ANALYSIS OF PERFORMANCE, POWER CONSUMPTION AND ENERGY EFFICIENCY OF MODERN FPGAS AND MICROPROCESSORS

Author:

JAVIER OLIVITO DEL SER

Supervisor:

DR. JESÚS JAVIER RESANO EZCARAY

DISSERTATION
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Universidad de Zaragoza

Grupo de Arquitectura de Computadores
Departamento de Informática e Ingeniería de Sistemas
Instituto de Investigación en Ingeniería de Aragón
Universidad de Zaragoza

March, 2017

Universidad Zaragoza

Analysis of performance, power consumption and energy efficiency of modern FPGAs and microprocessors

ABSTRACT

The irruption of mobile computing in the last years has highlighted energy efficiency more than ever. The computational requirements for battery-dependent devices are becoming more and more demanding while the power budget remains tight.

Heterogeneous computing has emerged as the mainstream to satisfy the increasingly computational requirements while keeping power consumption low. Systems-on-chip (SoCs) include not only general-purpose processors (GPPs) but also many specific processors to efficiently manage frequent tasks in mobile computing, such as communications, security, location, and others. However, those tasks that do not fit any of these specific processors rely on the GPP, where efficiency plummets.

Programmable logic is hardware whose functionality is not defined at the time of manufacture, and can be configured as many times as required, even at runtime. The main manufacturers have released SoCs that include programmable logic tightly coupled with the processing system, allowing the developers to improve efficiency by dynamically including specific processors for any computationally-intensive task.

In this dissertation, we explore the tradeoffs of this approach taking the artificial intelligence (AI) on board games as a case study. Board games are frequently found in mobile devices, and in fact, games as Chess are part of the benchmarks used to measure the performance of mobile SoCs. Board games are not only popular but also computationally demanding. Thus, they are good candidates to benefit from hardware acceleration.

As a first contribution, we designed bare hardware implementations for three different board games (Reversi, Connect6, and Blokus Duo) that increase energy efficiency by several orders of magnitude compared to delegating the work to the GPP. However, the design of complex hardware systems such as the AI for these games noticeably prolongs time-to-market.

As a second contribution, we designed solutions based on hardware/software codesign for these games, where only computationally-intensive tasks are delegated to hardware while the remaining functionality continues in software. This approach increases efficiency by an order of magnitude while keeping time-to-market acceptable.

Finally, as a third contribution, we measured the additional overheads caused by the configuration process needed to load the hardware accelerators in the SoC, and we have designed a hardware reconfiguration controller that drastically improves the efficiency of this process.

The results demonstrate the huge potential of SoCs that include programmable logic. This scheme improves performance and power consumption, and drastically increases energy efficiency, while the penalty in design complexity is affordable. We believe that it is very likely that, in the near future, SoCs that include programmable logic will be frequently found in mobile devices.

ACKNOWLEDGMENTS

I want to start by thanking my supervisor, Javier, for his help and guidance during these years, and with whom during this time I have built not only a professional, but also a relationship of friendship. Thanks also to my co-authors and to the rest of the members of the computer architecture group. To all of you, I hope you enjoyed working with me as much as I enjoyed working with you.

Special thanks to all my friends and colleagues who have been there during all these years, supporting and encouraging me at every step of the way. I hope you forgive me for not naming you all here. Finally, I have to thank my family, the ones that are and the ones that left, for all these years of support and patience, always believing in me.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

PRELIMINARIES

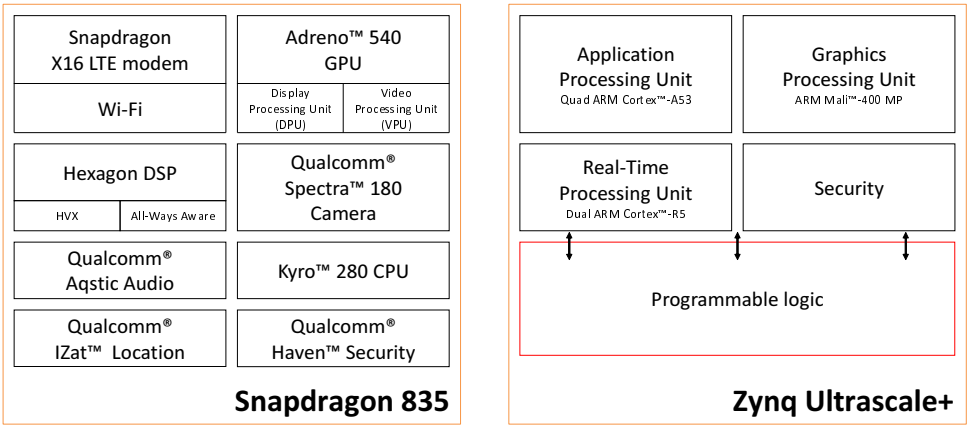
1 INTRODUCTION

1.1 RATIONALE

The impressive rise of smartphones, tablets, wearables, and internet of things (IoT) devices, has changed the computation landscape. This market is displacing the until recently predominant market of desktop and laptop computers, which is declining since 2011, and this trend is expected to emphasize in the next years. The limited power budget and the performance requirements of these devices has caused energy-aware computing to take on special relevance. The spectrum of applications developed for these devices has rapidly grown, narrowing the gap with respect to those hitherto reserved for desktop computers, where power is only limited because of thermal constraints. In this scenario, the semiconductor industry faces the challenge of delivering performance to support this new demand while keeping a satisfactory user experience.

Computer architects have found in heterogeneous computing an efficient solution to increase efficiency beyond advances in the semiconductor technology [3, 59]. The evolution of heterogeneous computing resulted in processors that contain not only general-purpose cores but also multiple specific processors to manage tasks such as graphics, audio, communications, security, location, and others. These devices are known as system-on-a-chip (SoC). SoCs have become more and more complex thanks to an increasingly smaller integration scale, including more specific processors and more powerful general-purpose cores. Figure 1.1a depicts the block diagram of the SoC that will be included in the Samsung Galaxy S8. This SoC includes many specific processors to manage frequently used tasks in today's smartphones, improving both performance and energy efficiency for these tasks. However, those computationally-intensive tasks that do no fit any of these specific processors must rely on the general-purpose processor where performance and efficiency plummets. Programmable logic helps to overcome this limitation by allowing the developers to implement any specialized hardware once the chip is manufactured, whose functionality can be changed as many times as needed, even at runtime. Hence, the same hardware resources can be used to provide hardware acceleration for different applications.

The main field-programmable gate array (FPGA) manufacturers, Xilinx and Altera, have released complete processor-based SoCs with an FPGA integrated in a single chip (Zynq-7000 SoC and Zynq UltraScale+ MPSoC by Xilinx [83]; Arria V [4] and Stratix 10 by Altera



(a) Snapdragon 835 (b) Zynq UltraScale+

Figure 1.1: Snapdragon 835 and Zynq Ultrascale+ SoCs

[63]). These platforms are similar to those found in mobile devices, but including a programmable logic fabric on chip, tightly integrated with the low-power processors. They allow software developers to use known programming environments, while logic designers can leverage the FPGA to introduce customized features to improve performance and reduce energy consumption. Figure 1.1b depicts the block diagram of the Zynq UltraScale+, released in 2017.

FPGAs are expected to play a key role in any field where energy efficiency is essential. Leading manufacturers like Intel, IBM, and Qualcomm have announced that they are preparing SoCs including processors and FPGAs for data centers applications [30, 55]. Furthermore, other companies such as Menta and Flex Logic have designed their own intellectual property (IP) FPGA cores, which can be included in any SoC at a reduced cost. Hence, FPGAs are expected to be frequently found in mobile SoCs in the near future, just as graphics processing unit (GPUs) are nowadays.

1.2 OBJECTIVES AND DISSERTATION OVERVIEW

The objective of this dissertation is to analyze hardware design and hardware/software codesign as ways to improve the energy efficiency on the growing market of battery-dependent devices. We selected the artificial intelligence (AI) of three board games as case study, and evaluated the performance, energy efficiency, and development effort of each alternative. The rest of this dissertation is organized as follows:

- Part I continues in Chapter 2 with a brief introduction to programmable logic. Chapter 3 presents board games as case study to evaluate performance and energy efficiency on programmable logic and general-purpose processors.
- Part II presents our hardware designs for the AI of three complex board games, Reversi, Connect6, and Blokus Duo. Chapter 4 discusses the context in which our designs were carried out and the methodology followed. Chapter 5 presents the design architecture of each case study, describing in detail the most relevant modules and the experimental results, focusing on the metrics performance, power, energy efficiency, and development effort.
- Part III explores hardware/software codesign on new SoCs that integrate general-purpose processors and programmable logic as a way to improve energy efficiency. Chapter 6 begins with a description of the platform where we carried out our experiments, and an analysis of the different interfaces to communicate the processing system with the programmable logic. Then, we present the software/hardware partitions considered for our cases studies, and finally we expose the experimental results.
- Part IV analyzes the overheads of dynamic partial reconfiguration in terms of performance, energy, and resources. In Chapter 7, we present a reconfiguration controller that minimizes the reconfiguration overheads. Chapter 8 quantifies the overheads of the components found in a system with reconfiguration capability, and shows the benefits of our controller on a real application.
- Part V concludes and discusses future research lines in Chapter 9.

1.3 CONTRIBUTIONS

At the time this dissertation is being written, a large part of the work presented here has been published in peer-reviewed national and international conferences and journals, or is currently under review process.

The main contributions of this dissertation are:

- Contributions on hardware implementation of the AI of board games (Part II):
  - We designed hardware implementations of the AI for three board games: Reversi, Connect6, and Blokus Duo. We propose designs based on well-known search algorithms and evaluation metrics that achieve a very efficient computation by fully exploiting the parallelism involved in board processing. These works have been published in the *International Conference on Field-Programmable Technology, in 2010, 2013, and 2014*. These works were also presented to the hardware design competition organized annually by that conference, and were awarded with the first prize in 2010 (3 competitors) and 2014 (14 competitors), with the second prize in 2012 (13 competitors), and with the fourth prize in 2013 (26 competitors).
  - We provide a comprehensive performance and energy efficiency analysis of the AI for the Reversi game implemented both on FPGAs and software. We optimized the hardware design for Reversi game presented in Part II, and also developed an optimized software application. We carried out a performance and energy analysis from a task-level to an overall perspective. This work has been published in *Microprocessors and Microsystems (March 2015)*.

- Contributions on hardware/software codesign applied to board games (Part III):
  - We designed several codesign schemes to split the AI engine of the games Reversi, Connect6, and Blokus Duo into software and hardware. New SoCs that integrate general-purpose processors and programmable logic in the same chip broaden the range of applications where codesign becomes profitable thanks to much lower communication overheads. We explored the benefits of codesign for board games by moving board processing to hardware accelerators while keeping the remaining tasks in software. This work has been accepted for publication in *IEEE Transactions on Computational Intelligence and AI in Games (August 2016)*.

- Contributions on dynamic partial reconfiguration on FPGAs (Part IV):
  - We propose an efficient reconfiguration controller in order to reduce the reconfiguration overheads, both in latency and energy consumption. We propose the inclusion of on-chip memory resources in the controller to achieve

peak performance and provide support for prefetching and caching. We also carried out a detailed analysis of the energy overhead of the reconfiguration process and the power penalty according to where the configurations are retrieved from. This work has been submitted to *IET Computer and Digital Techniques (submitted June 2016)*

2 PROGRAMMABLE LOGIC

Frequently used tasks in mobile devices are executed in specific application specific integrated circuits (ASICs) included in today's SoCs. These components offer the best performance and the lowest power consumption, but only the tasks they were designed for benefit from them, leaving many demanding tasks stayed out of the goodness of these components. Programmable logic is hardware whose functionality can be defined after manufacturing, putting hardware acceleration within reach of any application. In this chapter, we introduce programmable logic and the typical workflow in systems which include it.

2.1 INTRODUCTION

Programmable logic devices (PLDs) are electronic components whose functionality can be configured after manufacturing. The origin of PLDs dates back to 1960's and they have evolved giving rise to a variety of devices, such as programmable read-only memory (PROM), programmable logic array (PLA), complex programmable logic device (CPLD), or field-programmable gate array (FPGA). Among them, FPGAs are today the most broadly used programmable logic devices. They offer the highest amount of logic density, the most features, and the highest performance. They constitute a mature technology that has been proved to greatly increase performance and reduce energy consumption on many different applications ranging from data processing and storage, to instrumentation, telecommunications, and digital signal processing [11, 15, 16, 32, 42, 44].

An FPGA is an integrated circuit that contains an interconnected array of logic cells that can be configured to implement any custom functionality. Each logic cell includes look-up tables (LUTs) to implement logic functions, flip-flops (FFs) to implement sequential logic, and multiplexers to interconnect the different elements. Figure 2.1 outlines this architecture, where logic cells are represented by green boxes, and I/O is represented by orange boxes. In addition, FPGAs contain blocks of static memory, known as block random access memories (BlockRAMs, or BRAMs), and other hard blocks like digital signal processors (DSPs) and multipliers in order to increase performance and reduce power. The configuration of these logic blocks and its interconnections allows the developers to implement complex logic systems (the capacity of the latest FPGAs is equivalent to tens of millions of ASIC NAND gates).

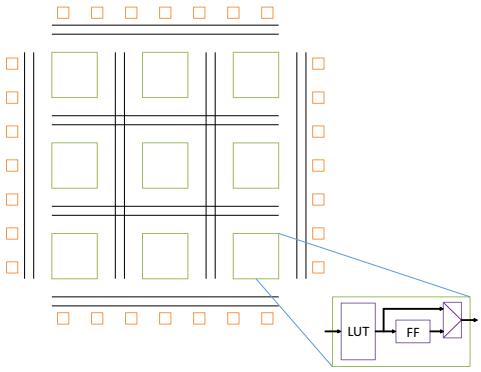


Figure 2.1: FPGA architecture

2.2 HARDWARE DESIGN ON FPGAS

The typical design flow to implement a digital system on an FPGA is represented in Figure 2.2. The process begins with the specification of the functionality and interfaces of the system components, and their interaction. This specification is coded in a hardware description language (HDL), like VHDL or Verilog, which allow hardware designers to write a modular description of the design architecture and functionality. It is possible to write a preliminary hardware description in HDL in time comparable to the required by software development in programming languages. However, writing HDL code ready to be translated into an efficient hardware implementation requires a good command on digital logic design, computer architecture, and parallel computing. Regarding this issue, FPGA vendors are doing a great effort to simplify the hardware design process. For instance, Xilinx has developed a C/C++ to HDL compiler that can directly map C/C++ code to an FPGA [77]. These tools are promising, but they still have much room for improvement.

The process continues iterating between coding and debugging. Hardware debugging at this stage relies on behavioral simulations. These simulations employ a high level of abstraction to model the hardware, leaving aside implementation issues such as resources utilization, routing, and timing. Once the functionality is verified, the design is synthesized, placed and routed. Synthesis translates the hardware description into hardware blocks, such as adders, registers, multiplexers, decoders, comparators, and so. The output of the synthesis is mapped to the target FPGA in the place&route process. In this stage, the hardware blocks are mapped to the FPGA resources (i.e, logic cells, DSPs, BlockRAMs, interconnections, I/O). Finally, the design implemented on the FPGA is verified.

Hardware design on FPGAs is significantly time consuming than software development, and the main responsible is the debugging process. Although we have stated that a qualified hardware designer is able to write a design specification and to code it in time compara-

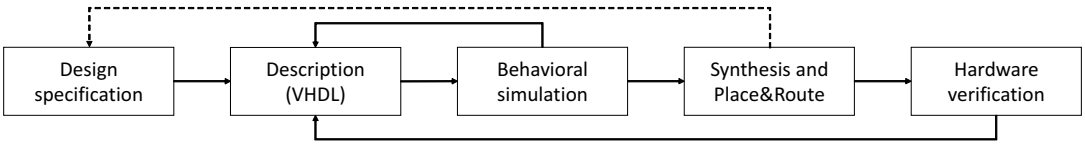


Figure 2.2: Hardware design flow on FPGAs

ble to a functionally equivalent software solution, definitely this does not apply to the debugging process. Behavioral simulations are much more time consuming than debugging software by means of native execution, and they usually have to be run many times. Furthermore, compilation time for hardware (i.e., time required by synthesis and place&route) is also much higher than for software.

2.3 HARDWARE/SOFTWARE CODESIGN ON GPP/FPGA PLATFORMS

Hardware/software codesign is a discipline that emerged in the early 1990s, in which system designers strategically partition an application into hardware and software blocks that interact among them, in order to improve at least one of the following metrics: time-to-market, performance, or power/energy consumption. This is especially relevant in embedded systems, where these metrics are usually considered design constraints, and solutions solely based on software or hardware cannot always meet them. Markets are governed by increasingly severe deadlines, and time-to-market frequently establishes the boundary between success and failure. Hardware design on FPGAs provides differentiation but the development cycle involved is unacceptable in many market segments. Hardware/software codesign greatly mitigates the effort of coding hardware by keeping most of the functionality in software, and moving to hardware only computationally-intensive kernels.

Despite codesign comprises any combination of hardware and software components, we focus on systems where software is executed on a hard-core general-purpose processor and hardware is implemented on an FPGA. Figure 2.3 shows the design flow we follow to develop efficient codesign solutions. First, the application is developed in software. Second, we profile the application in order to identify its hotspots (i.e., the most time-consuming tasks). These hotspots are the candidates to be moved to hardware. Then, an iterative process begins, where software developers and hardware designers must carefully refine their functionality and interface in order to minimize the communication overhead. Finally, once the metrics concerning the design constraints are satisfactory, hardware and software are integrated and codesign flow ends.

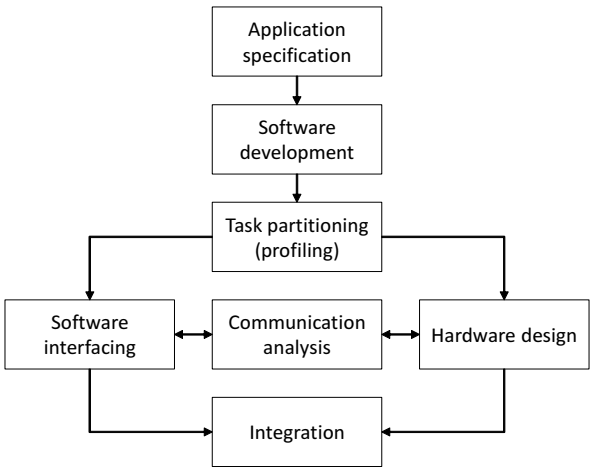


Figure 2.3: Hardware/software design flow on FPGAs

2.4 DYNAMIC PARTIAL RECONFIGURATION

Dynamic partial reconfiguration (DPR) is one of the most interesting features of FPGAs. DPR enables the reuse of the FPGA hardware resources for different tasks at run-time according to the system needs [33, 53]. This process takes place without interfering with other parts of the system, and therefore, FPGAs can be used to develop flexible platforms that can adapt themselves to the execution of different applications.

DPR provides several advantages. First, swapping hardware accelerators in and out on demand reduces the required chip area. Second, DPR leads to a reduction of power consumption. The reduction of required chip area allows for the use of smaller FPGAs, which are not only cheaper but also less power-hungry. Partial reconfiguration also helps to meet timing constraints, which is especially helpful in hard-real time.

The reconfiguration latency varies linearly with the size of the configuration bitstream. Partial reconfiguration enables the possibility to reconfigure only predefined regions of the FPGA while the remaining parts are unaltered. Thus, the bitstreams that configure the FPGA can be smaller, reducing the reconfiguration overhead. The reconfiguration latency also depends on other factors such as the reconfiguration protocol, and the bandwidth to the memory where the configurations are stored.

Figure 2.4 shows an FPGA where two reconfigurable regions (RRs) are defined and four tasks are executed. Hardware to process tasks a and c is allocated in RR_1, and tasks b and d are allocated in RR_2. The capability of reconfiguring a RR without interfering the other one allows time-multiplexing. This topic will be analyzed in Chapter 7 and Chapter 8.

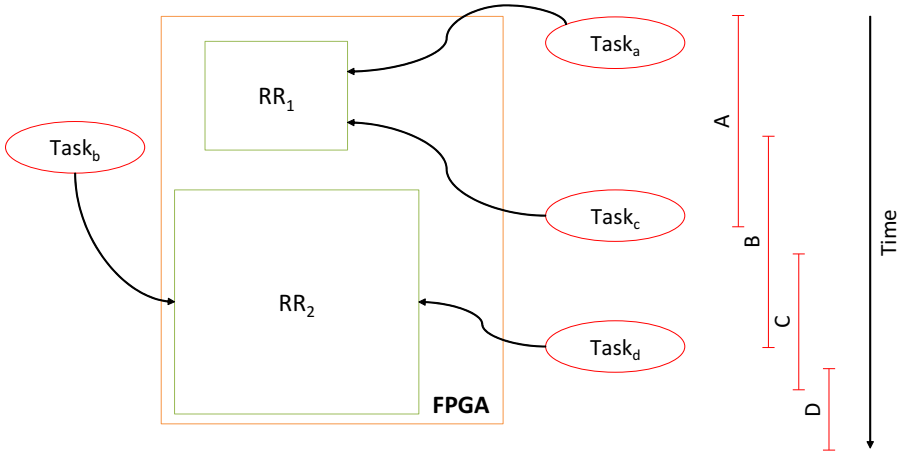


Figure 2.4: FPGA with two reconfigurable regions executing four different tasks

3 BOARD GAMES

Hardware acceleration plays a key role in those tasks that are computationally demanding. The artificial intelligence of board games is an excellent benchmark for the performance and energy efficiency analysis of this dissertation due to heavy workload involved in their execution and the parallel nature of their algorithms. We introduce board games in this chapter, stressing the importance of how the computations involved in the AI algorithms are performed. Finally, we describe the three games selected for this purpose, Reversi, Connect6, and Blokus Duo.

3.1 BACKGROUND

Board games attract the interest of the community, not only because of their popularity but also because their complexity poses the challenge of developing computer players strong enough to beat the top human players [26, 27]. Chess in the twentieth century, and Go at present, constitute the major icons of this interest.

