P.; Steffan, J. G. & Rose, J. Application-specific customization of soft processor microarchitecture *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays,* **2006**, 201-210

[Yiannacouras08] Yiannacouras, P.; Steffan, J. G. & Rose, J. VESPA: portable, scalable, and flexible FPGA-based vector processors *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems,* **2008**, 61-70

[Yasui91] Ikuo Yasui. Microprocessor with Harvard Architecture. US Patent 5,034,887, **1991**

[Yeh14] Yeh, T.-Y.; Marr, D. T. & Patt, Y. N. Author retrospective for increasing the instruction fetch rate via multiple branch prediction and a branch address cache *25th Anniversary International Conference on Supercomputing Anniversary Volume,* **2014**, 24-25

[Zeiser09] Zeiser, T.; Hager, G. & Wellein, G. Benchmark analysis and application results for lattice Boltzmann simulations on NEC SX vector and Intel Nehalem systems *Parallel Processing Letters, World Scientific,* **2009***, 19*, 491-511

[Zet] http://zet.aluzina.org/index.php/Zet_processor

[Zinenko14] Zinenko, Oleksandr, Stéphane Huot, and Cédric Bastoul. "Clint: A direct manipulation tool for parallelizing compute-intensive program parts." *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*. IEEE, **2014**.

[Zinenko15] Zinenko, Oleksandr, Cédric Bastoul, and Stéphane Huot. "Manipulating Visualization, Not Codes." *International Workshop on Polyhedral Compilation Techniques **2015** (IMPACT)*.

List of Publications

[Castells06] **Castells-Rufas, D.**, Jaume Joven, and Jordi Carrabina. "A validation and performance evaluation tool for ProtoNoC." *System-on-Chip, 2006. International Symposium on*. IEEE, 2006.

[Joven08] Joven, J., Font-Bach, O., **Castells-Rufas, D.**, Martinez, R., Teres, L., & Carrabina, J "xENoC-an experimental network-on-chip environment for parallel distributed computing on NoC-based MPSoC architectures." *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*. IEEE, 2008. **(Best Paper Award)**

[Castells09] **Castells-Rufas, D**., et al. "NocMaker: A cross-platform open-source design space exploration tool for networks on chip." *INA-OCMC Workshop, Paphos, Cyprus*. 2009.

[Joven09] Joven, J., **Castells-Rufas, D.**, Risueo, S., Fernandez, E., & Carrabina, J. "NoCMaker & j2eMPI A Complete HW-SW Rapid Prototyping EDA Tool for Design Space Exploration of NoC-based MPSoCs."*IEEE/ACM Design, Automation and Test in Europe (DATE)*. 2009.

[Castells10] **Castells-Rufas, D.**, et al. "MPSoC performance analysis with virtual prototyping platforms." *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. IEEE, 2010.

[Castells10b] **Castells-Rufas, D.**; Joven, J. & Carrabina, J. "Scalability of a Parallel JPEG Encoder on Shared Memory Architectures" *2010 39th International Conference on Parallel Processing (ICPP)*, 2010, 502-507

[Fernandez10] Fernandez-Alonso, E.; **Castells-Rufas, D.**; Risueno, S.; Carrabina, J. & Joven, J. A NoC-based multi-soft-core with 16 cores *Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on,* 2010, 259 -262

[Castells11] **Castells-Rufas, D.**; Fernandez-Alonso, E.; Carrabina, J. & Joven, J. Sharing FPUs in many-soft-cores *Field-Programmable Technology (FPT), 2011 International Conference on,* 2011, 1-6

[Joven11] Joven, J., Strict, P., **Castells-Rufas, D.**, Bagdia, A., De Micheli, G., & Carrabina, J. "Hw-sw implementation of a decoupled fpu for arm-based cortex-m1 socs in fpgas." *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*. IEEE, 2011.

[Castells12] **Castells-Rufas, D.**; Vila-Closas, O. & Carrabina, J. Design of a multi-soft-core based Laser Marking controller *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on,* 2012, 1-6

[Joven13] Joven, J., Bagdia, A., Angiolini, F., Strid, P., **Castells-Rufas, D.**, Fernandez-Alonso, E., Carrabina, J., and De Micheli, G.. "QoS-driven reconfigurable parallel computing for NoC-based clustered MPSoCs." *Industrial Informatics, IEEE Transactions on* 9.3 (2013): 1613-1624.

[Castells14] **Castells-Rufas, D.**; Carrabina, J.; González de Aledo Marugán, P. & Sánchez Espeso, P. "Fast Trace Generation of Many-Core Embedded Systems with Native Simulation" *In proceeding of: High Performance Energy Efficient Embedded Systems (HIP3ES 2014),* 2014

[Saa15] Saa-Garriga, A., **Castells-Rufas, D.**, and Carrabina, J.. "OMP2MPI: Automatic MPI code generation from OpenMP programs." *In proceeding of: High Performance Energy Efficient Embedded Systems (HIP3ES 2015),* (2015).

[Castells15] **Castells-Rufas, D.** & Carrabina, J. 128-core Many-Soft-Core Processor with MPI support *Jornadas de Computación Reconfigurable y Aplicaciones (JCRA),* 2015 **(Best Paper Award)**