The first game mastered by a computer was noughts and crosses (also known as tic-tac-toe) in 1952, when Alexander S. Douglas developed a video game that could play perfect games. Later in 1997, Logistello, developed by Michael Buro, won every game in a six-game match against world champion Takeshi Murakami, adding the complex game Othello to the list of games where humans are overcome by computers. Also in 1997, one of the most memorable milestones took place when the machine designed by IBM, Deep Blue, mastered Chess by beating the world champion at that time, Gary Kasparov. Chess is the most popular and deeply studied board game, and one of the most complex games of those where the community put their efforts. Recently in March 2016, the computer program developed by DeepMind, AlphaGo, beat the world champion Lee Sedol in a five-game match. It was the first time a computer Go program beat a 9-dan professional without handicaps [46]. This is the greatest milestone in the board game AI research since Go is one of the most complex board games (the number of positions is 10^{100} times larger than in chess). It was chosen by *Science* as one of the breakthrough of the year runners-up on 22 December 2016 [21].

Most of the board games currently remain unsolved because the number of possible states is far from being amenable with the current computational power. The design of a computerized board game player covers three different domains:

- Game states search

- Game strategy
- Computation

The first domain addresses the exploration of future game states as a consequence of making a movement. The game tree is a representation of sequences of player's movements and opponent's responses. Search algorithms make emphasis on smart explorations, trying to avoid non-relevant states in order to explore as many future movements as possible. Making deep searches has a great importance since the closer is the end of the game the more accurate are the movement quality estimations. The techniques applied in this domain are orthogonal with the game, thus, their algorithms are reusable for any board game.

The second domain involves many years of game study to reach a deep understanding of the game in order to conceive concepts which allows the game designers to reasonably estimate the quality of the movements. The game of Chess is the epitome with a large set of game concepts such as piece capturing, mobility, open lines, piece activity, pawn structure, King's role, and many more. Properly weighting these concepts allows an AI engine to guide the search within future movements towards the most promising one. Notice that this design domain is game-dependent and therefore it needs to be designed specifically for each game.

The third domain deals with the way the algorithms involved both in game-strategy knowledge and game-states search are executed. The main point here is data parallelism. The operations required to compute movements in board games exhibit many sources of parallelism, and therefore, a computation platform that efficiently exploits this would be able to process a larger number of future movements.

3.2 ARTIFICIAL INTELLIGENCE IN BOARD GAMES

Several alternatives can be taken when designing the AI engine for a board game. The most frequently found paradigm consists in the exploration of future movements and the estimation of the movements expectation based on strategy concepts of the game. The exploration of future movements takes knowledge from the classical tree-search problem, found in many applications of the AI. Several algorithms have been developed for this purpose, such as MiniMax [56], NegaMax [25], Negascout [10], SSS* [45] or B* [65]. We have selected the MiniMax algorithm because it is widely used in board games applications, it is easy to implement both in software and hardware, and performs well when combined with enhancements, such as alpha-beta pruning or move ordering.

The MiniMax algorithm assumes that we will try to make our best movement while the opponent also will, and therefore, we will try to

maximize our expectation and the opponent will try to minimize it. The procedure to select a movement is to explore future movements in a depth-first fashion until terminal nodes are reached. A terminal node is both a node where both players have no legal moves, or a node where a preset depth is reached. Terminal nodes are evaluated according to heuristics that estimate how promising looks the board to win the game. Then, the scores are backpropagated in such a way that even levels, which correspond to selections of our movements, are propagated with the maximum score, and odd levels, which corresponds to opponent's selections of movements, are propagated with the minimum score.

Alpha-beta pruning is a strong enhancement for this algorithm which avoids the exploration of nodes which are irrelevant. Figure 3.1 shows the operation of the MiniMax algorithm with alpha-beta pruning on a small game tree. Nodes in red with dashed lines are those ones not explored because of the alpha-beta pruning. For example, the red node with score 7 is not explored because of the scores already assigned of its parent nodes. Its parent node is already assigned with a score of 9 and it is a *max* level, therefore, further explorations of successors would lead to a score equal or higher than 9. Since the grandparent node is already assigned with a score of 8 and it is a *min* level, we can safely stop exploring that branch since it will not be relevant for the search outcome.

The effectiveness of this technique depends on the order the movements are explored. For an alpha-beta cutoff to occur, a good opponent's response has to be explored. Hence, the sooner the best movements are explored the higher the alpha-beta pruning efficiency. Being O^{bd} the baseline (i.e., no alpha-beta is applied), where b is the branch factor, and d is the exploration depth, a random move ordering reduces this to $O^{3/4bd}$, and an - unfeasible - perfect move ordering reduces the number of nodes explored to $O^{bd/2}$.

Move ordering can be achieved both relying on game knowledge, and on information inferred during the search. In the former, a deep understanding of the game strategy sets those movements which are likely to be better. In the latter, search is divided into incremental sub-searches in order to get movement quality estimations to be used in the subsequent deeper search. This algorithm is known as depth-first iterative deepening (DFID) [37]. Given a budget for the search, defined either by a preset maximum depth or a timeout, searches of increasing depth are performed. A search of depth d benefits from the outcome of the previous search of depth d-1. This algorithm introduces an overhead as a result of the repeated work that deeper searches already did in previous searches. This overhead, which depends on the branch factor b and the depth d, is shown in Equation 3.1.

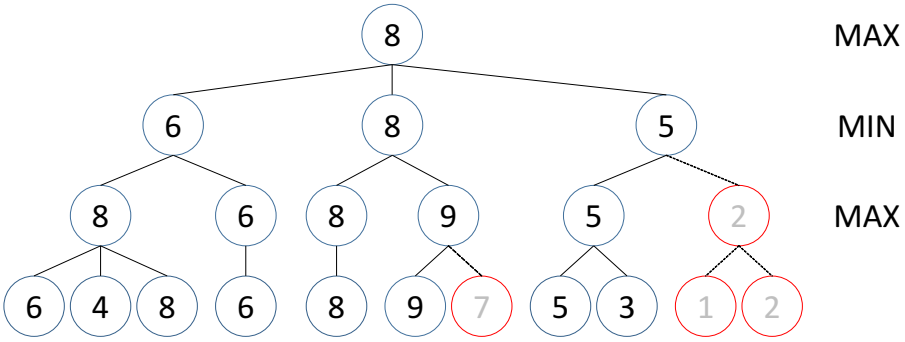


Figure 3.1: Minimax with alpha-beta pruning algorithm

This overhead is not only small but also reduces the effective branch factor of deeper searches, improving the overall performance.

$$overhead_{DFID} = \frac{db + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1}}{b^d} \tag{3.1}$$

There are other alternatives that follow other approaches, such as Monte-Carlo tree search (MCTS), where the selection of the movements relies on statistics, or machine learning, where the AI is trained from games played between top human players in order to learn how to select good movements. Monte Carlo makes a non-deterministic exploration based on playouts. A playout consists in a sequence of random movements from a given board until the end of the game is reached. Evaluation is always fully accurate in this algorithm since the end of game is always reached.

Machine learning has burst into board games in the last years allowing the AI designers to replace years of game study with emerged knowledge [20, 60, 82]. AlphaGo represents so far the most remarkable success beating the top human players; Go programs, before AlphaGo appeared on the scene, were able to beat only amateurs. We believe that machine learning algorithms are good candidates for hardware acceleration, and we will like to explore that in future works.

3.3 COMPUTATION IN BOARD GAMES

Mastering complex games such as Chess or Go relied not only on state-of-the-art algorithms but also on very powerful platforms capable of running their artificial intelligence engine extremely fast. Deep Blue was a supercomputer specifically designed to play chess, where 900 CPUs and 480 chess-specific hardware accelerators were the key to enhance its play strength [29]. AlphaGo was ran on the Google cloud platform when beat the world champion, distributing its execution on 1,920 CPUs and 280 GPUs [5]. The reason to allocate such computational power is that both the search into the game tree and the board processing exhibit a great parallelism, and exploiting it can

increase the number of positions explored per unit of time by several orders of magnitude. It is clear that hardware plays a crucial role to increase the play strength, however, moving these applications to the market makes it necessary to fit them to the hardware available in users' machines. Applications developed for desktop computers usually satisfy most users, but the question is: Are the solutions directly applicable to mobile devices? The fact is that there is a big gap between the strength of the board game applications developed for desktop computers, and those developed for mobile devices. At first glance, it might be thought that mobile processors do not provide enough performance. However, current SoCs developed for mobile devices include up to eight powerful out-of-order 64-bit cores running up to more than 2 GHz, providing a lot of computational power, which is more than enough to execute a strong board game player. The actual limit is the power budget. SoCs for mobile devices provide high peak performance, but running a computationally-intensive application drains the battery very fast, worsening the user experience.

A key to achieve gains in performance while keeping or even reducing the power budget is to use application-specific hardware. The accelerators included in Deep Blue were ASICs. ASICs provide the best performance and energy efficiency balance, but they also involves large development cycles and in most cases unaffordable costs [28]. The flexibility and low cost of programmable logic solves this drawback. Several works have described implementations of board games in FPGAs. Wong *et al.* [70] presented an implementation of the Reversi game. Their design reached a 3.67 speedup over an equivalent software running on a high-end processor. Other games like Connect6, Blokus, and Go have also been implemented by the FPGA developer's community. The works presented in [35, 58, 69, 80] detail FPGA-based implementations of these games and comparisons with software, reporting speedups of one or even two orders of magnitude. All these previous works focused on performance, but energy is crucial in mobile devices. A previous work was published in 2014 where not only performance but also energy was evaluated. Olivito *et al.* elaborated a comprehensive comparison of hardware and software implementations of Reversi in terms of performance and power, and pointed out that the hardware implementation on a low-cost FPGA was able to perform 25 times faster while consuming 400 times less power than the software implementation running on a high-end processor [51].

In the light of these results, it is clear that FPGAs outperform general-purpose processors both in performance and power in these games. However, the development of a board game application purely in hardware involves a much larger development cycle than in software, far from the demands of the industry.

Hardware/software codesign on hybrid CPU/FPGA platforms combines the flexibility and short development cycles of software design

with the higher performance and lower power consumption of FPGAs. Early codesigns were based on systems where the CPU and the FPGA were in different chips, communicated through a system bus.

One of the first applications of codesign to accelerate board games was published in 2002 [9]. This study presents a Chess player in which the move generation was accelerated by an FPGA and the remaining tasks of the AI were executed on a processor. In 2004, another successful use of processors and FPGAs to accelerate a Chess program was presented in [18]. Brutus was one of the strongest chess programs at that time and one of its key design strategies was to split the tree search into software and hardware. In these previous approaches, the CPU/FPGA communication overhead was a limiting factor both for the granularity and the speedups obtained by the tasks moved to hardware. The new heterogeneous SoCs, which integrate processors and FPGAs in the same chip, take weight off this issue. Moreover, manufacturers provide codesign environments with tools that automatically generate bus interfaces. Hence, communications are not only more efficient but also easier to manage. Some previous codesign works for board games on these new platforms have been developed [36, 50, 57, 64]. The common idea in these works is to delegate board processing to the hardware while commanding the search in software. This strategy leads to great speedups and energy consumption savings with a small development effort overhead.

3.4 REVERSI

Reversi (aka Othello) is a strategy board game for two players. Game unfolds on a 8×8 board placing discs which are colored black on one side and white on the opposite side. Each player shall be assigned to play a color, and the goal is to finish the game with more discs placed on the board than the opponent.

The game begins with blacks making a legal move. A legal move consists in placing a disc on an empty square in such a way that at least one opponent's discs is outflanked. Play then alternates between black and white. When a player has no legal moves, he forfeits his turn and the opponent plays again. Players are not allowed to voluntarily forfeit their turns. The game ends when both players have no legal moves. Therefore, it is possible to end a game before all the 64 squares are filled.

Reversi is a complex game which remains unsolved for the standard board size of 8x8 (the game tree is estimated to have 10^{58} nodes [1]). Hence, the design of a perfect player is unfeasible and it must rely on evaluation heuristics and search techniques which cannot guarantee that the best movement is selected.

It is a popular game thoroughly studied over the years, giving room to several concepts which make reasonable estimations about which

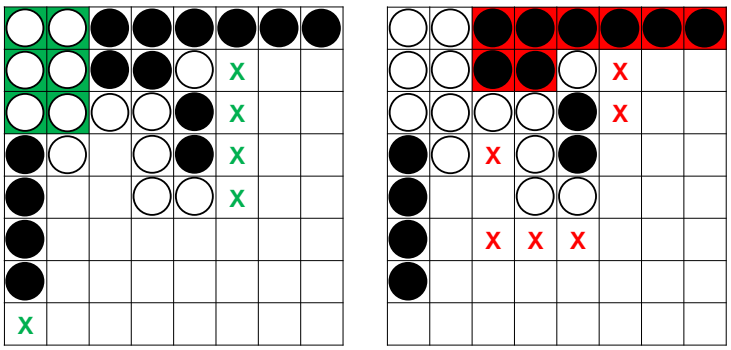


Figure 3.2: Evaluation metrics for Reversi

player is more likely to win the game. We selected two powerful concepts to evaluate boards in our design: mobility and stable discs.

Mobility measures the number of legal moves of each player. It is a crucial concept because a reduced mobility usually forces the player to make undesirable moves. Thus, looking for movements that maximize our mobility while minimizing the opponent's one makes it more likely to be in a better shape.

Sable discs are those ones that, once placed, they cannot be flipped anymore. Discs placed on the corners are always stable because there is no way to outflank them in any direction. Once a player has got a corner, he can turn more discs into stables. Since the goal of the game is to end the game with more discs than the opponent, having discs that definitely contribute to the final score is usually desirable.

Figure 3.2 illustrates the evaluation based on this two concepts on an example board. On the left, the board is evaluated for the white player, which has a mobility of 5 and 6 stable discs. On the right, the same board is evaluated for the black player, which has a mobility of 6 and 8 stable discs. The evaluation function computes the weighted difference for each metric.

3.5 CONNECT6

Connect-6 is a board game that was introduced in 2003 by Professor I-Chen Wu. This game belongs to the family k-in-a-row games. The game unfolds on a 19×19 board between two players, black and white, each one playing with stones of the corresponding color, which are placed on the intersections. Blacks moves first, placing only one black stone on an intersection. Subsequently, white and black take turns, placing two stones on two different unoccupied intersections each turn. The first player that gets six or more stones of his color in a row (horizontally, vertically, or diagonally) wins.

The complexity of this game is huge due to size of the board and the relaxed rule for making legal moves. The game-tree complexity of Connect(19,19,6,2,1) is estimated in 10^{140}nodes [78].

The key strategy concept in this game is the idea of *threat*. Threats are game positions which allow the opponent to win the game in a single movement if not defended, i.e., any combination of six contiguous squares, in any direction, with four stones of the same color and two empty squares, or five stones of the same color and one empty square. The extension of this concept to positions where a player may win the game not only in a single movement, but also in two or three, enables to identify not only immediate risks but also future risks. To this end, we name *threat-k* six contiguous squares, in any direction, which contain k stones of the same color and the remaining squares are empty.

Threats are valuable both for board evaluation and move generation. For the purpose of evaluating boards, we consider three categories: *threats-4* (t_4s), *threats-3* (t_3s), and *threats-2* (t_2s). Category *t4* includes also *threats-5* (t_5s), since they have the same value because a single stone is sufficient to defend the threat in both cases. Figure 3.3 illustrates the board evaluation based on threats: *threats-k* created by the black player are marked on the left (t_4s in read, t_3s in orange, and t_2s in yellow), and *threats-k* created by the white player are marked on the right (t_4s in dark blue, t_3s in light blue, and t_2s in green). The relative importance of each category was tuned empirically according to the performance against other Connect6 applications. For the purpose of generating movements, we also consider *threats-1* (t_1s) because there are situations, especially in the early game, where players do not have any other threat category. Making movements only in threat positions reduces drastically the search space by preventing the exploration of poor movements.

Figure 3.4 depicts this search space reduction on two boards, marking in black those intersections that are explored because they belong a *threat-k*, and marking in gray those intersections that are not explored. In the board on the left t_1s are explored because the black player does not have any other threat of higher category. In the board on the right only t_2s and higher are explored.

3.6 BLOKUS DUO

Blokus is an abstract strategy board game for two to four players, invented by Bernard Tavitian and first released in 2000, and played on a 20×20 board. Blokus Duo is a variant of Blokus designed for only two players and played on a smaller 14×14 board. This game is becoming increasingly popular because its rules are simple and games become fast and dynamic.

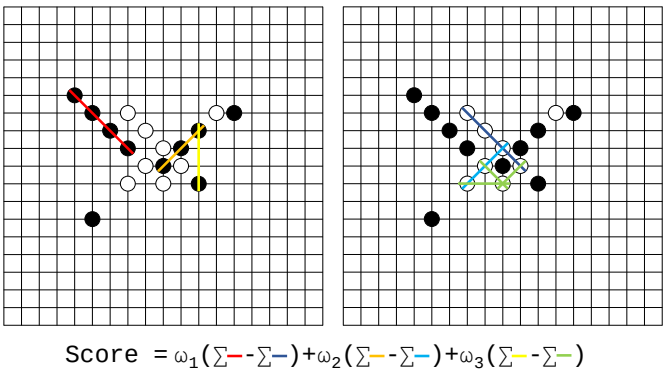


Figure 3.3: Evaluation metrics for Connect6

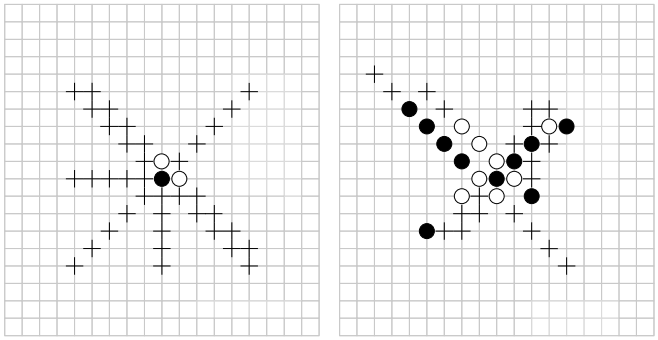


Figure 3.4: Search space cutoff based on *threats* in Connect6

Each player has a set of 21 different-shaped tiles, and can place them with eight different rotations. Figure 3.5 shows an example board and the remaining tiles for each player. The game begins with blue placing any tile, in any rotation, in such a way that the square [E,10] is occupied. Then, green must place any tile, in any rotation, occupying [J,5]. Thereafter, players have to place tiles in empty squares, sharing at least one corner with another tile of the same color already placed in the board (corner-to-corner rule), and not sharing any edge with any other tile of the same color (edge-to-edge rule). Each player places one tile at one time, and the game continues until neither of them can place tiles anymore. When the game finishes, the player that has occupied more squares wins the game.

The underlying strategy should prevent the opponent from placing tiles, especially the largest ones, while enabling us placing as many tiles as possible. A good metric that guides towards this objective is *accessibility*, which quantifies the number of squares that are potentially reachable in a given board. A square is reachable if can be occupied by means of a legal move. A player with higher accessibility than the opponent during the game has more chances to win the game. Figure 3.6 marks in light blue the blue player's accessibility for the game status presented in Figure 3.5, and in light green the green player's

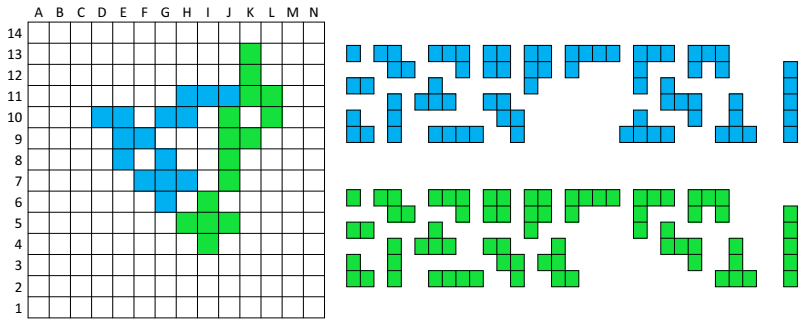


Figure 3.5: Blokus Duo game: board and remaining tiles

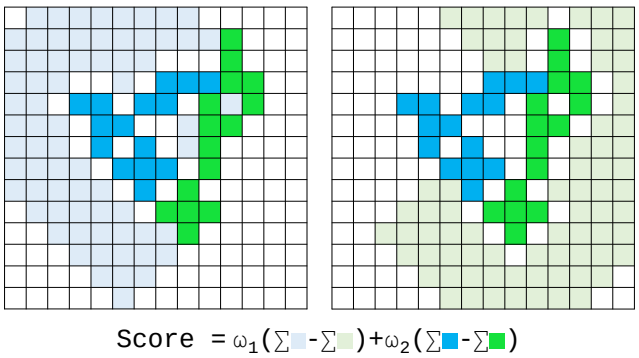


Figure 3.6: Evaluation metrics for Blokus Duo

accessibility. Boards are evaluated computing the weighted difference for both metrics, accessibility and occupied squares.

Accessibility can be also used to reduce the search space by exploring only movements in areas in dispute with the opponent. Placing tiles in these areas should have higher priority since they are not safe from being captured in the next opponent's movement. We define *overlapping map* as the union of the opponent's accessibility map and adjacent squares to his tile edges. We define *overlapping factor* as the number of squares of a legal move that are contained in the overlapping map, and finally, we define *overlapping threshold* as the minimum overlapping factor required to explore the legal move under consideration. Figure 3.7 illustrates the blue player's overlapping map, and two different legal moves, one that has an overlapping factor of 5, and another one that does not overlaps at all. The threshold varies along the game as shown in the same figure. This setup was tuned empirically following the same methodology than the previously exposed for determining the weights used in the evaluation function.

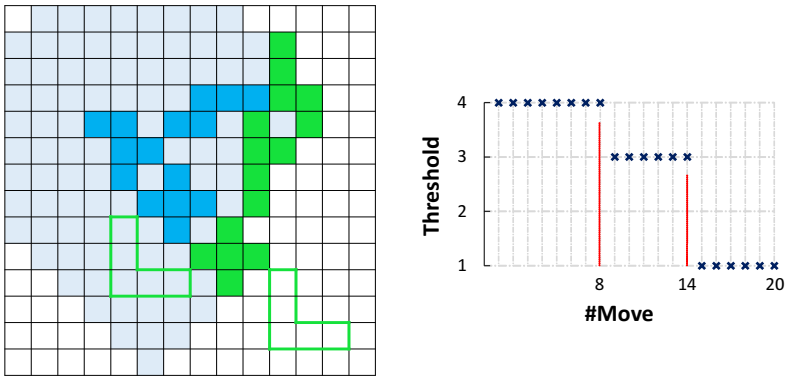


Figure 3.7: Search space cutoff based on *accessibility* in Blokus Duo

Part II

HARDWARE DESIGN

4 INTRODUCTION

This chapter describes the framework where the hardware designs presented in this dissertation were developed, and the methodology that was followed in accordance to that framework.

4.1 FRAMEWORK

The IEEE International Conference on Field-Programmable Technology [31] organizes each year a hardware design competition on FPGAs. The topic selected by the conference committee is board games: Sudoku in 2009, Reversi in 2010, Connect6 in 2011-2012, Blokus Duo in 2013-2014, and Traxx in 2015-2016. The designs presented in Chapter 5 were developed in order to participate in this competition. The fact that the competition details (board game, rules, and communication protocol) are disclosed a few months before the submission deadline conditions the workflow. It is a tight deadline to learn about each specific game, test strategies, and develop and validate the hardware design. Following the competition rules, the designs do not make any use of general-purpose processors, neither hard nor soft cores, and therefore they fully rely on custom hardware modules. These designs were subsequently optimized, analyzed in terms of performance, power and energy efficiency, and compared with algorithmically equivalent software applications.

4.2 METHODOLOGY

We split the work in pre-competition, where the time budget is very tight and the priority is to have a competing and validated hardware design, and post-competition, where further work such as adding new features, optimizing the hardware and software solutions, or taking measures is done.

4.2.1 *Pre-competition*

Our workflow to develop the hardware designs for the competitions was as follows:

1. Design of the AI engine

   The AI of all our designs is based on the exploration of movements in advance, and on the evaluation of boards according to a quantification of game strategy concepts. We opted for

the well-known minimax with alpha-beta prune to explore the game tree because is one of the most used search algorithms for board games and it is easy to implement. In order to design the board evaluation function, we got insight about the strategy of each game to pick some concepts easily amenable of being turned into metrics.

2. Development of a preliminary software version to test, explore, and adjust the strategy

   Making tests in hardware is much more time-consuming than in software. Therefore, it is worthy to develop a software application as a test bench to evaluate the quality of a given strategy, and explore other alternatives, by playing against different opponents: those supplied by the competition committee, and other stronger open source players. This software application was also very helpful to debug the hardware designs.

3. Development of the hardware design

   Those modules which are not strategy-dependent, such as the board generator, or the I/O, were developed in parallel with the preliminary software. Once we got a satisfactory strategy, we also designed its corresponding hardware modules.

4. Performance optimization and test

   The design must be tested in order to guarantee that it always provide valid moves, otherwise it would immediately loss the game. Testing an application that processes millions of board games per second was a challenging task. We carried out extensive simulations, but the simulation of a complete game was not feasible. Instead, we included in our hardware designs some specific support to store a trace and then compared the results with an equivalent software version. This methodology helped us to identify divergences between the hardware and the software version.

   In parallel with the testing process, we continued analyzing the games and our implementations in order to improve the strength of our strategy and accelerate its computation.

4.2.2 *Post-competition*

The analysis of the performance and energy of hardware and software solutions demands additional work on both the software and hardware developed for the competitions, namely:

1. Completion and optimization of the software developed before the competition

The purpose of the preliminary software developed before the competitions was to test and compare strategies as fast as possible, and therefore these applications were not fully optimized. In addition, some functionalities were only developed in hardware due to the severe deadline. Since a fair hardware/software comparative requires exactly the same functionality, we adapted our software in order to be algorithmically equivalent to the hardware designs.

2. Optimization of the hardware design

   The hardware designs which competed had much room for optimization in terms of performance, by better exploiting parallelism, and resources utilization. These optimizations have a remarkable impact in the energy efficiency of the design.

3. Data collection

   Data collection for performance measurements in software designs was obtained by profiling with third-party applications and also by adding little instrumentation. Regarding the hardware designs, data was collected by adding custom hardware counters and the required I/O to get the values.

   Power consumption was measured with a Yokogawa WT210 digital power meter.

5

CASE STUDIES: REVERSI, CONNECT6, AND BLOKUS DUO

This chapter introduces the hardware designs that were presented to the design competition organized annually by the International Conference on Field-Programmable Technology. These designs participated in the editions celebrated in 2010, 2012, and 2014. They demonstrated to be competitive, as they were awarded with the first prize in 2010 and 2014, and with the second prize in 2012.

5.1 CASE STUDY I: FPGA REVERSI PLAYER

5.1.1 *Design architecture*

In accordance with the AI engine paradigm introduced in Section 3.2, our designs comprise three main modules: Board generator, board evaluator, and search controller. In the Reversi game, the board generator consists of a module that identifies all the legal moves in a board (move checker), a module that computes the new board after making a legal move (disc flipper), and a module responsible for determining the order in which legal moves are explored (move selector). Board evaluator includes the module that computes the metric *stable discs*, and the arithmetic support to process the evaluation function. Minimax control includes the storage resources for the boards of the branch under exploration, and the hardware to guide the search. This architecture is depicted in Figure 5.1, and below the most relevant submodules are described.

1. Move checker

   Finding out whether a square corresponds to a legal move requires to check any flip pattern in each direction (N, E, S, W, NE, SE, SW, NW). This task exhibits a great degree of data parallelism as hundreds of patterns must be checked, and a hardware module can seamlessly exploit this parallelism since it is able to check all the patterns for all the squares at the same time. Figure 5.2 points out in red the legal moves for the white player in that example board and the hardware cell that checks all the patterns of a square in parallel. Each move checker consists of 60 cells like the one shown in the figure (the board has 64 squares but the game begins with four squares in the center, and then they cannot be legal moves). Each cell was customized to use only the minimum required resources, since the

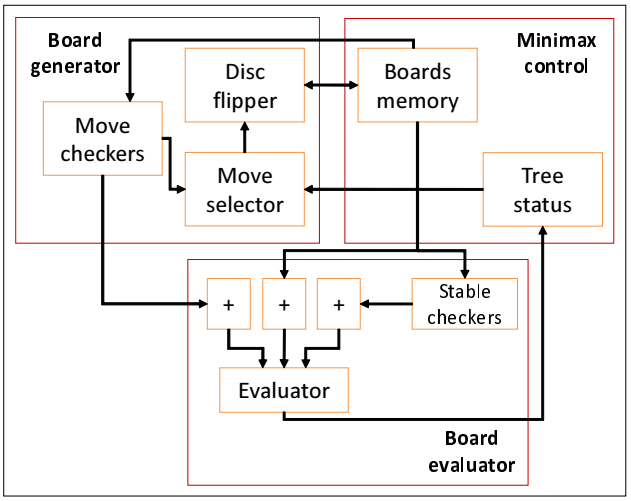


Figure 5.1: Design architecture of the FPGA Reversi player

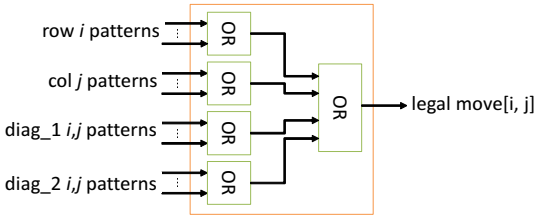


Figure 5.2: Move checker cell

patterns to check are different for each cell of the board. For instance, the cell corresponding to upper-left corner only looks for patterns in the S, E and SE directions. These cells are fully combinational, thus, they identify the legal moves in just one clock cycle. Since mobility is one the board evaluation metrics, our design includes two move checkers in order to compute in parallel this metric both for black and white players.

2. Disc Flipper

   This module returns the new board after making a legal move. It is implemented as an iterative network composed by 64 cells, one per each board square. This network works as follows:

   (1) The cell corresponding to the selected movement begins the pattern propagation to its neighbors in every direction.

   (2) Each cell propagates a new pattern in each direction based on the content of its square and its input pattern.

   (3) If a cell identifies a flip pattern, it stops propagating the pattern and returns a flip signal in the opposite direction. Every cell that receives this signal turns its color and keeps propagating this signal until it arrives to the initial cell.

   Figure 5.3 shows a simplified example of the operation of this network with four squares in one direction. A white disc is

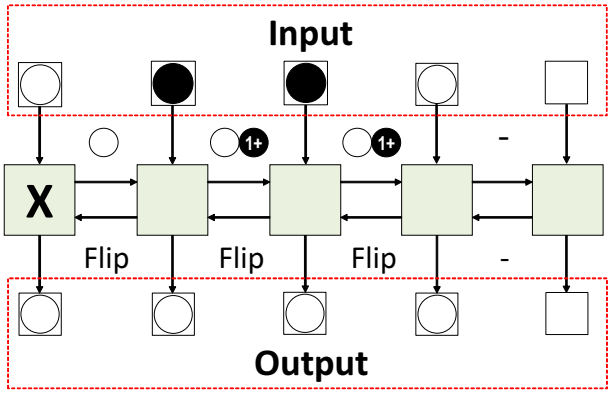


Figure 5.3: Section of the disc flipper iterative network

placed in the cell marked with an 'x'. This cell begins the pattern propagation by sending the corresponding pattern to its neighbor (a white disc, i.e., a white disc has been found so far). The next cell propagates a new pattern according to its input pattern and the content of its square (a white disc followed by at least one black disc). The third cell propagates the same pattern. The fourth cell finds another white discs that completes a flip pattern, so it returns a flip signal in the opposite direction that is propagated until the starting cell is reached. All the intermediate cells that receive this signal flip their discs.

As the previous module, this module is fully combinational, updating the board in one clock cycle.

3. Stable checker

   This module identifies the stable discs of a board. A disc is stable if there is no way for the opponent to outflank it, and therefore it will definitely contribute to the player score. A discs is stable if at least one of its neighbors in any direction (horizontal, vertical, diagonal, reverse diagonal) are stable. This is formally defined in Equation 5.1, where $S_{i,j}$ means that the square located in row i and column j is stable.

$$S_{i,j}\, if\ \left(S_{i,j-1} \vee S_{i,j+1}\right) \wedge \left(S_{i-1,j} \vee S_{i+1,j}\right) \wedge \left(S_{i-1,j+1} \vee S_{i+1,j-1}\right) \wedge \left(S_{i-1,j-1} \vee S_{i+1,j+1}\right) \tag{5.1}$$

We designed this module as an iterative network. However, as mapping this mutually recursive definition into a purely combinational hardware module generates combinational loops that seriously downgrade the performance, we added a flip-flop in each inner cell, in such a way that computation is sequentilized in layers. This network works as follows:

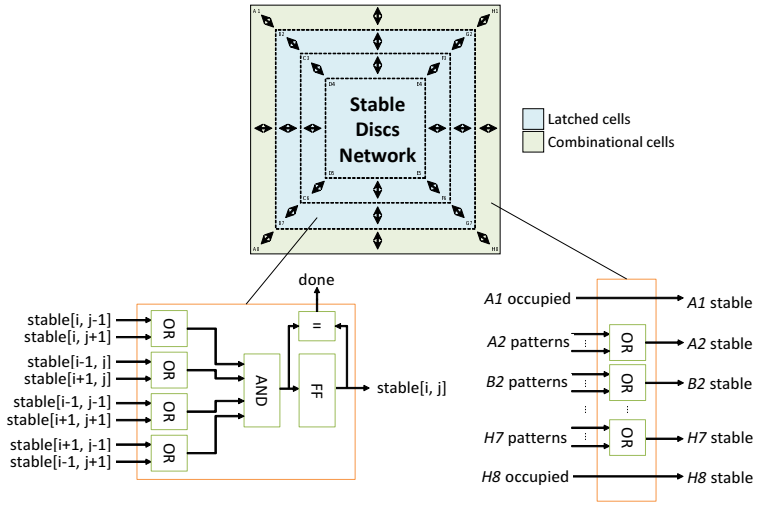


Figure 5.4: Stable discs checker

(1) In the first cycle, the combinational logic for the border squares determines which ones of them contain stable discs.

(2) In the following cycles, the intermediate results are propagated in a round-trip fashion until the results converge, i.e., none of the cells output changes with respect to the output in the previous clock cycle.

Figure 5.4 shows the architecture of the network of this module. The module on the right identifies stable discs on the boundaries. Corners are stable once they are occupied (their position prevent them from being outflanked), and the remaining squares just have to check patterns based on which squares on the boundaries are occupied. These results are propagated to the inner cells of the network (Latched cells). These cells implement the definition presented in Equation 5.1 with an OR/AND gate network, a flip-flop (FF) to store temporal outputs, and a comparator (=) to check when the output becomes stable. This network takes a variable number of cycles to converge, from 1 to 5.

5.2 CASE STUDY II: FPGA CONNECT6 PLAYER

5.2.1 *Design architecture*

Figure 5.5 shows the architecture for the Connect6 player. The core of this design is the threat identification, which is performed by the module *Threats finder*. Threats are used both to generate new positions (Board generator) and to evaluate boards (Board evaluator). The board generator also includes a filter to discard threat positions that are likely to have small expectation (Relevance zone filter), and a

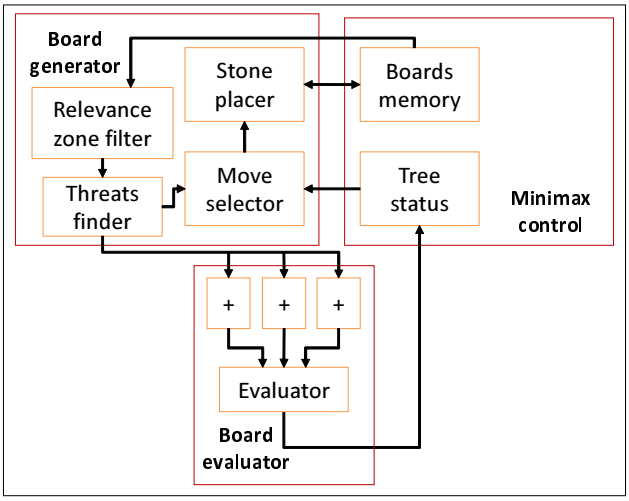


Figure 5.5: Design architecture of the FPGA Connect6 player

move selector that smartly combines threat positions in pairs to generate legal moves. The search controller is similar to the Reversi game. Below the most relevant modules are described.

1. Threats finder

   As explained in Chapter 3, we call *threat-k* six contiguous squares, in any direction, which contain k stones of the same color and the remaining squares are empty. Threats are involved both in move generation and board evaluation. On one hand, only positions belonging a threat are considered relevant in order to generate moves; on the other hand, *threats-k* is the only metric the board evaluation is based on. The identification of threats requires to analyze every row, column, and diagonal composed by at least six squares. We name each of them *section*, and we name *window* every possible combination of six contiguous squares within a section. Each section is analyzed following the algorithm detailed in [71]. This algorithm presents a data dependence since t_4s have to be identified prior to t_3s, and t_3s prior to t_2s. Figure 5.7 show a trace of the steps that the algorithm follows to find the threats: We first look for t_4s and we find three windows that satisfy its definition; we select the leftmost one and place a mark in its rightmost empty square. We have identified one t_4 so far, and the subsequent analysis reports one window with a t_4. We mark it and the subsequent analysis does not find any t_4 anymore. The next analysis follows the same process looking for t_3s, and finally the latest analysis will look for t_2s.

   The module that performs this computation is shown in Figure 5.6. *Threat finder* consists of n section processors to analyze in parallel all the sections in the board, three tree adders which

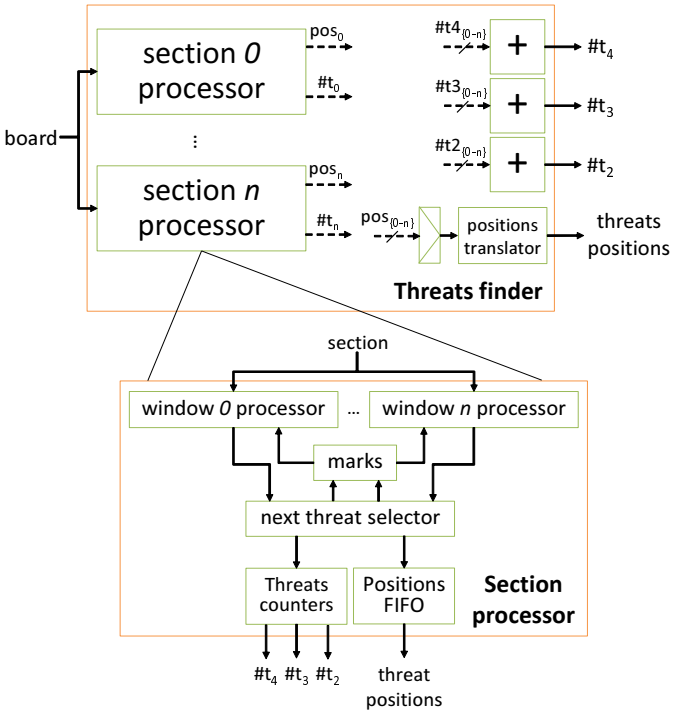
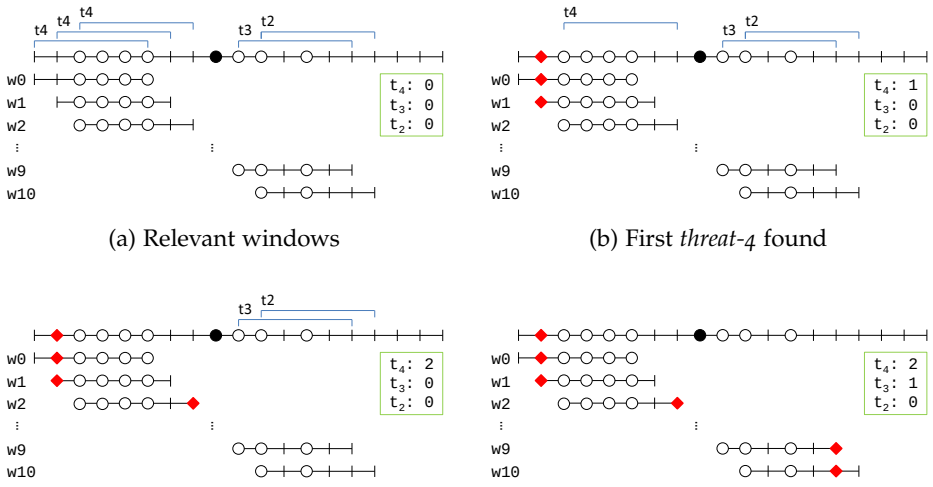


Figure 5.6: Threat finder architecture

add the threat count of each section and threat category, and a position translator which translates relative positions within a section into absolute positions. This module requires one clock cycle to find each *threat-k* within a section due to the data dependence of the algorithm that identifies threats. Hence, the number of cycles required to analyze a board is determined by the section with the largest number of *threats-k*.



(a) Relevant windows

(b) First *threat-4* found

(c) Second *threat-4* found

(d) *Threat-3* found

Figure 5.7: Threats identification process

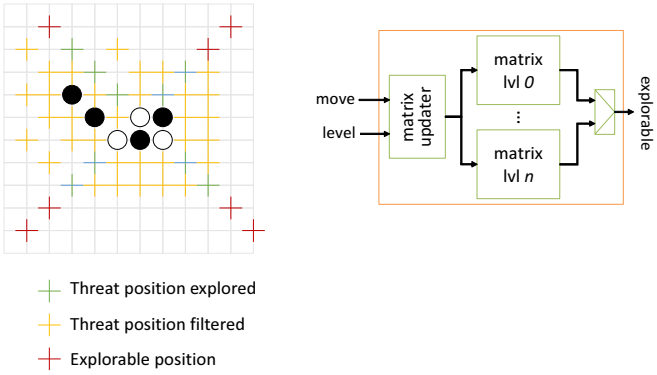


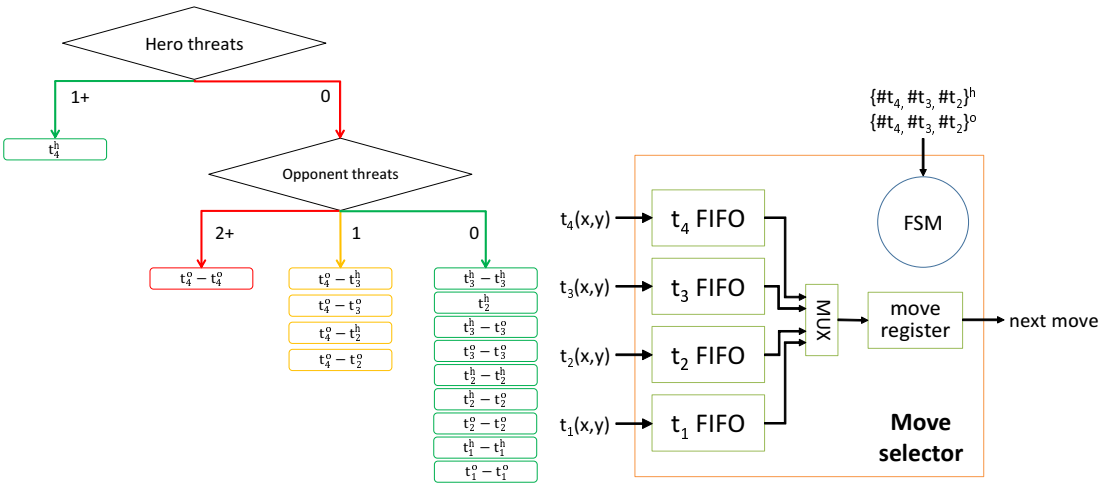
Figure 5.8: Relevance zone filter

2. Relevance zone filter

   Considering only *threats-k* to generate movements is the core strategy to smartly reduce the exploration search space, but even so it remains large. We added a second screen that discards any *threat-k* position which is far from the action zone. We define *action zone* as those squares that have a stone at distance 1 or 2 in any direction. Figure 5.8 illustrates this screening on a toy board, and the architecture of the module that performs this screen. In this example, the white player has four $t2$s for a total of 16 positions to analyze, and the relevance zone filter discards 7 of that 16 positions. The module that computes the relevance zone stores the exploration matrix of each level of the game tree in BlockRAMs. After a movement, the module *matrix updater* is responsible for both copying the matrix from level n to level $n+1$, and marking as explorable those squares affected by that movement.

3. Move selector

   Both players place two stones in each movement, except the first movement of the player who begins the game. We have seen that the identification of the positions belonging a threat introduces an initial and strong layer of intelligence, but exploring any combination of those threats positions generates many meaningless movements. Figure 5.9 depicts the scheme designed to select those movements to be considered from all the possible combinations of threat positions, and the priority assigned to each combination. The suffix h (from hero) in the categories refers to those threats belonging the player who is moving, and the suffix o (from opponent) refers to the player waiting for his turn. The hardware module consists of a set of FIFOs, one per category and player, that feed from Threat finder, and a finite state machine (FSM) that commands the extraction of threat positions



(a) Move generation scheme (b) Move selector architecture

Figure 5.9: Move generation scheme and move selector architecture

from the FIFOs in order to generate movements according to the categories presented in the scheme.

The scheme first checks whether the player who is moving has any real threat (i.e., *t4* or *t5*). If so, it selects a movement that turns the real threat into *six-in-a-row*. Otherwise, the priority is to defend from real threats, as any real threat not defended will cause an immediate defeat. Note that just one stone is sufficient to defend any *threat-k*, so the game will follow as long as the number of threats to defend is less or equal to 2. Then, the offensive movements are assigned with higher priority, exploring first movements that upgrade potential threats (i.e., *t3*s, *t2*s, and *t1*s, in this order). All the categories make up the movement by defending or upgrading two different threats, except *t2h*, where the two stones are placed in the same threat in order to upgrade from a *t2* to a *t4*. Note that this makes no sense on *t3* since a *t5* does not add any value regarding a *t4*.

In addition, we limited the number of movements to be considered up to 200. The reason is that, when there are no real threats involved, the number of combinations is usually too large, leading to shallow searches. Since the promising moves are explored first, the moves that are discarded beyond this limit are likely to have small expectation, and therefore they can be safely ignored.

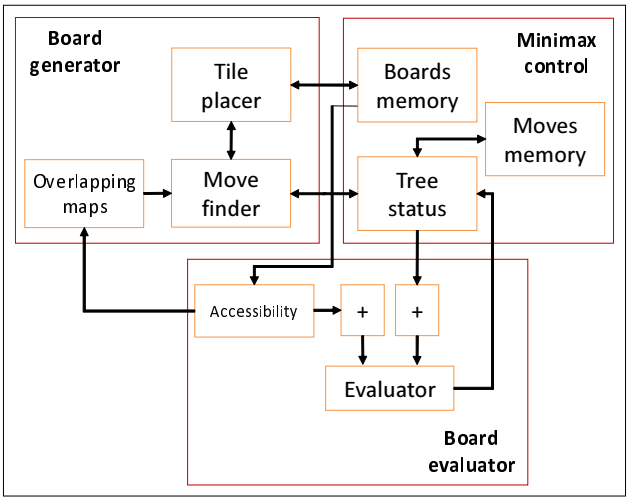


Figure 5.10: Design architecture of the Blokus Duo FPGA player

5.3 CASE STUDY III: FPGA BLOKUS DUO PLAYER

5.3.1 *Design architecture*

The design for this game follows the same approach than the previous case studies. The core of the design is the metric *accessibility*, i.e. the number of squares that are accessible for each player, and, in this case, the search controller includes a moves memory for the first four levels of the game tree in order to increase the prune efficiency. This architecture is depicted in Figure 5.10, and below the most relevant modules are described.

1. Tile placer

   Unlike the other two case studies, pieces in Blokus are not dimensionless. This fact adds some complexity to the modules which operates on the board, and opens the range of design decisions. Regarding the board, the straightforward approach is to code which squares are occupied by each player, and which ones are empty. However, we decided to also encode which squares are forbidden for each player, given the edge-to-edge restriction, because it simplifies other more complex modules involved in board processing. We also decided to keep two different board codifications, one that stores which squares are available for each player ($board_a$), and another one that stores which player is occupying each square ($board_v$). The first codification supplies the information to compute the accessibility with the optimal codification, whereas the second one provides information necessary to find the vertices.

   Regarding the tiles, we defined each combination of tile and rotation as an array of horizontal and vertical offsets with respect

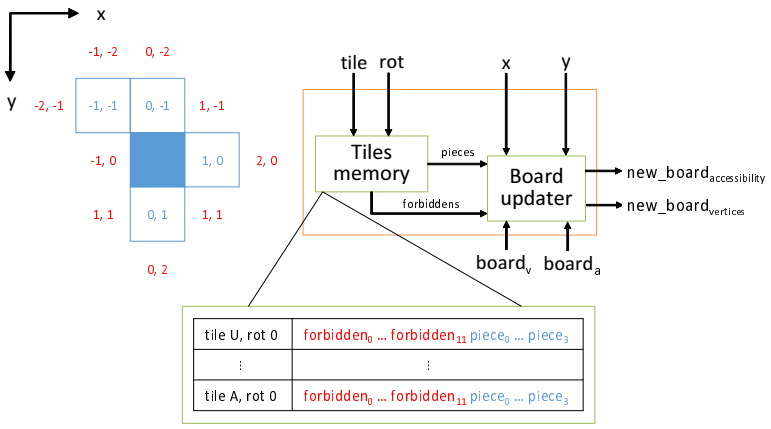


Figure 5.11: Tile placer

to its center (assigned in the game specifications). Additionally, each definition also includes the offsets of the squares that become forbidden once the tile is placed in order to simplify the hardware that updates the board after a movement. Figure 5.11 exemplifies this codification and shows the architecture of the module that codifies the board after making a movement. *Board updater* reads the tile definition, the coordinates of the square where the tile center is placed, and the boards aforementioned, and computes the new state of those squares affected.

2. Processing window

   Board processing, both for move generation and board evaluation is based on analysis from vertices, and the squares that are reachable from a vertex are determined by the shape of the tiles, giving rise to a shape like the one shown in Figure 5.12. We designed this module to feed those modules that perform any analysis from a vertex. Each cell of this module is composed by an adder that calculates the board coordinates of its corresponding square, and a multiplexer that selects the content of that square from the board, and a NOT gate. Our design includes two instances of *Processing window*, one designed specifically per each player. With the encoding shown in the figure, each instance has to check only one bit: a square is accessible for *hero* if it is free or it is forbidden for *opponent*, i.e., the most significant bit is zero, and a square is accessible for opponent if it is free or it is forbidden for hero, i.e., the least significant bit is zero.

3. Move finder

   Legal moves are found following two sequential steps:

   (1) Identification of available squares that share at least one corner with a placed tile. A square is available if both the square

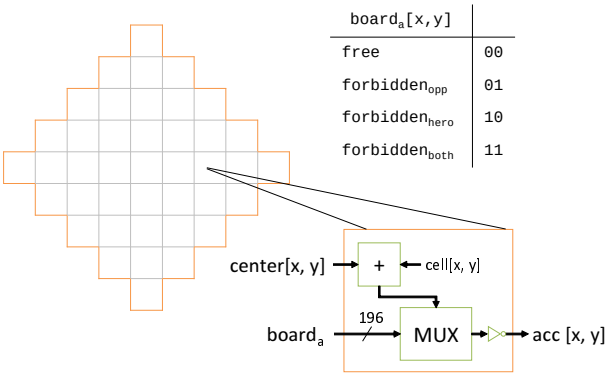


Figure 5.12: Processing window

is free, and it does not cause any edge-to-edge contact with an already placed tile of the same color.

(2) Verification of the availability of the remaining squares that would occupy the new tile.

There are 21 different tiles, 8 possible rotations for each tile, and several possibilities where to place the tile center to cause corner-to-corner contact. The combination of this three parameters leads to hundreds of potential legal movements, however, given the tiles symmetries and the edge-to-edge restriction, the maximum number of legal movements from a given a vertex is 127. We designed *Move finder* to be capable of analyzing in parallel all these possibilities. The architecture of this module is depicted in Figure 5.13.

Vertices Manager is responsible for identifying all the vertices in a board and returning the next one to be processed (Figure 5.14). In this module, *Vertices map* identifies in parallel all the valid vertices and its type through 169 *vertex detectors*, and then *Vertex selector* selects one of them given a preset order. We call *vertex* any intersection of four squares, and we define its type as 0, 1, 2 or 3 depending on which one of the four squares contains the piece of tile that originates a vertex.

Once a vertex has been selected, *Move checker* analyzes any of the 127 aforementioned {tile, rotation, center} combinations. This process requires to check whether those squares that would be occupied by the movement are available. Figure 5.15 details the architecture of this module. It includes one checker for each vertex type, each one analyzes 127 potential legal moves. Checkers for *v3u00* (vertex type 3, tile u, rotation 0, position 0) and *v3u01* are illustrated as an example.

Move selector is responsible for applying move filters and selecting the next move to be analyzed. Our design applies two filters in order to skip movements that are likely to have poor expec-

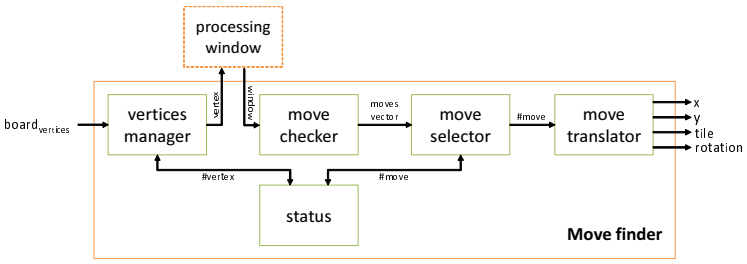


Figure 5.13: Move finder

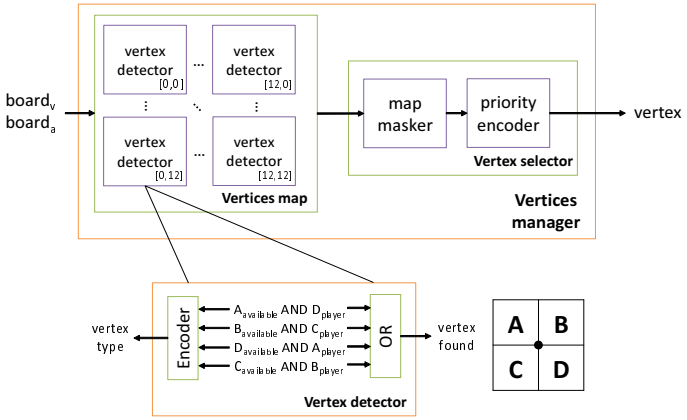


Figure 5.14: Vertices manager

tation. First, a size filter that masks movements of small tiles in the early game, and second, an overlapping filter that discards those movements that do not fight for areas in dispute. This filter checks how many pieces of the analyzed tile overlaps with the accessibility map of the opponent, and discards those movements whose overlapping factor is below the current overlapping threshold (unless there were no movements satisfying the overlapping threshold). These filters reduce the search space on early game by a factor of 2.4. Figure 5.16 shows the architecture of this module.

Finally, the movement selected from the movements vector is translated into {center position, tile, rotation} by means of a translation table.

4. Accessibility

   The squares that are accessible for a player are those ones occupied by all his legal moves. An implementation that benefits from hardware reusing would be to leverage the module described previously that finds legal moves and adding little support that would make it possible to compose the accessibility maps. However, this method would require as many cycles as legal moves are in the board under analysis, slowing down the board evaluation. For this reason, we designed a specific

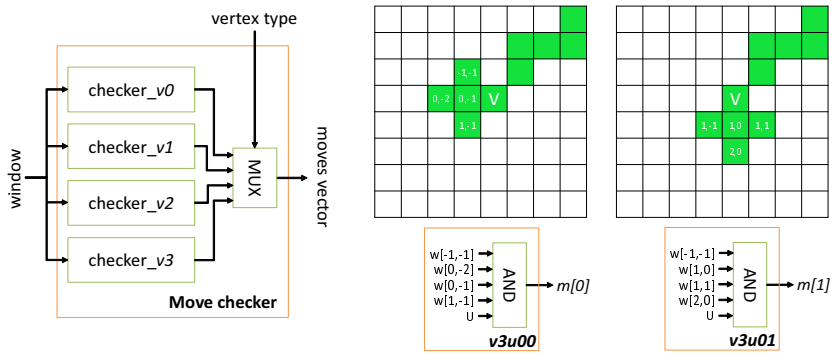


Figure 5.15: Move checker

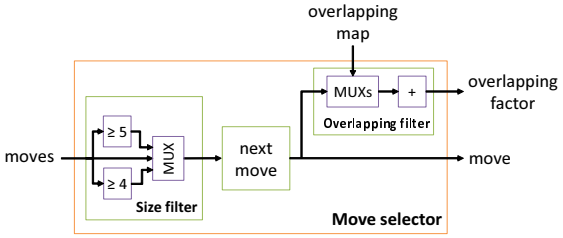


Figure 5.16: Move selector

module that accelerates the computation of this map. The underlying idea is to identify any pattern that qualifies a square as accessible. Consider the toy board represented in Figure 5.18: square labeled as *d5* is reachable if at least one the three movements illustrated is available. For example, the first movement is available if tile *K* has not been already placed, and squares *a2*, *b3*, *c4* and *d5* are available. In this example, just three patterns turns the square into accessible, but, the closer the square is to the vertex, the more patterns have to be verified. As a result, the analysis of the accessibility from a vertex checks up to 459 patterns. The module *Vertex processor* shown in Figure 5.17 analyzes all these patterns in parallel.

The accessibility evaluator shown in Figure 5.19 implements two evaluation units that traverses the board in opposite directions. Each evaluation unit identifies those squares that are ac-

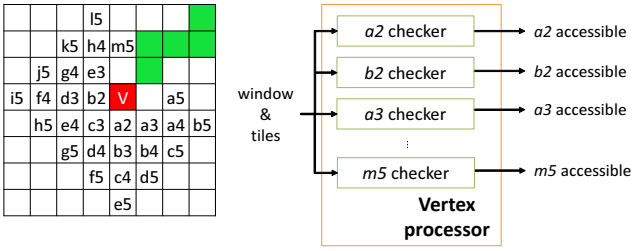


Figure 5.17: Vertex processor

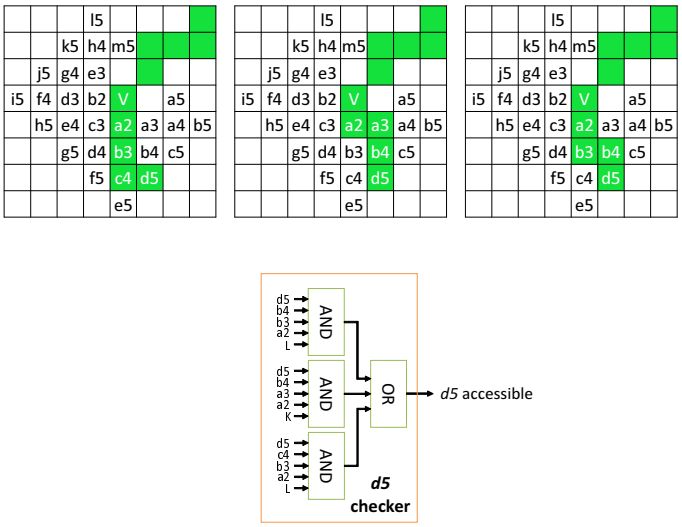


Figure 5.18: Patterns that qualify $d5$ as accessible

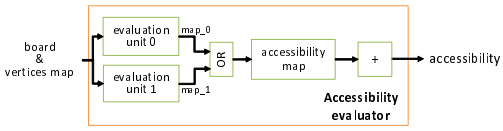


Figure 5.19: Accessibility evaluator

cessible from the vertices assigned to the unit, and the union of both partial maps give as a result the accessibility map. Finally, a tree adder calculates the accessibility from the map. Each evaluation units takes one clock cycle per analyzed vertex.

5. Moves memory

   In Chapter 3, we discussed the benefits of a good move ordering. Exploring first those movements with good expectation increases the prune efficiency, and therefore deeper searches are reached within a given time budget. The iterative deepening implemented in our search engine, which provides move ordering based on incremental searches, requires support to store the sequence of movements according to that ordering. The design of the moves memory is strongly linked to the branch factor of the game, the average search depth, and the memory resources available. Given the values of these parameters for our Blokus implementation, it is possible to store the whole ordered sequence of movements for the first two tree levels, and a partial order for levels 3 and 4. Support for deeper levels would exceed the on-chip memory resources.

   L1 and L2 memories are implementations of the well-known bubble sort algorithm (Figure 5.20a). We chose this algorithm due to its simplicity and because node ordering is fully overlapped with other computations. L1 consists of two memories

Memory	Method	Hits (%)	Collisions (%)	Size (entries)	Cutoff (%)
L1	Bubble	100.0	-	512	88.0
L2	Bubble	100.0	-	65,536	93.3
L3	Hash	83.5	2.0	16,384	94.2
L4	Hash	81.6	2.7	65,536	94.4

Table 5.1: Moves memory parameters

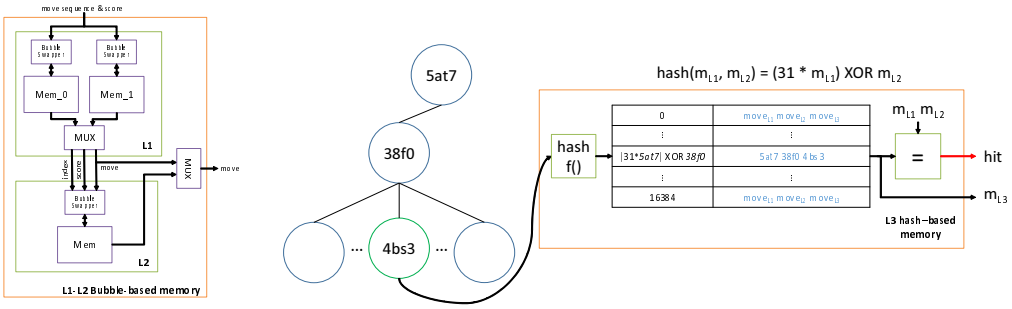
in order to be capable of supplying the sorting of the last search and storing the new sorting of the deeper search. This memory creates indices for each L1 node in the first search as references to their L2 successors. Alpha-beta prune is disabled in this initial search to get more accurate scores. L2 is a simplified version with just one memory as nodes of level two are not reordered in deeper searches.

L3 and L4 are hash-based memories that only stores the best scored successor for each one of the upper level nodes. Each memory entry stores the move chain from the root to the leaf. This piece of data is necessary to check the validity of the movement supplied by the hash memory, since entries are replaced on collisions. Figure 5.20b illustrates the operation of this memory. A toy game tree is shown labeling its nodes with the movements explored (notation is [x, y, tile, rotation]). Minimax search found that the most promising movement in level 3 was *4bs3* (node colored in green). It will be stored in the L3 memory in an address determined by a hash function taken from the JAVA library. On the next deeper search, the hash memory will be asked when exploring the move sequence *5at7*, *38f0*, and it will return *4bs3*, which was the movement stored in the previous shallower search.

Table 5.1 summarizes the main parameters and some performance metrics of each moves memory. Results show that hash-based memories offer high hit rates and low collisions rate despite its small size. The last column shows the reduction in the number of successors to be explored taking as baseline the search space of a minimax with alpha-beta pruning and no move ordering. The effect of move ordering gets smaller as performed in deeper levels since the resulting cutoffs are performed on smaller subtrees.

5.4 EXPERIMENTAL RESULTS

We evaluated both performance and energy efficiency of the hardware designs and the functionally equivalent software versions. Due



(a) Bubble-based memories for levels 1 and 2

(b) Hash-based memories for levels 3 and 4

Figure 5.20: Moves memories

to the fact that the designs of the three case studies were developed along several years, the target platforms where they were implemented and analyzed may differ in each case study. We decided to port all the hardware and software designs to the recent Zynq SoC. Evaluations on this platform make software and hardware comparisons more fair because differences due to the fabrication process are removed (both the ARM processor and the programmable logic fabric are built under 28nm technology in the same chip), and because it is a real target platform that might be the core of future mobile devices. Table 5.2 summarizes the specifications of the target platforms where the hardware designs were implemented, and Table 5.3 summarizes the specifications of the platforms where the software versions were executed. Frequency on programmable logic platforms is not specified because it depends on the design.

Performance evaluation of hardware designs was performed with a minimal instrumentation; adding some counters and small memories allows data collection for measures of both overall and task-level performance. This instrumentation has null impact on performance and negligible impact on power. In the case of the software versions, third-party tools were used: Intel VTune XE Amplifier for the Intel processors, and GNU gproof, included in the Xilinx Vivado tool suite, for the ARM processor. Intel VTune is a non-intrusive profiler based on statistical sampling that leverages dedicated hardware on the Intel processors to collect data. On the other hand, GNU gproof is an intrusive profiler also based on statistical sampling.

Regarding the evaluation of power, we considered the consumption due to the execution of our applications. Measures taken with the power meter include the power consumption of the whole platform (i.e., the PC when evaluating desktop processors, and the evaluation board when evaluating FPGAs and the processor of the Zynq SoC). Thus, it includes both the static and the dynamic power consumption of every platform component. In order to consider only the dynamic power required by our applications, we first measure the power con-

Platform	Processor	Tech (nm)	Year	Frequency
i7	Intel i7-2600	32	2011	3.4 GHz
Atom	Intel Atom D525	45	2010	1.8 GHz
Cortex-A9	ARM Cortex-A9	28	2012	667 MHz

Table 5.2: General-purpose platforms specifications

Platform	Board	Tech (nm)	Year	Logic Cells	BRAM (kB)
Virtex II-Pro	[74]	90	2002	30k	306
Virtex-5	[75]	65	2006	110k	666
Spartan-6	[6]	45	2009	43k	261
Zynq	[81]	28	2012	85k	560

Table 5.3: Programmable logic platforms specifications

sumption of each platform in an idle state, and then we repeat the measurements when the applications are running. To extract the dynamic power, we take the difference between both measurements.

Finally we computed the metric *Energy efficiency*, which measures useful work done per unit of energy. We define useful work as the number of boards analyzed; therefore, combining the performance metric *kBoards analyzed per second* with the power data, we obtain the energy efficiency metric as shown in Equation 5.2.

$$Energy\,efficiency = \frac{performance}{power} = \frac{\frac{kBoards}{s}}{watt} = \frac{kBoards}{J} \tag{5.2}$$

Tables 5.4, 5.5, and 5.6 show experimental results for each case study. *Ex. Time* stands for the time required to complete a game; *Power* represents the dynamic power consumption, i.e., the average increase in power consumption due to the execution of our application; numbers in the column *Energy* are the product of power and execution time, which represents the energy consumed during a game due to the execution of our application, and finally *Energy efficiency* connects the time to complete a game with the energy demanded applying Equation 5.2.

Despite hardware designs were conceived to decide each movement within one second as a consequence of the competition rules, we removed this timeout and set the AI engine to explore in advance a fixed number of moves to make fair comparisons. In this way, we guarantee the same useful work is done regardless the target platform. The Reversi application explores eight moves in advance, the Blokus application explores four moves, and the Connect6 application explores three moves. With these parameters, these applications

Platform	Ex. Time (s)	Power (W)	Energy (J)	E. eff. (kBoards/J)
i7	33.0	24.19	798.27	18.0
Atom	169.7	1.07	181.58	79.3
Cortex-A9	365.8	0.10	36.58	393.7
Virtex-II Pro	3.6	0.42	1.51	9,523.8
Virtex-5	2.1	0.15	0.32	45,714.3
Spartan-6	3.6	0.06	0.22	66,666.7
Zynq	3.6	0.02	0.07	200,000.0

Table 5.4: Reversi experimental results

Platform	Ex. Time (s)	Power (W)	Energy (J)	E. eff. (kBoards/J)
i7	96.0	29.98	2,878.08	5.0
Cortex-A9	1244.6	0.10	124.46	115.7
Zynq	9.1	0.03	0.27	52,747.3

Table 5.5: Connect6 experimental results

explore 14.4, 38.4, and 4.0 million boards during a game, respectively. Differences in execution time on the FPGAs are due to the frequency the designs can be clocked. In the Reversi implementation, the synthesis tool was able to clock the design at 60 MHz for the Virtex-5, and at 32 MHz for the others. The Connect6 design was clocked at 75 MHz, and the Blokus design works at 33 MHz in the two FPGA where it was implemented.

Hardware designs overwhelmingly outperform the GPPs, even the high-performance desktop processor, which drains much more power than the low-power desktop processor and the mobile processor. In particular, the results for the Reversi game show a efficiency gap sized from one to four orders of magnitude; numbers for Connect6 are similar to Reversi. The hardware implementation on the Zynq platform exhibits an efficiency four orders of magnitude higher than the i7, and two orders of magnitude higher than the mobile processor included in the same chip; the most impressive results are achieved with the Blokus application, bringing a 30× speedup over the desktop processor and a 300× over the mobile processor, leading to the best performance and energy savings from all the designs analyzed.

Figure 5.21 illustrates the energy efficiency for each case study in what we consider the most representative platforms: the ARM Cortex-A9 mobile processor and the FPGA, both built under the same scale integration, and Intel i7 high-performance desktop processor as a reference for performance. There is an efficiency gap sized from two to three orders of magnitude between the FPGA implementation and

Platform	Ex. Time (s)	Power (W)	Energy (J)	E. eff. (kBoards/J)
i7	852.0	28.56	24,333.12	0.6
Cortex-A9	8,615.5	0.10	861.55	16.7
Virtex-5	28.7	0.19	5.45	2,640.7
Zynq	28.7	0.03	0.86	16,724.7

Table 5.6: Blokus experimental results

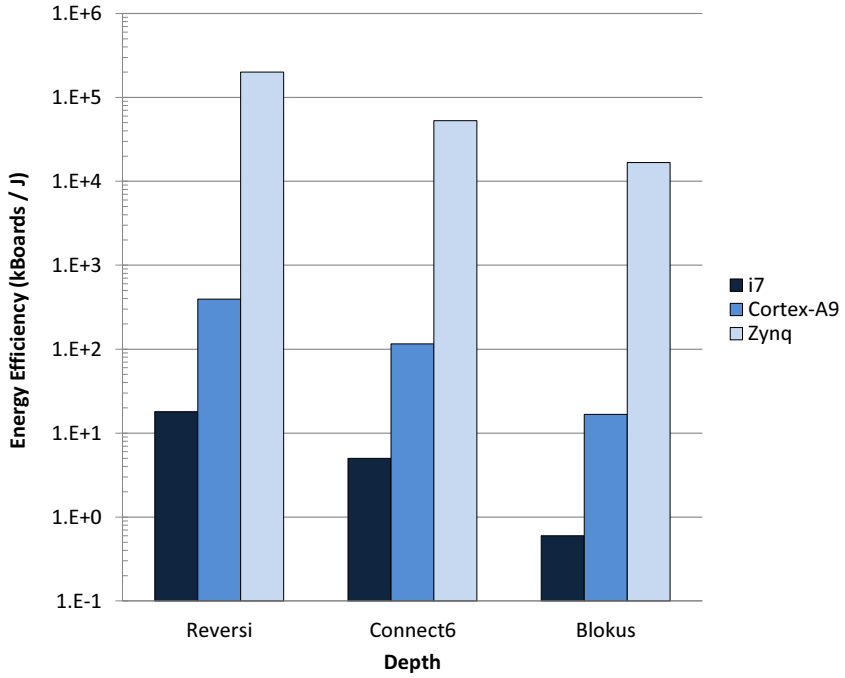


Figure 5.21: Energy efficiency

the mobile processor, and a four orders of magnitude gap between the FPGA and the high-performance desktop processor.

Table 5.7 shows the FPGA utilization of the hardware designs on the Zynq platform, the compilation time of both hardware and software solutions, and the time invested to implement and test the different solutions. Compilation time takes just a few seconds for software, whereas it takes hundreds or even thousands for hardware. Hardware compilation is a much more complex process, and this slows down the development cycle since designing and testing of complex designs requires a not negligible number of compilations.

Regarding the design time, there is a factor of about 2.5 between the development cycle of the hardware and software solutions. These numbers include the time to implement and test a defined design, i.e., this does not include the learning process about the game, and the design of the strategy. The complexity of the hardware design exploration can be reduced by developing a preliminary software so-

Design	LUTs (%)	FFs (%)	BRAMs (%)	Compi. time (s)	Design time (weeks)
Reversi ARM	-	-	-	1.7	7
Reversi FPGA	10.4	0.9	2.9	538.0	17
Connect6 ARM	-	-	-	2.4	8
Connect6 FPGA	81.4	8.0	2.2	1953.2	21
Blokus ARM	-	-	-	2.9	10
Blokus FPGA	48.7	5.7	69.6	1771.0	26

Table 5.7: Resources utilization

lution to carry out an initial analysis. With this approach, the design space is explored in software, and only the most promising strategy is designed in hardware.

Part III

HARDWARE/SOFTWARE CODESIGN

6

CASE STUDIES: REVERSI, CONNECT6, AND BLOKUS DUO

Based on the hardware designs presented in the previous chapter, we explored hardware/software solutions as a way to improve energy efficiency while keeping a competitive development effort. This chapter covers the work around this discipline. First, we introduce the platform where the experiments were carried out, and an analysis of the performance of the different communication interfaces between the processor and the programmable logic fabric. Then, we propose several codesign partitions based on a hotspot analysis of the applications, and finally, we expose the experimental results.

6.1 XILINZ ZYNQ-7000 EXTENSIBLE PROCESSING PLATFORM

Zynq-7000 is a SoC that integrates a dual-core ARM Cortex-A9 general-purpose processor, and an FPGA, all in a single chip. This heterogeneous platform joins up software flexibility and hardware efficiency, allowing developers to differentiate their products by increasing performance and energy efficiency. Figure 6.1 shows the block diagram of this platform, which is divided into what Xilinx names the processing system (PS), and the programmable logic (PL). The PS includes the general-purpose cores, a 256 KB on-chip memory, and a snoop control unit (SCP) responsible for managing the cache coherency both if caches are written by the ARM cores or by any module implemented on the PL. The PL includes configurable logic blocks (CLBs), memory blocks (RAM), and digital signal processing (DSP) blocks.

A critical aspect for hardware/software codesign to succeed is to enable an efficient communication between the processor and the programmable logic; the speedup achieved by the custom hardware must compensate for the communication overhead. Interconnection between the PL and the PS is based on the ARM advanced microcontroller bus architecture (AMBA) advanced extensible interface 4 (AXI4) [2]. This open-standard interface facilitates intellectual property (IP) integration, saving development time while providing high throughput and low latency. This bus offers several configurations, optimized to different traffic profiles. Our designs leverages AXI-Lite, which is suitable for small transfers, and AXI-Stream, which is suitable for larger transfers thanks to its burst mode. When using AXI-Lite, the hardware accelerator is assigned a set of 32-bit registers mapped into the processor memory space. Communicating hardware and software is as simple as writing or reading these registers. On the other hand, when using AXI-Stream, a DMA sends the data back

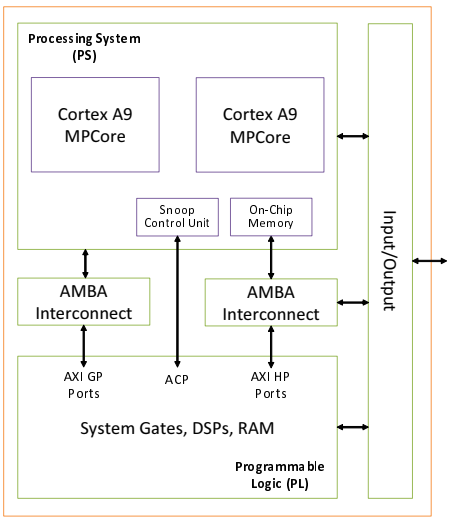


Figure 6.1: Xilinx Zynq Platform Block Diagram

and forth through the accelerator coherency port (ACP) or AXI high-performance (AXI HP) ports. We selected ACP because it ensures cache coherency when our hardware accelerator modifies the memory without processor intervention.

As a previous analysis to develop codesign solutions, we measured the communication latency of the two interfaces aforementioned according to the transfer length in order to select the most suitable alternative for each communication operation. As can be seen in Figure 6.2, the latency of the AXI-Lite interface is constant because each transfer always sends a single word. Instead, the time in AXI-Stream transfers decreases logarithmically as the transfer size increases. *AXI-Stream Custom* is a simplified version of AXI-Stream, where the DMA driver is simplified under the assumption that the source and destination addresses are always the same along the execution of an application, and the DMA has no simultaneous accesses. These assumptions apply in our case studies, and greatly increases the throughput for medium-sized transferences. Figure 6.3 shows the operations that the driver supplied by Xilinx performs, and how we modified them in our customized driver. Each one of the operations detailed involves a transference through the AXI-Lite interface, which is used to configure the DMA. Therefore, the Xilinx DMA driver performs five configuration transferences each time a DMA transference is requested, whereas our customized driver performs these operations only once, when the application is launched. The only remaining configuration-related transference is the writing of the transference length, which is the operation that triggers the DMA transference.

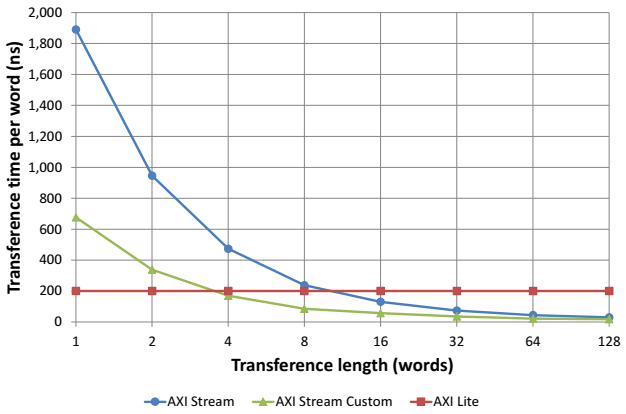


Figure 6.2: Transference throughput of each AXI interface

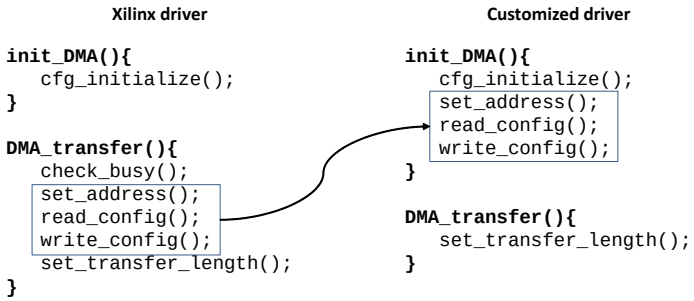


Figure 6.3: Operations on the Xilinx and our customized drivers

6.2 CASE STUDY I: REVERSI PLAYER

6.2.1 *Hardware acceleration*

The profiling of the software application revealed that 89.3% of the game time is invested in the evaluation of boards—48.6% computing stable discs, and 40.1% computing the mobility—, 9.6% of the game time is spent in the move generation, and only 1.1% is due to the execution of the game-tree search algorithm. Hence, we developed hardware accelerators for the two tasks involved in board evaluation: computation of mobility, and computation of stable discs. Figure 6.4 outlines the framework of the software method that computes these metrics in order to point out the sources of parallelism in these tasks. Loops in red are those fully unrolled in our hardware implementation, whereas loops in yellow are just partially unrolled due to data dependences.

6.2.2 *Codesign schemes*

We have implemented two codesign schemes for the Reversi game. The first one only moves to hardware the computation of the metric

Mobility

```
for each row
   for each col
      for each direction
         for each pattern
```

Stable discs

```
for each row
   for each col
      for each direction
```

Figure 6.4: Parallelism in the Reversi metrics

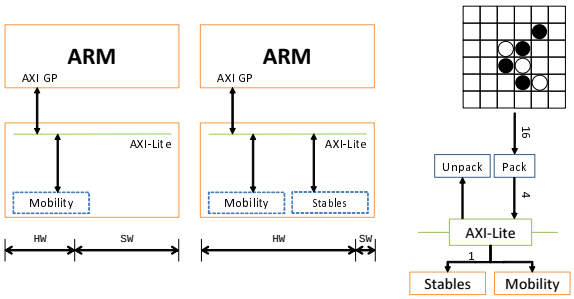


Figure 6.5: Codesign schemes for the Reversi application

mobility, and the other scheme moves to hardware the whole evaluation. These schemes are depicted in Figure 6.5.

The accelerators receive the board to process from the ARM processor and return the metrics values. Each board square stores three states: black, white, and free, and its encoding is not the same in software than in hardware: the hardware accelerators allocate 2 bits per square, whereas the software application allocates 8 bits. The overhead of sending the board as encoded in software, and then selecting in hardware only the worthy bits is much higher than performing bitwise operations in software in order to pack the data and then sending only those bits used by the hardware accelerator. The code we added in the software application to pack the board reduces the size of each transference from 16 words to 4 words. Hence, according to the analysis of the interface throughput presented in Section 6.1, both AXI-Lite, and AXI-Stream with a customized driver, offer a very similar throughput. We selected the AXI-Lite interface because of its simplicity.

The evaluation output is composed by the mobility of each player, and the number of stable discs of each player. All these values are ranged from 0 to 64, therefore each one of them is coded in 6 bits. These four values fit in a single word, so they are also transferred through the AXI-Lite interface.

Table 6.1 summarizes the features of each operation involving data movement. In this case study, packed data from the CPU to the FPGA perfectly fits the word size. Readback from the FPGA only uses a portion of the transference, 12 bits out of 32 (37.5%) in the first codesign scheme, and 24 bits out of 32 (75.0%) in the second codesign.

Operation	Transfer size (words)	Interface	Transfer occupancy
Feed accelerator	4	AXI-Lite	1.000
Readback mobility	1	AXI-Lite	0.375
Readback mobility & stables	1	AXI-Lite	0.750

Table 6.1: CPU/FPGA communication parameters in the Reversi application

Threats

```
for each section
   for each threat_category
      for each window
         for each pattern
```

Figure 6.6: Parallelism in the Connect6 metrics

6.3 CASE STUDY II: CONNECT6 PLAYER

6.3.1 *Hardware acceleration*

This application takes 90.4% of the execution time evaluating boards, 9.0% finding and selecting moves, and the remaining 0.6% is due to the minimax control. We only developed the hardware module that computes the number of threats, responsible for the hotspot of this application. Developing also the module to identify and select legal moves would be the next natural step, but the complexity of this hardware module is much higher than the module that evaluates the board, and the potential gain is much lower. Figure 6.6 outlines the framework of the software method that computes these metrics in order to show the sources of parallelism in these tasks. Loops in red are fully unrollable, whereas the loop in black is sequential because of data dependences.

6.3.2 *Codesign schemes*

For this case-study, we only implemented one codesign scheme (Figure 6.7). The dataflow from the software application to the hardware accelerator in Connect6 is the same than in Reversi. The input of the hardware accelerator is the board, and its squares stores three states: back, white, and free. As in the Reversi application, we followed the same *pack&send* approach, which reduces the transference size from 91 to 23 words. However, even after packing, the amount of data to transfer is large enough to use the AXI-Stream interface. The use of this interface makes it necessary to add a small module to store the board (*Board storage*). This module includes an array of 361 2-bit reg-

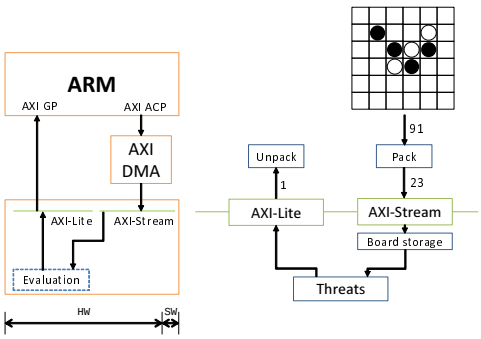


Figure 6.7: Codesign schemes for the Connect6 application

Operation	Transfer size (words)	Interface	Transfer occupancy
Feed accelerator	23	AXI-Stream	0.5938
Readback eval	1	AXI-Lite	0.8125

Table 6.2: CPU/FPGA communication parameters in the Connect6 application

isters and the control logic to store the board as it is being received from the ARM processor.

The evaluation output is composed by the number of $t4$s, $t3$s, and $t2$s of each player. These six values occupy 24 bits, fitting in a single word, so they are transferred through the AXI-Lite interface.

Given the board size in Connect6 and the word size, we decided send half a row in each transference (squares from 1 to 10, and from 11 to 19 respectively). The transference occupancy factor is poor, but it simplifies the code that packs the board and the hardware module that stores the board in the accelerator (*Board Storage*) at the expense of a very low impact on the overall performance since the communication overhead in this case study is low (a perfect occupancy factor would just reduce the execution time by less than 1%). Readbacks from the FPGA uses 26 bits out of 32 (81.25%). These data and other communication parameters are summarized in Table 6.2.

6.4 CASE STUDY III: BLOKUS PLAYER

6.4.1 *Hardware acceleration*

Profiling our Blokus software application showed that it spends 92.7% of the time evaluating boards, 5.3% finding legal moves and generating new nodes, 1.9% generating overlapping maps, and the remaining 0.1% executing the minimax search. According to these results, we decided to move the board evaluation to hardware. We also moved the generation of overlapping maps, despite not being one of the larger

Vertices

```
for each row
   for each col
      for each pattern
```

Accessibility

```
for each vertex
   for each window_square
      for each pattern
```

Figure 6.8: Parallelism in the Blokus Duo application

hotspots, because the hardware developed to evaluate nodes also provides such maps. Figure 6.8 outlines the framework of the software method that computes these metrics in order to show the sources of parallelism in these tasks. The task that identifies and selects the vertices to analyze is fully parallel, and the task that processes the accessibility surrounding a vertex, although it does not present any data dependence, only exploits a degree of parallelism of two thanks to a pair of units traversing the board in opposite directions. Including more units would exceed the hardware resources of the FPGA where it was implemented.

6.4.2 *Codesign schemes*

Figure 6.9 illustrates the codesign schemes developed for the Blokus application, one that computes in hardware the board evaluation, and another one that also computes the overlapping maps reusing the same hardware that performs the evaluation. The accelerator developed for this game has three inputs: the board that provides which player occupies each square to find the vertices, the board that provides which squares are forbidden for each player to identify the squares that are accessible from each vertex, and the tiles available for each player. A first approach would be to send all these data, but, since the board that stores which squares are forbidden is easily computable from the board that stores which squares are occupied by each player, we decided to send only the latter and to include a combinational module in the accelerator to compute the other board encoding (*Board composer*). This strategy halves the dataflow from the processor to the FPGA with no penalty on performance and a minimal extra development effort. Another small module to store the board (*Board storage*) is also necessary for the same reason we explained in the Connect6 application. Each board processing, after packing, demands to send 15 words from the processor to the accelerator - 13 to send the board, and 2 to send the array of tiles -. Readbacks return the evaluation values or the overlapping map. Evaluation values are two values of 7 bits, which fit in a single word, indicating the accessibility of each player. The overlapping map is an array of 196 bits (fits in 7 words). Following the interface selection guideline, the evaluation values are transferred through the AXI-Lite interface, and

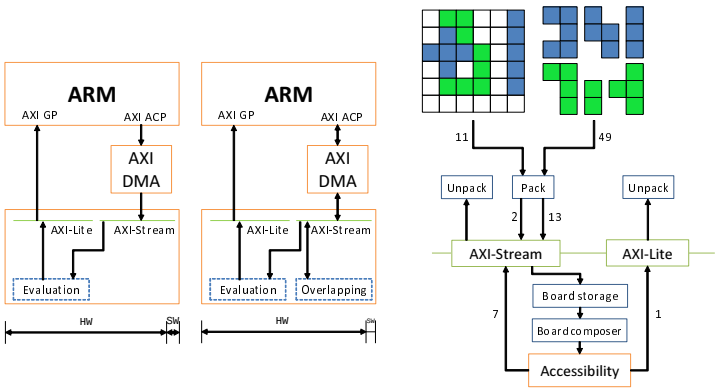


Figure 6.9: Codesign schemes for the Blokus application

Operation	Transfer size (words)	Interface	Transfer occupancy
Feed accelerator	15	AXI-Stream	0.875
Readback eval	1	AXI-Lite	0.4375
Readback map	7	AXI-Stream	0.875

Table 6.3: CPU/FPGA communication parameters in the Blokus Duo application

the overlapping map through the AXI-Stream. Notice that they are independent operations, evaluation values are only requested when processing terminal nodes whereas overlapping maps are requested when processing non-terminal nodes.

The features of each operation involving data movement for this application are shown in Table 6.3. We send a row in each transference, resulting in an utilization of 28 out of 32 bits (87.5%). Readback from the FPGA uses 14 bits out of 32 (43.75%) for evaluations, and 28 out of 32 bits (87.5%) for overlapping maps.

6.5 EXPERIMENTAL RESULTS

We evaluated both performance and energy efficiency of the all the codesigns described in the previous section and the software-only versions over the Zynq SoC. The software-only versions were also evaluated on the Intel i7 processor to add a reference of a performance-oriented processor. Results were gathered following the methodology described in Section 5.4. We also evaluated the impact of the communication between the processor and the FPGA in each codesign scheme, and, finally, the development effort of both the software-only and codesign versions.

6.5.1 *Performance and energy*

Tables 5.4, 5.5, and 5.6 show the experimental results for each case study. *Ex. Time* stands for the time required to complete a game; *Partitioning* details how the computation is distributed between the processor and the FPGA.; *Power* represents the dynamic power consumption, i.e., the average increase in power consumption due to the execution of our application; numbers in the column *Energy* are the product of power and execution time, which represents the energy consumed during a game due to the execution of our application; and *Energy efficiency* measures the quotient of performance and power. We obtained these measures with the search tree exploring eight moves in advance in the case of the Reversi, four moves for the Blokus, and three moves in the case of the Connect6. With these parameters, our Reversi application explores 14.4 million boards during a game, the Blokus application explores 38.4 million, and the Connect6 explores 4 million.

The results in Table 6.4 show that the codesign solutions offer remarkable speedups over the software-only version for the Reversi game. The first hybrid design achieves a 1.9 speedup over the software running on the Cortex-A9 by moving to the programmable logic fabric the computations responsible for the 53.5% of the original computation time, while the second codesign, which moves to hardware the whole board evaluation, is 6.3× faster. The latter, based on a low-power processor, approaches the Intel i7 processor performance while consuming 173 times less energy.

In the case of Connect6 (Table 6.5), moving the board evaluation to the hardware reduces execution time and the energy consumed by a factor of 11. As in the Reversi game, this codesign alternative with a low-power processor almost reaches the performance of the high-performance Intel i7, but requiring 250 times less energy. Codesign *a** slightly improves the results thanks to a customization of the DMA driver, taking advantage of the particularities of this application.

The results for Blokus, shown in Table 6.6, are impressive since the hybrid designs even outperform the Intel i7. The reasons are that the portion of the computation moved to the programmable logic fabric is greater, and that the size of the data transferences benefits from the high throughput offered by the AXI-Stream interface. Hybrid designs are from 14× to 20× faster than the software application running on the ARM processor. This improvement in performance leads to huge energy savings. Notice that codesigns *(a)–(a*)*, and *(b)–(b*)* have the same task partitioning and the only difference among them is the use of a customized DMA driver.

Figure 6.10 compares the energy efficiency of the most representative solutions. That is, the software running on the high-performance desktop processor (*i7*) and on the low-power mobile processor (*Cortex-*

Platform	Ex. Time (s)	Partitioning (CPU - FPGA)	Power (W)	Energy (J)	Energy Eff (kBoards/J)
i7	33.0	100.0% - 0.0%	24.19	798.27	18.0
Cortex-A9	365.8	100.0% - 0.0%	0.10	36.58	393.7
Codesign (a)	189.9	46.5% - 53.5%	0.08	15.19	947.9
Codesign (b)	57.7	8.1% - 91.9%	0.08	1.51	3119.6

Table 6.4: Reversi experimental results

Platform	Ex. Time (s)	Partitioning (CPU - FPGA)	Power (W)	Energy (J)	Energy Eff (kBoards/J)
i7	96.0	100.0% - 0.0%	29.98	2,878.08	5.0
Cortex-A9	1244.6	100.0% - 0.0%	0.10	124.46	115.7
Codesign (a)	116.5	9.6% - 90.4%	0.10	11.65	1236.1
Codesign (a*)	112.0	9.6% - 90.4%	0.10	11.20	1285.7

Table 6.5: Connect6 experimental results

Platform	Ex. Time (s)	Partitioning (CPU - FPGA)	Power (W)	Energy (J)	Energy Eff (kBoards/J)
i7	852.0	100.0% - 0.0%	28.56	24,333.12	0.6
Cortex-A9	8,615.5	100.0% - 0.0%	0.10	861.55	16.7
Codesign (a)	614.1	7.4% - 92.6%	0.10	61.41	234.5
Codesign (a*)	573.8	7.4% - 92.6%	0.10	57.38	251.0
Codesign (b)	475.8	5.4% - 94.6%	0.09	47.58	336.3
Codesign (b*)	427.8	5.4% - 94.6%	0.09	42.78	374.0

Table 6.6: Blokus experimental results

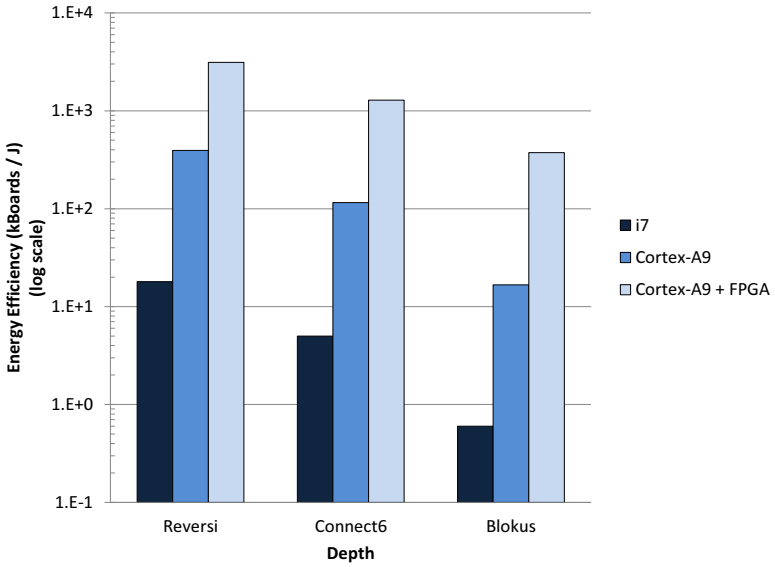


Figure 6.10: Energy efficiency of the most representative alternatives

A9), and the best codesign solution for each case study (*Cortex-A9 + FPGA*). Moving the computationally-intensive tasks to hardware allows the designers to increase by one order of magnitude the energy efficiency. This difference rises to two or even three orders of magnitude when compared with i7.

6.5.2 *Communication overhead*

We stated in Section 6.1 that one of the key points of codesign is the communication scheme. For this reason, we studied the interfaces of the Zynq SoC to select the most suitable one in each data movement operation, and we added additional code to remove useless data from the messages involved in that operations.

Table 6.7 quantifies the impact of the communication between the processor and the programmable logic in terms of performance for the hybrid schemes. We collected the data by measuring the time spent moving data among them. The overhead due to data packing and unpacking is included in the communication overhead presented in the table. Our codesign schemes send the board from the processor to the accelerator every time the accelerator is used and then sends back the evaluation values. The overhead in our most communication-optimized versions is lower than 10%, except in the Reversi because it cannot take advantage of the higher throughput of the AXI-Stream interface.

There are design alternatives that could reduce the communication overhead, but this kind of design decisions imply a compromise between efficiency and design complexity. For instance, a design alternative in our codesigns is to avoid board transferences, which are re-

Design	Communication overhead	Data transferred (MB)	Throughput (MB/s)
Reversi (a)	15.7%	596.8	20.0
Reversi (b)	55.5%	641.0	20.0
Connect6 (a)	8.9%	386.2	37.0
Connect6 (a*)	4.5%	386.2	73.6
Blokus (a)	14.4%	2543.2	28.7
Blokus (a*)	8.4%	2543.2	52.8
Blokus (b)	18.8%	2702.5	30.2
Blokus (b*)	9.0%	2702.5	70.4

Table 6.7: PS/PL Communication overhead

sponsible for most of the overhead. In a game-tree search, each board to be evaluated can be obtained making the sequence of movements of the current branch under exploration on the actual board, and the size of that sequence is much lower than the board. Figure 6.11 illustrates this idea. Each time our application is requested to find the next move, it would send the current board to the accelerator. Notice that now it would be done just once for each search. Then, every time our search algorithm reaches a node that has to be evaluated, only the move sequence is sent to the accelerator instead of the board. In the example illustrated on the figure, when node (a) has to be evaluated, the software would send the move sequence $\{m_{L1}, m_{L2}, m^1{}_{L3}\}$ to the accelerator. The preprocessor included would compute the resulting board after making that movements on the current board (module *Tile placer* described in item 1), and would forward it to the accelerator. This scheme would have a great impact in the communication overhead, since the size of a move chain is notably smaller than the board, but requires to develop additional hardware and to modify the interface of the software method that the hardware accelerator is replacing.

6.5.3 *Resource utilization and development effort*

The development of the software versions used in this study, including optimization and profiling, took from seven to ten weeks for each game. Regarding the hybrid versions, the development and debugging of the hardware accelerator for the Reversi required only four days due to its simplicity. The accelerators for the Connect6 and Blokus are more complex, especially in the case of Blokus, and their development took one week and three weeks, respectively. It is important to mention that the available design tools for codesign (in our case we use Xilinx Vivado [76]) are definitely helpful since the

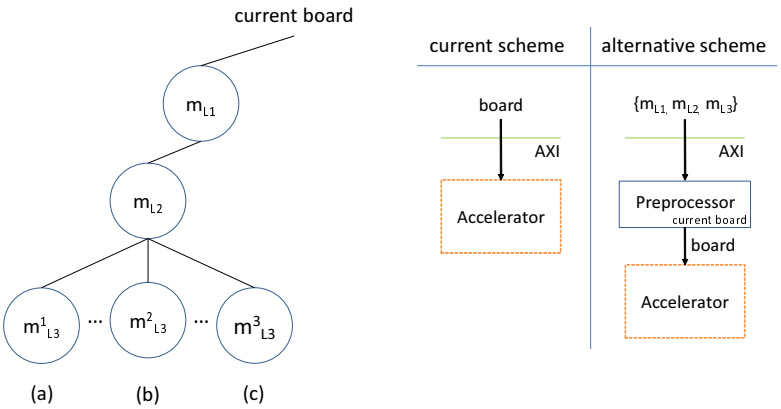


Figure 6.11: Alternative communication scheme

Design	LUTs (%)	FFs (%)	BRAMs (%)	Compi. time (s)	Design time (weeks)
Reversi software	-	-	-	1.7	7
Reversi codesign (a)	4.8	1.1	0.0	344.0	8
Reversi codesign (b)	7.4	1.2	0.0	422.0	8
Connect6 software	-	-	-	2.4	8
Connect6 codesign	72.0	6.4	0.7	1545.0	9
Blokus software	-	-	-	2.9	10
Blokus codesign (a)	29.7	4.9	1.4	1,139.0	13
Blokus codesign (b)	30.0	5.0	1.4	1,247.0	13

Table 6.8: Resource utilization in codesign solutions and development effort in software-only and codesigns

communication infrastructure and the hardware/software interfaces are generated automatically, and the software can interact with the hardware accelerator just as it is done with any other peripheral.

Table 6.8 summarizes the FPGA resources used in each codesign scheme, the compilation time for these schemes and for the software-only applications, and the design time invested. Notice that compilation time in hybrid solutions is similar to the numbers in hardware designs presented in Section 5.4, but the actual impact in the development cycle is noticeably lower since hardware in codesign is much simpler than in hardware-only designs, and therefore the development demands much less compilations. Thus, the overhead in terms of time of the codesign solutions with regard to the software-only solutions are 14.3% for Reversi, 12.5% for Connect6, and 30.0% for Blokus.

Part IV

DYNAMIC PARTIAL RECONFIGURATION

7 MULTI-MODE RECONFIGURATION CONTROLLER

This chapter presents the reconfiguration controller that we designed in order to increase efficiency in the reconfiguration process. We first discuss several techniques explored by the research community to reduce the reconfiguration overheads, and then we describe the architecture and the operation of our proposal.

7.1 INTRODUCTION

Dynamic partial reconfiguration (DPR) is one of the most interesting features of FPGAs. Reconfiguration enables the reuse of the FPGA hardware resources for different tasks that can be loaded at run-time according to the system needs. Hence, FPGAs can be used to develop flexible platforms that can adapt themselves to the execution of different applications. However, the reconfiguration process introduces overheads both in the execution time and in the energy consumption. The reason is that it involves not only using the reconfiguration circuitry to update the device configuration, but also moving large data sets from the memory where the configurations are stored to the reconfiguration port.

The analysis of the reconfiguration overheads has been the target of many research groups, mainly focused on time overheads. Many different strategies have been explored:

- *Reconfigurable computing architecture*: Despite FPGAs market is clearly dominated by single-context and fine-grained architecture, several works have proposed alternative reconfigurable architectures like coarse-grain architectures [24], which offer less flexibility but require smaller configuration bitstreams, or multi-context FPGAs [38], which permit loading a new configuration while another one is being executed.

- *Scheduling*: Scheduling techniques attempt to hide the reconfiguration latency by fetching the configurations in advance and storing them in idle reconfigurable regions. This approach is powerful in embedded systems where the task execution order is known. Some relevant works that propose reconfiguration scheduling techniques are [13, 22, 41, 49, 54, 61].

- *Compression*: Configurations are stored in bitstreams that are amenable of compression. Compressed configurations are fetched faster, but they have to be decompressed before writing them in the device, and this may involve additional time and energy

penalties. The inclusion of hardware support for decompression is crucial to minimize these penalties. Some relevant works on compression are [17, 40].

- *Customized storage resources*: The configuration process may benefit from a hierarchical storage composed by memories with different performance and energy profiles. For instance [14] proposes the inclusion of heterogeneous on-chip memory modules, ones optimized for performance and others optimized for low-power. With this approach the designer can explore different power/performance reconfiguration tradeoffs.

- *Caching*: The idea is to store configurations in different reconfigurable regions in the device and design specific replacement techniques to maximize the configuration reuse. Some interesting configuration caching techniques are presented in [12, 34, 39]. The application of the caching technique is not limited to reconfigurable regions, but it can be also applied on the memory hierarchy where configurations are stored.

Regarding the reconfiguration process, many works assume that the reconfiguration latency is a fixed time that can be calculated by dividing the size of the configuration by the peak bandwidth of the reconfiguration port. In fact, the peak reconfiguration bandwidth is usually the only value provided by FPGA vendors. However, is that value relevant? Can we really achieve that bandwidth? How can we do it? Several recent works point out that the actual reconfiguration latencies obtained in representative case studies are one order of magnitude, or even two, worse than the peak reconfiguration latency [19, 52]. The reason for these poor results is that configurations are usually stored in off-chip non-volatile memories, and the actual bandwidth of these memories is much smaller than the bandwidth of the reconfiguration port. Hence, the bottleneck is not the reconfiguration port, but the bandwidth of the external memories. Some recent works have demonstrated that it is possible to achieve almost peak-performance when using some specific external memories with some additional support. For instance in [66] the authors achieve almost peak reconfiguration performance for a Virtex-6 when using DDR3 memories, a DMA controller, and some additional FIFOs to hide the latency, and in [43], they retrieve configurations from an external SRAM through a customized DMA. However, the last generation of FPGAs can be reconfigured at 200 MHz (previous Xilinx FPGAs are limited to 100 MHz), and, even with this scheme, it is unfeasible to achieve peak reconfiguration performance. Moreover, in many systems energy overheads are as relevant as performance, and the use of external memories strongly penalizes in power and energy.

Some previous works propose the inclusion of memory resources inside the reconfiguration controller in order to preload configura-

tions. In [47] the authors analyze the reconfiguration latency of a system implemented in a Virtex-4 FPGA that includes a system bus that is used for all the data transfers, including those needed to carry out a reconfiguration, i.e. reading the configuration and sending it to the reconfiguration port. They measured the reconfiguration latencies taking into account the different access schemes provided by the bus, and including a DMA controller. The results demonstrate the impact of the communication scheme in the reconfiguration latency. In their analysis, they claim that the only way to achieve the peak reconfiguration bandwidth is to avoid accesses to the system bus. Even when the system bus is available and transfers are performed through a DMA, these accesses introduce significant delays.

Other interesting works are [8, 23]. In these articles, the authors propose partial reconfiguration controllers that reaches a throughput of 1.433GB/s and 2.2GB/s. These controllers allocate on-chip memory resources to store the configurations, and the Internal Configuration Access Port (ICAP) is overclocked up to 550 MHz. Although the manufacturer does not guarantee proper operation at frequencies higher than 100 MHz [67], these works are a proof of concept of the use of internal memory for future and faster versions of the ICAP. After analyzing these interesting previous works, it is clear that the reconfiguration latency drastically depends on where the configurations are stored and the communication scheme used to read them.

7.2 TARGET ARCHITECTURE

We made the evaluation on the XUPV5-LX110T Development System included in the Xilinx University Program. Our target architecture consists of three different memories that can be used to store configuration bitstreams, a reconfiguration controller that provides an interface with the ICAP , a processor, and at least one RR. Figure 7.1 depicts the elements of this architecture. In our experiments, the processor is a Xilinx MicroBlaze, the system bus is a PLB 4.6, and the memories are a 1GB Compact Flash, a 64-bit wide 256MB DDR2 SODIMM, and the FPGA internal BlockRAMs (BRAMs). Additionally, DMA and interrupt controllers supplied by Xilinx were added in order to evaluate their performance-energy tradeoffs.

The selection of the memories where configurations are stored, and how to move these configurations among these memories and the reconfiguration controller is not straightforward, as it depends on multiple factors such as the power budget, the performance requirements, and the needs of other system components (for example, if the DDR or the Flash memories are already used by other components, the static power of these controllers will not introduce an additional penalty). In the experimental results presented in Section 5, we an-

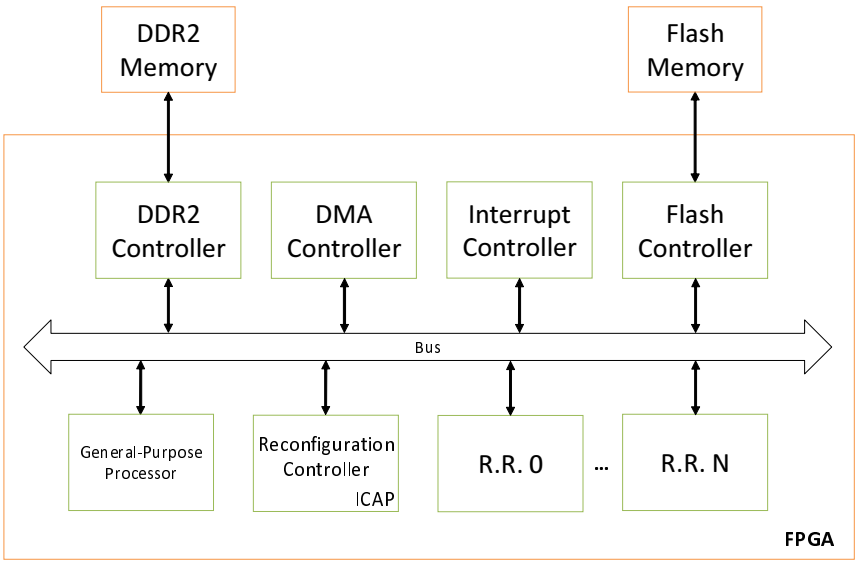


Figure 7.1: Target architecture

alyze the reconfiguration energy-performance trade-off in order to help designers to make their decisions.

7.3 PARTIAL RECONFIGURATION CONTROLLER

7.3.1 *Xilinx IP*

Xilinx provides the XPS_HWICAP IP core for the Virtex-5 FPGA to manage partial reconfigurations in processor-based systems [73]. It is composed by several control registers, two small FIFOs, a finite-state machine (FSM), and a PLB bus interface [72]. Xilinx also provides a driver to use this controller from the processor-side. This driver enables read configurations from any memory in the system and to send the data to the XPS_HWICAP controller through the PLB bus. This controller is easy to use and it should be the starting point for anybody who wants to carry out reconfigurations. However, this driver has not been designed to optimize the data transfers among the memories and the XPS_HWICAP. As a result, as explained in section 5, it only achieves a reconfiguration throughput of 12 MB/s, far from the peak 400 MB/s supported by the ICAP.

7.3.2 *Multi-Mode ICAP Controller*

In order to reach the maximum reconfiguration throughput, the ICAP should receive one 32-bit word per cycle at a 100MHz rate. As explained in the previous section, it is hard to achieve this bandwidth when configurations are stored in off-chip memories, or even when they are stored on-chip but they are accessed through a shared sys-

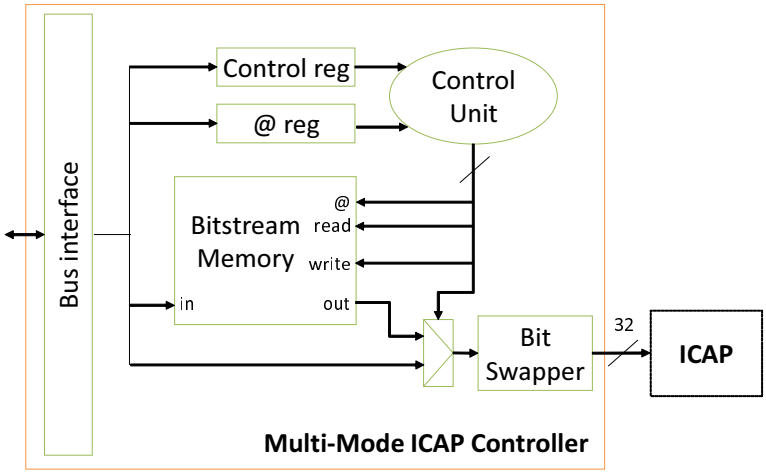


Figure 7.2: Multi-Mode ICAP Controller

tem bus. A solution is to store them inside the configuration controller. With this approach the reconfigurations can be carried out at full speed. In current FPGA architectures, this can be achieved by reserving part of the on-chip RAMs for the controller.

Figure 7.2 illustrates the architecture of our partial reconfiguration controller, called Multi-Mode ICAP. The bus interface was automatically generated by the Xilinx EDK tool, and the remaining blocks were designed in VHDL and integrated inside the EDK project. The Control and the Address registers are software accessible. Hence the processor can easily provide the information and check if the controller has finished. If the system includes an interrupt controller, the controller can generate an interrupt when the done bit is activated; again this is straightforward using the EDK tools. This architecture based on an internal memory, a register-based interface, and a control unit is very similar to those proposed in [8, 47]. The main difference is that our control unit provides support for several useful working modes.

The Multi-Mode ICAP supports four different working modes:

- *Mode 0: Configuration Load*. The controller receives a configuration and stores it in the Bitstream Memory. The controller does not send the configuration to the ICAP. This mode is useful to fetch configurations in advance from the external memories in order to reduce the reconfiguration latency since, once a configuration is stored, our controller sends it to the ICAP at the maximum supported speed.

- *Mode 1: External reconfiguration and configuration load*. The controller receives a configuration and forwards it to the ICAP. In parallel, it stores the configuration in the internal Bitstream Memory. In this mode the reconfiguration speed depends on how fast the configuration data are received from the bus, i.e.

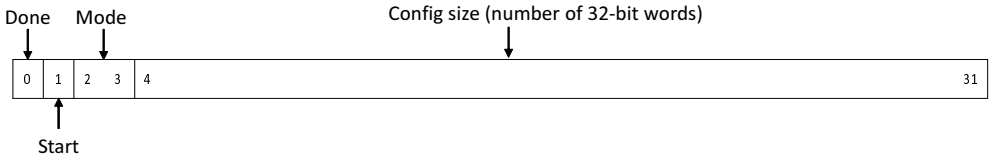


Figure 7.3: Control Register

our controller does not reduce the reconfiguration latency in this mode. However, if the same configuration is required again, it can be loaded from the internal Bitstream Memory at the maximum speed.

- *Mode 2: External reconfiguration*. The controller receives a configuration and forwards it to the ICAP as in the previous case, but in this mode the configuration is not stored in the Bitstream Memory.

- *Mode 3: Internal reconfiguration*. The controller sends to the ICAP a configuration previously stored in the Bitstream Memory. The reconfiguration is carried out at full speed.

In order to support this functionality, only two software accessible registers (Control and Address registers) and little additional control logic are necessary. The control register (Figure 7.3) is used to configure our controller. Bit 0 reports when an operation has finished, bit 1 triggers the start of an operation, and bits 2 and 3 are used to select the working mode. Finally the remaining 28 bits are used to set the size of the configuration expressed in 32-bits words. The address register specifies the initial address of the Bitstream Memory to store or load a specific configuration.

The control unit (Figure 7.4) only requires three states: *Idle*, *Forward*, and *Internal Reconfig*. The controller is initially in the *Idle* state. When the start bit is activated it will move forward to state *Forward*, or *Internal Reconfig* according to the mode set on the control register. In *Forward*, every configuration word received from the bus is forwarded to the proper destination: Bitstream Memory for modes 0 and 1, and the ICAP for modes 1 and 2. In *Internal Reconfig*, the controller reads the configuration stored in the Bitstream memory included in the reconfiguration controller, and forwards it to the ICAP. In all cases, a counter is used to know how many words have been processed and to update the next Bitstream Memory address to be read. When the counter reaches the number of words requested the Done bit is activated.

The implementation of this control unit requires a 2-bit register to store the current state, a counter that keeps track of how many words have been received, an adder to update the Bitstream Memory address, and a comparator to know when all the words have been

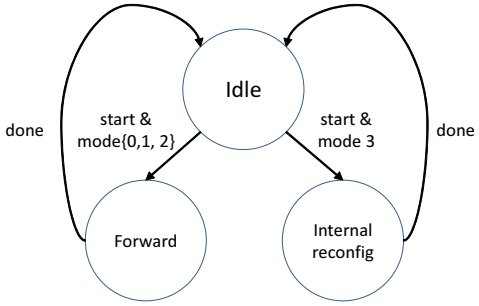


Figure 7.4: Multi-Mode ICAP Finite State Machine

processed. We have also included a bit swapper module. The reason is that Xilinx Plan Ahead, which is the tool that we used to generate the bitstreams for run-time reconfiguration, does not generate the configuration data in the same bit order required by the ICAP [67]. Hence, the bit swapper module reorders each configuration word by swapping the bits within each byte, i.e. from ($b_{31}..b_{24}$, $b_{23}..b_{16}$, $b_{15}..b_8$, $b_7..b_0$) to ($b_{24}..b_{31}$, $b_{16}..b_{23}$, $b_8..b_{15}$, $b_0..b_7$). This step might be instead done in software once a configuration bitstream is generated, and then this module could be removed. We decided to do it in hardware because it does not introduces any hardware overhead and because it simplifies the use of the controller.

The available on-chip RAM varies a lot depending on the FPGA used. The current tendency in computer architecture is to include more and more on-chip memory resources. In fact, Xilinx has recently announced a new family of FPGAs, called UltraScale+TM, which brings a breakthrough including up to 65,913 Mbits of on-chip RAM [68], making it feasible to store several configurations even for large reconfigurable regions. However, small FPGAs have less than 1 Mbit. Hence, the benefits of including a Bitstream Memory embedded in our controller depend on the availability of on-chip memory and the size of the reconfigurable regions. If there are enough on-chip memory resources to store a significant part of the needed configurations our controller will help to optimize the reconfiguration process. At this point, other orthogonal techniques such as compression would play an important role. Adding compression support to our controller is as simple as including a decompression module before the bit swapper.

In our controller the size of the Bitstream Memory is a generic parameter that can be instantiated with different values. In our implementation we can use up to 256 KB. The FPGA includes more RAM resources, but they are used to implement other elements of the system. This size can be used to store the configuration of a reconfigurable region of 8000 LUTs. If this is not enough, the configuration can be partially stored in the Bitstream Memory, and later the reconfiguration can be carried out combining modes 2 and 3. This is a key feature to maximize the use of the on-chip memory.

7.3.3 *Configuration Prefetching and Caching support*

Our controller has been designed to provide support both for configuration prefetching and configuration caching. As explained in the related work section, prefetching and caching have been proved to be powerful techniques to reduce the reconfiguration latency. These techniques are usually applied by loading and storing the configurations in different reconfigurable regions.

The idea behind configuration prefetching, as applied in the articles [13, 22, 41, 49, 54, 61] cited in Section 7.1, is to carry out the reconfigurations in advance. For instance, while a task is executed on a RR, we may prefetch the following task by storing it in another RR.

Configuration caching in FPGAs consists in temporally storing some selected configurations in idle RRs. Hence, when these tasks need to be executed, they are already loaded and no reconfiguration is needed (this is called a configuration cache hit). Some examples already mentioned in Section 7.1 are [12, 34, 39].

However, the application of these techniques onto RRs presents a strong limitation: it is only feasible in systems where some of the RRs are idle and fits the configuration to be preloaded. Including an on-chip memory inside the reconfiguration controller in order to store configurations helps to overcome this limitation since even when all the reconfigurable regions are busy, it is still possible to apply these techniques. Of course the benefits are different: if a configuration is stored in a RR, it can be immediately used when required without carrying out a reconfiguration, whereas if it is stored in the Bitstream Memory of our controller, the reconfiguration is still necessary, but it can be carried out at full speed.

Hence, our controller supports configuration prefetching and caching at two levels, RRs and on-chip memory. Chapter 8 presents a case study that illustrates the benefits of this additional level. In this study we have included only one RR and, we have used the Bitstream Memory for configuration caching and prefetching.

8 EXPERIMENTAL RESULTS

In this chapter we expose the experimental results of the reconfiguration controller presented in the previous chapter. We measured reconfiguration latency and the power demanded both for retrieving the configurations from different memories, and for loading the configuration onto the FPGA. Systems with DRP capability usually include other components, such as a DMA, memory controllers, or an interrupt controller, and these components also have an impact in the overheads. We analyze the penalty in terms of resources and power caused by their inclusion. Finally, we analyze the performance of our controller on a 3D rendering application, using the prefetch and caching capabilities.

8.1 LATENCY

We have measured the reconfiguration latency for both Xilinx controller and our controller retrieving configuration data from three different memories: a non-volatile off-chip memory (Flash), a volatile off-chip memory (DDR2), and a volatile on-chip memory (BRAM). Figure 8.1 depicts the normalized reconfiguration latencies for all the evaluations. Notice that the results are represented in logarithmic scale. The red line points out the minimum latency according to the ICAP bandwidth.

With the XPS_HWICAP controller, reconfigurations from Flash take 2,900 ms/MB. In the case of transferences from the Flash memory, the inclusion of a DMA controller does not reduce significantly the latency because the Flash controller limits transferences to only 2 bytes. If data are read from DDR2, the latency decreases to 117 ms/MB without DMA and 34.7 ms/MB with DMA, still far from taking fully advantage of the reconfiguration port bandwidth. Finally, retrieving the configurations from on-chip memory is the fastest choice, with a latency of 79 ms/MB without DMA, and 28 ms/MB with DMA.

In the case of our controller, both Flash and DDR2 alternatives got similar latencies with the XPS_HWICAP controller since in these cases the controller is not the bottleneck. However, the on-chip reconfiguration is clearly faster. The reason is that our on-chip memory is embedded in the controller. Hence, no accesses to the system bus are needed. For this reason, our controller achieves the minimum reconfiguration latency (one write operation per cycle at 100MHz, i.e. 400 MB/s), which is 31 times faster than the XPS_HWICAP. In fact, our controller can be clocked at 500 MHz, but, as explained before, Xilinx

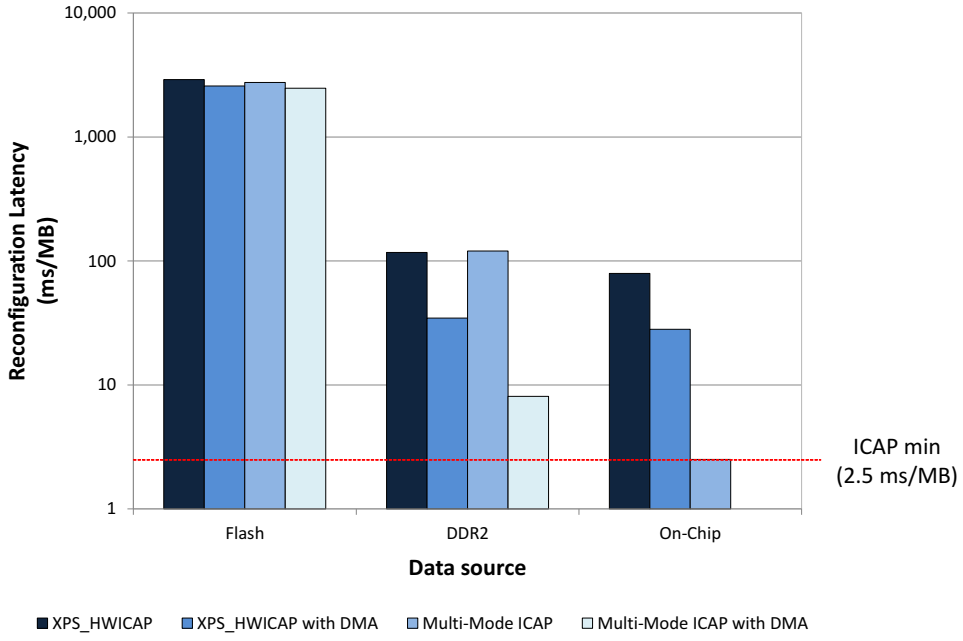


Figure 8.1: Reconfiguration latencies

does not guarantee proper operation for frequencies above 100 MHz in this FPGA [67].

We have also evaluated the benefits of including a DMA controller to manage the transactions from the different memories without processor intervention. As it can be seen in the figure, the DMA controller reduces all the latencies significantly, but even in the best case is still eight times slower than using the embedded Bitstream memory. Moreover the original Xilinx functions for the XPS_HWICAP do not support DMA transactions. Hence they have to be modified by the user.

8.2 RESOURCES AND POWER OVERHEADS

In order to analyze the performance-energy tradeoff, we took power measures of reconfiguration. Most previous work on this is based on models [7, 62], and therefore they do not perform any actual measurement. A previous work that actually performs measurements is [48]. A. In this work, Nafkha *et al.* carried out power measures of the reconfiguration process on a Xilinx ML505 development board using a high-speed digital oscilloscope and the shunt resistor method. For our measurements, we decided to use a Yokogawa WT210 digital power meter, which is an accepted device by the Standard Performance Evaluation Corporation (SPEC) for power efficiency benchmarking [79].

Table 8.1 summarizes the FPGA resources and the static power overheads of each component. To measure the static power consump-

Component	LUTs (%)	FFs (%)	BRAMs (%)	Static power (W)
DDR2 controller	3.66	5.34	3.38	3.54
Flash controller	0.15	0.31	0.00	0.32
Interrupt controller	0.12	0.18	0.00	0.02
DMA controller	1.01	0.81	0.00	0.64
Multi-Mode ICAP (128KB)	0.64	0.51	21.6	0.28
Multi-Mode ICAP (256KB)	0.65	0.52	43.24	0.45
XPS_HWICAP	1.04	1.02	1.35	0.16

Table 8.1: Resources utilization and static power consumption

tion of each component, we first measured the power consumption of a system with all the components in an idle state, and then we removed them one by one and repeated the power measures. The difference between these sequent measures corresponds to the static power of each component removed. Ambient temperature was constant in all the measures.

Although these data may change from one FPGA to another, we believe that it provides an interesting hint for any designer in order to decide whether to include or not any component based on the overheads, the resources available, the power budget, and the system requirements. For instance, in systems with a very constrained power budget, avoiding the use of the DDR2 memory, if possible, will significantly help to meet this power budget.

LUTs and FFs required by the Multi-Mode ICAP controller keep almost constant and very low regardless its storage capacity (the small variations are due to the additional addressing bits required for higher memory capacities), and the static power consumption linearly increases with the memory size by a factor of 1.6.

Figure 8.2 depicts the dynamic power consumption due to the reconfigurations, both for our controller and for the XPS_HWICAP. Each bar is divided into two terms: *Data Transference*, which includes the dynamic power consumption due to data movement from the external memories to the reconfiguration controller, and *Reconfiguration*, which includes the power consumption of the reconfiguration controller and the FPGA reconfiguration circuitry.

To split the power consumption between reconfiguration and data transference we carry out two different measures. First we carry out the reconfiguration. This involves reading the configuration from the given memory and carrying out the reconfiguration process. Second, we repeat the process but this time we use our reconfiguration controller in mode 0. In this mode we read the configuration and send it to the controller, but the controller does not carry out the recon-

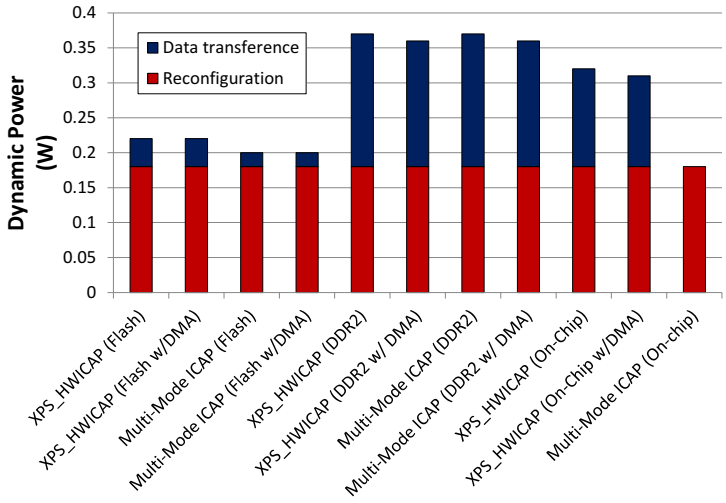


Figure 8.2: Dynamic power consumption

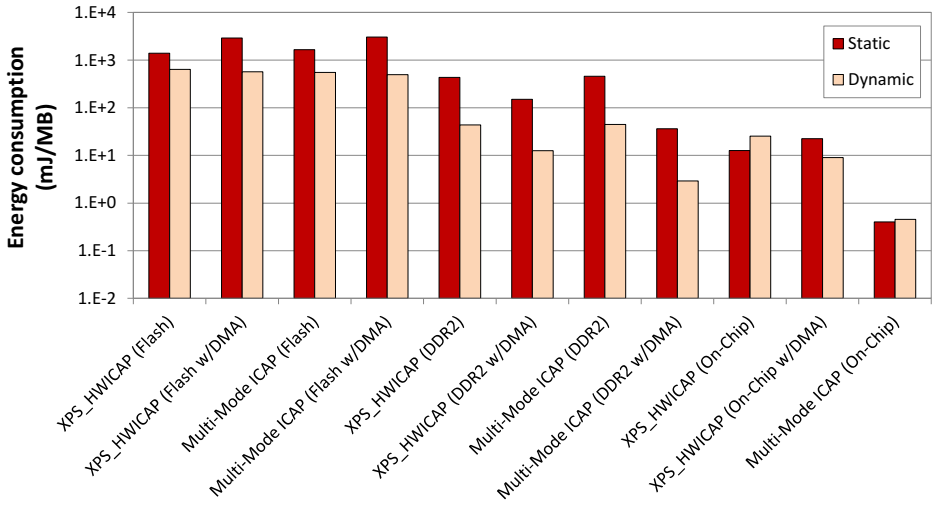


Figure 8.3: Energy consumption

figuration. The difference between these two values corresponds to the power term associated to the reconfiguration. In all the cases we repeat the process inside a while loop for several minutes, and took the average value to avoid noisy power data.

The term 'Reconfiguration' keeps almost constant about 0.15W in all the setups. On the contrary, the *Data Transference* term greatly varies depending on the case. This term is null for *Multi-Mode (on-chip)* because all the process is carried out inside the controller, whereas setups retrieving configurations from DDR2 present the highest values due to the use of the DDR2 controller and the access to an off-chip memory. The term 'Data Transference' is reduced by 26% if the configurations are stored on-chip instead of on DDR2, and it is even lower when using the Flash memory. However, in the latter case the reason for the reduction is the slow access to that memory.

Figure 8.3 shows the energy consumed, in miliJules, per MB reconfigured for the setups evaluated in Figure 8.2. The setups that retrieve the configurations from the Flash memory exhibit the worst results, taking into account both static and dynamic energy. Despite consuming less power than the DDR2-based setups, the long latency of this memory (see Figure 8.1) causes a strong energy penalty. On the contrary, setups that use an on-chip memory are the most energy-aware. It is remarkable the fact that our Multi-Mode ICAP controller requires two orders of magnitude less energy than the XPS_HWICAP (on-chip). In fact, the energy required by this setup is, at least, one order or magnitude lower than any other setup. Hence a bitstream memory embedded in the reconfiguration controller drastically reduces the reconfiguration energy overheads.

8.3 CONFIGURATION PREFETCHING AND CACHING

Our multi-mode controller has been designed to provide an additional level for configuration caching and prefetching. Using the Bitstream Memory, we can apply these techniques even when all the reconfigurable regions are busy. In this section, we present a case study to illustrate the benefits of this approach.

We have selected a 3D rendering application based on the open source Pocket-GL library (Pocket GL). This application includes 20 different sequential task graphs that are combinations of ten different tasks. We analyzed the same application in a previous article that presented a run-time scheduler for reconfigurable systems [13]. This scheduler included support both for configuration prefetching and caching taking advantage of idle reconfigurable regions to reduce the reconfiguration overhead. However, since these are sequential graphs, the system designer may decide to include only one reconfigurable region to execute this application, and in that case there would be no idle regions available. Hence, we have extended our scheduler to apply prefetching and caching using the Bitstream Memory of our controller whenever no idle regions are available.

The results are depicted in Figure 8.4. The leftmost column shows the initial overhead when the reconfigurations are carried out on demand and without applying neither prefetching nor caching. We assume that the configurations are stored in the DDR2 memory, and that the system includes a DMA controller. In this case the application needs 70% more time due to the reconfigurations. If we use the Bitstream Memory to apply a prefetch approach we can reduce that overhead to 38%. The implemented approach is very simple, while executing one task the following one is stored, totally or partially depending on the available time, in the Bitstream Memory, and when the stored task is executed our controller carries out the reconfiguration of the portion stored in the Bitstream Memory 3.2 times faster.

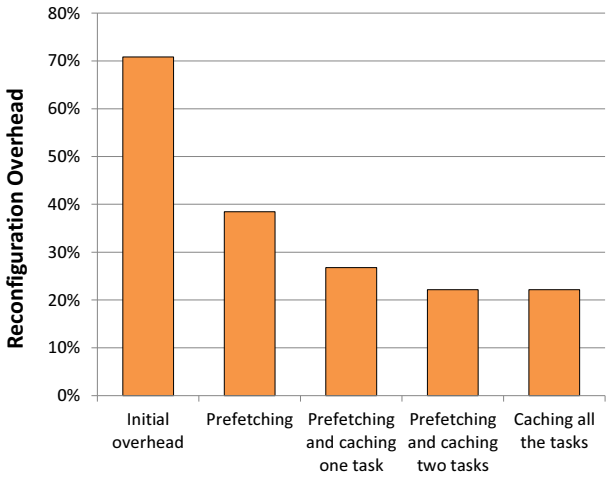


Figure 8.4: Reconfiguration overheads for the Pocket GL application when using the Bitstream Memory to apply configuration prefetching and caching

These results can be further improved by caching some critical tasks. The idea is to store in the Bitstream Memory the configuration of those tasks that generate the largest reconfiguration overheads. In the figure we can see that when the configuration of the most critical task is cached the overhead is reduced to 27%, and if the two most critical tasks are cached it is 22%. In the rightmost column we can see that caching the remaining 8 tasks provides no further reductions. The reason is that the prefetch technique is already reducing the reconfiguration latency of those tasks. In fact when the two most critical tasks are cached the reconfiguration overhead is reduced by a factor of 3.2, which is the best result that can be achieved in this scenario. Hence, if the Bitstream Memory provides enough storage space to store three configurations (the two cached ones plus the one that is prefetched each time) the system will provide the same performance with a system that stores all the configurations on-chip.

Part V

CONCLUSION

9 CONCLUSIONS AND FUTURE WORK

This chapter highlights the conclusions of this dissertation and points out future lines of investigation.

9.1 HARDWARE DESIGN

The increasing relevance of energy-aware computing is giving more prominence than ever to hardware design. Heterogeneous computing has settled in the computation roadmap, and the design of ASICs is getting prohibitively expensive because of rising costs associated to state-of-the-art integration scales, shrinking the class of applications and markets that can afford them. FPGAs are an excellent alternative by providing a balance between performance, power consumption, and programmability. Corporate movements in the main chip manufacturers, such as the acquisition of Altera by Intel, or recently signed deals between IBM, Qualcomm and Xilinx, demonstrate the importance of this technology.

In Part II of this dissertation, we have explored the potential of programmable logic to accelerate the artificial intelligence on board games. Board games are an interesting case study because are popular applications, frequently found in user's mobile devices, and they demand heavy computations. We analyzed three complex board games: Reversi, Connect6, and Blokus Duo. The analysis considered the metrics performance, power, energy efficiency, and development effort, of hardware and software versions algorithmically equivalent. The hardware implementations improve energy efficiency by three orders of magnitude compared with the software implementations executed on a mobile processor. When compared with a performance-oriented desktop processor, the energy efficiency gap grows up to four orders of magnitude. These impressive results come from both higher performance and lower power consumption. However, these improvements do not come for free, as development in hardware is harder than in software. Hardware is parallel by nature, and this increases the design complexity. Debugging complex designs, where many modules are working and interacting at the same time turns both the design and its validation into a hard undertaking. The development cycle in our hardware designs, developed by experienced hardware designers, was about 2.5× longer than in our software applications, also developed by experienced software developers.

Many market niches governed by tough time-to-market may find unacceptable this results, preventing the use of this technology. FPGA

vendors are putting their efforts in new tools to shorten the development cycle, such as high-level synthesis tools, which, based on software-like specifications, try to automatically extract the parallelism and to translate into a custom hardware system. This tools are now in an early stage, and have much room for improvement.

9.2 HARDWARE/SOFTWARE CODESIGN

Hardware highlights in performance and energy, whereas software do in flexibility. Moving to hardware all the tasks within an applications increases the development effort, whereas the potential profit on those non computationally-intensive tasks is limited by the Amdahl's law. Hardware/software codesign emerges as a discipline to bring together goodnesses from both software and hardware worlds. New platforms released in the last years, coupling general-purpose processors and programmable logic in the same chip, broaden the range of applications where this discipline becomes profitable by facilitating hardware/software integration, and by minimizing the communication overhead.

In Part III of this dissertation, we propose codesign solutions for the AI of the board games explored in the Part II. We identify board evaluation as the most computationally-intensive kernel. We also explore alternatives regarding the hardware/software communication in order to minimize its impact. Our codesigns improves energy efficiency by one order of magnitude over the software-only implementations executed on a mobile processor. In spite of the fact that development on FPGAs adds some complexity to the design process, hybrid hardware/software platforms pays-off the harder development cycle since noncritical tasks remain executed in the general-purpose processor, and the FPGA is reserved for specific and demanding tasks.

9.3 DYNAMIC PARTIAL RECONFIGURATION

FPGAs capability of modifying the operation of specific regions of the programmable logic fabric without halting the remaining ones is a powerful feature that increases the flexibility of the FPGA-based systems, and reduces the chip area required, leading to smaller, cheaper, and less power-hungry silicon devices. It also reduces the overhead associated to the load of the hardware accelerators, since only a portion of the FPGA has to be configured, and therefore configuration bitstreams become smaller.

While the benefits of DRP are clear, the overheads associated to this process, and the optimal setup regarding its management are not so clear. In Part IV of this dissertation, we present an analysis of the overheads in the reconfiguration process, in terms of performance, power and energy. Designers should explore the different tradeoffs offered

by the variety of memories where the set of configuration bitstreams can be stored, and by the components involved in the process, such as the memory controller, a DMA, or the reconfiguration controller, in order to meet the operation constraints and to improve efficiency.

We propose a reconfiguration controller that drastically improves energy efficiency by leveraging on-chip memory, and by including support for prefetching and caching. This controller achieves peak performance and reduces the power penalty of moving data from external memories to the reconfiguration port when configuring from its on-chip memory. The viability of this approach is endorsed by the impressive increase of on-chip memory resources in the latest FPGA architectures.

9.4 FUTURE WORK

We will like to analyze FPGA implementations of AI engines for board games based on machine learning. Machine learning is a field that enjoys enormous popularity today, since it has been proved to succeed in many different knowledge fields, and artificial intelligence in board games is one of them, as the recent victory of the DeepMind artificial player over the human world champion in the game Go proved. In addition, the regular structures of the kernels used in machine learning, and their parallelism degree, make them ideal for hardware acceleration. Furthermore, there is a huge interest in porting these algorithms to mobile devices as they can be used in many different fields, and the main issue is finding energy-efficient solutions.

BIBLIOGRAPHY

[1] V. Allis. "Searching for Solutions in Games and Artificial Intelligence." PhD thesis. Maastricht, The Netherlands: University of Limburg, 1994 (cit. on p. 20).

[2] *AMBA AXI4 Interface Protocol*. URL: https://www.xilinx.com/products/intellectual-property/axi_interconnect.html (cit. on p. 55).

[3] *ARM big.LITTLE*. URL: https://www.arm.com/products/processors/technologies/biglittleprocessing.php (cit. on p. 3).

[4] *Arria V SoC FPGA hard processor system*. 2016. URL: https://www.altera.com/products/fpga/arria-series/arria-v/overview.html (cit. on p. 3).

[5] *Artificial Intelligence and Go*. URL: http://www.economist.com/news/science-and-technology/21694540-win-or-lose-best-five-battle-contest-another-milestone (cit. on p. 18).

[6] *Atlys Spartan-6 FPGA Trainer Board*. URL: https://www.xilinx.com/products/boards-and-kits/1-27b7ol.html (cit. on p. 49).

[7] R. Bonamy, D. Chillet, S. Bilavarn, and O. Sentieys. "Power consumption model for partial and dynamic reconfiguration." In: *International Conference on Reconfigurable Computing and FPGAs*. 2012 (cit. on p. 80).

[8] R. Bonamy, P. Hung-Manh, S. Pillement, and D. Chillet. "UPaRC. Ultra-fast power-aware reconfiguration controller." In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2012, pp. 1373–1378 (cit. on pp. 73, 75).

[9] M. Boulé and Z. Zilic. "An FPGA based move generator for the game of Chess." In: *IEEE Custom Integrated Circuits Conference*. 2002, pp. 71–74 (cit. on p. 20).

[10] H.J. Chang, M.T. Tsai, and T. Hsu. "Advances in Computer Games SE." In: ed. by H. J. Herik and A. Plaat. Vol. 7168. 26. Springer Berlin Heidelberg, 2012. Chap. Game Tree Search with Adaptive Resolution, pp. 306–319 (cit. on p. 16).

[11] W. N. Chelton and M. Benaissa. "Fast elliptic curve cryptography on FPGA." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16.2 (Feb. 2008), pp. 198–205 (cit. on p. 9).

[12] J. A. Clemente, J. Resano, and D. Mozos. "An approach to manage reconfigurations and reduce area cost in hard real-time reconfigurable systems." In: *ACM Transactions on Embedded Computing Systems* 13.4 (Nov. 2014), 90:1–90:24 (cit. on pp. 72, 78).

[13] J. Clemente, J. Resano, C. Gonzalez, and D. Mozos. "A hardware implementation of a run-time scheduler for reconfigurable systems." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 19.7 (July 2011), pp. 1263–1276 (cit. on pp. 71, 78, 83).

[14] J.A. Clemente, E. Perez, J. Resano, D. Mozos, and F. Catthoor. "Configuration Mapping Algorithms to Reduce Energy and Time Reconfiguration Overheads in Reconfigurable Systems." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.6 (June 2014), pp. 1248–1261 (cit. on p. 72).

[15] J. Cong, V. Sarkar, G. Reinman, and A. Bui. "Customizable Domain-Specific Computing." In: *IEEE Design & Test of Computers* 28.2 (2011), pp. 6–15 (cit. on p. 9).

[16] B. Cope, P. Y. K. Cheung, W. Luk, and L. Howes. "Performance comparison of graphics processors to reconfigurable logic: A case study." In: *IEEE Transactions on Computers* 59.4 (Apr. 2010), pp. 433–448 (cit. on p. 9).

[17] A. Dandalis and V. K. Prasanna. "Configuration compression for FPGA-based embedded systems." In: *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGAs)*. 2001, pp. 173–182 (cit. on p. 72).

[18] C. Donninger, A. Kure, and U. Lorenz. "Parallel Brutus: The first distributed FPGA accelerated Chess Program." In: *18th International Parallel and Distributed Processing Symposium*. 2004 (cit. on p. 20).

[19] E. El-Araby, I. Gonzalez, and T. El-Ghazawi. "Exploiting partial runtime reconfiguration for high-performance reconfigurable computing." In: *ACM Transactions on Reconfigurable Technology Systems* 1.4 (Jan. 2009), 21:1–21.23 (cit. on p. 72).

[20] B. Freisleben. "A neural network that learns to play Five-in-a-Row." In: *2nd New Zealand International Two-Stream Conference on Artificial Neural Networks and Expert Systems*. 1995, pp. 87–90 (cit. on p. 18).

[21] *From AI to protein folding: Our Breakthrough runners-up*. Science. Dec. 2016. URL: http://www.sciencemag.org/news/2016/12/ai-protein-folding-our-breakthrough-runners (cit. on p. 15).

[22] W. Fu and K. Compton. "Scheduling intervals for reconfigurable computing." In: *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2008, pp. 87–96 (cit. on pp. 71, 78).

[23] S. Gimle, D. Koch, and J. Torresen. "High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro." In: *IEEE International Parallel & Distributed Processing Symposium*. 2011, pp. 174–180 (cit. on p. 73).

[24] R. Hartenstein. "A Decade of reconfigurable computing: a visionary retrospective." In: *Conference on Design, Automation and Test in Europe (DATE)*. 2001, pp. 642–649 (cit. on p. 71).

[25] G. T. Heineman, G. Pollice, and S. Selkow. "Algorithms in a Nutshell." In: 1st. O'Reilly Media, 2008. Chap. Path Finding in AI, pp. 213–217 (cit. on p. 16).

[26] H.J. van den Herik and H. Iida, eds. *Lecture Notes in Computer Science*. Vol. 1558. Springer, 1999 (cit. on p. 15).

[27] H.Jaap van den Herik, Jos W.H.M. Uiterwijk, and Jack van Rijswijck. "Games solved: Now and in the future." In: *Artificial Intelligence* 134 (2002), pp. 277–311 (cit. on p. 15).

[28] F.-H. Hsu. "A two-million moves/s CMOS single-chip chess move generator." In: *IEEE Journal of Solid-State Circuits* 22.5 (Oct. 1987), pp. 841–846 (cit. on p. 19).

[29] F.-H. Hsu. "Chess Hardware in Deep Blue." In: *Computing in Science & Engineering* 8.1 (2006), pp. 50–60 (cit. on p. 18).

[30] *IBM and Xilinx Announce Strategic Collaboration to Accelerate Data Center Applications*. 2107. URL: https://www-03.ibm.com/press/us/en/pressrelease/48074.wss (cit. on p. 4).

[31] *International Conference on Field Programmable Technology*. URL: http://www.icfpt.org/ (cit. on p. 29).

[32] S. Jin, J. Cho, X. D. Pham, K. M. Lee, S. K. Park, M. Kim, and J.W. Jeon. "FPGA design and implementation of a real-time stereo vision system." In: *IEEE Transactions on Circuits and Systems for Video Technology* 20.1 (Jan. 2010), pp. 15–26 (cit. on p. 9).

[33] M. Al Kadi, P. Rudolph, D. Gohringer, and M. Hubner. "Dynamic and partial reconfiguration of Zynq 7000 under Linux." In: *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. 2013, pp. 1–5 (cit. on p. 12).

[34] R. Kalra and R. Lysecky. "Configuration locking and schedulability estimation for reduced reconfiguration overheads of reconfigurable systems." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 18.4 (Apr. 2010), pp. 671–674 (cit. on pp. 72, 78).

[35] K. Koizumi, Y. ishii, T. Miyoshi, and K. Yoshizoe. "Triple line-based playout for Go-An accelerator for Monte Carlo Go." In: *International Conference on Reconfigurable Computing and FPGAs*. 2009, pp. 161–166 (cit. on p. 19).

[36] A. Kojima. "FPGA implementation of Blokus Duo player using hardware/software co-design." In: *International Conference on Field-Programmable Technology*. 2014, pp. 378–381 (cit. on p. 20).

[37] R. E. Korf. "Depth-first iterative-deepening: An optimal admissible tree search." In: *Artificial Intelligence* 27.1 (Sept. 1985), pp. 97–109. DOI: 10.1016/0004-3702(85)90084-0 (cit. on p. 17).

[38] D. I. Lehn, K. Puttegowda, J. H. Park, P. Athanas, and M. Jones. "Evaluation of rapid context switching on a CSRC device." In: *International Conference of Engineering of Reconfigurable Systems and Algorithms (ERSA)*. 2002, pp. 209–215 (cit. on p. 71).

[39] Z. Li, K. Compton, and S. Hauck. "Configuration caching management techniques for reconfigurable computing." In: *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2000, pp. 22–36 (cit. on pp. 72, 78).

[40] Z. Li and S. Hauck. "Configuration compression for virtex FPGAs." In: *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2001, pp. 147–159 (cit. on p. 72).

[41] Z. Li and S. Hauck. "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation." In: *ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGAs)*. 2002, pp. 187–195 (cit. on pp. 71, 78).

[42] O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, and O. Mencer. "Beyond traditional microprocessors for geoscience high-performance computing applications." In: *IEEE Micro* 31.2 (2011), pp. 41–49 (cit. on p. 9).

[43] S. Liu, N. Pittman, and A. Forin. *Minimizing Partial Reconfiguration Overhead with Fully Streaming DMA Engines and Intelligent ICAP Controller*. Tech. rep. Microsoft Research, 2009 (cit. on p. 72).

[44] S. Lopez, T. Vladimirova, and C. González. "The promise of reconfigurable computing for hyperspectral imaging onboard systems: A review and trends." In: *Procedings IEEE* 101.3 (Mar. 2013), pp. 698–722 (cit. on p. 9).

[45] U. Lorenz and T. Tscheuschner. "Advances in Computer Games SE." In: vol. 4250. 16. Springer Berlin Heidelberg, 2006. Chap. Player Modeling, Serach Algorithms and Strategies in Multi-player Games, pp. 210–224 (cit. on p. 16).

[46] *Match 1 - Google DeepMind Challenge Match: Lee Sedol vs AlphaGo*. Mar. 2016. URL: https://www.youtube.com/watch?v=vFr3K2DORc8&t=1h57m (cit. on p. 15).

[47] L. Ming, W. Kuehn, L. Zhonghai, and A. Jantsch. "Run-time Partial Reconfiguration speed investigation and architectural design space exploration." In: *International Conference on Field Programmable Logic and Applications (FPL)*. 2009, pp. 498–502 (cit. on pp. 73, 75).

[48] A. Nafkha and Y. Louet. "Accurate Measurement of Power Consumption Overhead During FPGA Dynamic Partial Reconfiguration." In: *International Symposium on Wireless Communication Systems (ISWCS)*. 2016 (cit. on p. 80).

[49] J. Noguera and R. M. Badia. "Multitasking on Reconfigurable Architectures: Microarchitecture Support and Dynamic Scheduling." In: *ACM Transactions on Embedded Computing Systems* 3.2 (May 2004), pp. 385–406 (cit. on pp. 71, 78).

[50] J. Olivito, J. Resano, and J. L. Briz. "Accelerating board games through hardware/software codesign." In: *IEEE Transactions on Computational Intelligence and AI in Games* (2016) (cit. on p. 20).

[51] J. Olivito, R. Gran, J. Resano, C. González, and E. Torres. "Performance and energy efficiency analysis of a Reversi player for FPGAs and general-purpose processors." In: *Microprocessors and Microsystems* 39.2 (2015), pp. 64–73 (cit. on p. 19).

[52] K. Papadimitriou, A. Anyfantis, and A. Dollas. "An effective framework to evaluate dynamic partial reconfiguration in FPGA systems." In: *IEEE Transactions on Instrumentation and Measurement* 59.6 (June 2010), pp. 1642–1651 (cit. on p. 72).

[53] A. A. Prince and V. Kartha. "A framework for remote and adaptive partial reconfiguration of SoC based data acquisition systems under Linux." In: *10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. 2015, pp. 1–5 (cit. on p. 12).

[54] Y. Qu, J. Pekka Soininen, and J. Nurmi. "A parallel configuration model for reducing the run-time reconfiguration overhead." In: *Design, Automation, and Test in Europe Conference (DATE)*. 2006, pp. 965–970 (cit. on pp. 71, 78).

[55] *Qualcomm and Xilinx Collaborate to Deliver Industry-Leading Heterogeneous Computing Solutions for Data Centers with New Levels of Efficiency and Performance*. 2015. URL: https://www.qualcomm.com/news/releases/2015/10/08/qualcomm-and-xilinx-collaborate-deliver-industry-leading-heterogeneous (cit. on p. 4).

[56] S. Russell and P. Norvig. *Artificial intelligence: a modern approach*. 3rd. p. 1152. Prentice Hall Press, 2009 (cit. on p. 16).

[57] K. Sano. "SW and HW co-design of Connect6 accelerator with scalable streaming cores." In: *International Conference on Field-Programmable Technology*. 2011 (cit. on p. 20).

[58] K. Sano and Y. Kono. "FPGA-based Connect6 solver with Hardware-Accelerated Move Refinement." In: *ACM SIGARCH Computer Architecture News* 40.5 (Dec. 2012), pp. 4–9 (cit. on p. 19).

[59] A. Shan. "Heterogeneous Processing: a Strategy for Augmenting Moore's Law." In: *Linux Journal* 142 (Feb. 2006), p. 7 (cit. on p. 3).

[60] D. Silver, A. Huang, C. J. Maddison, A. Guez, and L. Sifre. "Mastering the game of Go with deep neural networks and tree search." In: *Nature* 529.7587 (2016), pp. 484–489 (cit. on p. 18).

[61] J. Sim, W.-F. Wong, G. Walla, T. Ziermann, and J. Teich. "Interprocedural placement-aware configuration prefetching for FPGA-based systems." In: *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2010, pp. 179–182 (cit. on pp. 71, 78).

[62] P. Stenstrom, ed. *Transactions on High-Performance Embedded Architectures and Compilers IV*. Vol. 6760. Lecture Notes in Computer Science. Springer, 2011 (cit. on p. 80).

[63] *Stratix 10 FPGA and SoC*. 2017. URL: https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html (cit. on p. 4).

[64] N. Sugimoto and H. Amano. "Hardware/software co-design architecture for Blokus Duo solver." In: *International Conference on Field-Programmable Technology*. 2014, pp. 358–361 (cit. on p. 20).

[65] Y. Tsuruoka, D. Yokoyama, and T. Chikayama. "Game-Tree Search Algorithm Based on Realization." In: *International Computer Games Association* 25.3 (2002), pp. 145–152 (cit. on p. 16).

[66] K. Vipin and S. A. Fahmy. "A High Speed Open Source Controller for FPGA Partial Reconfiguration." In: *International Conference on Field-Programmable Technology (ICFPT)*. 2012, pp. 61–66 (cit. on p. 72).

[67] *Virtex-5 FPGA Configuration Guide UG191 (v3.11)*. 2012. URL: https://www.xilinx.com/support/documentation/user_guides/ug191.pdf (cit. on pp. 73, 77, 80).

[68] *Virtex UltraScale+ Product Table*. URL: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html#productTable (cit. on p. 77).

[69] T. Watanabe, R. Moriwaki, Y. Yamaji, Y. Kamikubo, Y. Torigai, Y. Nihira, T. Yoza, Y. Ueno, Y. Aoyama, and M. Watanabe. "An FPGA Connect6 Solver with a two-stage pipelined evaluation." In: *International Conference on Field-Programmable Technology*. 2011, pp. 1–4 (cit. on p. 19).

[70] C. K. Wong, K. K. Lo, and P. H. W. Leong. "An FPGA-based Othello solver." In: *International Conference on Field-Programmable Technology*. 2004, pp. 81–88 (cit. on p. 19).

[71] I-C. Wu and D.-Y. Huang. "Advances in Computer Games." In: ed. by H.J. van d. Herik, S.-C. Hsu, T. s. Hsu, and H.H.L.M. Donkers. Vol. 4250. Springer, 2005. Chap. A New Family of k-in-a-Row Games, pp. 180–194 (cit. on p. 37).

[72] *Xilinx DS531 Processor Local Bus (PLB) v4.6 (v1.05a)*. 2010. URL: http://www.xilinx.com/support/documentation/ip_documentation/plb_v46.pdf (cit. on p. 74).

[73] *Xilinx DS86 LogiCORE IP XPS HWICAP*. 2010. URL: http://www.xilinx.com/support/documentation/ip_documentation/xps_hwicap.pdf (cit. on p. 74).

[74] *Xilinx University Program Virtex-II Pro Development System*. URL: https://www.xilinx.com/univ/xupv2p.html (cit. on p. 49).

[75] *Xilinx University Program XUPV5-LX110T Development System*. URL: https://www.xilinx.com/univ/xupv5-lx110t.htm (cit. on p. 49).

[76] *Xilinx Vivado Design Suite*. URL: https://www.xilinx.com/products/design-tools/vivado.html (cit. on p. 66).

[77] *Xilinx Vivado high-level synthesis*. 2016. URL: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html (cit. on p. 10).

[78] C.-M. Xu, Z.M. Ma, J.-J. Tao, and X.-H Xu. "Enhancements of Proof Number Search in Connect6." In: *Chinese Control and Decision Conference*. 2009, pp. 4525–4529 (cit. on p. 22).

[79] *Yokogawa WT210/WT230 Digital Power Meters*. URL: http://tmi.yokogawa.com/discontinued-products/digital-power-analyzers/digital-power-analyzers/wt210wt230-digital-power-meters/ (cit. on p. 80).

[80] T. Yoza et al. "FPGA Blokus Duo Solver using a massively parallel architecture." In: *International Conference on Field-Programmable Technology*. 2013, pp. 494–497 (cit. on p. 19).

[81] *ZedBoard Zynq-7000 ARM/FPGA SoC Development Board*. URL: https://www.xilinx.com/products/boards-and-kits/1-elhabt.html (cit. on p. 49).

[82] D. Zhao, Z. Zhang, and Y. Dai. "Self-teaching adaptive dynamic programming for Gomoku." In: *Neurocomputing* 78.1 (Feb. 2012), pp. 23c–29 (cit. on p. 18).

[83] *Zynq-7000 SoC & Zynq UltraScale+ MPSoc, 2016*. 2017. URL: https://www.xilinx.com/products/silicon-devices/soc.html (cit. on p. 3).